Introduction to 6811 Programming

Fred G. Martin[*]

September 9, 2004

Abstract

This document provides an overview of the Motorola 68HC11 processor, focusing on programming the E1 series chip using the open-source `as6811` assembler.

1 Bits and Bytes

Most humans, having ten fingers, think in decimal numbers. In computers, information is represented with voltages, and it is most convenient for the voltage levels to represent only two states: a binary one or binary zero. Thus computers process binary digits, or bits.

For convenience, microprocessors group bits together into words. The first microprocessor, the Intel 4004, operated on a word composed of four bits. Many small microprocessors use 8-bit words, called *bytes.*

In an 8-bit numeral, 256 different states can be represented (since $2^8 = 256$). Programmers use these 256 states to represent different things. Some common usages of a byte of data are:

- a natural number from 0 to 255;
- an integer in the range of -128 to 127;
- a character of data (a letter, number, or printable symbol).

When programmers need to represent larger numerals, they group bytes together. A common grouping is two bytes, often called a 16-bit *word*, or a *short* (a "regular" word is 4 bytes of data). A word can have 65536 states, since $2^{16} = 65536$.

Decimal numbers are painful to use when talking about binary information. To make life easier, programmers started to use the base 16 *hexadecimal* (or *hex* for short) numbering system when talking about bits, bytes, and other binary data.

The hex system uses 16 different digits to represent each place value of a numeral. Using hex, one would count as follows: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F, 10 ...etc. The letters A though F are then used to represent the values of (decimal) 10

[*]Department of Computer Science, University of Massachusetts Lowell, 1 University Avenue, Lowell MA 01854. E-mail: `fredm@cs.uml.edu`. This document is Copyright ©1992–2004 by Fred G. Martin. It may distributed freely in verbatim form provided that no fee is collected for its distribution (other than reasonable reproduction costs) and this copyright notice is included. An electronic version of this document is available at the author's home page `www.cs.uml.edu/ fredm`.

binary	decimal	hex
0b0000	0	0x0
0b0001	1	0x1
0b0010	2	0x2
0b0011	3	0x3
0b0100	4	0x4
0b0101	5	0x5
0b0110	6	0x6
0b0111	7	0x7
0b1000	8	0x8
0b1001	9	0x9
0b1010	10	0xA
0b1011	11	0xB
0b1100	12	0xC
0b1101	13	0xD
0b1110	14	0xE
0b1111	15	0xF
0b10000	16	0x10
0b11100011		0xE3
0b10011100		0x9C
0b11111111	255	0xFF
	256	0x100
	1024	0x400
	65535	0xFFFF

Figure 1: Some Numeric Conversions

through 15, respectively. This is wonderful, because a hex digit (of 16 possible states) is equivalent to four bits exactly. Then, a byte can be represented by exactly two hex digits, and a sixteen bit word by four of them.

The following conventions are typical for representing values in using the different numeric bases. *Binary* numbers are represented by the prefix `0b`. *Hexadecimal* numbers are specified by `0x`. *Decimal* numbers don't have a prefix. (These aren't the only conventions that are used in the computer world, but they will be standard throughout these notes.)

Let's examine some of the numeric conversions in Figure 1. Notice that four bits equal one hex digit. This is helpful in converting binary to hex. Notice some entries don't have their decimal values filled in. This is to make the point that it's easy to transcribe between binary and hexadecimal representation, but using decimal is often cumbersome.

It's good to know some general quantities. For example, eight bits, or one byte, is 256 values. Then the largest unsigned integer representable in a byte is 255. The largest integer representable in two bytes is 65535.

A byte can be used to represent one character of information. A standard has been devised for this, called the American Standard Code for Information Interchange standard, or ASCII code[1]. ASCII is almost universally accepted for representing the English character set, including upper and lower case letters, numbers, and typical

[1]ASCII is pronounced as in "*ass-key.*"

Memory Address	Function	# bytes (decimal)
0x00–0x1ff	RAM memory	512
0x1000–0x013f	special registers	64
0xb600–0xb7ff	EEPROM memory	512

Figure 2: Memory Map of the 68hc11e1 Microprocessor

punctuation (like `!@#$%&*()`). An old competing IBM standard, the EBCDIC code, is largely defunct except on some of their mainframes, but modern computer scientists are presently devising a 16-bit international character code.

