University of Glasgow | School of Computing Science

An Animated Emulator of MC6809 for Teaching Purpose

Hui Huang

School of Computing Science

Sir Alwyn Williams Building

University of Glasgow

G12 8RZ

A dissertation presented in part fulfillment of the requirements of the Degree of Master of Science at the University of Glasgow

September 2013

Abstract

Students who learning computer organization and assembly language programming often have difficulty on understanding the relationship between hardware functions and software instructions [17]. To tackle this problem, many educational institutions use Instruction-Set Emulators as teaching tools. However, most of existing emulators are designed as software development assistants, but not for teaching purpose. They tend to focus on functionality of debugging and testing, but not provide proper means to present execution information that would be necessary for teaching and learning. To address the above issues, the project developed an emulator of MC6809 microprocessor, named AEM6809, to help users understanding important concepts and topics in computer organization and assembler language. It provides the function to display animations of data interaction at register level, thus students can actually "see" what happened during the execution. Results from user evaluation proved that this approach is effective and efficiency for the purpose of teaching and learning.

Education Use Consent

I hereby give my permission for this project to be shown to other University of Glasgow students and to be distributed in an electronic form.

Name:	Signature:

Acknowledgements

I would like to thank my parents who sacrificed so much to fund my graduate studies, and my supervisor Dr. Lewis Mackenzie who provided me a lot of suggestions and feedback for my project.

Particularly, I would like to express my deeply grateful here to Dr. Helen Purchase for her consideration and courtesy when I was suffering a hard time during the master project stage.

Thanks to all the wonderful people I met at Glasgow University.

Contents

Chapter 1 Introduction

The term “emulator” was created by IBM in 1963 during their development of the famous IBM 360 [16]. Nowadays, the word “emulate” has been commonly used in the context of software simulation. Emulation is a strategy in digital preservation to combat obsolescence. Even though it is time-consuming and difficult to rebuilding a computer environment in host machine, but valuable because of its ability to maintain a closer connection to the authenticity of the simulated targets. This technique has been widely used in many educational institutions for teaching computer architecture, as it provides appropriate practical experience to support the theoretical concepts of subjects [13].

The project aims at developing an emulator based on MC6809 to help students learning computer organization and assembly language programming. It would allow user to execute MC6809 machine codes, observe the execution of instructions. Usable and friendly user interfaces was implemented to present useful information as well as interact with users.

1.1 Motivation

Students who learning computer organization and assembly language programming often have difficulty on understanding the relationship between hardware functions and software instructions [17]. To tackle this problem, one of the most straightforward approaches is to run codes on real hardware, and then use Integrated Drive Electronics (IDE) to monitoring the execution of that piece of codes, so as to give students insights into principles of processor. However, this approach is difficult to be adopted as it is infeasible to have enough hardware platforms available to every student in lab session, and a significant amount of time are required to train students to use these platforms [17]. In addition, migrating object program from host machine to target machine is time consuming, and any changes in source code will inevitably result in another round of code migrating. Furthermore, repeatedly writing program into nonvolatile storage, for example, an EPROM, tends to reduce the lifetime of that device, which causes more expense.

The other approach that has been widely used in many educational institutions is to use Instruction-Set Simulators [15]. Unfortunately, most of these simulators only cover some aspects of computer systems [3], and they were designed as assistants for software development rather than teaching tools.

Some common characteristics of these emulators are:

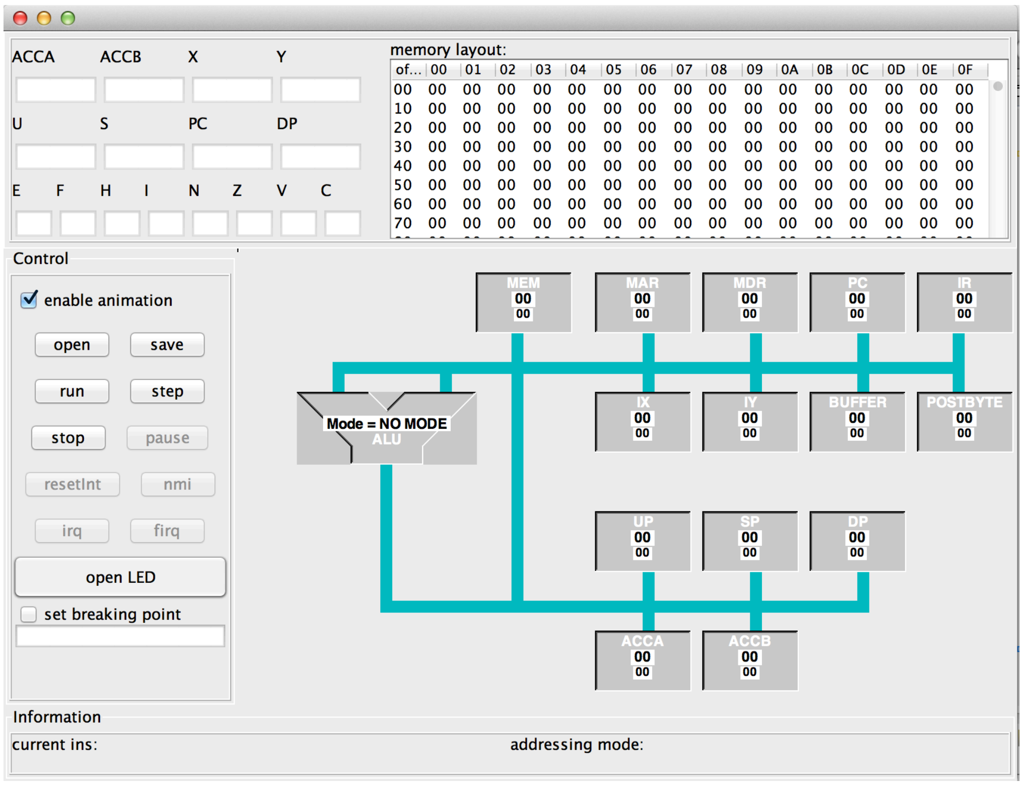
1. Tend to overlook the importance of user interface design, some even do not provide graphical user interface
2. Few efforts are used in showing the data interaction at a register level
3. Difficult to be extended to implement additional functionality

An undeniable fact is that potential users of these emulators are well-experienced engineers, who are familiar with the basic concepts of microprocessor and computer architecture. But novices might have difficulties when using them as learning tools. For example, instruction execution cycles are complicated and dynamically. It is expected to have an emulator, which addresses the three above drawbacks, for teaching purpose. Students could use it to actually "see" the fetch and execute cycles for different instructions with addressing modes, so that accelerate the progress to understand the relationship between hardware and software.

The above example exposes that one of the difficulties students have is being able to visualize how internal components interact with each other to execute different instructions, which is also the major motivation of the project. The main objective of this project is to develop a new MC6809 emulator, named AEM6809 (Animation EMulator) to help students understanding varieties of concepts of computer architecture. The emulator should be capable to present a graphical representation of the processor, and show data interactions at a register level by the mean of animation.

1.2 Project Outcome

Figure 1 shows the resulting emulator. As specified, it is capable of loading S-record files containing machine codes of MC6809, and then simulates the execution of each instruction. Note that, the project does not require providing an integrated assembler due to the time constrains. Users are suggested to use existing MC6809 assemblers to assemble programs (see Appendix D: User Manual) into S-record format file. The emulator then could load it into memory to execute. Data interactions between registers and other components are animated to provide a visualized mean for users.



memory layout:

of...	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
10	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
20	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
30	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
40	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
50	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
60	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
70	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00

Figure 1: Initial screen of the MC6809 emulator

Most of the requirements in the project brief were satisfied, except the emulator only provides one I/O device, a LED panel, because of the limited time available for the project. However, the emulator was designed and implemented as an extendable framework application. Thus it is easy for user-developers to add vir t ual I/O devices if they are necessary in future (see section 5.x). This was also agreed with the project supervisor so that more time could be spent on additional requirements collected during the requirement gathering stages.

Features of the emulator are:

1. Visualizing the execution cycles of each MC6809 instructions, data interactions between registers and other components are animated.

2. The content of memory subsystem could be modified manually to provide a straightforward mean for debugging and testing.
3. The emulator can load MC6809 executable code and data in S-record format.
4. Users are able to trigger hardware interrupts manually during the simulation of instruction execution.
5. A built-in LED panel was implemented, additional I/O device could be added in by simply extending from an abstracted I/O device class.

The rest of chapters are arranged to demonstrate the important aspects during the development of the AEM6809 emulator, especially decision that were made. In Chapter 2, a survey is presented to provide necessary background materials that would be useful during the development of the proposed emulator. It also looks into three previous emulators to find their advantages and drawbacks. The software development processes, from requirements gathering to testing, are illustrated in Chapter 3, 4, 5 and 6, respectively. The last chapter describes the status of final project outcome, and possible further works.

Chapter 2 Survey

To develop a functional emulator that fulfills objectives of the project, it is important to make an investigation of the 6809 architecture, as well as related tools and data structures that would be used to support the implementation.

The first section of the chapter gives an overview of MC6809 architecture, and how instructions are executed. The second section provides some related background materials that would be used during the development of the project. Finally, three previous emulators are evaluated to find their weaknesses and strengths, so that to help the design of the proposed emulator.

2.1 MC6809 Microprocessor

The MC6809, which are the top range processor in the Motorola MC6800 family of microprocessors, were introduced in the fall of 1978. It is an advanced processor of the 6800 family and offers greater throughput, improved byte-efficiency, and the ability to handle new software methods [7]. Compared to its predecessor, MC6809 is still based on 8-bit techniques but also support 16-bit features. One of the most significant enhancements introduced in MC 6809 is the improved programming model. Additional registers such as Y index register and U stack pointer introduced more flexibility of software design. Two 8-bit accumulators can be concatenated to form a 16-bit accumulator, so that operations involved with 16-bit calculation are easier to be performed. It also supports modern programming techniques of that age such as position-independent, system independent and reentrant programming.

2.1.1 Programming model of MC6809

To simulate the execution of MC6809, it is important to understand its programming model. The programming model of MC6809 contains four 8-bit and five 16-bit registers that are programmable to developer. Figure 2 shows the layout of these registers.

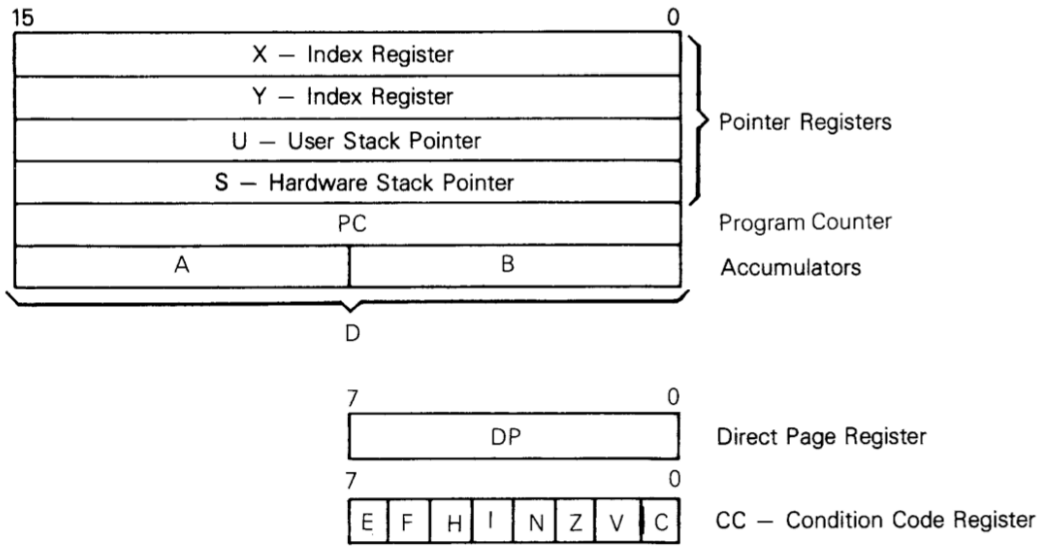


Figure 2: Programming model of MC 6809

Index Registers

Two index registers, IX and IY, are responsible for store base address in indexed addressing modes. Each index register holds a 16-bit address, which could add with an offset to form an effective address. This address then may be used to point directly to data or may be modified by a particular constant to produce another effective address.

Stack Pointer Registers

A stack refers to a data structure for spilling registers organized as a last-in-first-out queue [13]. The Stack Pointer (SP) contains value denoting the most recently allocated address in a stack [13], which could be used to track the old value in the top of a stack or indicate where new data should be pushed in. MC 6809 provides two SP. One is user stack pointer (U) that is exclusive controlled by the programmer. The other one is hardware stack pointer (S). It is controlled by processor in subroutine call and interrupts, but also can be used by programmer.

Program Counter

Program Counter (PC) register holds the address of next instruction to be executed, so that processor can execute a sequence of instructions one by one. For some particular addressing modes, PC plays the role of base-address to calculate effective address

Accumulator Registers

The accumulator registers (A, B) are 8-bit registers that could hold operands for calculations and data manipulation. The most improved feature implemented in MC 6809 is that the two 8-bit registers can be concatenated to form a 16-bit register, where the value held by Accumulator A (ACCA) positioned as the Most-Significant Byte (MSB). This newly formed register is referred to as accumulator D (ACCD).

Direct Page Register

The direct page register (DP) holds the MSB of the address in the direct addressing mode. A 16-bit effective address thus can be calculated by concatenate the DP with operand provided by a direct addressing mode instruction. The point of this register is to allow the direct mode to be used at any place in memory.

Condition Code Register

The condition code register (CC) is an 8-bit register that reflects the state of the processor at any given time. As mentioned above, the results of ALU might set some particular bits in CC. However, it can indicate more states. The format of CC is shown as Figure 3.

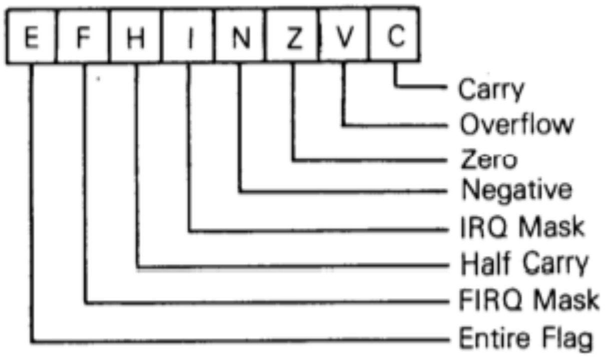


Figure 3: The format of CC register

Bits in CC can be divided into three groups. One is used to indicate the results of ALU, which include Carry bit, Overflow bit, Zero bit, Negative bit and Half Carry bit. The other group includes IRQ Mask, FIRQ Mask bit and Entire Flag. They are used to reflect the states of interrupt. The following table shows the details of these bits.

Table 1: Meaning of bits in CC register

Position	Name of the bit	The meaning when set to 1
0	Carry (C)	A carry or a borrow was generated from bit seven
1	Overflow (V)	The previous operation caused a signed overflow
2	Zero (Z)	The result of previous operation was zero
3	Negative (N)	The MS bit of the result of previous data operation
4	Interrupt Request Mask (I)	Any interrupt request input is disabled
5	Half Carry (H)	A carry was generated from bit three
6	Fast Interrupt Request Mask (F)	Any fast interrupt request is disabled
7	Entire Flag (E)	All the registers were stacked during the last interrupt

2. 1. 2 Interrupts

Interrupt is a signal or software generated event that causes the processor to stop execution of its main program and jump to a special program, an interrupt service routine, that response to the need of the external device [7]. Note that the processor will first finish the execution of current instruction, and then jump to interrupt service routine. Interrupts in MC6809 could be categorized into two classes: hardware interrupts and software interrupts.

The purpose of hardware interrupts is to provide control and act on feedback from peripheral devices such as line printer or machinery controllers. These interrupts include non-maskable interrupt (NMI), fast maskable interrupt request (FIRQ), normal maskable interrupt request (IRQ) and reset interrupt (RESET), which are all caused by signal changes on corresponding pins. On the contrary, the software interrupts are caused by instructions. They are typically used for program debugging and for calls to an operation system. Three software interrupts are available for MC6809: SWI, SWI2 and SWI3.

To provide more flexibility for programming, the MC6809 adopts vectored interrupt technique. Interrupt service routines could be put in anywhere in memory. When a particular interrupt is triggered, the processor will push the content of internal registers into stack as specified, and then load a pair of address stored in reserved memory location into PC. These reserved memory locations are vector addresses. Table 2 shows the vector addresses of MC6809.

Table 2: Interrupt vector addresses

Interrupt Description	Vector addresses	
	MSB	LSB
Reset	0xFFFE	0xFFFF
Non-Maskable Interrupt (NMI)	0xFFFC	0xFFFD
Software Interrupt (SWI)	0xFFFA	0xFFFB
Interrupt Request (IRQ)	0xFFF8	0xFFF9
Fast Interrupt Request (FIRQ)	0xFFF6	0xFFF7
Software Interrupt 2 (SWI2)	0xFFF4	0xFFF5
Software Interrupt 3 (SWI3)	0xFFF2	0xFFF3
Reserved	0xFFF0	0xFFF1

RESET Interrupt

The RESET interrupt is used to start the program. When a RESET is initiated, the process fetches the address stored in RESET vectors (0xFFFE-0xFFFF), then transferred to the PC register. Typically, the RESET vectors contain the start location of program.

IRQ Interrupt

Typical application scenario of IRQ interrupt is when peripheral devices need to communicate with the processor. When an IRQ is initiated, the processor will save current program status by pushing program model into stack, and then load address stored in IRQ vector. To run interrupt service routine without being interrupted, the processor also sets the ‘I’ bit in CC register. The interrupt service routine should end with an RTI instruction to recovery the execution of main program.

NMI Interrupt

As its name implies, the NMI would not be affected by the 'I' bit in CC register. It is usually used to provide response for events that could be catastrophic such as power failure. Same as IRQ, the processor will first store the current program status, and then load NMI vectors into PC.

FIRQ Interrupt

The FIRQ interrupt is a newly added feature of MC6809. Not like IRQ interrupt, FIRQ interrupts only require the processor to push PC and CC registers into stack before accessing the FIRQ vectors.

SWI Interrupt

As mentioned above, three software interrupts are available to MC6809. Compared to hardware interrupts, they are triggered by instructions but not hardware signals. Among these software interrupts, the SWI have the highest priority than others. When processor is processing a SWI interrupt, it will set both I and F bit in CC register to prevent other interrupts from affecting the execution of current interrupt service routine. The rest software interrupts, SWI2 and SWI3, do not have the same priority of SWI. Thus both 'I' and 'F' bit will not be affected when processing them.

2.1.3 Memory mapped I/O

I/O devices in MC6809 system have at least two registers, control/ st at us register and data register. Control/status register is usually used to store codes of operations or current status code of the device. It provides information that is necessary for processor to control its operations. Data register holds I/O data going between the external device and processor. For some devices, they also contain a third register that indicates the direction of data in data register.

In order to control a peripheral, the processor must be able to access the internal registers of that device. In general, there are two approaches to perform I/O operations. One is memory mapped I/O, another one is separate I/O. The MC6809 adopted the former one approach to control and communicate with I/O devices. With this technique, each internal I/O register is assigned with a unique identifier through which it is addressed by the CPU [18]. Thus both memory devices and I/O devices actually shares the same address space.

