A Microprocessor Sumulator: Design and Implementation

by Ken Clowes

This document describes the basic architecture (design) and implementation of a microprocessor simulation engine. The primary features of the architecture are not tied to any specific processor. However, the Motorola MC68HC11 is used for a reference implementation.

Table of contents

1. History

In the Fall of 2001 I wrote a simple (very partial) simulator for the Motorola 6811 microprocessor. My only goal was to give students a feel for object-oriented techniques using a non-trivial example that connected with their previous knowledge. (The students knew what a 6811 microprocessor was and how it behaved.)

After presenting the basic design of the simulation engine, I pointed out that some details had been glossed over. The most obvious limitations were the complete absence of a user interface and the very limited machine language instructions that could be simulated.

Nonetheless, I felt that the basic design worked well and that it would be interesting to pursue it. The result is this project.

It would be disengenuous to claim that "filling in the glossed-over details" was simple. It was not! Nonetheless, the original design has proved to be basically sound. Alas, nurturing the idea from conception through childhood and eventual usability has taken more time than I had anticipated.

This paper describes the basic simulation engine for a processor.

Note: Two important aspects of a useable simulator are not described here:

1. The user interfaces to the underlying simulation engine.
2. How devices are simulated.

These are important topics that deserve their own description. My TODO list includes writing these descriptions.

2. Goals

The goals set for the project are:

1. Document the complete design and implementation of the basic simulation engine. This remains one of the main goals and is the reason I am writing this document.
2. Make the implementation portable so that it can run in most computers (e.g. PCs running Windows, Apples, Unix and Linux boxes) and make the product freely available.
3. Allow for simulation of a pure 6811 CPU with any kind of configuration and with no required access to source code.
4. Allow any kind of device that can be mapped to the memory space of the 6811 system to be simulated.
5. Provide basic mechanisms so that user interfaces (graphical, text based, batch, etc.) can be built.
6. Provide a warning mechanism whereby the simulation engine can detect probable

program bugs such as writes to ROM, stack overflow, etc.

7. Provide at least one user interface so that the simulator can be usefully deployed. (Note that user interface design is not discussed here; the simulation engine simply provides mechanisms for implementing user interfaces.)

3. Analysis of simulation strategies

There are many possible approaches to designing a microprocessor simulation architecture, but the granularity of the model is one of the first things to consider.

The system could be simulated at the internal model level, the external signal level, the bus-cycle level or the functional level. Let's look at these possiblities.

The maximum granularity would be achieved by simulating the processor using a model of its internal organization. For example, a VHDL description could be used. However, this level of detail is clearly overkill for a simulator of a processor that is already available and known to work. (The internal model simulation would, of course, be appropriate during the design of the processor itself.) Furthermore, not only is the detail unnecessary, it would substantially slow down the speed of the simulation. Even more importantly, it would be a great deal of work to create the model and it is unlikely that the manufacturer would make its model freely available. In short, it is a "no brainer" to reject this level of granularity for the simulator.

The next level of granularity simulates the processor's external connections. For example, simulating the 6811 at this level would involve the crystal clock, signals such as AS and the multiplexed bus. The specifications for these signals are published by the manufacturer and readily available. Such fine-grained simulation would offer the considerable advantage of plugging the simulator into a general purpose circuit simulator thus allowing the complete simulation and debugging of the hardware and software components of a complete system.

However, this level of detail will slow down simulation and is not required when only the software and functionally-defined external hardware is being simulated and debugged. Implementing all aspects of this level including setup and hold times, rise and fall times loading factors and so forth would be onerous.

The coarsest granularity needed for software simulation is the bus cycle. (Yes, assembly language programmers do consider bus cycles.) Each bus cycle (read, write or idle) is simulated. This gives almost as much power as the external signal level, but will run faster and be less tedious to implement. Furthermore, external hardware whose functionality is defined can be simulated at this level.

The coarsest granularity possible for a simulator of machine language would be a functional one. This could offer the fastest performance, but could make the simulation of arbitrary

devices unduly complex. (Actaully, I am not sure about this. I think a functional approach is possible with reasonable programming effort; it would offer a considerable performance boost.)

I have chosen to implement the simulator at the *bus cycle* level of granularity. Frankly, I chose this method at the outset and did not even consider other levels of granularity. It was only after writing this analysis that I became quite interested in examining simulation based on higher and lower levels of granularity.

3.1. A closer look at alternative strategies

3.1.1. Clock simulation

In the case of the 6811, moving the simulation to the crystal clock level (instead of the bus cycle level) is probably not too difficult. Basically, the bus cycle is split into four phases and the appropriate external signals are set or monitored at each phase. This kind of crystal-clock simulation would degrade speed by a small amount. Alas, taking timing parameters (setup/hold/rise/fall times) and loading factors (resistive and capacitive) would add considerable complexity.