In the back of the Motorola HC11 programmer's handbook is a table of the ASCII codes. The most important thing to know is first that it exists, but then some other details. First, notice that it only uses seven of the eight bits in a byte. So, there are actually only 128 ASCII characters, using the values 0x00 to 0x7F hex.

Printable characters start at 0x20 hex (32 decimal). The codes from 0x0 to 0x1F are used for things like cursor control, line feeds, and the like. Knowing the ASCII characters become important if when doing interactive programming on your HC11, in which case the user might type ASCII information to the HC11 over the serial line, and it would respond in kind. Then, the programmer must deal with the characters as bytes, and the ASCII codes become important.

2 Introduction to the HC11

2.1 Memory Map

Microprocessors store their programs and data in memory. Memory is organized as a contiguous string of *addresses*, or locations. Each memory location contains eight bits of data (this is because the HC11 is an 8-bit micro; other processors can have 16 or 32 bits of data at each memory location).

The entire amount of memory that a processor can access is called its *address space.* The HC11 has an address space of 65,536 memory locations, corresponding exactly to 16 bits of address information. This mean that a 16-bit numeral can be used to point at, or address, any of the memory bytes in the address space of the HC11. Thus four hexadecimal digits (4 bits per digit $\times$ 4 digits) can exactly specify one memory location (in which one will find one byte of information).

In general, any area of memory should be equivalent to any other. Because the HC11 is a special-purpose chip, and all of its memory is etched right on the microprocessor chip itself, its designers had to dedicate portions of its memory to particular functions. Figure 2 shows a "memory map" of the HC11 chip. Let's go over this map in detail.

The first area of memory, from address 0x00 to 0x1ff, is the chip's *random access memory*, or RAM. RAM can be both written and erased. It's "volatile," which means that when power is removed from the chip, it loses its state. RAM is typically used for storing programs and data.

When you program the HC11, however, RAM is not typically used to store programs, because there's only 512 bytes of it (0x00 to 0x1ff is 512 bytes). It is normally used it to store data and variable values that the program will use while it's running.

Programs will reside normally in the EEPROM, an acronym for *electrically erasable programmable read-only memory.* EEPROM is the culmination of a trend in programmable, yet permanent, memory technology.

Read-only memory (ROM) means what it suggests: that memory can only be read, not written to like RAM. It is programmed at the factory, in mass quantities. This is good for companies that are selling a production version, but to satisfy R & D engineers, PROM (*programmable read-only memory*) was developed.

PROM chips can't be erased, so when in order to make changes to code, the chip is throw away and a new one is used. PROM chips aren't horribly expensive, but this process still imposes a high development cost.

EPROM, or *erasable programmable read only memory*, was the next step. Most EPROM chips are erased by exposing the chip to ultraviolet light for half an hour. This is a vast improvement over PROM, but unless there is a large supply of blank chips for reprogramming, the programmer will have a long wait time between code downloads.

The Motorola 68hc11e1 chip uses recent ROM technology: EEPROM, which is electrically erasable. This means that the chip can erase its own ROM, and download new data to be written into it. It's the ultimate thing for microprocessor hackers, because new programs can be downloaded into the chip in just ten seconds or so. Also, because it's ROM, when the micro is powered down, its program doesn't go away.

EEPROM isn't a substitute for RAM: writing new data in is extremely slow by RAM standards, and can only be done a finite number of times (about one to ten thousand erase/write cycles, to be exact).

The EEPROM resides from addresses 0xb600 to 0xb7ff. There's just 512 bytes of it.

In the middle part of the address space, starting at address 0x1000, is an area for special control registers. By storing and reading values from this area of memory, you can control input/output functions like the serial ports, sensors and motor ports, and a host of other HC11 special functions. These features are discussed in detail in Section 4.