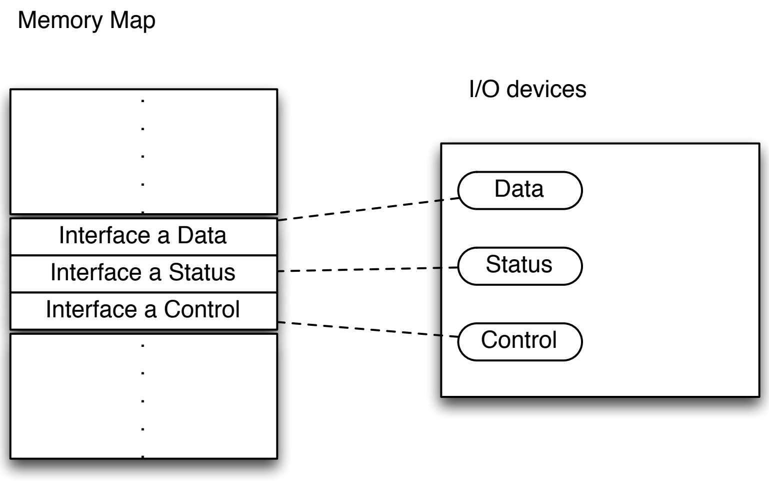


Figure 4: Illustration of a memory-mapped I/O interface.

Figure 4 shows the mapping between memory and a hypothetical I/O device. There are three internal I/O registers, each of them are allocated a unique address among the same address space of memory system. From the standpoint of software, Reads and writes of the device are equivalent to reads and writes of memory, so that the same instructions for memory access could also be used for communication with I/O device. Note that, addresses allocated to I/O registers are no longer available to memory devices. In other words, I/O ports create holes in the memory address space, leaving less room for programs and data [3].

2.1.4 Software execution and instruction cycles

From the point of view of machine, instructions are a series of binary numbers that is stored in memory. Microprocessors fetch and interpret these binary numbers to perform logical and arithmetical functions as expected. Each instruction must specify:

1. The sources of operand
2. The operation to be performed
3. The destination of result

In general, the execution of an instruction could be divided into two stages, the fetch stage and the execute stage, where op-code and operand are prepared at the first stage, and then operations on data are performed at the second stage.

A computer starts to execute instructions by fetching the first byte of an instruction. It first sends address in Program Counter address to Bus interface Memory Address Register (MAR), and then reads the byte of instruction addressed by MAR, store it into Memory Buffer Register (MBR). For most of instructions, the first byte of instruction is an 8-bit op-code. The op-code is then loaded into Instruction Register (IR) to be decoded to determine what to do next.

In most cases, CPU then executes a second fetch cycle to get the address of operand required by the instruction. After everything for execution has been prepared, it then enters into the execute stage to actually perform the instruction. When it has completed the current execution, it returns to fetch stage and reads the next instruction from memory. Note that, during the execution, the PC will be incremented by 1 for every fetch cycle, so that make sure the address of PC keeps pointing to the first byte of the next instruction after current execution.

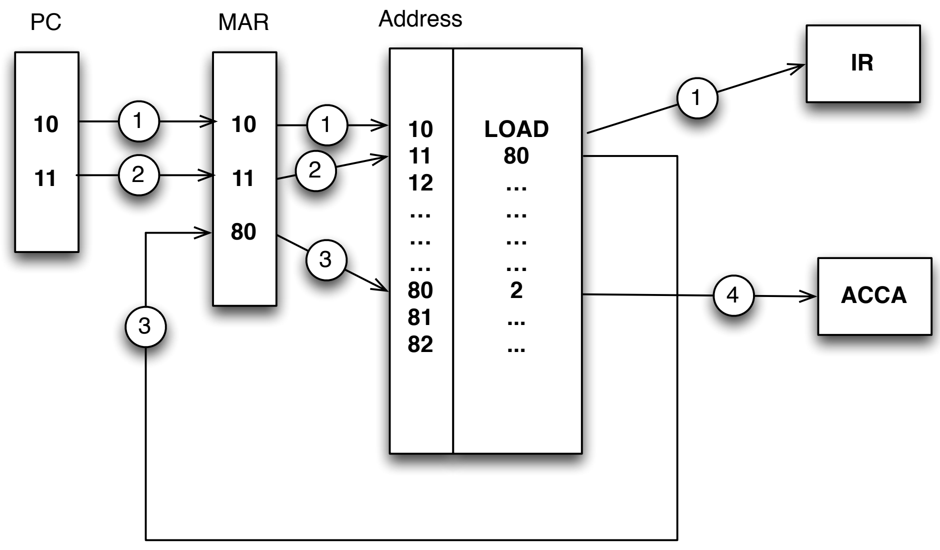


Figure 5: Execution of a load instruction [7]

Figure 5 shows an example of execution of a *LOAD* instruction. In the beginning, the ‘10’ is transferred from PC to the MAR, and the CPU reads the op-code from memory location 10, put it into IR (path 1). After that, another fetch cycle is needed to get the address of operand. In this case, the address of the operand is the next byte following the op-code (path 2). Then the CPU then fetches the operand, and loads it into ACCA (path 3 and path 4).

2.2 Background Material

2.2.1 S-record format

The S-record format is an ASCII hexadecimal text encoding for binary data, which is also known as the SREC or S19 format. A S19 file Format for output modules is for encoding programs or data files in a printable format for transportation between computer systems [10]. Machine codes of programs could be converted into this format, and then loaded into memory to be executed.

An S-record format file consists of a number of specially formatted ASCII strings, each line of string is a record of the program, which specifies the address of memory and the data of that address to be loaded in. The general format of an S-record is shown as the following figure:

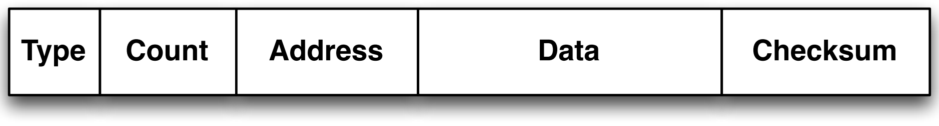


Figure 6: The format of a record in S-record format

Type field

The type field is a 2-byte field. It describes the type of the record and the length of the address interpreted by tis record. There are totally 8 record types, as shown as the following table.

Table 3: 8 record types of S-record

Record	Description	Address Bytes
S0	Block header	2
S1	Data sequence	2
S2	Data sequence	3
S3	Data sequence	4
S5	Record count	2
S7	End of black	4

S8	End of black	3
S9	End of black	2

The S0 record contains vendor specific data rather than program data. It could be used to specify the file name and version info. S1, S2 and S3 record are data sequence of program. The difference between these three types is the size of address needed. S5 record does not have data field, it is the count of S1, S2 and S3 record appearing in the file. S7, S8 and S9 contain the starting address for the program, they also do not have data field.

Count Field

This field contains hex-decimal number with two-byte length. It displays the count of remaining character pairs in the record.

Address

These characters grouped and interpreted as a hexadecimal value, display the address at which the data field is to be loaded into memory. The length of the field depends on the number of bytes necessary to hold the address. A 2-byte address uses 4 characters, a 3-byte address uses 6 characters, and a 4-byte address uses 8 characters.

Data

These characters when paired and interpreted as hexadecimal values represent the memory loadable data or descriptive information.

Checksum

These characters when paired and interpreted as a hexadecimal value display the least significant byte of the ones complement of the sum of the byte values represented by the pairs of characters making up the count, the address, and the data fields.

2.2.2 Related abstract data structures

List ADTs

A list is a sequence of elements, where elements could be added or removed at any position. The elements of a list could be anything from numbers to more complicated objects. List ADTs are usually used to organize data that would be operated in a fixed order. In the case of the project, it was used to hold different kinds of objects such as registers and path information of data moving.

Typically, a list could be implemented by array or linked-list. An array list holds data in an array data structure. Advantage of this implementation is the time complexity of random access is in O (1). However, the size of an array list is fixed during its initialization, which is usually a disadvantage. It needs to be dynamically expended when more slots are necessary, but require copying data. Figure 7 shows the structure of an array list.

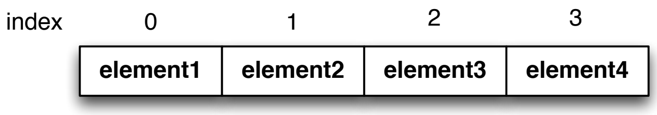


Figure 7: An array list of length 4

The linked-list data structure can also be used to implement lists, whose size could not be determined in advance. Compared with array lists, one of the greatest advantages of this approach is that the list can grow as large as necessary, without the need to copy data from the full array to another one. On the other hand, retrieving an existing element from this kind of list requires to traversal to locate the desired one, which takes more time than array lists. Figure 8 shows the structure of a list implemented by a linked-list data structure.

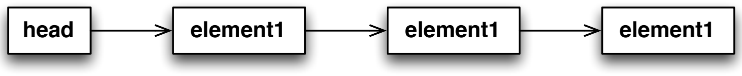


Figure 8: A list that is implemented by linked-list data structure

Graph ADTs

The graph ADTs is important to the project, especially for the implementation of data moving animations. A graph consists of a set of vertices and a set of edges, where each vertex contains a single element and each edge connects two vertices in an arbitrary manner. A graph is usually to store topology information of objects that connected each other. In the case of the project, components that would be displayed in the user interface together with busses that connect them could be treated as an un-directed graph.

The implementation of graph ADTs are varying, one of method is to represent a graph using edge-set. As shown as in figure 9

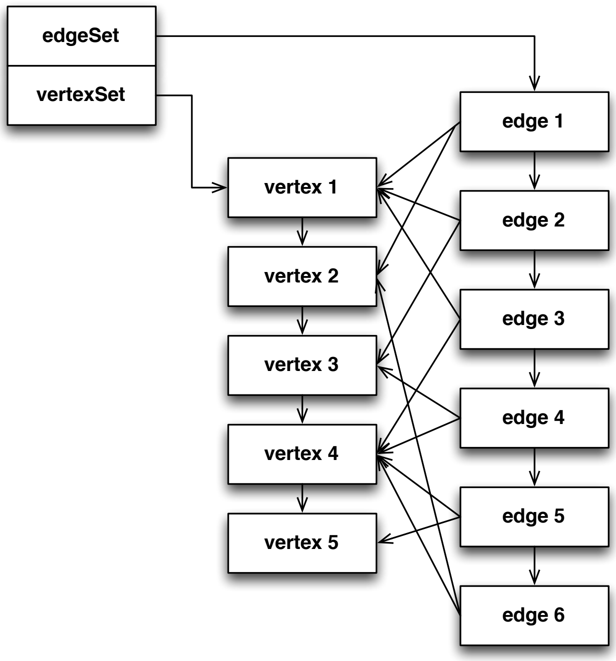


Figure 9: implement a graph by using edge-set

The vertices and edges are organized into linked-list data structure, and each edge inside edge set maintains two references of vertices in vertex set, so as to indicate that there is a connection between them.

The most frequently application scenario of graph is to search for a path between any two vertices. Since the graph in the project is un-weighted, it is suitable to adopt breadth-first algorithm to find a path between two components. The algorithm of breadth-first is shown as the following figure.

```
# To find a path between V1 and V2 in graph g. starting at V1:
```

```
1. Make vertex-queue contain only V1, and mark start
   as reached.
2. While vertex-queue is not empty, repeat:
   2.1. Remove the front element of vertex-queue into v.
   2.2. Visit vertex v.
   2.3. For each unreached successor w of vertex v, repeat:
      2.3.1. Add vertex w to vertex-queue, and mark w as
         reached.
      2.3.2. if w=V2, return the path
3. Terminate.
```

Figure 10: Algorithm of breadth-first search [6]

2.3 Related Works

2.3.1 Sim6809

Sim6809 [9] is a MC6809 simulator implemented by C language, which contains a debugger and disassembler. Because it supports for Intel Hex binary files, the Integrated Development Environment (IDE) was not implemented, users need to write program by a suitable editor and then assemble it into binary format. The emulator was developed under GUN license, so that the source code is open to everyone. This feature enables users to extend this emulator for implementing additional functionalities or vir t ual devices such as LCD screen and IDE interfaces.

Another interesting features implemented is this emulator can display the number of 6809 cycles for a given program or instruction. It is useful to help student understanding the efficiency issues with each instructions. There are always different ways to implement a program for the same functionality, use suitable instructions and addressing modes are important to improve the execution efficiency.

Besides, the debugger of the emulator is also powerful. It could not only monitoring the states of a running program, but also allow users to manually control the program flow. The content in memory and registers could be dumped and displayed in any time when programs are executed in step fashion.

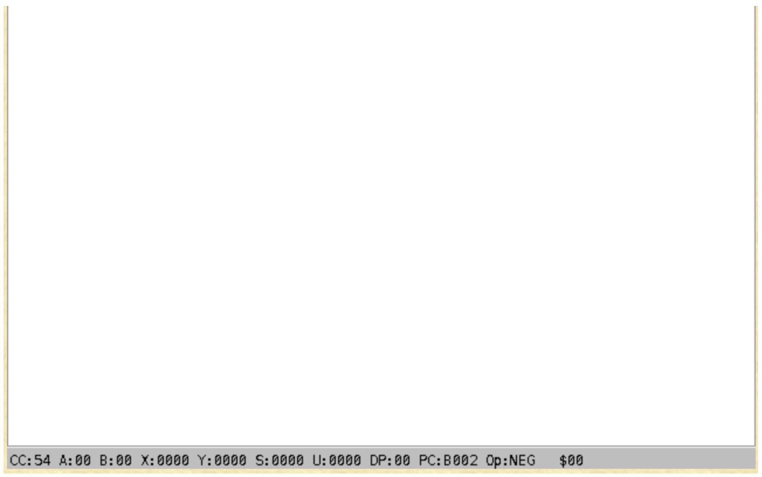
However, a most significant disadvantage is that the emulator does not provide any user interface. The emulator is totally line-oriented, student need to spend time on being familiar with its commands.

2.3.2 J6809

J6809 [19] is a Java version 6809 emulator and assembler, which also provide a simple IDE to edit assembler language. The most unique feature of this emulator is it can run as an application or as an applet, so that users can use it from local or distance. The other features inheriting from Java is the platform independently.

Distance accessibility and platform independent feature is essential for teaching purpose as this could totally bypass any problems in distributing the emulator to each student. All students' need is a web-browser.

However, from the teaching point of view, J6809 still have some disadvantages. One of these disadvantages is that the emulator does not appropriately present the states of a running program. As shown as figure 11, after a program written by assembler language is assembled into machine code, user can then click "tools-->Execute" to run this program. But the text area of the IDE is cleared then, and the only hint about the states of the program is a line of text string on the bottom of the user interface. It is not capable to clearly illustrate the effects of instructions to novice student.



```
CC:54 A:00 B:00 X:0000 Y:0000 S:0000 U:0000 DP:00 PC:B002 Op:NEG   $00
```

Figure 11: The states of microprocessor when a program is running

In general, this emulator is sufficient for emulating the behavior of MC 6809, but lack of capability as a teaching aid. It acts like a “black box” emulator because it hides most detailed information about the states of a running program.

2.3.3 Motorola MC6800 graphical emulator

As stated by its name, this is an MC6800 emulator, rather than MC6809. However, it is still worth to be assessed here because the graphical simulator also aims at teaching purpose. The motivation of the graphical simulator is to help the user to understand the cycle-by-cycle operation of the M6800 microprocessor, thus it gives a good illustration of the microprocessor blocks, all connected by single line busses.

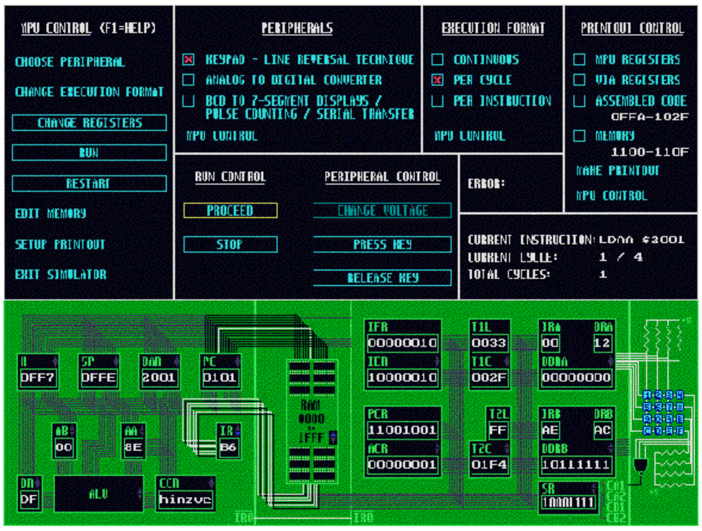


Figure 12: Graphical user interface of MC6800 graphical simulator

Figure 12 shows the graphical user interface of MC6800 graphical simulator. At the top area, several control windows provide variety execution options. The bottom area shows the internal architecture of MC6800, single line busses would change color to show the path of data moving. It also allows several execution formats, program could be executed continuously, step or even run per clock.

In general, it is a good emulator that achieves its original purpose. However, it could have been made more user-friendly. The first drawback of the simulator is that the direction of data moving was not shown by animation. Novices might be confused to distinguish the source and destination of each data moving. Another disadvantage is that the simulator reveals too much detail of the internal architecture and components, which seems too complicate to be used by students.

Chapter 3 Requirements Analysis

This chapter demonstrates the requirement gathering process of the project as well as a general picture of key functional and non-functional requirements identified during this stage

3.1 The requirements process

The purpose of requirements gathering is to clarify what a system should do to correct the problem. Thus its success or failure can serious affects the rest of the development activities. During the requirements gathering and analysis phase, it is expected to build a relative firm and stable base for future development activities.

Because of the particularity of master projects, the first stage of the requirements process was to identify correct and suitable stakeholder of the project. Due to constrains such as time and resources available to the project, the stakeholders were identified and limited to the project sponsor, Dr. Lewis Mackenzie and a group of MSc IT students who attended the System and Network course.

The gathering process started with a project brief that generally describes the problem and expected project outcomes. The project brief was then extended to a set of vague requirements to build an initial project scope. To elicit more detailed requirements, the following approaches were adopted:

1. Questionnaire. Questionnaires were compiled and distributed to 10 MSc IT students. Since the initial project scope is vague and open, questions are open and request natural language unstructured response.
2. Evaluation of previous emulators. Three previous emulators were assessed to find their strengths and drawbacks, as specified in section 2.3. It is not usual to collect requirements from previous works. However, due to the nature of the master project, these existing emulators could inspire the author as well as help to avoid design and implementation defects in advance.
3. Interview. The above two approaches were used to collect requirements from the perspective of students and developers. To collect requirements from

course lecturer, a number of interviews were conducted with the project supervisor.

The collected data then was combined and analyzed to generate raw user requirements for the system. These requirements were documented and organized to produce a collection of use cases.

These use cases then were refined and validated. Firstly, all requirements were examined by the following four criteria: necessary, reality, testable and quality, learning from the Requirements Engineering course. Requirements that could not satisfy one of these criteria must be revised or even deleted to make sure the project is feasible under the limited time and resources.

Since the main factor that could seriously affect the success of the project is the limited time that is available for the development, requirements were prioritized using the MoSCoW prioritization method [1]. The most important requirements would be fulfilled first, and the less important ones would be implemented only if there is adequate time left. This development tactics helped the author to focus on requirements essential for meeting users' goals.

3.2 Functional Requirements Overview

It is beyond the purpose of this chapter to discuss the gathered requirements agreed with the stakeholders. However, to help understanding the scope of the project, this section presents key functional requirements that would be critical to the project.

Functional requirements describe a set of function of the system. According to the requirements process, it is clear that the most fundamental functional requirement is an application that could simulate the behavior of MC6809. To achieve this goal, a set of functions that could control the simulated code execution process are necessary, such as load program into memory, run program continues or step and stop or reset processor.

Another important functional requirement, which is also the key point to fulfill the teaching purpose of the emulator, is to display animation of each instruction cycle on the user interface. Controls of animation are also necessary, the animation should be able to be disabled, and the speed of animation should be adjustable.

3.3 Non-functional Requirements Overview

Non-functional requirements refer to the emergent properties of a system that cannot be tested by the presence or absence of a feature. According to the check list presented in [2], the identified non-functional requirements include:

1. Platform independent, compatible with mainstream operating system
2. Easy to be extended in future
3. Be able to accept machine code in the form of S-record
4. The emulator should be intuitive and easy to use.