3.1.2. Functional simulation

The functional approach is also very interesting and might provide a significant performace boost (maybe a factor of 2 or even an order of magnitude!). With the current approach, every single bus cycle is simulated; this is prudent, but most bus cycles have no side effects (such as setting a status bit waiting for this particular cycle). A clever functional-level simulator could detect instructions (or even groups of instructions) that cannot have side effects related to cycle count. These instructions could be simulated in-line without the overhead of cycle-by-cycle simulation.

4. Fundamental Design

The most important design decision was to limit the project to the pure simulation engine and defer questions relating to user interface issues to a separate software development stage. (However, both stages could, and did, procede concurrently.)

The importance of this decision **cannot be overemphasized**. There were two major consequences:

1. Designing the simulation engine was focussed on that single problem leading to (I think) a better design.
2. Since the design was unhindered by the specifics of any particular kind of user interface,

I could concentrate on providing mechanisms to support arbitrary user interfaces and avoid the pitfall of a simulation engine designed primarily for a GUI or a line-oriented command user interface.

As a collorary to this decision, there is no code in the engine that communicates directly with the user or reads user-defined files. The engine deals only with Memory Spaces, Cells and CPUs. Some of the things the engine does not deal with are:

1. The engine has no user-friendly mechanism (such as a configuration file) for defining what addresses are mapped to specific devices (such as RAM, ROM, EEPROM, internal/external devices, etc.) However, the engine *does provide* mechanisms that a user interface can exploit to accomplish these objectives.
2. Any user interface must allow for the initialization of memory from (for example) Motorola S19 or Intel Hex records. Once again, the engine has absolutely no code to parse S19 or Intel Hex records; it merely provides an API that a third party can exploit to furnish these important user interface features.
3. The engine has absoultely no knowledge of mapping between source code (C, C++, assembler or whatever) and its machine language image. The engine itself deals only with machine language (although it does have a simple disassembler.) Yet again, however, the engine provides mechanisms allowing a user interface to exploit such mappings.

Despite the separation between user interface and basic engine, I was certainly aware of the potential needs of user interfaces as the basic engine was being implemented. Indeed, there are some basic mechanisms built into the engine specifically designed for user interfaces. These include the Warning and CellChange features that will be described later.

5. Bus cycle simulation

I chose the Bus Cycle Simulation model for the simulation engine. This means that every bus cycle (initiated from the core processor) is simulated. Each bus cycle either READs something from somewhere, WRITEs something somewhere, or twiddles its thumbs in an IDLE cycle. (Of course, the IDLE cycle is not totally brain-dead; these cycles are used by the core processor to perform necessary tasks.)

This strategy implies that there is at least one object that can initiate bus cycles and that there are at least one object that can respond to READ and WRITE bus cycles.

I use a single CPU object to initiate bus cycles. The READ/WRITE cycles communicate with a *Memory System* which is composed of one or more MemorySpaces. Each MemorySpace contains a collection of Cells; it is the individual Cells within a MemorySpace that collaborate with READ/WRITE bus cycles.

To summarize, the bus-cycle design implies three basic classes:

1. A CPU that initiates bus cycles.
2. A Memory System that interacts (indirectly) with READ/WRITE bus cycles.
3. Cells (composed within a Memory System) that interact directly with READ/WRITE cycles.

The following sections treat each of these fundamental objects.

6. The Memory System

The fundamentals of the memory system were designed for any type of processor. In particular, the abstract design does not assume a specific memory space size or the number of bits transferred in a bus cycle.

In addition, no assumptions are made about the number of memory spaces used by a processor. Some processors have only a single memory space (such as the 68hc11 or the 68xxx) that is mapped to RAM, ROM, and devices. Other processors (such as many DSP chips that use a Harvard architecture in which the data and program memory spaces are distinct or the Intel 80x86 family that has separate I/O and ordinary address spaces) have more than one memory space.

The design of the memory space abstraction does not limit the number of memory spaces that a processor may support nor does it define the size of memory space cells or the size of a memory space. (In short, the abstraction allows arbitrary address bus and data bus widths for each memory space.)

6.1. Memory Spaces

A *Memory Space* is a concept. It is not an implementation. *What* it does (not how it does it) is important.

A memory space is a collection of *cells*. Cells are also abstractions; they may be bytes, 16-bit entities or anything else. The Cell abstraction is examined later; the Memory Space abstraction makes no assumptions about how the Cell abstraction works.