2.2 Registers

A microprocessor does its work by moving data from memory into its *internal registers*, processing on it, and then copying it back into memory. These registers are like variables that the processor uses to do its computations. There are two different types of registers: *accumulators*, and *index registers.*

Accumulators are used to perform most arithmetic operations, like addition, subtraction, or performing logical and bit operations (and, or, invert). Results of such operations often are placed back into a register; for example, an instruction may add something to the "A" register, and place the sum back into that same register. It's for this reason that the name accumulator is appropriate for these register type: they accumulate the results of on-going computations.

Index registers are used to point at data that is located in memory. For example, in the add operation just described, the addend (the number getting "added in" to the sum) might be indexed by the "X" register, meaning that the X register is being used to indicate the address of the data in memory.

Figure 3 shows the "programmer's model" of the registers of the HC11.

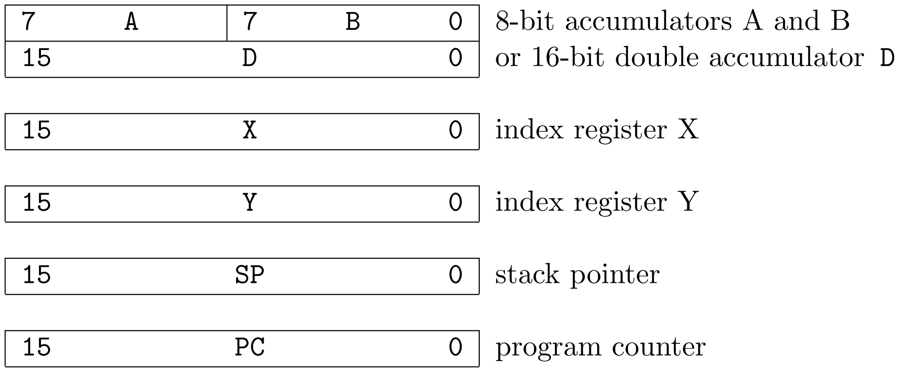

Figure 3: Programmer's Model of HC11 Registers

The HC11 has two accumulators, labelled A and B. Each are 8-bit registers: they hold one byte of data.

The general-purpose index registers are the X and Y registers. These are 16-bit registers and are most commonly used to address data in memory.

The A and B registers can be used together as a 16-bit arithmetic register, in which case they are named the D register. As indicated in the diagram, the A register forms the "high bits," or most significant digit, in this mode.

The Stack Pointer, or SP register, is used to store the location of the *program stack*. The stack, which is explained in detail later, is used for temporary storage of data, and to store the return address before a subroutine is called.

The Program Counter, or PC, is used to keep track of the current instruction being executed. The PC is automatically incremented as the microprocessor proceeds through the instruction stream.

3 Programming the HC11

When a microprocessor runs a program, it advances sequentially through memory, fetching and executing one instruction at a time. As mentioned earlier, the PC (program counter) register keeps track of the address of the instruction currently being executed. The microprocessor automatically advances the PC to the next instruction after it is finished executing the current one.

Let's look at typical instruction: load a number into the A register. The machine code for this instruction is (in hex): `86 nn`, where `nn` is the byte to be loaded into the register. The hex value `0x86` is called the *operational code*, or *op-code*, that signifies the "load A register" instruction.

Instructions may be one, two, three, or four bytes long, depending on what their function is. When the microprocessor encounters the byte `0x86` in the instruction stream, it knows, "I'm going to fetch the next byte of data, and load that into my A register." After the microprocessor evaluates the first byte of an instruction, it knows how many more bytes it needs to fetch to complete the instruction, if it is longer than one byte. Then it executes the next instruction, and so on, ad infinitum.

Instructions take varying numbers of *machine cycles* to execute, depending on their complexity. The HC11 we're using operates at a frequency of 2 megahertz (Mhz.), meaning that it executes 2,000,000 machine cycles per second. The period of a machine cycle is then 0.5 microseconds (μsec), so an instruction that requires 3 machine cycles will take 1.5 μsec of real time to execute.

In general, longer instructions (those needing two, three, or four bytes) take longer (more machine cycles) to execute, although there are some exceptions to this rule.

3.1 Machine Code vs. Assembly Language

People often speak of *machine code* and *assembly language*. Both of these terms refer to the same thing: the program that is executed directly by the microprocessor.