Chapter 4 Analysis and Design

This chapter illustrates the evolution of the design process, as well as discussion of design alternatives and how decisions were made. The first section explains the solution of unsigned number storage problem cause by Java programming language. The design processes of software architecture and user interfaces are demonstrated in 4.2 and 4.3 sections, respectively.

4.1 Store Unsigned Numbers in Java

How to store unsigned binary numbers is the most fundamental design decision to be made. In high-level programming language such as C or C++, simply put numbers into unsigned integer data type with suitable length could solve this problem. However, Java does not provide unsigned data type, contents in both ***byte*** (8-bit) and ***short*** (16-bit) data type are automatically treated as two's complement numbers. Hence it is infeasible to use them to store 8-bit and 16-bit unsigned binary numbers directly. For example, the range of byte data type in Java is from -128 to 127, but the range of an unsigned 8-bit number is from 0 to 255.

One possible solution is to use the java build-in class ***Stream***, which is designed to help Java programs to interact with disk files and other sources of bytes and characters [8]. Methods of ***DataInputStream*** class can read unsigned byte from data stream. But this scheme is not suitable for the project because the coding would be too complicate as it requires converting every number into data stream type before manipulate on them.

Since the nature of binary number is just a serials of 0 and 1. Whether it is an unsigned number, signed-magnitude number or two's complement number depends on how to interpret it. Solution for this project is to use ***int*** data type to store both 8-bit and 16-bit unsigned binary number, and ignore sign-bit of the original data. Numbers thus are treated as unsigned numbers from the point of view of simulator. When data needs to be converted to two's complement number, the 7th bit of the number is the sign bit. The following figure shows the bit layout of an 8-bit number stored in ***int*** data type.

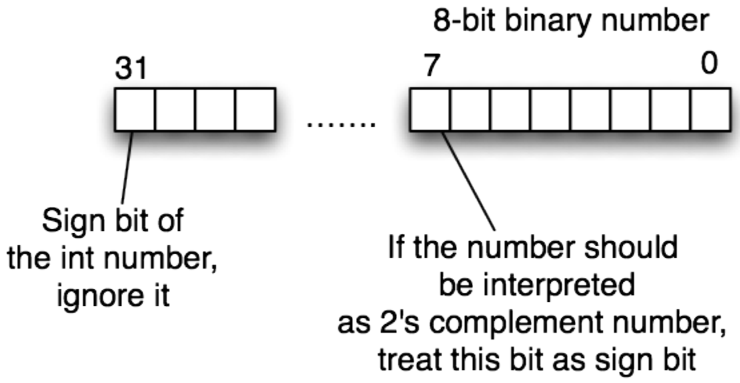

8-bit binary number
31
7
0
....
Sign bit of the int number, ignore it
If the number should be interpreted as 2's complement number, treat this bit as sign bit

Figure 13: use ***int*** data type to store an 8-bit binary number

Another issue related to this scheme is how to perform unsigned arithmetic operations and determine the status of results. Considered that in real microprocessor, when an unsigned overflow occurred, the processor will cut the result off to fit its' word length. Based on this fact, unsigned addition and subtraction could be performed by using the following formulas:

$$\mathrm{re}sult = a + b \,\&\, LIMIT$$

$$\mathrm{re}sult = \begin{cases} a - b & if\ a\text{-}b \geq 0 \\ a - b + LIMIT & if\ a\text{-}b < 0 \end{cases}$$

Figure 14: unsigned arithmetic operations of the project

Where LIMIT equals to 0xFF when operands are 8-bit, 0xFFFF when operands are 16-bit.

To test whether an operation caused a two's complement overflow, approach from [14] was adopted. For addition operation, overflow occurs when adding two positive numbers and the sum is negative, or vice versa. For subtraction, overflow occurs when subtract a negative number from a positive number and get a negative result, or vice versa. Algorithm of bit V test is:

```
# test two's complement overflow for a + b = c, or a – b = c
if a + b
  if sign(a)=sign(b) and sign(c)!=sign(a)
    claim two's complement overflow
else
  if sign(a)!=sign(b) and sign(c)=sign(b)
```

```
        claim two's complement overflow
    Terminate and claim no overflow occurred
```

Figure 15: algorithm for bit V test

Approaches to determine the rest of status of results are shown in the following list:

- `Z bit: set Z if result=0`
- `C bit: set C if result > LIMIT or result < -LIMIT`
- `N bit: set N if result & 0x80 != 0`

4.2 Analysis and Design Process

Fair software architecture is critical if a project is to succeed. In general, there are two common approaches to design software architecture. The first one is that designers start with nothing, and then build-up an architecture from existing software components until it satisfies the requirement of proposed system. The second is that designers start with the system as a whole entity, without any partitions, or boundaries between internal components. Software engineering principles or design patterns can then be applied step by step to split components that are needed from the whole system. The project adopted the second one, as this approach is better to understand the system context.

4.2.1 Starting with a single class

According to [5], a null style describes a system in which there are no distinguished boundaries between components, which could be treated as starting point during the design process. From a Java programming perspective, a null style represents a single class in a single package that implements all the functionality of intended system.

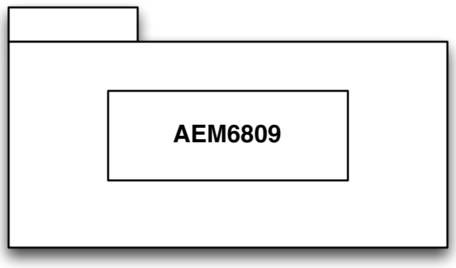


Figure 16: Null style of the project

Figure 16 shows the null style of the project, the single class could be thought as a whole and integrated emulator. Not only user interfaces but also core functionality and event handling process are implemented within it.

4.2.2 Separation of concerns

It is obviously that coding for null style is too complicated as it includes everything into a single structure. To dominate the complex, the first software engineering principle added to the system is separation of concerns. Model-view-controller (MVC) design pattern was adopted to meet the requirement. The advantages of MVC are that it takes apart the user's interaction from the event handling and the processing of information, so that systems are easy to be developed and maintained. System architecture after adopted MVC design pattern is shown as the following figure.

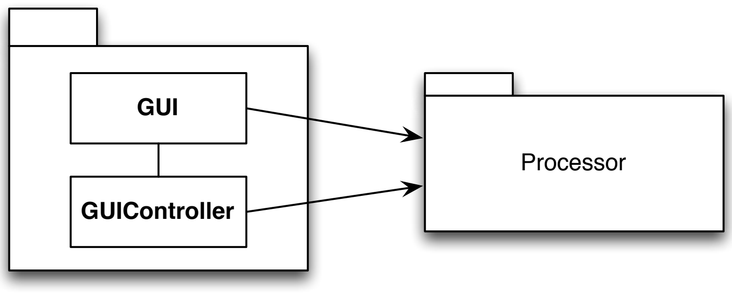


Figure 17: Architecture after adopt MVC design pattern

The ***GUI*** class plays the role of view in MVC design pattern; it shows a graphical user interface and display information getting from *Processor* package. ***GUIController*** is responsible for event handling; it directs user inputs to processor, collects response, and then sends the response back to user interface. *Processor* package models data of the system, and provides core functionality that the system needs to simulate the execution of MC6809.

4.2.3 Deriving memory subsystem

The next step is to detail the design of *Processor* package, continue splitting components from it to follow the rule of separation of concerns. The first component that was separated away is the memory. It is a natural decision as memory subsystem is independent from processor in most of computers. This arrangement can reflect the structure of computer, so that help other programmers who need to extend the project to understand the source code.

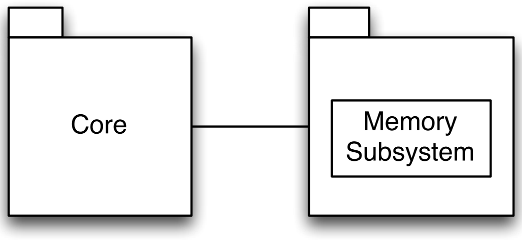


Figure 18: Split memory subsystem from core

As shown as in figure 18, a memory package is derived from the Processor package now. Inside this package, ***MemorySubsystem*** class maintains an array of 8-bit storage data structure (see implementation section) to represent the main memory, provides methods to access data stored of different memory addresses and protect read only memory (ROM) from illegal write operation.

4.2.4 Split execution cycles into different phases

One of the challenges of the project is how to design reasonable and simple software architecture for altogether 1464 available op-codes with unique addressing mode. As required by abstraction principle, the execution of the instructions should be modularized to reduce its complexity. The most intuitive approach is to divide execution cycles into different phases, and then abstract and extract the most common operations as re-usable code fragments or methods. Based on the features of mc6809, execution cycle could be divided into four phases.

Op-code fetching

The first phase is to fetch op-code from memory, then load it into IR register. This phase is necessary to every instruction's execution. Op-codes are usually in byte length, but if the op-code is 0x10 or 0x11, another byte next to the op-code need to be fetched to determine the exactly instruction.

Determine addressing mode

The next phase is to determine the addressing mode specified by the op-code fetched in the first phase. According to Table 9 of MC6809 manual book [11], addressing modes of most of instructions are specified by some bits of their op-code. For example, if the op-code & 0xf0 equals to 0x00, 0x90 or 0xd0, then the addressing mode is in direct addressing.

Operand fetching

After the addressing mode is determined, the third phase is to fetch operand that is needed for the instructions from memory. This phase might not be necessary for all instructions because operands of inherent or register addressing mode are specified by op-code, but not stored in memory.

Operations on operand

The final phase is to manipulate operands as specified. Even though different instructions have different behaviors, however, common features of them could be further abstracted. The clue is whether the instruction will affect the content of CC register, if so, operations that specified by the instruction must involves the using of ALU. These operations that will affect one or more bit of CC register thus were extracted into **ALU** class.

According to the above analysis, the **MC6809** package was elaborated further, as shown as in figure 19

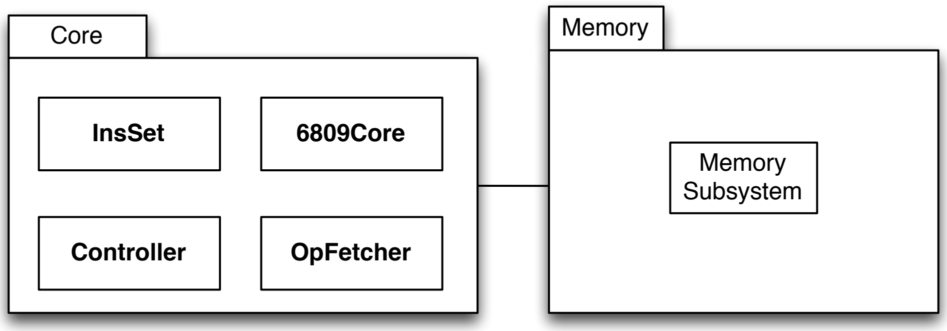


Figure 19: Architecture after elaboration

The ***OpFetcher*** class is responsible for fetching op-code, determine addressing mode and then fetching operand. The ***ALU*** maintains a list of ***boolean*** variable to represent the CC register, provide functions that might affect one or more bit in CC, such as integer arithmetic and logical operations. The ***InsSet*** includes a set of methods to describe the operations on data for each instruction.

The purpose of ***Controller*** class is to takes control of data moving from one register to another one, or in another word, which plays the role of an internal bus controller. This class seems un-necessary, because most of existing simulators do not simulate the behavior of internal bus. The reason for this arrangement is the proposed simulator requires recording paths of data moving so as to display animations for each instruction on user interfaces. For example, if one instruction needs to pass the value of accumulator A to PC, simply assign this value to a variable that represents PC register is not enough. Controller class should perform the assignment. Path information of this transmission then is stored. When the execution is completed, user interface can display animation by interpreting the path. Details of the controller class and data structure to store paths information are explained in section 5.3.1.

4.2.5 I/O devices

The next step was to design the I/O devices sub-system of the proposed emulator. Since there is varieties of I/O devices could be connected to MC6809 system, it is impossible to implement all of them due to the limited time available for the project development. The I/O devices sub-system should be flexible and extendable so that virtual devices that are necessary for teaching could be added in future. For achieving this purpose, a hierarchy structure of I/O devices was built-up, as show as in figure 20.

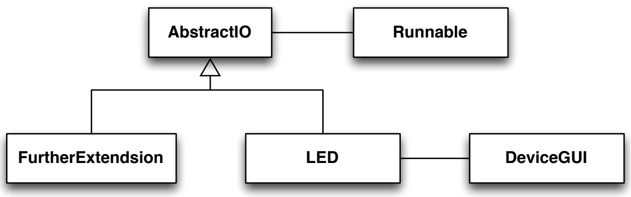


Figure 20: UML class diagram of I/O devices sub-system

The highest-level class in the hierarchy is the ***AbstractIO*** class. It implemented ***Runnable*** interface so that I/O devices inherit from it could run in a separated thread. Sub-classes need to specify operations of I/O devices. For example, the ***LED*** class should light up or extinguishes LED lights as specified by content in data register.

Note that ***AbstractIO*** does not provide GUI for each I/O device. Programmers who extend this class are responsible for implement the graphical user interface if necessary.

4.2.6 Class hierarchy of storage unit

Now that the basic architecture of MC6809 Package has been built up, it is necessary to design the class hierarchy of different storages. In general, there are two types of storage units: a) registers and b) 8-bit memory cell. From the point of view of emulator, the only difference between registers and memory cell is the length of data as some registers are able to store 16-bit value. Recall that the AEM6809 uses ***int*** data type to store both 8-bit and 16-bit value because Java does not provide unsigned data type, thus this difference should be reflected by conditions of modification, but not using different type of class instance.

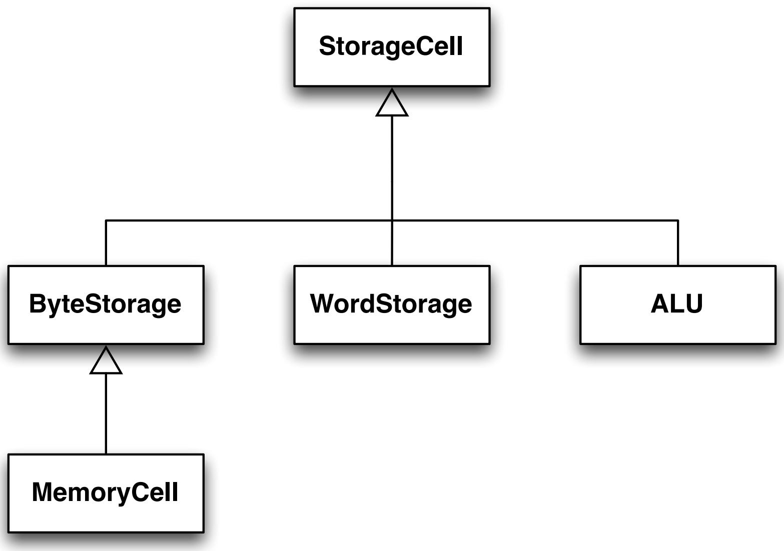


Figure 21: Class hierarchy of storage units

Figure 21 shows the class hierarchy of storage units. The ***StorageCell*** class is the most abstracted class where every subclass is inherited from it. The ***WordStorage*** class would be used to represent 16-bit registers such as PC, SP and MAR. The ***ByteStorage*** represents 8-bit registers. Note that the memory cell could be further classified into ROM and RAM, where the former one could not be modified during the running of the simulator. But considered that users might want to modify the content in ROM manually for the purpose of debugging or testing, it might be inappropriate to protect ROM from writing when the emulator is not executing instructions. Thus there is no need to create two separated classes to represent ROM and RAM location. The responsibility of protecting ROM from modifying should be assigned to higher-level classes.

Extending ***ALU*** class from ***StorageCell*** was slightly tricky. The ***ALU*** class should not have been added into this hierarchy, as it does not store data. The reason for this arrangement is because the ***ALU*** could also be the source or destination for data moving, for example passing the data hold by ACCA into ALU, shift it left and then send it back to ***ACCA***. The same appearance of ***ALU*** and other registers simplifies the implementation of data transmission functionality of ***Controller*** class.

4.2.7 Overall design

The final step of software design process is to refine the software design. After combine all the components and packages together, a three-tier system architecture was built up, as shown as in figure 22.

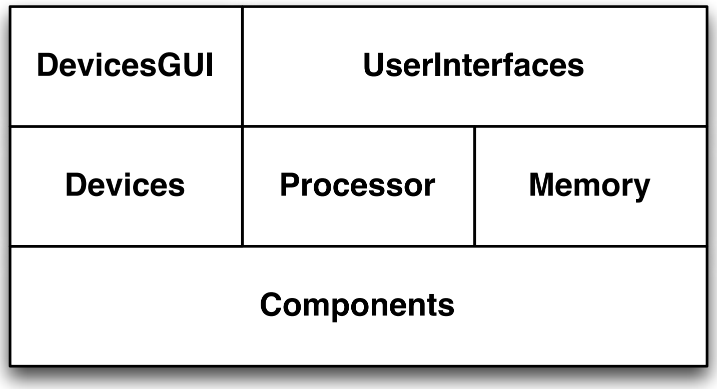


Figure 22: Overall design of the proposed simulator

The most concreted layer is the Components layer, which contains classes that extend from ***StorageCell*** class. The functionality layer contains processor, devices and memory packages. It actually performs the simulation of MC6809 microprocessor and connected peripherals. The highest layer is the user interface layer. The responsibility of this layer is to interact with users and present data getting from functionality layer.

4.3 Design of User Interface

Since the basic architecture of MC6809 Package has been refined, it is necessary to decide how to interact with users as required by the project requirement specification. How to present data interactions between registers is the start point to determine internal components that are necessary to be display in user interface. Other requirements are then added in to form an integrated graphical user interface.

4.3.1 Design of CPU Panel

The CPU Panel consists of a set of registers interconnected by a common bus, this area is used to display animation for the fetch and execute cycle. The first step to design the CPU Panel is to determine which registers and components should be displayed on the screen.

As discussed in section 4.2.4, the execution cycle starts with fetching op-code from memory, therefore registers includes IR, PC, MAR and MDR must be put on the panel. For showing interactions between processor and memory subsystem, memory location that is currently accessed by processor should also be displayed. After the op-code is loaded into IR, the processor will decode the IR, calculate effective address specified by addressing mode, and then fetch operand when needed. Hence, registers of 6809 programming model are all necessary to illustrate the progress clearly. ALU is also included in, as most of instructions need the help of ALU to perform integer arithmetic and logical operations on data. The prototype of CPU Panel is shown in figure 23

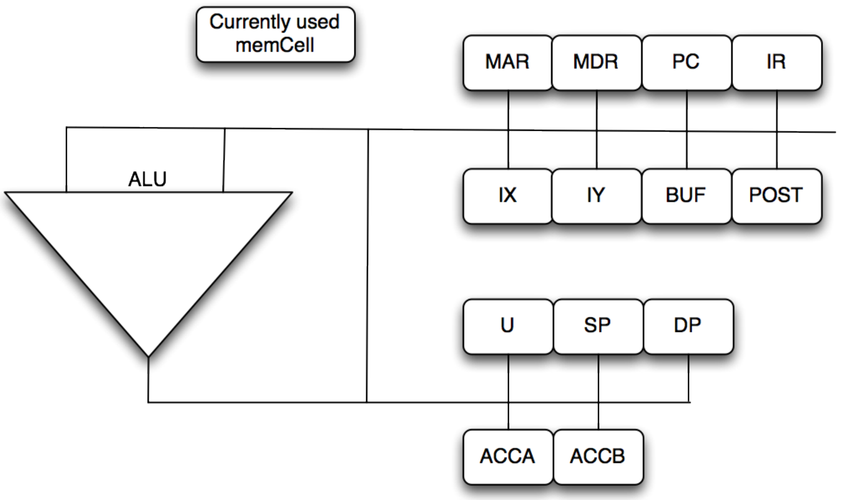


Figure 23: Prototype of CPU panel

Note that in figure 23, there are two components that do not exist in real mc6809 processor, BUF and POST. The BUF block is used to assemble word operand fetched from memory, otherwise 16-bit operations could not be animated. The POST block is design to hold the post byte of indexed addressing mode, so that student could clearly see the process of post byte decoding in indexed mode.