In Java, the most general way to describe an abstract conceptual design with no implementation is with an *interface*. The Memory Space abstraction merely states that a Cell is associated with each possible address. An abridged version of the interface is given here:

```
public interface MemorySpace {
    Cell getCell(int address);
    void setCell(int address, Cell cell);
    void init();
}
```

The interface does not describe how methods like `getCell(int)` are implemented. If the

address space is big (eg. 32 bits) and (hence) it is likely that it is not fully populated, then some form of hashed implementation is appropriate. Our target, however, is a simple processor with a mere 64K (16-bit) address space. Consequently, the only implementation of a `MemorySpace` provided is one for 64K Cells which are maintained in a simple array. An abridged version of the source code is:

```
public class MemorySpace64K implements MemorySpace
{
    private Cell[] cells = new Cell[0x10000];
    private static final MemorySpace INSTANCE = new MemorySpace64K();
    public static MemorySpace getInstance() {return INSTANCE;}
    private MemorySpace64K() {}  //private to ensure singleton property
    public final Cell getCell(int address) {return cells[address];}
    public void setCell(int address, Cell cell) {cells[address] = cell;}
}
```

Note that the tiny 64K addressing space can be easily implemented as a simple array. Note also that the `MemorySpace64K` class is implemented as a *singleton*. (Refer to GoF for definition of what a *singleton* is.) In this case, there is NO public constructor; the only way to get a `MemorySpace64K` object is with `MemorySpace64K.getInstance()`.

6.2. Cells

A Cell is mapped to an address in a Memory Space. Specific cells are obtained using the `MemorySpace` interface. Once a cell has been obtained, its contents can be examined or modified subject to constraints based on the specific nature of the cell.

The Cell interface allows the contents to be obtained or modified in two different ways: "get/read" to obtain the contents and "set/write" to change them. Broadly speaking the "get/set" methods allow the direct manipulation of a cell's content; the "read/write" methods simulate read (write) bus cycles.

The distinction between "set/write" is particularly clear when referring to ROM cells. By definition, you cannot write to ROM. (Note that a real processor would not blink when a write bus-cycle had a ROM cell as its destination; the CPU would consume some bus cycles, but the READ-ONLY memory location would not be changed.) BUT you have to able to set the contents of ROM locations. In real life, you would have to burn your desired contents into a ROM. In simulated life, the `set` method makes it possible to directly change the contents of a ROM cell; a write cycle would, of course, have no effect. Note that an instruction that modifies memory (such as a `staa` instruction) would initiate a Write cycle to modify the memory; it would *not* use the `set` method. If the destination of the write cycle were in ROM, the contents would not be changed.

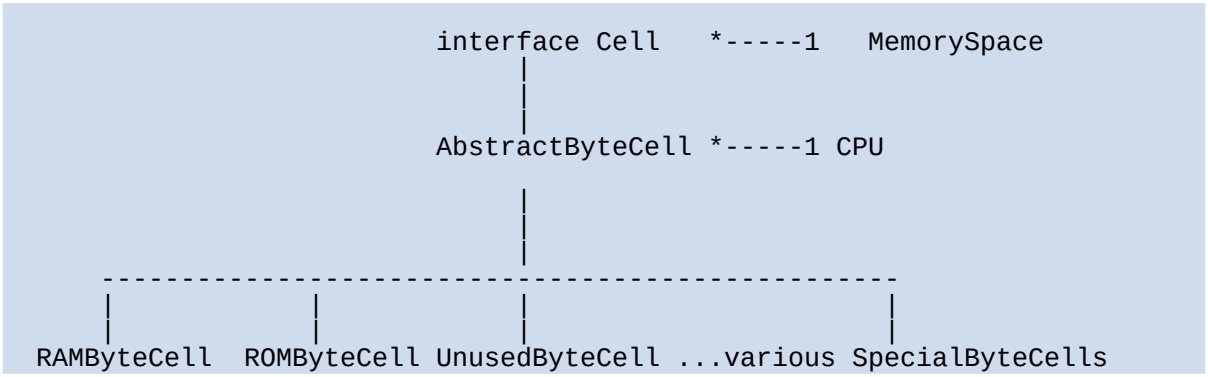
The (abridged) `Cell` interface is:

```
public interface Cell {
    void write(int contents);
    int  read();
    int get();
    void set(int contents);
    void reset();
    int getAddress();
    void addCellChangeListener(CellChangeListener l);
    void removeCellChangeListener(CellChangeListener l);
}
```

Note the `addCellChangeListener()` and `removeCellChangeListener()` methods. The intent is to give user interface implementors a hook so that they can be informed of the changes to the contents of a cell.

6.3. Byte-sized Cell Implementation

The implementation of cells is roughly described in the diagram:



Note that the Cell interface is associated with a MemorySpace. The parent class for all concete implementations of byte-sized cells is the `AbstractByteCell` class. This class is associated with a CPU class that we will discuss later on.