However, these terms refer to that program in different states of development. Let me explain.

Machine code usually refers to the raw data stored as a microprocessor's program. This is commonly described in the hexadecimal notation we've been using.

Assembly language is a set of *mnemonics*, or names, and a notation that is a readable yet efficient way of writing down the machine instructions. Usually, a program that is written in assembly language is processed by an *assembler program*, that converts the mnemonic instructions into machine code. This output from the assembler program is often called the *object code*, which can then executed directly by the microprocessor.

In the HC11 assembly language, the "Load A register" instruction that we discussed earlier is written as follows:

```
ldaa #0x80
```

The word "`ldaa`" is the assembly language mnemonic for "LoaD Accumulator A." Then, the `#0x80` is the hexadecimal value to be loaded (the value 0x80 was selected at random).

When a HC11 assembler program processes an input file, it knows the mnemonics for all of the HC11 instructions, plus their corresponding op-codes. It uses this information to create the object code file.

The assembly process is a straight-forward, mechanical operation. Each assembly-language instruction is converted to one machine-language instruction (though that instruction may be one to four bytes in length). Assembler programs have none of the sophistication that high-level language compilers must have.

But, assemblers typically have features to make writing assembly programs easier. These features allow the creation of symbolic labels for constant values or memory addresses, perform arithmetic in binary, decimal, and hex format, and convert character strings to binary values (amongst other functions).

Rather than presenting an overview of assembly language all at once, HC11 instructions are introduced throughout this document in a progressive fashion.

3.2 Addressing Modes

In our previous example (`ldaa #0x80`), the hex value 0x80 is loaded into the A register. This method of loading data into the register is called *immediate addressing*, because the data to be loaded is located "immediately" in the instruction itself. Immediate addressing is commonly used to load a known piece of data into a register.

There are other ways to address data bytes that need to be operated on. These different methods are known as *addressing modes*. Other than the immediate addressing mode, most addressing modes provide ways of accessing data that is stored somewhere in memory.

The *extended addressing mode* is a one way to access data stored in memory. In this mode, the 16-bit address of a memory byte is specified in the instruction. For example, the instruction

```
ldaa 0x1004
```

will load the A register with the contents of memory location 0x1004. This instruction uses three bytes of memory: one byte is the op-code, and two more bytes are needed to specify the 16-bit memory address.

The *direct addressing mode* is similar to the extended mode, but works only for data stored in the first 256 bytes of the chip's address space, from addresses 0x00 to 0xFF. This happens to be the chip's RAM, as shown in Figure 2, the HC11 Memory Map. So the direct mode is used to store and load data to the RAM.

In the direct mode, only one byte of address data is required to specify the memory address, since it is known to be in the first 256 bytes of memory. Instructions using direct addressing may require only two bytes: one for the op-code, and one for the address information. They execute in fewer cycles as a result of this savings. The HC11 assembler will automatically choose the direct addressing mode if the address specified is in the range 0x00 to 0xFF. Extended addressing could also be used to access this portion of memory, but it would rarely be preferable.

The *indexed addressing mode* uses the X or Y register as a pointer into memory. The value contained in the index register is and an offset byte are added to specify the location of the desired memory byte or word.

Let's look at an example to make this clear. Suppose the X register currently has the value 0x1000. Then the instruction

```
ldaa 0,X
```

will load the A register with the contents of location 0x1000, and the instruction

```
ldaa 5,X
```

will load the A register with the contents of location 0x1005.

The offset value is contained in one byte of data, and only positive or zero offsets are allowed. As such, only offsets in the range of 0 to 255 decimal are possible.

Why would a programmer use the indexed addressing mode, when the extended addressing mode will access the desired byte directly? The reason to use the indexed modes, with their associated offset bytes, is when repeatedly accessing locations from a particular region of memory.

For example, the HC11 special register area begins at location 0x1000 and ends at location 0x103F. Suppose there were a series of instructions that accessed the registers located in this area. We could then set up the X register as a *base pointer*—pointing to the beginning of this area of memory (we'd load the X register with 0x1000: `ldx #0x1000`). Then, we can use the two-byte indexed instructions to do a series of loads, stores, or whatever to the locations in this region that we were interested in.