4.3.2 Design of Component diagram

The general display options have been specified, it is necessary to choose the information that needed to be display inside each component diagram. In general, each diagram of the processor should contain a label of name, so that user can see the information required. For diagrams represent registers and memory cells, it is expected that the previous value could be kept on the screen so that students can compare it with the new value. For the ALU diagram, the operation that is currently invoked and the result of the operation are required. Diagrams of registers and ALU is shown as figure 24 and 25

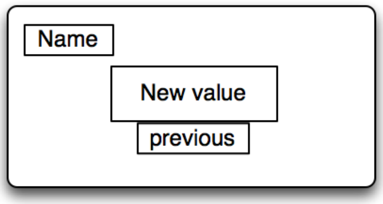


Figure 24: Diagram of registers

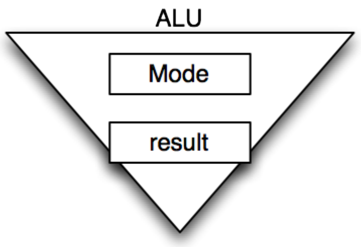


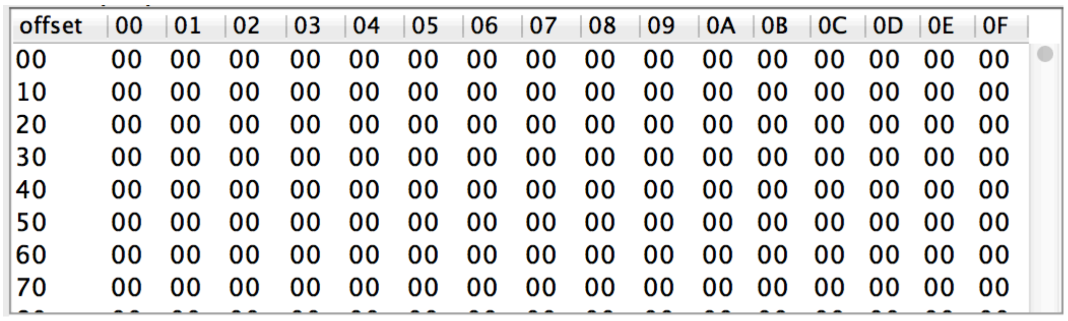
Figure 25: Diagram of ALU

4.3.3 Design of Animation Format

There are a few ways to animate the data interaction between internal registers. The simplest animation would be t o represent active of source block, destination block and buses connected these two components by changing their colors. This fashion gives sufficient information to users about the sequential changes in the values of CPU registers. However, the direction of data flow could not be illustrated clearly by this level of animation, as it provides little information about the source and destination of that data moving. This could be resolved by showing the value of the data being transferred actually "floating" along the path between the two blocks. This scheme would be ideal as it shows the execution of the instruction with very explicit data movement details.

4.3.4 Memory layout and internal registers list

As specified in the requirement specification, it is expected that the simulator could display contents of main memory and allows users to modify value of a chosen address. A memory table is used for displaying memory content, as shown as in figure 26.



offset	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
10	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
20	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
30	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
40	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
50	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
60	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
70	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00

Figure 26: Memory layout table

The first row of the table is the address offset, from 0x0000 to 0xfff0. Each column displays the value from address offset to offset+0xf. For example, the content of address 0x0041 is located in the fifth row, which means offset of 0x0040, and the second column, which means the location of 0x0041.

About the internal registers, even though the CPU panel has already contained registers of the programming model. However, one of the requirements gathered is that the animation should be able to be disabled to accelerate the execution of program. Hence a list of the content of registers is still useful when users want to run program without showing animation.

Chapter 5 Implementation

After the requirements and software architecture were established and refined, the simulator was developed by a down to up fashion. The most concrete layer was implemented first, and then the middle layer that provides simulation functionality. The final stage is the implementation of the user interface layer so as to build an integrated simulator that can fulfill the initial requirements of the project.

This chapter outlines the important aspects of the implementation of the simulator by the order of development. Algorithms and data structures that were relatively difficult to implement, or affect the functionality of the system are also discussed.

5.1 Implementation of Components Layer

The components layer was the most concrete part in the simulator. This layer contains only one package, components package. As specified in section 4.2.6, classes within this package were organized into hierarchy.

5.1.1 Super class implementation

The most abstracted class is the ***StorageCell*** class, which describes common characteristics among different kinds of storage unit. Technically speaking, the ***StorageCell*** could be implemented as an interface. The reason for building it as a class is that most behaviors of subclasses are same. This arrangement can reduce repeated codes.

According to analysis, contract of ***StorageCell*** class is:

1. An *int* type class instance to store the value holds by the storage unit
2. A *String* type class instance to store the name of the storage unit
3. An *int* type class instance to store the ID or address of the storage unit
4. A *boolean* type class instance to indicate if it represents a register or a memory location
5. A number of setter and getter methods to access the above class instances

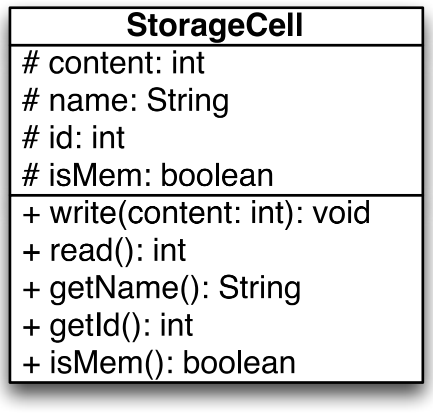


Figure 27: UML diagram of StorageCell

Figure 27 shows the refined UML class diagram of the ***StorageCell*** class. Note that this class is decorated as abstract class to protect it from instantiating. In addition, implementation of *write()* method was not specified here, as it behaves vary among different kinds of storage units. Subclass must override the method to implement required operations.

5.1.2 Sub-classes implementations

As discussed in Chapter 4, the difference between ***ByteStorage*** and ***WordStorage*** is the length of binary number that could be put into them. ***ByteStorage*** should only be able to hold 8-bit unsigned numbers while 16-bit unsigned numbers for ***WordStorage***. There are two implementation options to protect them from illegal writing. The first is to transfer the responsibility to callers. Class should throw an exception when writing numbers that are out of range into the data structure. Advantage of this option is data could be protected safely from unexpected modification. However, flexibility of program will be degraded, as exception handling would be needed when trying to modify the content of storage units.

Option two is simpler; values out of range would be cut-off to fix the length of the storage unit. This option was adopted in the project because real chips also take the same actions. Algorithms of write method in ***ByteStorage*** and ***WordStorage*** are shown in figure 28, 29.

```
# write a into ByteStorage
if a < 0
   return
```

```
else
  ByteStorage.content = a & BYTE_LIMIT
```

Figure 28: write a number into ByteStorage

```
  # write a into WordStorage
if a < 0
  return
else
  WordStorage.content = a & WORD_LIMIT
```

Figure 29: write a number into WordStorage

As discussed in section 4.2.5, the ***MemoryCell*** class could be used to represent both ROM locations and RAM locations. It maintains a private *boolean* class instance, *isRom*, as the flag if the location is a ROM location. Note that if the class implemented a new write method to protect ROM from illegal writing, then users would not be able to modify the content of ROM location manually when the emulator is not executing instructions. This responsibility should be transferred to ***Controller*** class, because all data moving of instruction cycles was implemented there (see section 5.3). Thus it is enough to simply add a new public method, ***isRom()***, to return the type of the memory location.

5.2 Implementation of memory package

The implementation of functionality layer is important to the success of the project, as it provides functions for processor or I/O device simulation. The layer contains two packages, memory and processor. Considered that the structure of memory system is relatively simple, and functionality of processor and devices packages relay on memory access, the *memorySystem* package was implemented first.

5.2.1 Data structure to hold the memory locations

The purpose of *memorySystem* package is to simulate the behavior of main memory and provide processor with memory access operations. The first thing need to be considered is how to maintain a list of ***MemoryCell*** objects to represent the main memory. Since the size of main memory could be fixed during the initialization, and the most frequently operations are read and write, an array data structure was used to store the objects of ***MemoryCell***, indices of each slot would be used to represent addresses of each memory location. Advantage of this implementation is its simple, and the time complexities of read and write operations are both in O (1).

5.2.2 Separate RAM and ROM

Recall that the responsibility of protecting ROM from modifying during the execution was transferred to ***Controller*** class. The memory array should be separated into two parts, one for store RAM locations, and another one for store ROM locations. Typically, the highest addressable space of MC6809 is used as interrupt vectors, which are usually sitting in ROM. Thus the emulator would assume that the RAM is always starts with 0x0000, and end with a user specified address, which is hold by an *int* type class instance, *ramEnd*. Thus slots in memory array whose indices are less than *ramEnd* stores RAM object (the *isRom* variable is false), and the rest of slots stores ROM object.

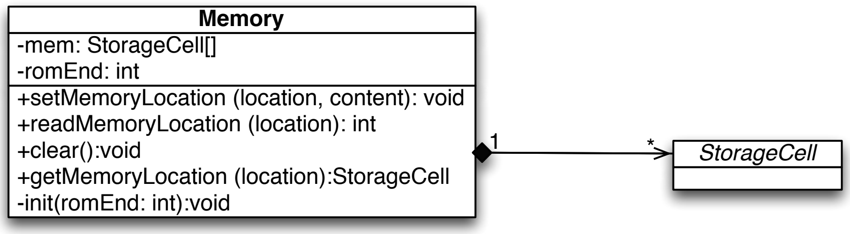

Memory
-mem: StorageCell[]
-romEnd: int
+setMemoryLocation (location, content): void
+readMemoryLocation (location): int
+clear():void
+getMemoryLocation (location):StorageCell
-init(romEnd: int):void
1
*
StorageCell

Figure 30: UML class diagram of memory subsystem

Figure 30 shows the refined UML class diagram for memory subsystem. The association between ***Memory*** class and ***StorageCell*** class is one-to-many and directed. Within Memory class, Two class instances, *mem* and *ramEnd*, were used to store a list of memory location and the end address of RAM location, respectively.

5.3 Implementation of Processor Package

After the implementation of memory subsystem, the next step was to implement the processor package. The package was developed by a general-to-specific fashion. The first stage is to build up a framework application that would allow for executing MC6809 instructions. Then instructions were added into this framework to complete the functionality of the simulated microprocessor. During the development, it is proved that the strategy could remarkably reduce the need for modifications.

5.3.1 Building the framework application

Four classes that are necessary for the execution of instructions were created and put into the processor package. These are ***Controller***, ***OpFetcher***, ***MC6809*** and ***InsSet***. This section illustrates the important aspects of framework development.

Controller class

The ***Controller*** class is the most basic component within the processor package, whose purpose is to take control of data moving on bus, whilst record information that is necessary for showing animation of that moving in user interface. To describe a data moving, the following parameters are necessary:

1. Source: register or memory locations where the data comes from
2. Destination: register or memory locations where the data goes
3. Data: data being transferred
4. Mode: operations on data, before storing it to destination

Note that in fact, it is enough to describe a data moving by just using the first three parameters. The reason for *Mode* parameter is to provide more information to user when showing animation of data moving. For example, as shown as in figure 31, in the case of incrementi ng PC, it would be clearer if a label marked as "+1" is passed into PC register, rather than simply carry the new value of PC into the register.

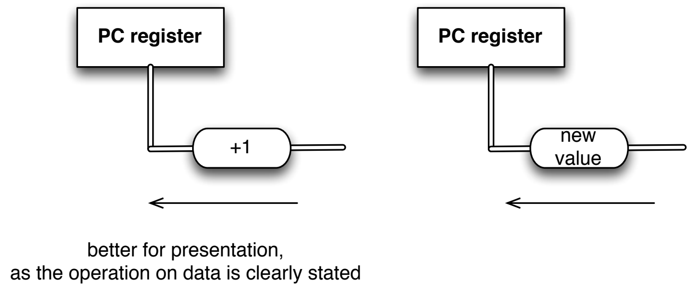


Figure 31: Two different ways to show animation of PC+1

Table 4 illustrates all possible modes that are required. In most of the cases, *PASS* mode would be used to express the simple data transmission between two registers.

Another four modes were mainly used to express procedure of effective address calculation in indexed addressing mode, relative addressing mode and directed addressing mode.

Table 4: Modes of Controller class

Mode	Meaning
PASS	Transfer the content from source to destination
PLUS	Plus the content with the value of destination, and then assign the result to destination
MINUS	Subtract the content from the value of destination, and then assign the result to destination
PLUS_2	Convert the content into two's complement number, plus the number with the value of destination, and then assign the result to destination
CONCAT ENATE	Concatenate the content with the old value of destination, and then assign the result to destination

According to the above analysis, each data moving thus could be described by a vector, which contains four variables. For example, the following vectors describe the process of op-code fetching:

```
1. (PC, MAR, PC.value, PASS)
2. (MEM[MAR], MDR, MEM[MAR].value, PASS)
3. (MDR, IR, MDR.value, PASS)
4. (PC, PC, 1, PLUS)
```

A class called ***PathVector*** was created to store the information of each data moving. Objects of the ***PathVector*** was organized into a linked-list data structure, so that path information of one instruction could be stored first and then feed back to user interface to be interpreted.

Figure 32 shows the UML class diagram of ***Controller*** class. A list data structure in the type of ***PathVector*** was created to hold information of data moving for one instruction. The most important method of ***Controller*** is the public method *path()*. It first checks if the destination of the data moving is a ROM location. If not, operations on data specified by mode parameter are performed, and then generate a ***PathVector*** object to describe that data moving. The *showAnimation()* method passes recorded path vectors to user interface, so that user interface could interpret them to generate animations. This method usually is invoked after the execution of one instruction.

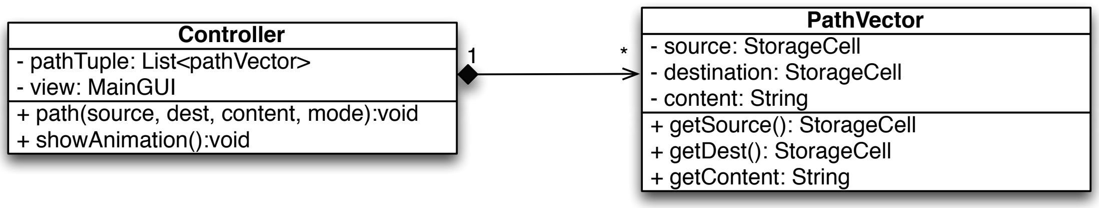


Figure 32: UML class diagram of controller class

OpFetcher class

The ***OpFetcher*** class is responsible for memory access during the instruction execution. A public method *fetchOpCode()* was implemented to fetch op-code from memory. Note that not like other existing emulators, the purpose of the AEM6809 is to present the execution cycles by the means of animation, so details of memory access procedure must also be simulated.

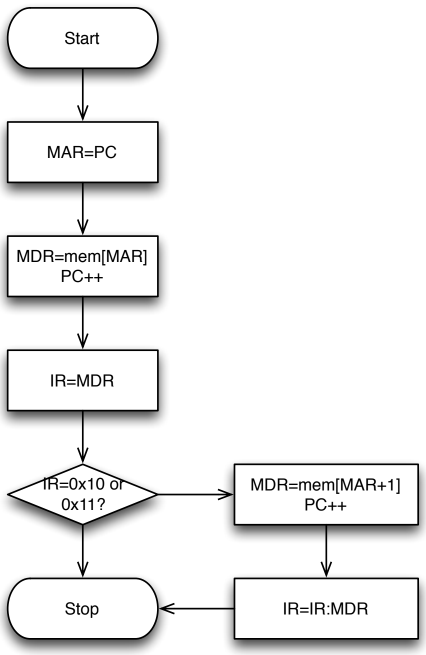


Figure 33: flow chart of fetchOpCode ()

Figure 33 shows the flow chart of method *fetchOpCode()*. Firstly, the content of PC is loaded into MAR register to indicate the address of this memory access. Then memory subsystem reads the content of the addressed memory location, put it into MDR and increment PC by one. The MDR now holds the op-code, which then is transferred to IR. Because for some instructions, such as long-branch, op-codes are in

16-bit length, so if the fetched byte is 0x10 or 0x11, another byte must be fetched again and concatenate it with the content in IR to form the op-code.

Before fetching operand from memory, the emulator needs to know the addressing mode specified by op-code. The private method *selectAddrMode()* was implemented to perform this task. The op-code in IR is tested by a switch compound structure to return its addressing mode. The algorithm of *selectAddrMode()* is based on table 9 in MC6809 manual [11].

As discussed in section 4.2.4, the ***OpFetcher*** should provide functions for fetching operand. However, one problem emerged during the implementation is that the addressing modes only specify how to get the effective address of operand, whether to fetch a byte, a word or just treat the effective address as the operand is determined by op-code. For example, addressing mode of *lda*, *ldd* and *clr* could be directed addressing, but *lda* requires fetching a byte, *ldd* requires fetching a word and *clr* only need to know the effective address. In addition, it is also difficult to find an effective and integrated algorithm to determine which kind of operand is needed for a given op-code.

Solution for this problem is the ***OpFetcher*** class provides three public methods, *fetchOperand()*, *fetchWordOperand()* and *fetchEA()*. The first method fetches a byte from memory location that is addressed by effective address. The second one fetches a word, while the last one simply returns the effective address. The responsibility of invoking the correct method to get the operand was then transferred to ***insSet*** class (see below).

InsSet Class

The ***InsSet*** is responsible for manipulating fetched operand as specified op-code, in another word, the class is a set of MC6809 instructions. Main method of the class is *doOp()*. It takes op-code as input, and invoke one of the three operand fetching methods in ***OpFetcher*** to get the correct operand, then do operations on that operand specified by corresponding instruction. Detailed implementation of the method was ignored at the first stage of development (See section 5.3.2 for details of the implementation of ***InsSet*** class).

MC6809 class

After the development of the ***Controller***, ***OpFetcher*** and ***InsSet***, the ***MC6809*** was created to integrate the their functionality together to build up a framework application. Responsibilities of this class are to manage resources that are needed by microprocessor, and invoke methods of other classes in a particular order, so as to simulate the execution of each instruction. In addition, the class is also responsible for handling different level of interrupts.

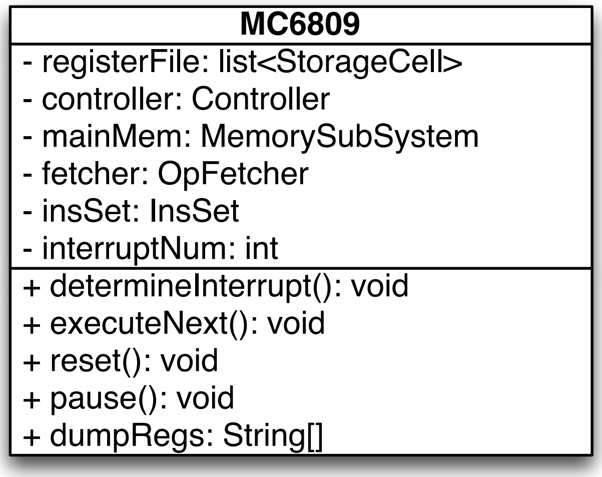


Figure 34: UML class diagram of MC6809

Figure 34 shows the refined UML class diagram of ***MC6809***. Resources that are required for the class are references of ***Memory***, ***OpFetcher***, ***Controller*** and ***InsSet***. During the construction stage, a set of internal registers were created and stored in the class instance *registerFile*.