6.4. AbstractByteCell

An abridged version of the AbstractByteCell.java implementation is:

```
public class AbstractByteCell implements Cell {
    int contents;
    int nWrites;
    int nReads;
    int address;
    ArrayList listeners = new ArrayList();
```

```
    CPU6811 cpu =  CPU6811.getInstance();

    public AbstractByteCell(int  address, int contents) {
        this.address = address; this.contents = contents;
        nWrites = 0; nReads = 0;
    }

    public int get() {return  contents;}

    public int read() {
        nReads++;
        cpu.incrementReadCycles();
        return  contents;
    }

    public void set(int contents) {this.contents = contents;}

    public void write(int contents)
    {
        nWrites++;
        cpu.incrementWriteCycles();
        set(contents);
    }

    public int getAddress() {return address;}

    public void addCellChangeListener(CellChangeListener l) {}

    public void removeCellChangeListener(CellChangeListener l) {}
}
```

NOTES:

1. Much of the implementation (for example the `get/set` methods) is trivial.
2. The `read/write` methods keep track of the number of times the cell is read(written) and inform the CPU of the read/write cycles. (The CPU maintains a global count of the total number of cycles, read cycles and write cycles.)

   Why keep track of the number of times an individual cell is read or written to? The short answer is: "Why not?" It is very easy to do and the penalty in computation and memory usage is tiny. However, it is also a fact that the user interfaces to the simulation engine currently being developed (Summer 2002) make no use of this information.

   The question then becomes: "Why track individual cell read/writes if nothing is being done with the information?" Good question! But the answer is simple: "This information could be used to perform code coverage analysis (by checking how many memory locations that correspond to instructions have been read), profiling and other things". The mere fact that no current user interface (as of mid-September 2002) exploits this mechanism does not mean that these capabilities could not be exploited by future user interface designers.

3. The `getAddress()` method implies that Cells are aware of their address. The astute reader may notice a bug here: the address of a Cell is insufficient to identify it unless the memory space is known. Fortunately, the only concrete implementation of the simulator framework is for the 68hc11 which has only one memory space. (The memory space is implemented as a singleton and, hence, is always knowable within the package.) If a processor with more than one address space were to be simulated (e.g. the Intel 80x86 with its separate memory and I/O address spaces), each Cell would have to know both its address and its memory space.

7. The CPU

The CPU itself is the third major component of the simulation engine. (The other two major components are the MemorySpace and Cell interfaces and implementations.) Unlike the the other two majors, the CPU does not (at present) have a higher level abstraction such as a Java interface or abstract class. Only the `CPU6811` is implemented.

The `CPU6811` class is implemented as a singleton. Clients obtain a reference to the cpu with code like:

```
    CPU6811 cpu = CPU6811.getInstance();
```

Thus all clients are guaranteed to be working with the same 6811 cpu engine.

The (abridged) source code in CPU6811.java that ensures this is:

```
public class Sim6811 {
    private static final CPU6811 instance = new CPU6811();
    public static CPU6811 getInstance() {return instance;}
    private CPU6811() {} //Private constuctor; clients CANNOT create
instance
}
```

The implementation of the CPU6811 class is divided into several components including:

1. Registers
2. Statistics
3. Execution

Each of these portions is discussed in greater detail in the following sections.

7.1. CPU registers

Each register is declared as a simple primitive data type and given arbitrary initial values as shown in the following code segment. Note that each bit in the Condition Code register is given its own variable.

```
    private int AccA = 0x12;
    private int AccB = 0x34;
    private int PC = 0x6000;  //Standard starting address at RyersonU
    private int IX = 0;
    private int IY = 0;
    private int SP = 0xff;   //?Standard BUFFALO monitor stack?
    private boolean CC_Z = false;
    private boolean CC_N = false;
    private boolean CC_C = false;
    private boolean CC_V = false;
    private boolean CC_H = false;
    private boolean CC_I = true;
```

Each register has its own public get/set methods. For example:

```
    final public int getAccA()   {return AccA;}

    final public void setAccA(int i)  {AccA = i;}
```

7.2. Statistics

The CPU engine maintains statistics on the total number of reads, writes, number of instructions executed and number of bus cycles simulated with the following variables and initial values:

```
    private int totalWrites = 0;
    private int totalReads = 0;
    private int totalNInstructions = 0;
    private int nCycles = 0;
```

The CPU is informed when a read, write or idle cycle occurs. For example, if the execution of an instruction performs a read cycle, it must also invoke the following CPU method:

```
    public final void incrementReadCycles()
    {
        totalReads++;
        incrementCycles();
    }
```

Whenever any bus cycle occurs, the CPU must be informed with the `incrementCycles()` method: i.e. this method is invoked for *every* E-clock cycle. Thus it is invoked often--one million times per simulated second--and must be efficient. The code is:

```
    public final void incrementCycles()
    {
        nCycles++;
        TimedEvent event = (TimedEvent) pendingTimedEvents.tickAndRemove();
        if (event != null) {
            event.doAction();
        }
```

```
    }
```

Clearly, the `incrementCycles()` method invokes the method `tickAndRemove()` every time. Even without knowing why we do this and what the method does, it is evident that it must be fast if the simulation engine is to be fast. It is indeed an important element in the architecture of the simulator; however, the details of what it does is deferred until later in this document.