This is good programming practice because each indexed instruction saves a byte over the extended instruction. Once the cost is paid of loading the X register with the base address (a three byte instruction), each use of an indexed instruction will save code space and execution time.

Indexed addressing really is most useful when working with arrays of common data structures. Then, one can set up an index register to point at the base of each data structure, and use indexed operations to access individual fields of that data element. To move to the next data element, only the index base pointer needs to be changed, the offsets will then access the subsequent structure.

Finally, there are a few instructions that do not support the extended addressing mode (they support only direct and indexed addressing), so if one must work with a byte not in the direct addressing area, then indexed addressing must be used.

Here is a list of all of the addressing modes that are supported on the HC11 architecture:

Immediate Data is part of the instruction itself. This mode is specified with the use of the prefix "`#`" before the data byte or word. Example: `ldaa #0x80` loads the A register with the hex number 0x80.

Direct Data is located in RAM (within addresses 0x0000 to 0x00ff). One byte is used to specify which RAM location is to be used. Example: `staa 0x80` stores the A register to the memory location 0x0080.

Extended Location of data is specified by a 16-bit address given in the instruction. Example: `staa #0x1000` stores the contents of the A register at memory location 0x1000.

Indexed Location of data is specified by the sum of a 16-bit index register (register X or Y) and an offset value that is part of the instruction. Example: `ldaa 5,X` loads the A register with the memory byte located at the address that is the sum of the value currently in the X register and 5 (decimal). Offsets range in value from 0 to 255.

Inherent Data is "inherent" to the microprocessor and does not require and external memory address. Example: `tab` transfers the contents of the A register to to the B register. No external memory address is required.

Relative Location is specified by an offset value from the address of the instruction currently being executed. Example: `bra 5` causes a branch that skips five bytes ahead in the instruction stream. Relative addressing is only used in branching instructions. offsets range in value from -128 to +127, allowing jumps both forward and backward in the instruction stream.

3.3 Data Types

The HC11 supports a few different "data types," or ways of representing numbers. Most high-level languages (like C) support many data types, such as integers, floating point numbers, strings, and arrays. In assembly language, a programmer is given only "the bits," and must build more complex data types with subroutine libraries.

The HC11 has two data types: 8-bit numbers and 16-bit numbers. This means that there are instructions that process numbers of length eight bits (bytes) and there are instructions that process numbers of length sixteen bits (words).

It's good to keep in mind the range of an eight-bit number versus a sixteen-bit number. An eight-bit number can have 256 different values ($2^8 = 256$), and a sixteen-bit number can have 65536 different values ($2^{16} = 65536$).

3.4 Arithmetic Operations

Microprocessors give the programmer a standard set of arithmetic and logical operations that can be performed upon numeric data.

The HC11 is a particularly nice processor because it provides instructions that work on both eight-bit data values (such as the A or B registers or memory bytes) and sixteen-bit data values (such as the X and Y index registers). Earlier processors provided only eight-bit operations; the programmer had to combine them to get sixteen-bit ones. Also, the multiplication and division instructions are by no means standard amongst 8-bit microprocessors like the HC11.

The HC11 supports the following instructions:

Addition for both 8-bit and 16-bit values.

Subtraction for both 8-bit and 16-bit values.

Multiplication of two 8-bit values to yield a 16-bit result.

Division of two 16-bit values to yield an integer or fractional result.

Increment of both 8-bit and 16-bit values. The increment operation adds one to its operand.

Decrement of both 8-bit and 16-bit values. The decrement operation subtracts one from its operand.

Logical AND for 8-bit values. This instruction performs a bit-wise "and" operation on two pieces of data. The result of an AND operation is 1 if and only if both of its operands are 1. (e.g., `0b11110010` ANDed with `0b11000011` yields `0b11000010`.

Logical OR for 8-bit values. This instruction performs a bit-wise "or" operation on two pieces of data. The result of an OR operation is 1 if either or both of its operands is 1.

Logical Exclusive OR for 8-bit values. The result of an exclusive-OR operation (called "EOR") is 1 if either, but not both, of its inputs are 1.