The simulation of interrupt handling was implemented as the following fashion. Firstly, there is a public *int* type class instance, named *interruptNum*, to indicate the current interrupt status, where 0 means no interrupt, and 1 to 4 for four levels hardware interrupts. Users or I/O devices then could trigger hardware interrupts by set the flag when necessary. A private method *determineInterrupt()* was created to handling interrupts. It checks the *interruptNum* variable to see if the value is not zero. If so, do operations as specified by each type of interrupt and then clear the flag.

The most important method of the class is *executeNext()*. Different phases of instruction cycles are performed here to actually execute each instruction. The method first call *determineInterrupt()* to processing possible interrupts, and then execute next instruction by invoking different phases of execution cycles. Figure 35 shows the flow chart of method *executeNext()*.

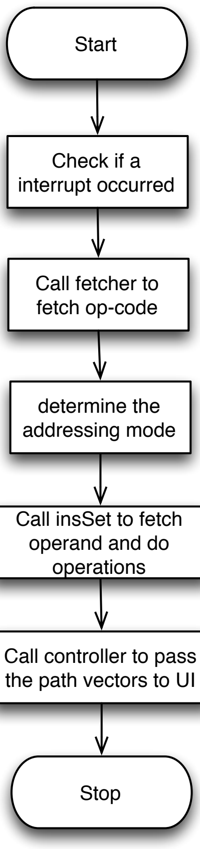

Start
Check if a interrupt occurred
Call fetcher to fetch op-code
determine the addressing mode
Call insSet to fetch operand and do operations
Call controller to pass the path vectors to UI
Stop

Figure 35: flow chart of executeNext()

Note that this method does not contain a while block to repeatedly execute the next instruction, in another word, the method only provide the step execution functionality. Because of the animation feature, execution of the next instruction would not start until animations for the previous instruction are finished. Hence the implementation of continuously execution was put into the user interface level by repeatedly calling executeNext() method. This arrangement could simplify the relationship between user interface thread and low-level threads.

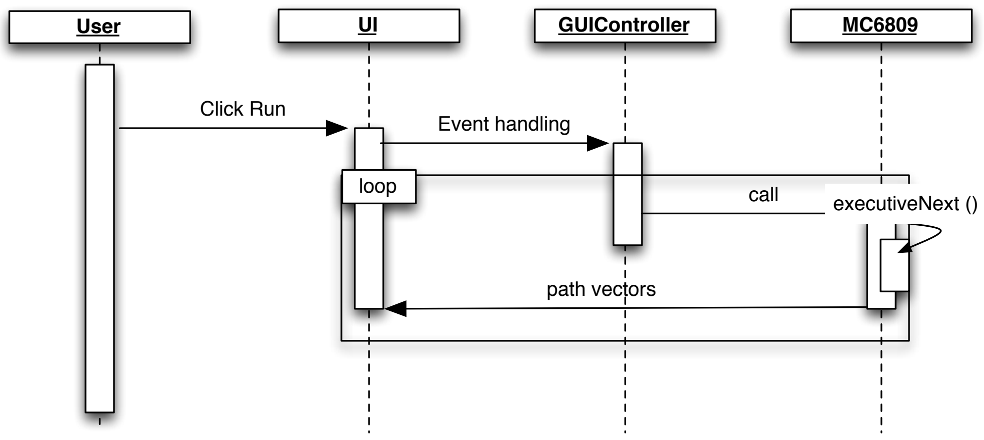


Figure 36: Sequence diagram of the interaction between UI and functionality layer

As show as in figure 36, when a user requests to execute continuously, the user interface generates an event and pass it to ***GUIController***. ***GUIController*** creates a new thread to run *executeNext()* method. Since the last step of *executeNext()* is to inform user interface to show animation and wait for return, the newly created thread would be naturally blocked until animations for this instruction finished.

5.3.2 Adding instructions into the framework

After the development of framework application, it is necessary to complete the implementation of instruction set. The process involves two stages. The first stage is to create an ***ALU*** class. It contains operations that would affect content in CC register, such as integer arithmetic and logical operations. The next stage is to complete the implementation of ***InsSet*** class, so that the emulator would be able to execute MC6809 instructions.

ALU class

As discussed in section 4.x, the ALU class contains highly frequently used operations for most of MC6809 instructions. The first problem need to be resolved is to extract these common operations on data. The clue is whether one operation would affect the content of CC register. If so, then the operation should be extracted and included in the ALU class. According to analysis, the ALU was programmed to support 22 operations (see source code for details).

These operations were classified into two categories: unary and binary operations. Two private methods *unaryOp()* and *binaryOp()* were implemented, where the former one takes only one parameter as input and the second on takes two input.

The second problem need to be resolved is caused by the animation feature of the proposed simulator. Recall that ***Controller*** class is responsible for data moving from one component to another one. However, in the case of moving data from register into ALU, it is not enough to just specify the destination as ALU.

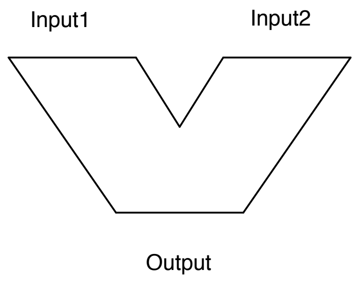


Figure 37: diagram of ALU

As shown in figure 37, a typical ALU has two entrances to receive inputs, and one exit to output calculation results. To describe a data moving that involves with ALU, information about the entrance must be specified. For example, in the case of *adda*, the animation flow should be shown as the following order:

```
1. ALU receives operand one from ACCA via input 1
2. ALU receives operand two from MDR via input 2
3. The result of adding operand one and operand two is stored in ACCA
      via output.
```

If translate the above order into path vectors:

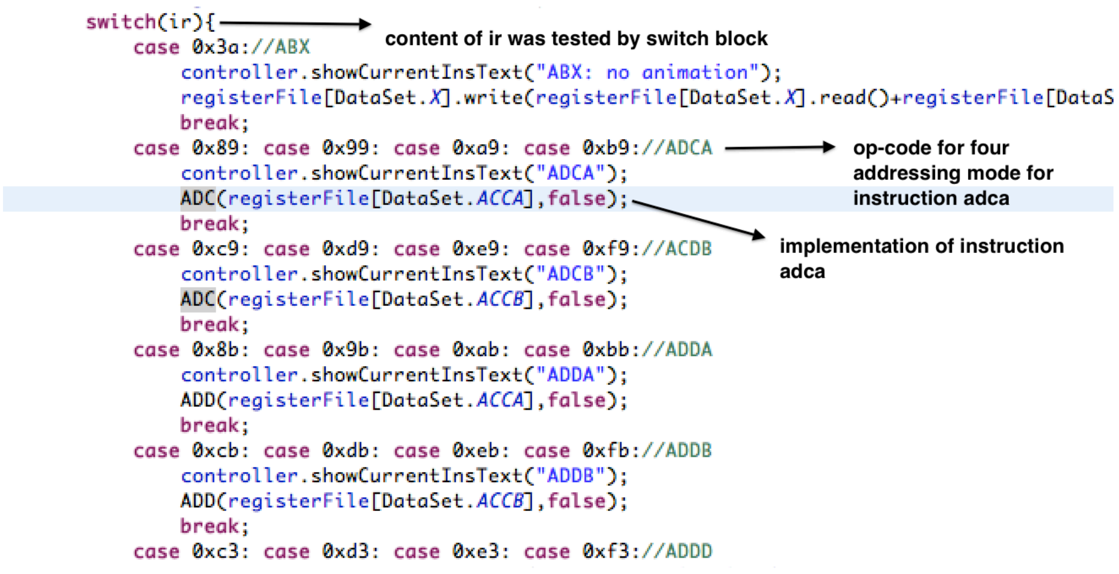
```
1. (ACCA, ALU_INPUT1, ACCA.value, PASS);
2. (MDR, ALU_INPUT2, MDR.value, PASS);
3. (ALU_OUTPUT, ACCA, result, PASS);
```

Hence, three class instances, *aluI1*, *aluI2*, *aluOutput*, with the type of ***StorageCell*** were created to represent two input entrances and one output. A public method *calculate()* was created, which takes four parameters. The first parameter is an *int* type parameter to specify the operation want to be invoked. The rest three parameters, with the type of ***StorageCell***, specify two input sources and the destination of calculation result. The method first calls *path()* method of ***Controller*** to move data from source to internal class instances *aluI1* and *aluI2*, thus to specify the entrance of input data.

InsSet class

It is necessary to complete the ***InsSet*** class to provide full functionality of MC6809 instruction set. One Implementation option of the class is to implement a super class, which contains the most common features, as an abstracted instruction. Then other concrete instructions extend it to specify their unique features. The ***InsSet*** class maintains a list of instruction objects to be invoked when necessary. This strategy provides a good structure of instructions, so that easier to be extended, but also causes implementation overhead due to the massive number of instructions.

The project adopted a more intuitive and simple approach, where each instruction was implemented as a method of ***InsSet***, rather than an object. A public method *doOp()* was implemented as the instruction selector. It takes an op-code as input, and then finds the matched method to be invoked for a given op-code. The following code fragment shows the structure of *doOp()*.



```
switch(ir){
    case 0x3a://ABX
        controller.showCurrentInsText("ABX: no animation");
        registerFile[DataSet.X].write(registerFile[DataSet.X].read()+registerFile[DataS
        break;
    case 0x89: case 0x99: case 0xa9: case 0xb9://ADCA
        controller.showCurrentInsText("ADCA");
        ADC(registerFile[DataSet.ACCA],false);
        break;
    case 0xc9: case 0xd9: case 0xe9: case 0xf9://ACDB
        controller.showCurrentInsText("ADCB");
        ADC(registerFile[DataSet.ACCB],false);
        break;
    case 0x8b: case 0x9b: case 0xab: case 0xbb://ADDA
        controller.showCurrentInsText("ADDA");
        ADD(registerFile[DataSet.ACCA],false);
        break;
    case 0xcb: case 0xdb: case 0xeb: case 0xfb://ADDB
        controller.showCurrentInsText("ADDB");
        ADD(registerFile[DataSet.ACCB],false);
        break;
    case 0xc3: case 0xd3: case 0xe3: case 0xf3://ADDD
```

content of ir was tested by switch block
op-code for four addressing mode for instruction adca
implementation of instruction adca

Figure 38: adding new instruction into InsSet

As shown in figure 38, the process of adding an instruction into ***InsSet*** class involves two stages. The first stage is to create a method contains the behavior specified by the instruction. Then add a case block into the switch compound structure in where the method of that instruction is invoked.

5.4 Implementation of Devices Package

The *devices* package is responsible for simulating behaviors of I/O devices as well as the communication process between peripherals and CPU. As discussed in section 4.x, the package contains an abstracted super class to describe the most common features among varieties of I/O devices in MC6809 system. Concrete devices that are necessary then could be added by extending this class.

5.4.1 Super class implementation

The most common features among I/O devices in MC6809 system are:

1. Contains a set of internal registers to store I/O data, or code of different operations and device status.
2. Each internal register is assigned with a unique address among the address space of MC6809
3. I/O devices are relative independent to the processor. They keeps monitoring the contents in registers and taking corresponding operations when it changes.

An abstracted super class, ***IODevice***, was created to describe the above features. The class maintains an array in ***StorageCell*** type to store its internal registers. Note that since different I/O devices might have different number of registers, sub-class who extends from this class should specify the length of the register array. To map each internal register into the same address space of memory subsystem, each slot of the array refers to an object of memory array (see section 5.2.1). This means that there is a memory location shares a same object with the register, or in another word, they are using the same address. When the content of internal registers changed, this change would be reflected to processor immediately, and vice versa.

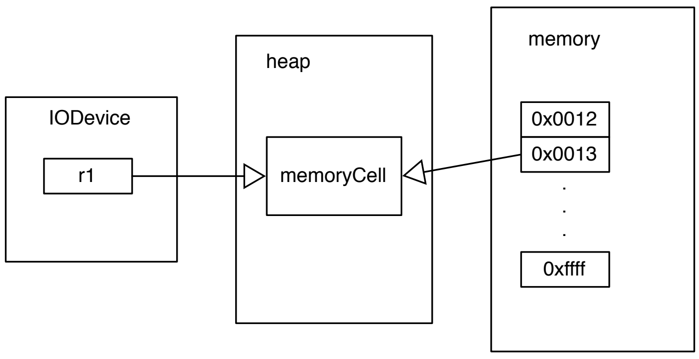


Figure 39: Map internal registers into the same address space of memory

Figure 39 shows an example of mapping an internal register of a hypothetical device into memory address space. The device has only one internal register, named r1, which should be assigned with the address 0x0013. For achieving this goal, both r1 and slot in memory array with index 0x0013 refers to a same ***MemoryCell*** object sitting in heap. The processor thus can access the internal register of the I/O device using the address 0x0013.

As discussed in section 4.2.5, the class implements ***Runnable*** interface so that it could be run as a thread. The *run()* method contains a endless while loop, where the method *doOperation()* is invoked repetitively. For each round of loop, there is an adjustable delay to simulate the different speed of I/O device. Implementation of *doOperation()* is empty, which should be specified by sub-classes.

5.4.2 Create concrete I/O devices

Due to the time constrains, the project only implemented a hypothetical LED panel. The device is assumed to have eight LED lights, which could be controlled by the processor.

The class ***LEDPanel*** is the implementation of the device. As specified in the above section, it extends from ***IODevice***, has two internal registers, one for control register, and another for data register. Addresses of the registers were assigned to 0x140 and 0x141. Method *doOperation()* is re-write to specify the expected behavior of the device.

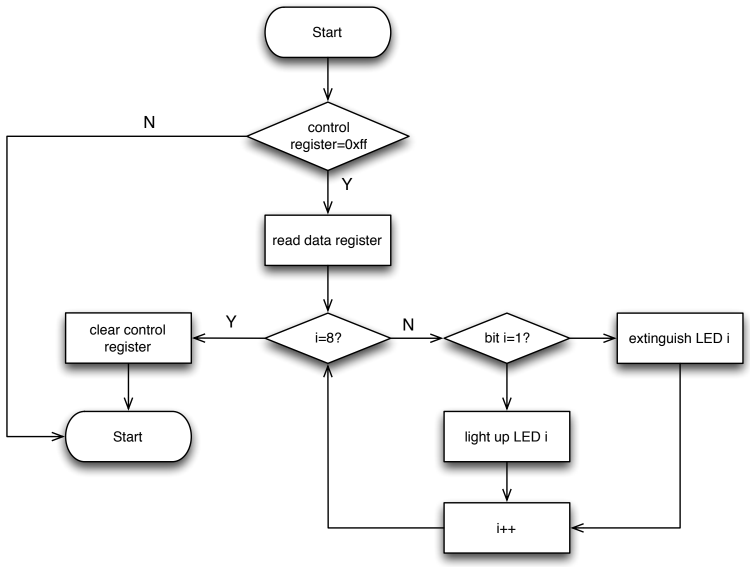


Figure 40: flow chart of *doOperation()* method

Figure 40 shows the flow chart of the override *doOperation()* method. It first checks the value of control register. If the value is 0xff, it then reads content in data register to determine the status of each LED lights. The control register is cleared to wait for next command from processor.

Implementations of other I/O devices, for example, A/D and switch banks, are same as the implementation of ***LEDPanel***. They need extends from ***IODevice***, specify the number of registers required, assign addresses for these registers and then re-write the *doOperation()* method to detail their expected behaviors.

5.5 Implementation of GUI layer

After the implementation of functionality layer, the next step was to implement the graphical user interface. As specified in section 4.3, this layer is responsible for interacting with users, and displaying information generated during execution. This section illustrates the process of GUI development; especially focus on the implementation of CPU panel, as it is the area that shows animation of each instruction.

5.5.1 Find paths between components

The most fundamental problem of the GUI implementation is how to implement the animation of data moving between different graphical components sitting in the CPU panel. As decided in section 4.3.3, animations of a data moving would show the value of the data being transferred actually "floating" along the path between the two diagrams. Thus, finding buses between source and destination is critical to the implementation of animation.

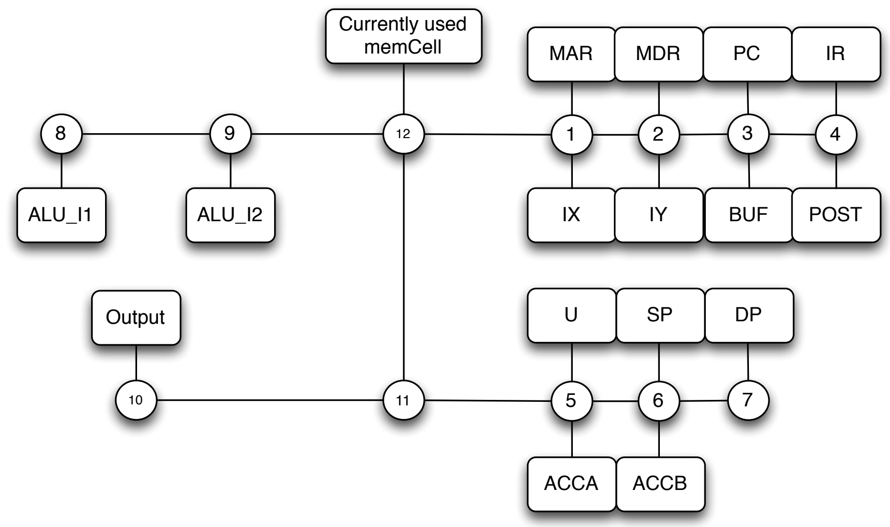


Figure 41: Topology of components in CPU panel

The problem could be abstracted into a more general problem: finding a path between two vertices within a graph. As shown in figure 41, components (denoted as squashed rectangles), and cross points (denoted as circles) could be treated as vertices. Buses connecting them are edges. Note that the ALU component were spited into three vertices because of the reason discussed in section 5.3.2.

It is clear that these vertices and edges could form an un-directed graph. Because the graph does not contains any cycles, so there is only one path between any two vertices. Finding a path between vertices thus could be resolved by simply using the breadth-first search algorithm.

Since Java does not have built in support for the graph data structure, a package *graph* was implemented to provide functionality of graph search. Within the package, a ***Vertex*** class is created to represent each component (including cross points) in CPU panel. Except ordinary attributes of other graph vertices, each vertex also stores a set of coordinator to represent its position in CPU panel. Edges of the graph are represented by ***BusCanvas*** class (see section x).

The graph was implemented by using edge-sets approach. A ***ComponentGraph*** class organizes these vertices and edges into a graph, and provides method to return a path between any two vertices within it. The data structure and algorithm was implemented using a modified and debugged version of the code presented by David Watt in his Algorithms and Data Structure course.

5.5.2 Implementation of diagram

Before the development of CPU panel, it is necessary to implement a number of diagrams to represent each register and bus that would be displayed in the user interface. According to analysis, these elements in CPU panel could be organized into a certain class hierarchy.

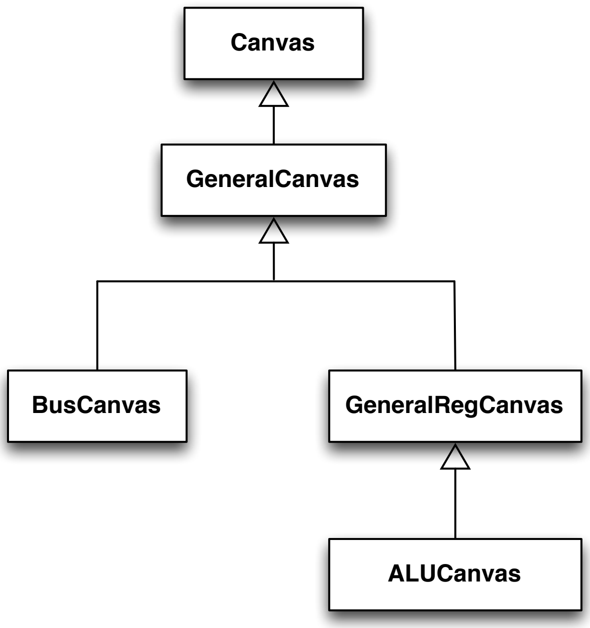


Figure 42: Class hierarchy of registers and buses

Figure 42 shows the hierarchy of registers and buses. The super-class is a Java build-in class, ***Canvas***, which was designed to represents a blank rectangular area of the screen. Further analysis suggested that diagrams of register and bus share certain common features:

1. Dimensions of each register diagram are same
2. Each register diagram should be able to be activated and inactivated
3. Each register diagram should be displayed as the same colors

A ***GeneralCanvas*** class extends from ***Canvas*** class, was created to describe the most common features of each register and bus. The overridden *paint()* method draws an edged rectangle on the location specified by two public class instance, *x* and *y*. Methods *setActive()* and *setInActive()* change the colors parameters of the object, and then call *repaint()* to reproduce a rectangle.