7.3. Instruction fetch/execute cycles

We have now described most of the scaffolding in the `CPU6811` class and the associated `MemorySpace` and `Cell` classes that allow the CPU to perform its central job: fetching and executing instructions stored in its Memory Space.

An abridged (and slightly inaccurate) version of the CPU's `step()` method, which fetches and executes a single insruction, is shown below:

```
    public final InstructionEffects step()
    {
        Instruction inst = fetchInstruction();
        InstructionEffects effects = inst.execute();
        totalNInstructions++;
        checkInterrupts();
        return effects;
    }
```

I hope that the code is (almost) self-explanatory: we fetch an instruction, execute it, update the number of instructions statistic and, when the execution is complete, check if there are any pending interrupts. (Note that, once again, the simulation code mimics what the real processor does; like the code, the real hardware only checks for interrupts *after* and instruction has finished.)

But *how* is an instruction fetched and then executed? Let's look at these details. (These are implementation details that are written as private methods.)

Let's start with another private method: `fetchOpCode()`:

```
    private final int fetchOpCode()
       {return mem.getCell(CPU6811.PC++).read();}
```

This simple method (it's only a single line of code) relies on many of the archtectural features of the simulation engine's design. First, `mem` is a private variable in the CPU6811's object that contains the Memory Space involved in the simulation. We then obtain the specific Cell (or byte in this case) at the address specified by the Program Counter (and we also increment the PC just like the real hardware in the 6811 CPU.) Finally, we invoke the `read()` method on that cell and obtain its contents as an `int`; in short, we get the first byte of the instruction.

(Note that this also mimics what is going on in the real hardware state machine implemented in the actual 6811 CPU.)

Let's now look at how an instruction is fetched with the `fetchInstruction()` method:

```
    private final Instruction fetchInstruction()
    {
        int opCode = fetchOpCode();
        switch (opCode) {
          //pre-byte instruction stuff not shown
        default:
            Instruction inst = OpCodes.Opcodes[opCode];
        }
        return inst;
    }
```

The op-code of all 6811 instructions is one or two bytes long. Over 90% of the instructions have the op-code encoded into the first byte of the instruction. The other instructions have their op-code encoded in two bytes: a pre-byte (which must be 0x18, 0xCD or 0x1A) followed by a second byte which completely specifies the instruction and its addressing mode. For simplicity, the abridged code above and the following discussion assume that only instructions with a one-byte op-code are simulated. Note, however, that the op-code is not the whole instruction; most instructions (except for Inherent mode instructions) also require an operand (encoded in machine language) following the op-code. (The truly curious reader is invited to examine the real source code to see how instructions with a pre-byte op-code are handled.)

Once we have an op-code (which will be an integer in the range 0--255), we can retrieve an `Instruction` from an array of `Instructions` indexed by the op-code.

But what is an `Instruction`?

An `Instruction` is an interface:

```
public interface Instruction {
    //constants omitted from abridged code
    public int execute();
    public String toDisassembledString(); //Not discussed here
}
```

Every Instruction object in the array of instructions (indexed by op-code) implements this interface. But every one is different! Each one is a real class (not an interface). In addition, each one extends an abstract class that implements the interface. This class is `AbstractInstruction`.

Let's have a look at this `AbstractInstruction` class.

Before getting into details, let me first note that it is the biggest class (measured by lines of

source code) in the engine package. (It is about 600 lines of code including comments.) The size of this class is enormous because it allows its sub-classes (the real instructions) to inherit (use) its library of useful methods.

There are dozens of such utility methods in the abstract class. As a simple example, many instructions need to read a byte from some address; they can use the inherited `read8()` method shown below:

```
    int read8(int address) //AbstractInstruction method
    {
        int data;
        data = (int) mem.getCell(address).read();
        data &= 0xff;
        return data;
    }
```

(Note that the `read()` method for a cell will increment the number of cycles so the programmer does have to remember to do this.)

For a more complex example, many instructions need to add two bytes and set the N, Z, V, C and H bits of the Condition Code register according to the result. Rather than clutter all of these instructions with the messy details of how this is done, they can use the inherited `add8()` method shown below:

```
    final public int add8(int p, int q) //AbstractInstruction method
    {
        int result = (p + q)&0xffff;
        setNZ8(result);
        cpu.CC_C = (result > 0xff);
        cpu.CC_V = (result > 0xff) != (((p&0x7f) + (q&0x7f)) > 0x7f);
        cpu.CC_H = ((p&0xf) + (q&0xf) > 0xf);
        return (result & 0xff);
    }
```

Every actual instruction extends `AbstractInstruction` (or another derived abstract class) and fills in a few details such as what the instruction really does (in the `execute()` method.) For example, the `AbaInh` class (which implements the `aba` inherent mode instruction that adds Accumulator B to Accumulator A) is shown below:

```
public class AbaInh extends AbstractInstruction
{
    private CPU6811 cpu =  CPU6811.getInstance();
    final public int execute()
    {
        cpu.setAccA(add8(cpu.getAccA(), cpu.getAccB()));
```

```
        cpu.incrementCycles();
        return -1;
    }

    final int getAddressingMode() {return Instruction.INH;}
    final protected String getOpSymbolic() {return "ABA";}
}
```

7.4. Simulating at full speed

When a user interacts with the simulation engine to run at full speed, the engine may be busy for a long time. The simulation will stop when a breakpoint or stop instruction is reached or when the user explicitly terminates the run. Since the user input has to be monitored and a possibly infinite number of instructions have to be simultaneously simulated, a separate thread of execution is required for the simulator while the main thread continues to monitor user requests.

The CPU's `go()` method creates and returns the Thread. An abridged version of the source code is shown below:

```
    public Thread go(int maxNInstructions)
    {
        setState(RUNNING);
        Thread goThread = new Thread ( new Runnable() {
                public void run() {doFastSimulation(); }
        } );

        return goThread;
    }
```

The `doFastSimulation()` (code not shown) calls `fastStep()` in a tight loop and checks if a breakpoint has been hit or the user has requested to stop the simulation. The `fastStep()` method code is shown below:

```
    public final int fastStep()
    {
        int rc;
        Instruction inst = fetchInstruction();
        totalNInstructions++;
        rc = inst.execute();
        checkInterrupts();
        return rc;
    }
```

8. Instruction Set

When I demonstrated my initial very limited version of the 6811 simulator (which simulated a half dozen instructions but did incorporate most of the core ideas described in this

document) to my colleague Peter Hiscocks, he commented, "Ken, this looks interesting but, you know, there are over 300 instructions in the 6811." My initial reaction was "So what... the design is architecturally sound...I'll worry about the details later..."

However, I soon encountered the abyss: "Oh... If each instruction requires a separate class of about 15 lines of Java code and there are 300 instructions, I have to write nearly 5000 lines... Oops!"

I decided to write a text file describing each instruction in one line. Some typical lines are:

```
aba inh 1b f
adca imm8 89 t
adca dir8 99 t
adca ext8 b9 t
adca ix8 a9 t
```

The fields indicate the symbolic assembler, the addressing mode, the machine language opcode and whether there are more than one addressing modes. I wrote another program (a Perl script) to read these lines and translate each one into Java source code for the instruction. The generated source code had to be completed by hand; this meant implementing the `execute()` which was simple to do. (Yes, it was somewhat tedious, but *much* less so than writing these 5000 lines of code by hand!)

The treatment of instructions with more than one address mode was different. In these cases, an abstract version of the instruction was generated and a separate concrete class for each addressing mode was created.

For example, the `ldaa` instruction generated the following abstract class:

```
public abstract class LdaaAbstract extends AbstractInstruction
{
    private CPU6811 cpu =  CPU6811.getInstance();
    final public int execute()  //This method was written manually
    {
        int op = getOperand();
        cpu.setAccA(op);
        setNZ8(op);
        cpu.setCC_V(false);
        return -1;
    }
    final protected String getOpSymbolic() {return "LDAA";}
}
```

The automatically generated code for the `ldaa` using the Extended addressing mode is:

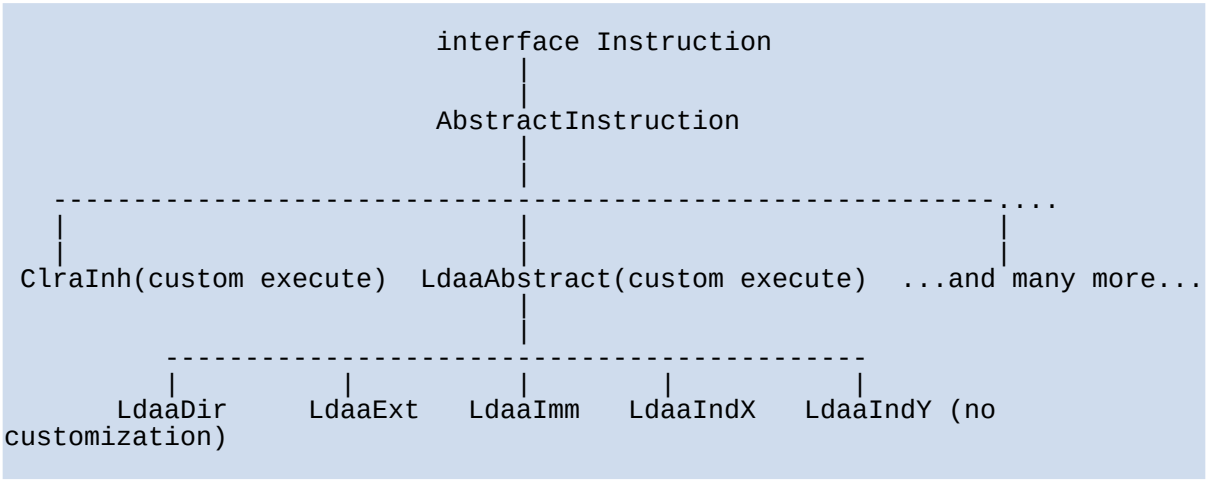
```
public class LdaaExt extends LdaaAbstract {
    private CPU6811 cpu =  CPU6811.getInstance();
    final public int getOperand() {return  getEXT8Operand();}
```

```
    final int getAddressingMode() {return Instruction.EXT8;}
}
```