Two classes were implemented as extensions of ***GeneralCanvas***. The first one is ***GneralRegCanvas***, which is responsible for creating diagram of all 8-bit and 16-bit registers. Except features inherited from ***GeneralCanvas***, it also contains current value, previous value, and a label to specify the name of the register. Figure 43 shows an example diagram of PC register.

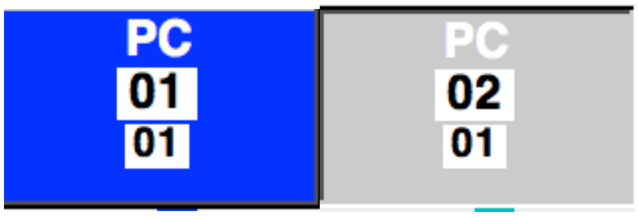


Figure 43: Active and inactive diagram of PC register

The second one is BusCanvas, which is responsible for creating buses between registers. Recall that The ***BusCanvas*** also represents edges in ***ComponentGraph***, thus it contains two class instances with the type of ***Vertex*** to store the connected nodes. In addition, the dimensions of buses are not fixed, but calculated by the start position and the end position. If the x coordinate of start position is equal to the x coordinate of end position, then the bus should be placed vertically. Otherwise, the bus should be placed horizontally. The following pseudo-code shows the algorithm of the calculation.

```
#Determine the dimensions of bus that starts
#at (xStart,  yStart), ends at (xEnd, yEnd)
if xStart = xEnd
  width = 10
```

```
        height = yEnd - yStart
    else
        width = xEnd-xStart
        height = 10
```

Figure 44: Algorithm to determine width and height of BusCanvas

The diagram of the ALU was chosen to be "Y" shaped, as this is the representation commonly used in textbooks. This diagram is difficult to draw on the screen, as it is not simply rectangular, like all the other diagrams, instead there are four sets of diagonal lines. The *paint()* method inside ***ALUCanvas*** was override again to draw the required shape of ALU. In addition, another label attribute was added to show current operations that being invoked. The following figure shows an example diagram of ALU.

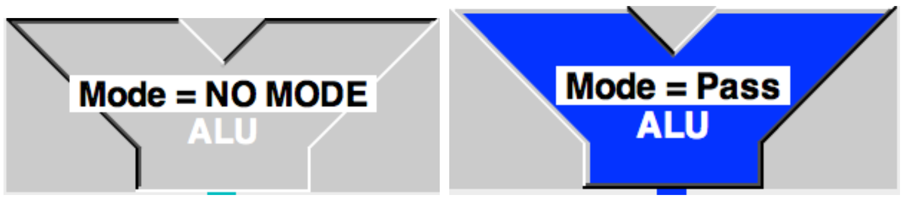


Figure 45: Diagram of ALU

5.5.3 CPUPanel implementation

The ***CPUPanel*** is responsible for displaying block diagrams of MC6809, as well as showing animations for each fetch and execute cycle. During the construction stage, the class creates a list of ***GeneralRegCanvas*** objects, put them into predefined positions, and then use a set of ***BusCanvas*** to connect them together as specified by the prototype in section 4.3.1. After that, a components graph is created to hold topology of these diagrams, and gives path from one node to another node when necessary.

To display animations of instruction that being executed, a public method *dataMove()* was created to interpret path information getting from ***Controller***, and then generate animation as desired. It first get the name of source and destination registers, highlight the corresponding diagrams by calling their *setActive()* method. Then a label contains the content of this data moving is generated. It floats along the path calculated by

ComponentGraph to simulate the process of transferring data from one register to another.

5.5.4 Implementation of MainGUI

The final stage of GUI layer implementation is to create a window that integrated all functionality together. The process of adding GUI components into the window is simple and intuitive as they were already implemented stable and firm. In the top left corner, a set of ***JTextFiled*** was created to display contents of register file. A ***JTable*** object to display contents in memory subsystem, as specified as section 4.3.4, occupied the top right corner. It allows users to manually modify content of each memory location for the purpose of debugging or testing. The bottom right corner is the CPU panel area; animations of each instruction are displayed here.

The bottom left corner is the control panel. Recall that the simulator was decided to adopt MVC design pattern, Mouse events are listened by ***GUIController***, which implements a ***ActionListener*** interface and has an override method *actionPerformed()*. When buttons were clicked, mouse event generated by that click would be passed to ***GUIController***. The ***GUIController*** then create a new thread to do jobs required by the event. Details of the functionality of buttons and other components in control panel could be found in user mannual.

Chapter 6 Testing and Evaluation

6. 1 Software Testing

Software verification and validation plays an important role in the software development process. The purpose of testing is to determine the correctness of a system against its pre-defined requirements. Note that software testing should not be considered as a separated phase of the development process, but occurs simultaneous with the implementation activities. Testing were conducted frequently and periodically to find defects of developed components

Section 6.1.1 explains the testing strategy that was adopted. Then section 6.1.2 and 6.1.3 illustrate detailed testing approaches with regard to unit testing and overall testing, respectively.

6.1.1 Testing strategy

In general, testing strategy could be categorized into two classes: black box testing and white box testing. The former one does not have to access the internal implementation details of test targets. The only information provided is the specification of interfaces, which describe the expected behavior of component or the entire system. The second one requires tester have access the internal implementation details to develop test cases that would cover as much the components or system's behavior as possible.

Even though a risk of black box testing is it tends to miss defects due to incomplete test cases, the project still adopted the black box testing as the basic testing strategy. The first reason is the time constrains of the project make the white box testing infeasible, as it requires spend amount of effort to analysis all possible paths of program flow to build up full coverage test cases. In addition, since the developer is also the tester, it is difficult for the tester to avoid the affection of same assumption of developer in the using of the system.

As mentioned above, the implementation of each package was carried out accompany with a set of unit testing, so that to build a concrete and stable base of the system. The tool that was used to conduct unit testing of packages in functionality layer and component layer is Junit, which is a simple toolkit to write repeatable tests for test target. It is easy to be accessed, as the Eclipse IDE has integrated this framework into its development environment.

6.1.2 Unit testing

It is desirable to test every possible input for each method of test targets. However, exhaustively testing is very inefficiency for most of cases. In general, unit tests were conducted by building equivalence classes for each method of test targets. Equivalence class refers to a set of inputs that would be treated as the same manner. Inputs at or near its boundary, and some random value within the class were tested. In addition, pre-condition defects were also considered into testing. If any the defect was detected, more specific test cases would be created to help locating debugs or mistakes that lead to the defect.

Because the project contains altogether 33 classes, the time constrains do not allow to develop full unit testing for each class. To provide as much coverage as possible, for each package, a set of critical classes were identified as the test targets. Details of the critical classes and their test cases could be found in source code.

6.1.3 Overall testing

After the system was built up, an overall testing was conducted to find out any other possible defects. As most of classes in functionality layer and components layer had been tested in unit testing stage. The overall testing focuses on determine the correctness of GUI layer, especially the animations.

The most straightforward testing approach is to generate a S-record forma file that contains every op-codes with different addressing mode. The file then should be loaded and executed to verifying the correctness of animations for each instruction. However, the verification of animation is difficult to be processed by automatically methods. The tester thus is required to observe each animation to find mistakes. Hence, this approach requires a significant amount of time, as there are altogether 59 instructions, and for each instruction there are a number of available op-codes with different addressing mode, altogether 1464 op-codes, which mean there are 1464 test cases need to be verified manually.

Since execution of instructions could be separated into two discrete phases: fetch and execute, where the first phase involves with fetching op-code and operand according to different addressing mode, and the second phase is to perform operations on data. The test cases could be reduced by testing animation of different addressing mode and testing behavior of each instruction separately.

Hence, a set of instructions was selected to test the animation of fetching data with different addressing mode. Then 59 instructions were tested with the simplest addressing mode to verify their operations on data. This strategy not only reduced the size of test cases, but also provides as much coverage as possible.

6.2 Evaluation of Project Outcome

The main purpose of testing is to verify the functionality of AEM6809, most related to functional requirements that were collected in requirement gathering stage. On the other hand, user evaluation aims at investigating the usability of the emulator, which related to non-functional requirements that cannot be measured directly.

According to ISO9241, usability of software is usually examined by the following factors:

1. Effectiveness: How accuracy and completeness of users' tasks when using the design
2. Efficiency: How quickly users can perform the tasks, once they have learnt the design
3. Satisfaction: How pleasant is it for users to use the design

Thus, a user evaluation was developed to assess the above three factors. Section 6.2.1 explains the evaluation approaches that were adopted during the evaluation process. The results then are summarized in Section 6.2.2

6.2.1 Evaluation process

In general, there are two approaches could be used to assess the usability of a software, criteria-based approach and tutorial-based approach. The project adopted the tutorial-based approach because it can provide the developer a practical insight into how the software is approached and any potential technical barriers that prevent adoption [12].

Since the nature of a tutorial-based evaluation is a set of reflections of participants' subjective experiences in learning, building, installing, configuring and using the client's software [12], it is essential to recruit a group of participants who have similar backgrounds and experiences with potential users of the project. According to the project brief and requirement specification, potential users are students who need to learn knowledge of computer organization and assembler language, or anyone who need a crash course in MC6809 architecture. Thus ideal participants should be students of School of Computing, who attended System and Network course last year.

Seven participants, include 4 master students of System and Network course and 3 undergraduate computing science students, were recruited to help conduct user evaluation of the project. Note that, the participant size is apparently relative smaller than normal software evaluation, due to the limited time available for the project. Thus the evaluation results are not as general as expected. This is a limitation of the evaluation.

The evaluation centered on carrying out typical tasks (see Appendix C) using the emulator. Because the emulator does not provide integrated assembler, participants were first given a tutorial of an external MC6809 assembler, named A-09. To examine how intuitive and efficiency the design is, the evaluator would not give any instructions or hints when participants are carrying out tasks, unless they cannot find solutions from the user manual. The evaluator would record the time consumed for performing each tasks, as well as any difficulties on operations during the evaluation. Then participants were required to answer a questionnaire (see Appendix C: Questionnaire) to provide their subjective opinions for the emulator. The collected data were analyzed, and the results are summarized at the next section.

6.2.2 Result summary

As described in Appendix C: Task list, there are three general tasks to examine how easy to use the emulator. In general, all participants were able to perform most of the tasks quickly and correctly. However, the general tasks exposed one shortcoming of the emulator that should be improved further. Participants felt that it is not easy and intuitive to enter machine code into memory manually by using the memory table on the top-right corner, especially when entering a long sequence of codes. It is expected to adopt more convenient methods rather than fill in the code into corresponding grids one by one.

After the general tasks, participants were required to perform three learning tasks to examine the usability in terms of learning and teaching purpose. All participants agree that the animation of execution cycles help them to understand the concept of addressing mode and interrupt handling of MC6809 (Appendix C: figure C-1). Most of them (6 over 7) were able to give correct descriptions on how does the processor fetch operand with different addressing modes, and how does it process hardware interrupts, after they performed these learning tasks.

However, one drawback was identified from the learning tasks. Five participants (Appendix C: table C-2) reported that the absence of integrated assembler degrades the teaching capability of the emulator. Because it is difficult to remember machine codes for every instruction, participants would like to enter programs in assembly language form into the emulator directly, and then observe the data interaction between registers and internal components. In addition, the use of external assembler also degrades the efficiency of the emulator, as users have to open at least three applications to edit, assemble and execute one piece of program.

Details of the evaluation results could be found at Appendix C. In general, the results suggested that the AEM6809 satisfies most of the non-functionality of the project, especially has a high usability for the teaching purpose.

Chapter 7 Conclusion and Further Work

The project aims at creating a MC6809 emulator for teaching purpose. The developed emulator, AEM6809, allows users to load and execute programs in the format of S-record, and then learning important concepts such as interrupts, memory mapped I/O and different addressing modes from observing the execution of programs.

Compared to existing 6800/6809 emulators that were assessed in section 2.3, it provides similar functionalities such as execution control and interrupt handling. However, the most different and unique features is that the emulator can display animations of data interaction between registers and other internal components, so that users or students could actually "see" what happened during each execution cycles. The whole procedure of how processors execute and how internal component interact with each other for different instructions thus are properly visualized and presented. Results from user evaluation agree that this design is effective and helpful for students in learning computer architecture and assembly language.

In addition, the emulator is capable to include additional integrated I/O devices to simulate I/O operations between processor and peripherals. It provides a framework of memory mapped I/O device from where new virtual devices that are necessary for teaching could be extended in future.

Generally speaking, the outcome emulator meets most of the original functional requirements (except provides a number of I/O devices), and particularly fulfills the requirement of teaching purpose. However, results from user evaluation also exposed several drawbacks existed in the emulator, which are worth to be improved in future.

7.1 Suggestion for Further Work

Integrated assembler

During the user evaluation, all participants reported that lack of an integrated assembler degrades the teaching ability of the AEM6809. It is expected that programs in assembly language format could be entered into the emulator directly, so that the links between animation of execution and instruction could be clearer and more intuitive, as it is not easy to remember all op-codes for different instructions.

In addition, from the technique perspective, using external assembler also affects the platform independent feature of the AEM6809 emulator. The emulator would be limited to be used in the same environment of the external assembler.

More I/O devices

Because of the limited time available for the project, the emulator only includes one I/O devices, a LED panel. It is expected that more I/O devices could be added into the system, especially some typical devices such as switch bank, DAC and ADC.

Support varieties of means for program debugging

Results from user evaluation suggested that the emulator is usable in teaching users the concepts and topics related to computer architecture and assembly language. However, most of users reported that they probably would not use it for the purpose of development. The reason is that the emulator only provides one method for program debugging: set breaking point on memory address. Typically speaking, users are preferred to debug a program by setting breaking point on program line, or a given status of internal registers. It is also expected that the emulator would allow user to set multiple breaking points at same time.

Shortcut keys

Only a few shortcut keys are provided for the system. Participants in evaluation suggested that more shortcut keys could improve the accessibility of the emulator.

References

[1] Ash, S. (2007) *MoSCoW Prioritisation Briefing Paper*, http://certifications.groupsite.com/beta/discussion/topics/310632/messages, Last Accessed: 20/08/13

[2] Mynatt, Barbee Teasley. *Software engineering with student project guidance*. Prentice-Hall, Inc., 1990.

[3] Dumas, Joseph D. *Computer architecture: fundamentals and principles of computer design*. CRC Press, 2006.

[4] Edler, Jan, and Mark D. Hill. *"Dinero IV trace-driven uniprocessor cache simulator."* available at http://www.cs.wisc.edu/~markhill/DineroIV 1998. [accessed: 03/24/2013].

[5] Fielding, Roy. "Representational state transfer." *Architectural Styles and the Design of Netowork-based Software Architecture* (2000): 76-85.

[6] Goodrich, Michael T., and Roberto Tamassia. *Data structures and algorithms in Java.* John Wiley & Sons, 2008.

[7] Greenfield, Joseph D., and William C. Wray. *Using microprocessors and microcomputers: the Motorola family*. John Wiley & Sons, Inc., 1988.

[8] Horstmann, Cay S. *Big Java: Compatible with Java 5, 6 and 7.* John Wiley & Sons, 2009.

[9] Jérôme Thoen, *Sim 6809, Motorola 6809 simulator.* Available at:http://membres.multimania.fr/jth/6809.html [accessed:03/24/2013].

[10] Motorola Inc., *M68000 8-/16-/32-bit microprocessors: programmer's reference manual.* Prentice Hall, 1986.

[11] Motorola Inc*., MC6809 Preliminary Programming Manual,* USA, 1979

[12] Mike Jackson, et al. “Software Evaluation: Tutorial-based Assessment”. Available at: http://software.ac.uk/sites/default/files/SSI-SoftwareEvaluationTutorial.pdf,[accessed: 20/08/2013].

[13] Newsome, Mark, Cherri M. Pancake, and Christopher Ward. "Visual execution of assembly language programs." *Proceedings of the 1993 ACM conference on Computer science.* ACM, 1993.

[14] Patterson, David A., and John L. Hennessy. *Computer organization and design: the hardware/software interface*. Morgan Kaufmann, 2009.

[15] Patti, Davide, et al. "Supporting Undergraduate Computer Architecture Students Using a Visual MIPS64 CPU Simulator." *Education, IEEE Transactions on* 55.3 (2012): 406-411.

[16] Pugh, Emerson W. Building IBM: *shaping an industry and its technology*. Mit Press, 1995.

[17] Skrien, Dale. "CPU Sim a computer simulator for use in an introductory computer organization-architecture class." *Journal of Computing in Higher Education* 6.1 (1994): 3-13.

[18] Summerville, Douglas H. "Embedded Systems Interfacing for Engineers using the Freescale HCS08 Microcontroller II: Digital and Analog Hardware Interfacing." *Synthesis Lectures on Digital Circuits and Systems* 4.1 (2009): 1-139.

[19] Sun-Cheung Wong. *J6809.* Available at:http://www.evenson-consulting.com/swtpc/MC6809Applet.htm [accessed:03/24/2013].

Appendix A Requirements Specification

INTRODUCTION

The requirements specification covers requirements that are needed by both potential users of the system and the system itself. Included in this document are several key components that make up the requirements specification. The key components are identified collectively in the next section, and are then expanded upon in later sections of the document.

This document includes information regarding both the system and user's requirements. Identified documentation:

- Use Case Diagrams
- Use Case Descriptions
- Non-Functional Requirements

Requirements Elicitation Procedure

The original source of the system requirement is the initial project brief. Most of main functional requirements of the system could be extracted from this material. More details were gathered from experienced users, which have been identified as previous S/N students of the University of Glasgow, by distributing questionnaires.

A set of use cases thus can be derived from the above information. The use cases were grouped and documented as use case description. Because the time constrains, the MoSCoW rule was adopted to prioritize them, which would reduce the risks of project failures.

System Scope

The focus of this project is to design and implement a graphical emulator of the MC6809 microprocessor for the teaching purpose. The aim is to create an application that could accept machine code in the form of S-record, and show the simulated execution of that program, as it would actually happen in MC6809. For achieving the above purpose, the fetch/execute cycle and stack operations should be animated to show the interactions between. Figure A-1 gives an overview of the system scope that will be further detailed in the use case descriptions.