(Notes: the `getOperand()` concete method is used by the simulator to obtain the actual operand; the `getAddressingMode()` method is only used by the disassembler.)

With this hierarchy of classes, I only had to customize the `execute()` method in the abstract class rather than in each individual addressing mode instance of the LDAA instruction. Besides reducing typing (or cutting and pasting), any bugs in the implementation of `execute()` were conveniently encapsulated in one place.

To summarize, the architecture for implementing Instructions is broadly described in the following diagram:



9. Events

Some cells are mapped to devices that are time sensitive. For example, when the Analog-to-Digital subsystem is told to convert 4 analog channels it needs 128 bus-cycles to complete the conversions. There is a flag bit (Converstion Complete Flag or CCF) that is cleared when the conversion process begins and set when it is completed.

10. Testing

Testing software is often forgotten or pushed to the background during development. Professors may gripe about this and insist on some sort of testing procedure, but they are sometimes just as guilty in ignoring this essential aspect of software implementation.

I have certainly been guilty of this sin, justifying the poor testing with the standard excuses, "Hey, I have to get this thing out the door! Let's worry about testing later. Indeed, let the

users (aka "guinea pigs") do the testing for me."

Recently, however, I have discovered the magic of unit testing and the Junit implementation of this technique. My mastery of this technology is still weak (mastery involves knowing how to use it and, more importantly, practising the discipline of using it), but I have tried to exploit it while the engine code was being developed.

An aphorism that describes unit testing is "Code a little, test a little". In the past, I often implemented this simple idea by littering code with `println` debugging statements. Unfortunately, the litter had to be removed (or commented out) at some later stage; furthermore, the litter was sometimes only informative (eg. a certain block of code has been reached with such and such conditions) or, at other times, actually did some verification.

Unit testing is as simple as inserting debugging code, but offers the huge advantage of not clutterring the implementation code. You never have to delete debugging code from the implementation and can continue to run the tests no matter how the implementation code evolves. This gives you automated regression testing.

In addition, unit testing forces the programmer to write tests that actually test something (instead of simply providing information.) This tends to focus the mind at the problem at hand and can result in cleaner designs and implementations.

One surprising advantage to unit testing during development involves writing the test code *prior* to coding the implementation and even writing tests you expect to *fail*.

10.1. Example of Unit Testing: the DeltaList

Most of the intellectual portion of the design/implementation of the simulation engine is found in the abstract object-oriented architecture (the design of its *data* rather than its *algorithms*.) One of the abstractions used in the design whose algorithmic implementation was a tad more complex than most was the `DeltaList`. The steps in its design and implementation (including the use of unit testing) are summarized here.

10.1.1. Background

Recall that one aspect of the simulation engine's efficiency is the way it can handle bus cycle timed events. My first decision was to use a delta-time list as both the design and implementation of this aspect; indeed, the object implementing the basic idea is called a "DeltaList".

Before implementing code, I realized that the standard implementation of this idea was not necessarily the best one in all circumstances. (For example, the idea could be implemented with a standard Priority Queue algorithm instead of a linked list.) Consequently, I tried to

write the public documentation for the `DeltaList` class so that it did not imply any particular implementation.

10.1.2. Starting

Initially, I wrote a test case for a `DeltaList` object and implemented it with stubs.

The initial test was:

```
    // Add a single item, check size and delete when expired.
    public void testSimpleDelta1() {
        DeltaList deltaList = new DeltaList();
        assertEquals(deltaList.size(), 0);
        //Add an event to fire after 3 ticks
        TimedEvent item3 = new TimedEvent(3);
        deltaList.add(item3);

        assertEquals(deltaList.size(), 1); //List only contains single item
        assertEquals(deltaList.tickAndRemove(), null);
        assertEquals(deltaList.tickAndRemove(), null);
        assertEquals(deltaList.tickAndRemove(), item3);
    }
```

The initial implementation used stubs. An abridged version is:

```
public class DeltaList {
    public DeltaList(){}
    public int size() { return 0;}//STUB size is ZERO
    public void add(DeltaTimed item){} //STUB does NOTHING
    public DeltaTimed tickAndRemove(){return null} //STUB does NOTHING
}
```

Why, you may ask, go to all the bother of creating test cases that will fail because the implementation itself (only stubs) guarantee failure? The big advantage: the basic code and testing framework can be written, compiled and run. (Note that public documentation should be written at this stage as well.)

Even with this stub implementation, the first test (`assertEquals(deltaList.size(),0);`) will actually pass! The next test (checking that the size of the list is one after adding an item) is expected to fail. Indeed, this is exactly what happened. This expected result (including the failure) gives a programmer confidence that they are doing at least something right and that it is time to start replacing the stubs with real implementations.

10.1.3. Oops

I added implementation and more tests. I concentrated on "boundary condition" tests (where,

experience has taught me, bugs often occur). Happily, these tests all worked.

I then implemented a test to mimic the example given in the public documentation of the class. (The details are described in the public documentation for the `DeltaList` class.) An abridged version of this test is shown here:

```
    // Try out the Moo, Foo Goo example in the public docs
    public void testSimpleDelta5() {
        DeltaList dl = new DeltaList();
        TimedEvent foo5 = new TimedEvent(5); dl.add(foo5);
        TimedEvent bar8 = new TimedEvent(8); dl.add(bar8);
        //do 3 ticks... assertEquals(dl.tickAndRemove(), null); // 3 times
        //now add goo (after 4 additional ticks) and moo(1 tick later)
        TimedEvent goo7 = new TimedEvent(4);
        TimedEvent moo4 = new TimedEvent(1);
        dl.add(moo4); dl.add(goo7);
        assertEquals(dl.tickAndRemove(), moo4);
        assertEquals(dl.size(), 3);
        assertEquals(dl.tickAndRemove(), foo5);
        assertEquals(dl.size(), 2);
        assertEquals(dl.tickAndRemove(), null);
        assertEquals(dl.size(), 2);
        assertEquals(dl.tickAndRemove(), null);
        assertEquals(dl.tickAndRemove(), null);
        assertEquals(dl.tickAndRemove(), goo7);
        // assertEquals(dl.tickAndRemove(), goo7);
        // assertEquals(dl.tickAndRemove(), bar8);

    }
```

Alas, the test failed...oops. What to do when a test you *expect* to work does not work?

Some of the choices:

1. I could have examined the algorithm source code and figured out (using brain power) what dumb thing I had coded that caused the bug.
2. I could have added informative debugging infomation to the test (NOT to the implementation) to help me zero-in on the bug.
3. I could have used a debugger (including an IDE such as Forte or Jbuilder).

I decided to add information to the test. The (abridged) test now looked like:

```
    public void testSimpleDelta5() {
        //...(deleted)...same as before...
        //do 3 ticks... assertEquals(dl.tickAndRemove(), null); // 3 times
        //now add goo (after 4 additional ticks) and moo(1 tick later)
        TimedEvent goo7 = new TimedEvent(4);
        TimedEvent moo4 = new TimedEvent(1);
        dl.add(moo4); dl.add(goo7);
        assertEquals(dl.tickAndRemove(), moo4);
        assertEquals(dl.size(), 3);
```

```
        assertEquals(dl.tickAndRemove(), foo5);
        assertEquals(dl.size(), 2);
        assertEquals(dl.tickAndRemove(), null);
        assertEquals(dl.size(), 2);
        assertEquals(dl.tickAndRemove(), null);
        assertEquals(dl.tickAndRemove(), null);
        assertEquals(dl.tickAndRemove(), goo7);
        // assertEquals(dl.tickAndRemove(), goo7);
        // assertEquals(dl.tickAndRemove(), bar8);

    }
```

I could now run the test, redirect `stdout` to a file and examine it. The output was:

```
DELTA test moo, foo, bar, goo...
        New delta list: []
        After adding foo5: [Delta: 5]
        After adding bar8: [Delta: 5, Delta: 3]
        After 3 ticks: [Delta: 2, Delta: 3]
        After adding moo4: [Delta: 1, Delta: 1, Delta: 3]
        After adding goo7: [Delta: 1, Delta: 1, Delta: 4, Delta: 1]
```

All went well until:

```
        After adding goo7: [Delta: 1, Delta: 1, Delta: 4, Delta: 1]
```

which should have been:

```
        After adding goo7: [Delta: 1, Delta: 1, Delta: 2, Delta: 1]
   // i.e. 3 ticks already so    ^moo4     ^foo5     ^goo7     ^bar8
```

This erroneous list representation caused all subsequent tests to fail.

I now had a fairly detailed view of the internal representation of the delta list and could focus on the "dumb code" I had written. For the mobidly curious, only one line had to be added:

```
        //...and set the delta of the item being added accordingly
        item.setDelta(itemAddTime);
```