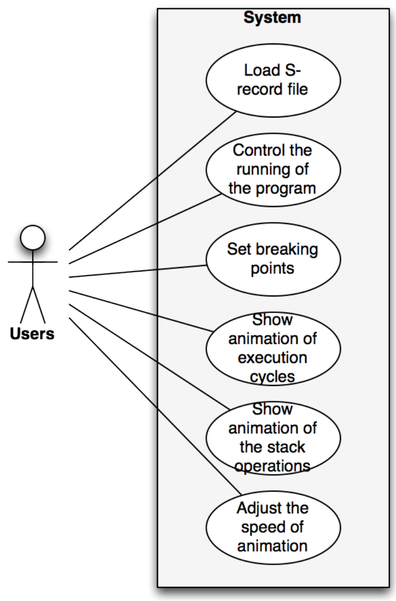

System
Load S-record file
Control the running of the program
Set breaking points
Show animation of execution cycles
Show animation of the stack operations
Adjust the speed of animation
Users

Figure A-1: System scope

Functional Requirements

This section describes the required functionality for the MC6809 emulator. The core use cases could be categorized into two classes:

Control of execution

- Load S-record file
- Run program continuously/step
- Pause program
- Stop program
- Set breaking point

Animation feature

- Show animation of the fetch/execution cycles
- Adjust the speed of animation
- Disable animation

Memory related operations

- Display memory layout
- Modify content of a given memory location manually

I/O and interrupts

- Connect I/O devices
- Trigger hardware interrupts manually

Control of code execution

Use Case Descriptions

Use Case 1: Load S-record file

Name	Load S-record file
Description	Users must load a program formatted in S-record before any operations
Rationale	The program won't provide additional assembler, program written in assembly language must be translated in S-record and then loaded in the emulator
Preconditions	
Scenario	The user selects the load button The system will pop-up a dialog The user choose a s-record file and then click ok The system will check the program and then load the program into memory subsystem

Alternative Scenario	The file content is illegal An error dialog pop-up
Post-conditions	Run and Step buttons are enabled
Actors	Emulator Users
Priority	Must have

Use case 2: Run program

Name	Run program
Description	User clicks run button to execute programs
Rationale	After the program is loaded, user can click Run button to run the program
Preconditions	A program has been loaded
Scenario	The user clicks the Run button, Program is executed from current PC
Alternative Scenario	
Post-conditions	Stop and Pause button enabled
Actors	Emulator Users
Priority	Must have

Use case 3: Run program in step mode

Name	Run program in step mode
Description	User click step button to execute the program in step mode
Rationale	It is necessary to run the program step by step
Preconditions	A program has been loaded
Scenario	The user click step button The system will run only one instruction pointed by current position and then paused at the beginning of next instruction

Alternative Scenario	
Post-conditions	
Actors	Emulator Users
Priority	Must have

Use case 4: Pause program

Name	Pause program
Description	User can pause program when it is running
Rationale	User can pause program when it is running
Preconditions	A program is running
Scenario	A program is running User click pause button The system will pause the program after the current instruction is completed
Alternative Scenario	
Post-conditions	Run and Step buttons are enabled
Actors	Emulator Users
Priority	Must have

Use case 5: Stop program

Name	Stop program
Description	User can stop program when it is running
Rationale	User can stop program when it is running
Preconditions	A program is running
Scenario	A program is running User click stop button The program stops, and the emulator is initiated

Alternative Scenario	
Post-conditions	Run and Step buttons are enabled
Actors	Emulator Users
Priority	Must have

Use case 6: Set breaking point

Name	Set breaking point
Description	The system will automatically pause the program when it meets the breaking point set by users
Rationale	The breaking point is an address of memory location
Preconditions	
Scenario	User enter an address, which is the beginning of an instruction User enable the address as a breaking point The program pauses at the breaking point
Alternative Scenario	User enter an address, which is not the beginning of an instruction User enable the address as a breaking point The system alert user that it is not the beginning of an instruction and then continue executing
Post-conditions	
Actors	Emulator Users
Priority	Should have

Animation of code execution

Use Case Description

Use case 7: Show animation of the fetch/execution cycles

Name	Show animation of the fetch/execution cycles
Description	The animation of fetch/execution cycles will be shown when the program is running

Rationale	It is necessary to show more details when program is running because of the teaching purpose
Preconditions	
Scenario	A program is running The animation of fetch/execution cycle for each instruction is shown on the main panel
Alternative Scenario	
Post-conditions	
Actors	Emulator Users
Priority	Should have

Use case 8: Adjust the speed of animation

Name	Adjust the speed of animation
Description	The speed of animation could be adjusted
Rationale	User can look at the unfamiliar concept with slower speed
Preconditions	
Scenario	
Alternative Scenario	
Post-conditions	
Actors	Emulator Users
Priority	Should have

Use case 9: Disable animation

Name	Disable animation
Description	Animation of execution cycles could be disable
Rationale	It is necessary when user want to focus on execution results

Preconditions	
Scenario	User clicks the enable animation select box. Animation is disabled when execute program
Alternative Scenario	
Post-conditions	
Actors	Emulator Users
Priority	Should have

Memory related operations

Use case 10: Display memory layout

Name	Display memory layout
Description	Content of memory locations are presented in main GUI
Rationale	This information is important for debugging
Preconditions	
Scenario	
Alternative Scenario	
Post-conditions	
Actors	Emulator Users
Priority	Should have

Use case 11: Modify content in a given memory location

Name	Modify content in a given memory location
Description	The memory could be modified manually
Rationale	This is important when user needs to modify a small part of program.
Preconditions	

Scenario	User double clicks on a grid in memory table A valid new value is entered The content of that memory location is modified
Alternative Scenario	User double clicks on a grid in memory table A invalid new value entered The system alarms the user and stops this modification.
Post-conditions	
Actors	Emulator Users
Priority	Should have

I/O devices and interrupts

Use case 12: Connect I/O device

Name	Connect I/O device with processor
Description	A virtual I/O device is selected and plugged into the system
Rationale	To demonstrate the memory mapped I/O technique
Preconditions	
Scenario	User chooses a I/O device from device lib The device is activated, internal registers are mapped into address space
Alternative Scenario	
Post-conditions	
Actors	Emulator Users
Priority	Could have

Use case 13: Trigger hardware interrupts manually

Name	Trigger hardware interrupts manually
Description	A hardware interrupt is triggered when click corresponding button
Rationale	This function allows user to focus on the interrupt handling
Preconditions	

Scenario	User clicks on corresponding button to trigger a hardware interrupt during the simulation of execution. The system start processing the interrupt after the current instruction is completed
Alternative Scenario	
Post-conditions	
Actors	Emulator Users
Priority	Could have

Non-Functional Requirements

This section describes the non-functional requirements that have been recognized. These requirements are:

1. Platform independent, compatible with mainstream operating system
2. Easy to be extended in future
3. Load codes in S-record format
4. User interface should be intuitive and easy to use

Appendix B Design Documents

Packages design

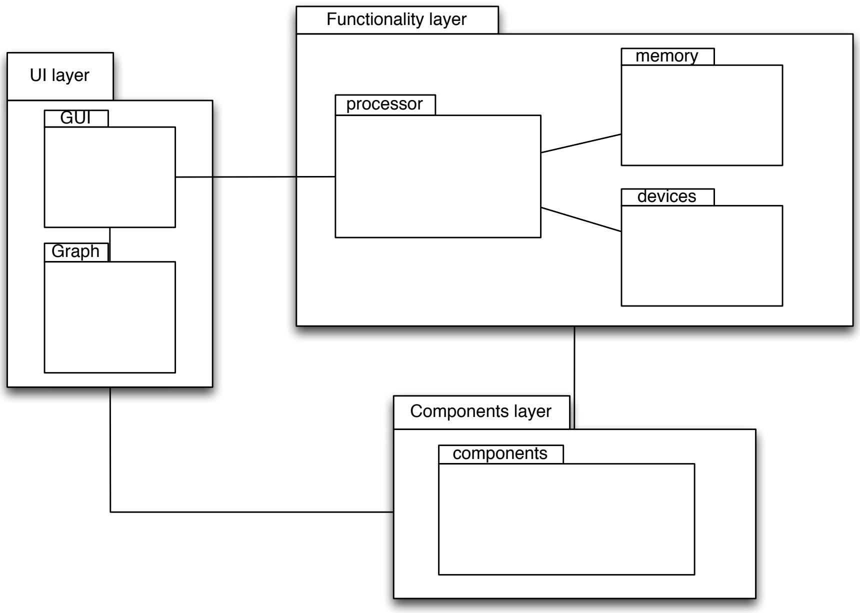


Figure B-1: Packages design

Functionality and Components layer

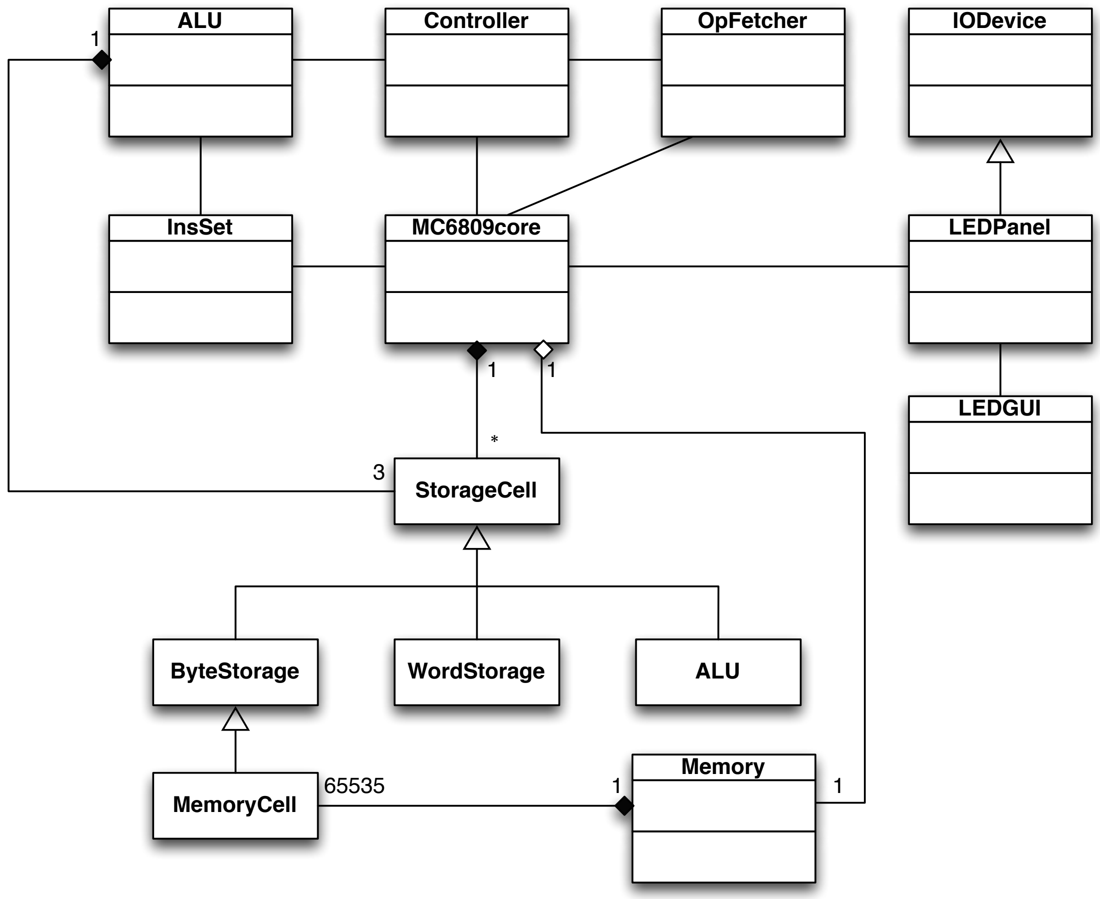


Figure B-2: UML diagram of functionality layer

GUI layer

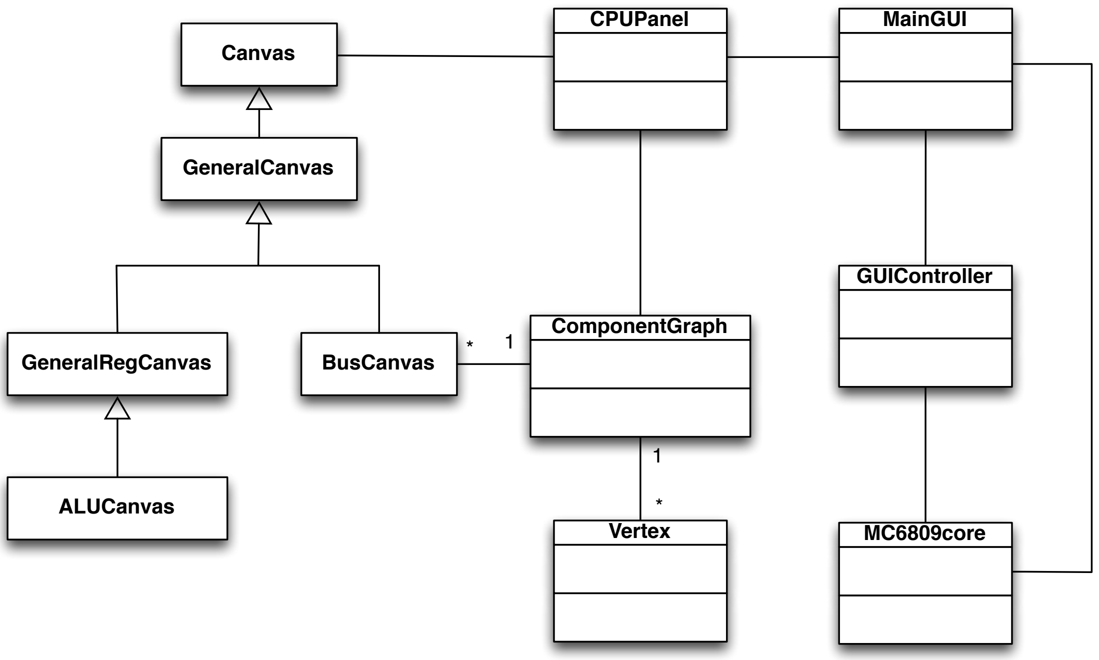


Figure B-3: Class diagram of GUI layer

Appendix C User Evaluation Documents

Task sheet

Part 1: General usage tasks

1. Enter the machine code of ***Program 1*** into memory, and then execute it. During the execution, you are free to click buttons on control panel.
2. ***Program 2*** is written by assembler language, please assemble it using the A-09 MC6809 assembler, choose S-record format as the output. And then load the S-record file into emulator to execute.
3. The emulator provides a number of methods to control the simulation of code execution. Please re-load the program in task 2, and then perform the following sub-tasks.
    a. Execute the program by step mode
    b. Set breaking point at random address
    c. Adjust the speed of animation
    d. Disable animation and then execute the program by step mode

Part 2: Learning tasks

1. One of the objectives of the emulator is to help student understanding different addressing modes in MC6809 by showing animation of execution cycles. The ***program 3*** contains machine codes of instruction LDA with different addressing mode. Please execute them one by one, and observe the behavior of them. After that, you are required to give descriptions for each addressing mode as your learning outcomes.
    a. Immediately addressing mode
    b. Directed addressing mode
    c. Extended addressing mode
    d. Indexed addressing mode (Zero-offset)
    e. Indexed addressing mode (Constant-offset)
    f. Indexed addressing mode (Auto-increment/decrement)
    g. Indexed addressing mode (Accumulator-offset)

2. To demonstrate the interrupt concept, the emulator allows user to trigger interrupts manually, by clicking four buttons on control panel. Please load ***program 4*** into the emulator and then trigger interrupts when you want. After that, you are required to give descriptions for each kind of interrupts as your learning outcomes.
3. The emulator implemented an integrated LED panel. Please active the LED panel, write a program to light up LED lights one by one. You can find the specification of the LED panel at the user manual.

Questionnaire (includes average marks)

1. Mark your experience from 5 to 1 in using the MC6809 emulator for learning purpose.
   a. Effective for learning (4.71)
   b. Easy to use (4.28)
   c. Information presentation (4.85)

2. Would you prefer using the emulator for the following purpose? Please give your justification:
   a. Learning computer organization and assembler language
      Yes (7)   No   Unsure

   b. Developing program for MC6809
      Yes (2)   No (3)   Unsure (2)

   c. Debugging
      Yes (1)   No (4)   Unsure (2)

3. How useful did you found the animation feature for learning architecture of MC6809? Mark your experience from 5 to 1. (5)

4. How intuitive is the graphical user interface? Mark your experience from 5 to 1. (4.28)

5. How easy is it to load and execute program? Mark your experience from 5 to 1 (3.28)

6. What are the new features you would like to have, but not provided by the emulator for teaching purpose?
   5 of 7 participants reported that it would be better to have an integrated assembler. 3 of 7 suggested that the emulator should support more approaches for debugging.

Questionnaire results

Overall results obtained from participants questionnaire. For all questions, 5 means excellent experience, 0 means bad experience.

Table C-1: over results of questionnaire

Participant	Q1.a	Q1.b	Q1.C	Q2.a	Q2.b	Q2.c	Q3	Q4	Q5
P1	5	4	5	Y	Y	N	5	4	3
P2	4	4	5	Y	N	N	5	4	3
P3	5	5	5	Y	N	N	5	5	4
P4	5	4	4	Y	Y	Y	5	4	3
P5	4	4	5	Y	U	U	5	4	3
P6	5	5	5	Y	U	N	5	5	4
P7	5	4	5	Y	N	U	5	4	3
Average	4.71	4.28	4.85	N/A	N/A	N/A	5	4.28	3.28

1. Mark your experience from 5 to 1 in using the MC6809 emulator for learning purpose.

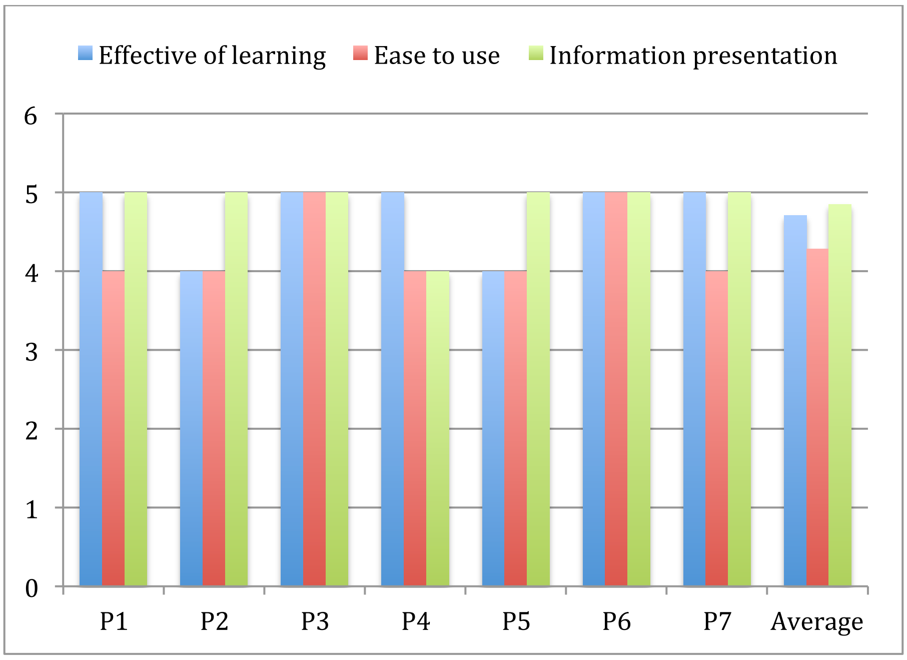


Figure C-1: Results for Q1

2. Would you prefer using the emulator for the following purpose? Please give your justification:

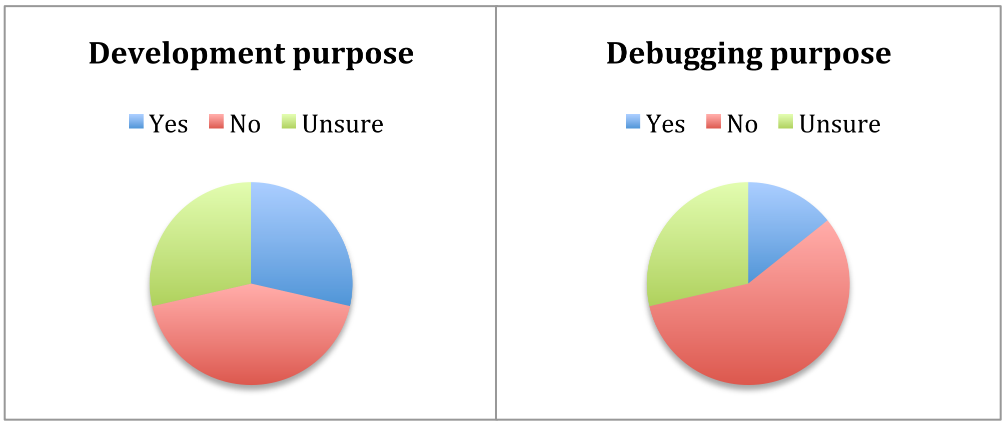


Figure C-2: Results for Q2

3. How useful did you found the animation feature for learning architecture of MC6809? Mark your experience from 5 to 1.
4. How intuitive is the graphical user interface? Mark your experience from 5 to 1.

5. How easy is it to load and execute program? Mark your experience from 5 to 1

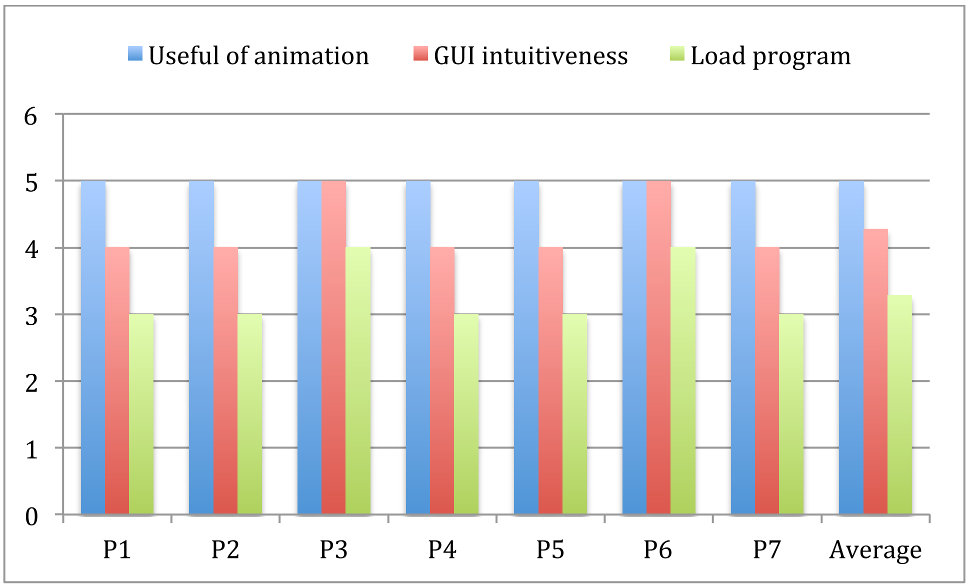


Figure C-3: Results for Q3, Q4 and Q5

6. What are the new features you would like to have, but not provided by the emulator for teaching purpose?

Table C-2: Results for Q6

Participant	Assembler	Debugging tools	Others
P1	Y		Shortcut keys
P2	Y		N/A
P3		Y	RAM Size should be adjustable after initialization
P4		Y	Mask the CPU panel when animation feature was disabled
P5	Y		N/A
P6	Y	Y	N/A
P7	Y		N/A

Appendix D User Manual

Overall View

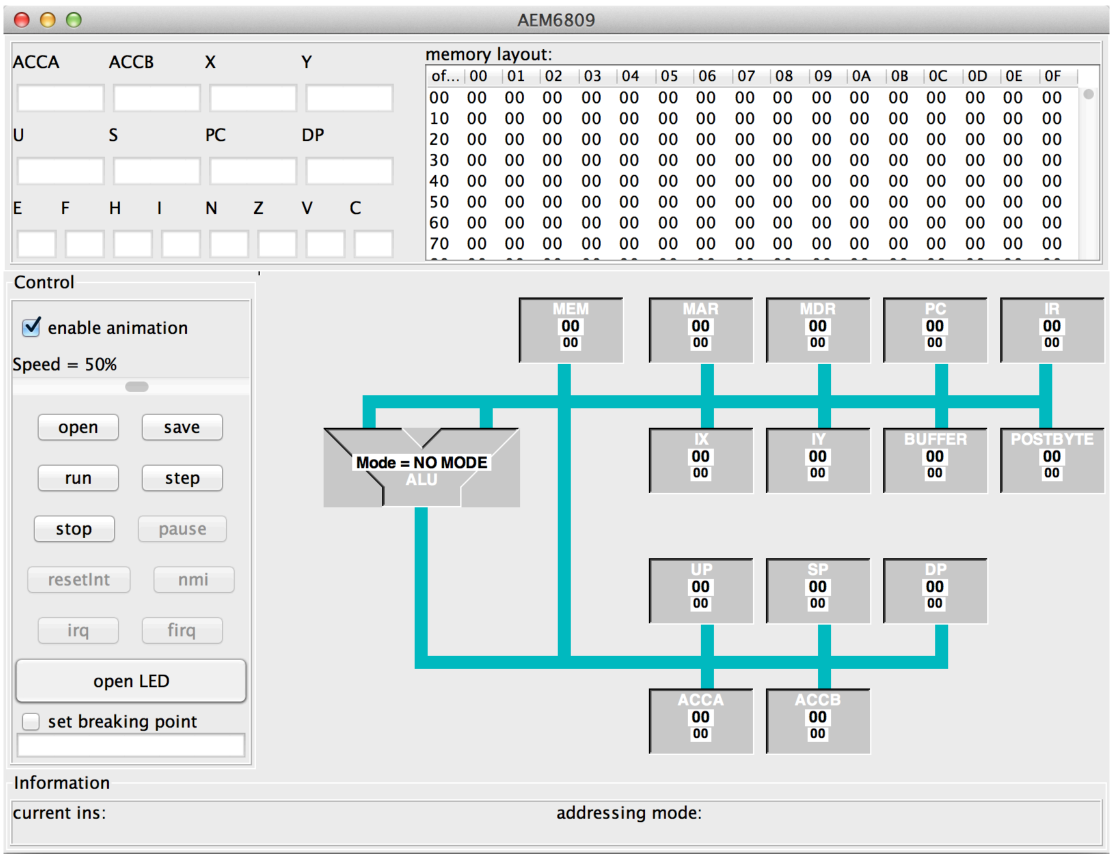
The AEM6809 is a MC6809 emulator for teaching purpose. It allows users to load S-record files that contain machine codes of MC6809, and then execute programs just as expected. Compared to existing 6809 emulators, the emulator provides similar functionalities such as execution control and interrupts handling. However, the most different and unique features is that the emulator can display animations of data interaction between registers and other internal components, so that users or students could actually "see" what happened during each execution cycles. The whole procedure of how processors execute and how internal components interact with each other for different instructions are properly visualized and presented.

Features of AEM6809 are:

1. Visualizing the execution cycles of each MC6809 instructions, data interaction between registers and other components are animated.
2. The content of memory subsystem could be modified manually to provide a straightforward mean for debugging and testing.
3. The emulator could load S-record format file containing MC6809 machine code.
4. Users are able to trigger four kinds of hardware interrupts manually during the simulation of instruction execution.
5. A built-in LED panel were implemented, additional I/O device could be added in by simple extends from an abstracted I/O device class.

User Interface Layout

As shown in figure D-1, the graphical user interface of AEM6809 consists of five parts.



of...	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
10	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
20	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
30	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
40	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
50	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
60	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
70	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00

Figure D-1: Graphical user interface of AEM6809

The first part is the Register Panel, which is sitting on the top-left corner. It displays execution result of all user programmable internal registers and Code Condition register (CC) for each instruction.

The Table in top-right corner is a Memory Layout Table that shows content of memory subsystem. Each memory location is editable when the emulator is not executing programs.

The Control Panel allows users to control the execution of programs, set breaking point and trigger interrupts.

The CPU Panel displays the internal architecture of MC6809. Animation of date interactions of instructions will be displayed here when executing program. The animation could be disabled.

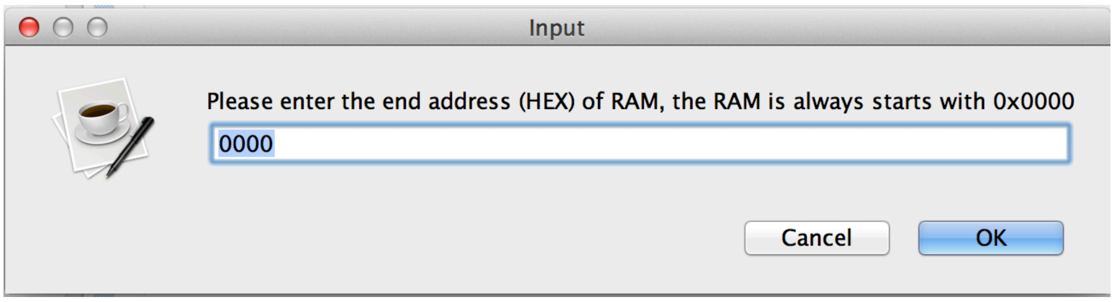
The Information Panel in bottom shows information of execution such as name of current instruction, current addressing mode and errors.

Details on functionality of each part are illustrated in the following section

Open the AEM6809 emulator

Since the AEM6809 is developed by Java language, it could be executed under any computer as long as it has installed Java Runtime Environment (JRE). Suggested version of JRE would be not lower than 1.7.0, as the project was tested under this version.

The AEM6809 emulator could be opened by simple execute the AEM6809.jar file under the directory "/AEM6809/". When the jar file is executed, an input window will be pop-up to ask user to specify the end address of RAM (the start address of RAM is assumed always at 0x0000), as shown as in figure D-2.


Input

Please enter the end address (HEX) of RAM, the RAM is always starts with 0x0000

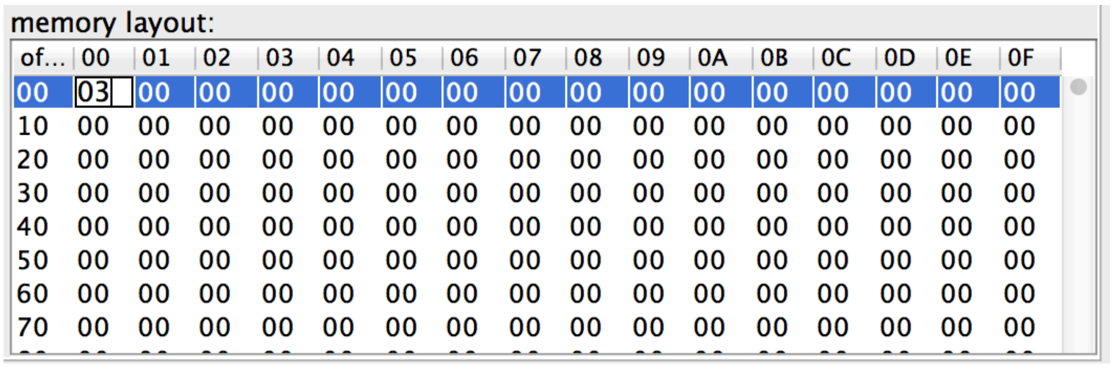
0000

Figure D-2: A pop-up window ask for the end address of RAM

Illegal inputs are from 0x0000 to 0xffff, in the format of hexadecimal. After the user entered an end address, the main window of the emulator will be shown.

Load a program

There are two ways to load a program into AEM6809. The first one is to enter executable code and data into memory by modifying Memory Layout table. This method is suitable for shorter program that consisted by only a few instructions. As shown as in Figure D-3, grids in memory layout table are editable when the processor is not executing codes. User can click on grid of a given address and then enter the new value (in hexadecimal). Note that, the biggest value that is allowed is 0xff.

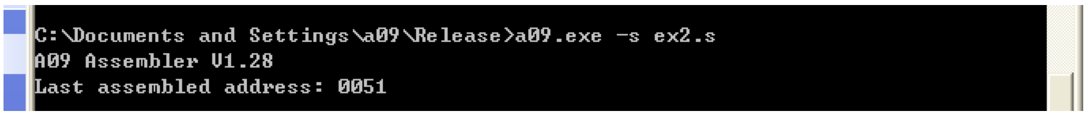


```
memory layout:
of... 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
00    03 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
10    00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
20    00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
30    00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
40    00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
50    00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
60    00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
70    00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

Figure D-3: Enter 0x03 into address 0x0000

The emulator also allows user to load S-record file that contains executable code and data. Since it is beyond the scope of the emulator to provide integrated assembler, users need to assemble programs written in assembly language must by external mc6809 assembler. The software package has included an assembler named A-09, which could be used under Windows O/S directly.

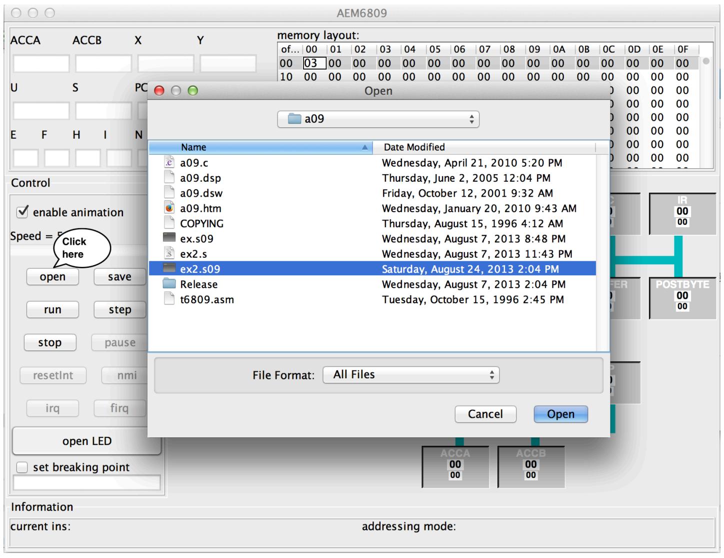
For example, to load a program named "ex2.s" into the emulator, first execute the command as shown as in figure D-4.



```
C:\Documents and Settings\a09\Release>a09.exe -s ex2.s
A09 Assembler V1.28
Last assembled address: 0051
```

Figure D-4: Assemble a program named "ex2.s" into a S-record file by using A-09

Then the executable code and data is output into a S-record file named "ex2.s09". Next, click on "Open" button of the emulator; choose the file "ex2.s09", as shown as in Figure D-5

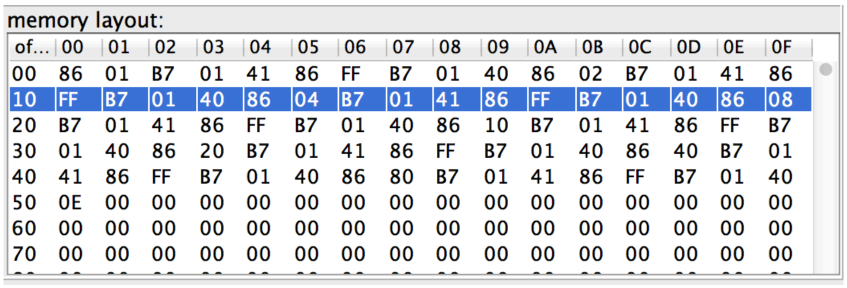


Name	Date Modified
a09.c	Wednesday, April 21, 2010 5:20 PM
a09.dsp	Thursday, June 2, 2005 12:04 PM
a09.dsw	Friday, October 12, 2001 9:32 AM
a09.htm	Wednesday, January 20, 2010 9:43 AM
COPYING	Thursday, August 15, 1996 4:12 AM
ex.s09	Wednesday, August 7, 2013 8:48 PM
ex2.s	Wednesday, August 7, 2013 11:43 PM
ex2.s09	Saturday, August 24, 2013 2:04 PM
Release	Wednesday, August 7, 2013 2:04 PM
t6809.asm	Tuesday, October 15, 1996 2:45 PM

File Format: All Files

Figure D-5: open the S-record file

The program then is loaded into the memory, which could then be executed by the emulator, as shown as in Figure D-6.



memory layout:

of...	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
00	86	01	B7	01	41	86	FF	B7	01	40	86	02	B7	01	41	86
10	FF	B7	01	40	86	04	B7	01	41	86	FF	B7	01	40	86	08
20	B7	01	41	86	FF	B7	01	40	86	10	B7	01	41	86	FF	B7
30	01	40	86	20	B7	01	41	86	FF	B7	01	40	86	40	B7	01
40	41	86	FF	B7	01	40	86	80	B7	01	41	86	FF	B7	01	40
50	0E	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
60	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
70	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00

Figure D-6: Memory layout after loaded program ex2.s

Execute a program

After a program is loaded into the memory, the emulator then could execute the program. Click the "Run" button on control panel to execute the program continuously, or click "Step" button to execute the program step by step, as shown as in figure D-7.

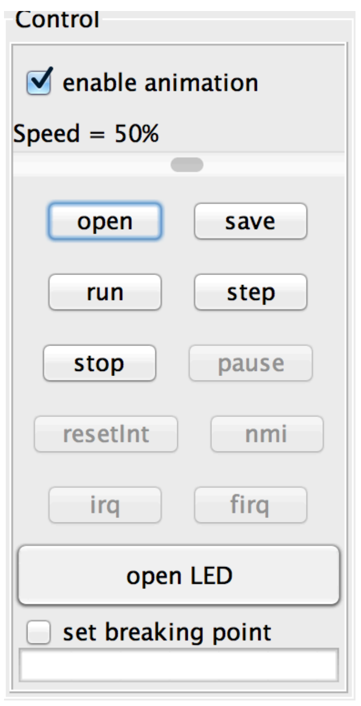


Figure D-7: Execute a program

When start executing, if the "enable animation" check box is selected, the CPU Panel will show animation of instruction that being currently executed. Otherwise, the animation feature will be disabled, and then program is executed in normal speed.

The speed of animation is also adjustable; user can slow down or accelerate the animation by using scroll bar under "enable animation" check box.

Other Features

Except simply load and execute programs, the AEM6809 also provides other features.

Trigger hardware interrupts manually

The emulator allows user to trigger hardware interrupts by clicking buttons on the Control Panel, as shown as in Figure D-8.

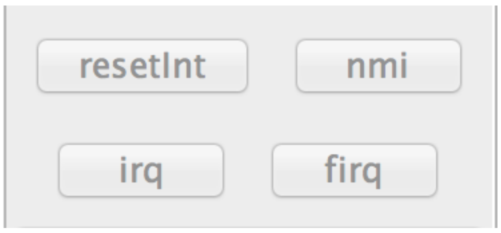


Figure D-8: Buttons for triggering hardware interrupts

These buttons will be activated when the emulator is executing programs. Once the user click on one of them, the emulator will start processing interrupt and displaying animation of interrupt handling process after the current instructions has been completed, if the animation check box is selected.

Set breaking point

As existing emulator, the AEM6809 is also capable to set breaking point. However, this emulator only allows setting breaking point by address. If the breaking point is not the first byte of an instruction, the emulator will pop-up an alarm window and then continues execution.

Activate LED panel

The AEM6809 provides an integrated I/O device, a LED panel. User can activate it by clicking on the "Open LED panel" button on The Control panel. It contains eight virtual LED lights as shown as figure D-9.

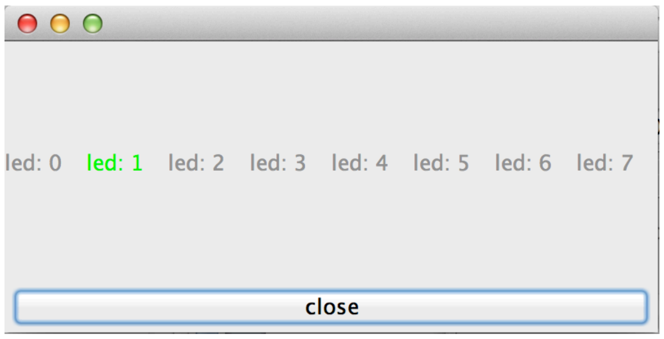


Figure D-9: LED panel

The LED panel has two internal registers: control register and data register. The control register is mapped into 0x0140. The only available command code is 0xff, which means the content of data register has been changed. The data register is mapped into 0x0141. Bits in data register indicate the expected status of corresponding LED light, 1 for light-up, 0 for extinguished.