Arithmetic Shift operations on 8-bit and 16-bit values. The Arithmetic Shift operation moves all the bits in an operand to the left or to the right by one bit position. This is equivalent to a multiplication or division by 2 (respectively) upon the operand.

Rotation operations on 8-bit values. These are similar to the shift operations except that the bit that gets shifted out of the high or low bit position (depending on the direction of the rotate) gets placed in the bit position vacated on the other side of the byte. Example: rotate right (`ROR`) of `0b11011001` produces `0b11101100`.

Bitwise Set and Clear operations on 8-bit values. These operations set or clear bits at specified bit positions within an eight-bit data byte.

Clear operations on 8-bit memory bytes or registers. This instruction is equivalent to writing a zero into the memory location or HC11 register, but does so more quickly.

There are a few more arithmetic instructions not mentioned here, but they are relatively obscure.

3.5 Signed and Unsigned Binary Numbers

There are two methods of representing binary numbers that are commonly used by microprocessors. Using these two methods, the same string of 1's and 0's that comprise a byte or word can represent two different numbers, depending on which method is being used.

The two methods are: *unsigned* binary format and *two's-complement signed* binary format.

The unsigned format is used to represent numbers in the range of 0 to 255 (one byte of data) or 0 to 65535 (one word of data). This is the more simple way of representing data; it's easy to understand because there is a direct translation from the binary digits into the actual numeric value. But, the unsigned format has the limitation that values less than zero cannot be represented.

Here are some unsigned binary numbers and their decimal equivalents:

binary	decimal
`0b0000`	`0`
`0b0001`	`1`
`0b0010`	`2`
`0b0011`	`3`
`0b0100`	`4`
`0b0101`	`5`
`0b0110`	`6`
`0b10011100`	`156`
`0b11100011`	`227`
`0b11111111`	`255`

Signed values are represented using the "two's complement" binary format. In this format, a byte can represent a value from −128 to +127, and a word can represent a number from −32768 to +32767.

The highest bit (most significant, or left-most bit) of the number is used to represent the sign. A "0" in the high bit indicates a positive or zero value, and a "1" in the high bit indicates a negative value.

If the number is positive or zero, then the signed representation is exactly equivalent to the unsigned one. For example, the largest binary number representable in a byte, using the signed format is `0b01111111`. The leading zero is the sign bit, indicating a non-negative number; the seven ones that follow are the significant digits.

If the number is negative, then the following process determines its value: invert the significant digits (change zero's to one's and one's to zero's), and add one. Put a minus sign in front of the number, and that is the equivalent value.

For example, let's figure out the value of the signed number `0b10011011`. We know this is a negative number, since its high bit is one. To find its value, we take the significant digits (`0b0011011`) and invert them, obtaining `0b1100100`. We add one, and obtain `0b1100101`. This value converted to decimal is 101; thus, our original number was equal to −101.

This bizarre method is employed because it has one tremendous property: signed binary numbers can be added together like unsigned ones, and results of standard addition and subtraction processes produce correct signed values.

I won't go through the proof of this because I don't know if it is interesting or not. But let's look at an example, which shows an addition of two signed binary numbers to produce a valid result:

```
   10011011     (-101 decimal)
 + 01110000     ( 112 decimal)
 ----------
(1)00001011     (  11 decimal)
```

Ignoring the carry out of the highest bit position, we can see that performing regular binary addition on the two numbers gives us the correct result. This is important, because then the microprocessor doesn't have to implement different types of addition and subtraction instructions to support both the signed and unsigned data representations.

3.6 Condition Code Register and Conditional Branching

Whenever the HC11 performs any type of arithmetic or logical operation, various *condition codes* are generated in addition to the actual result of the operation. These condition codes indicate if the following events happened:

- The result of the operation was zero.
- The result of the operation overflowed the 8- or 16-bit data word that it was supposed to fit in. This condition is based on interpreting the data operands as two's complement values.
- The result was a negative value. Example: subtracting 50 from 10.
- The result generated a *carry* out of the highest bit position. This happens (for example) when two numbers are added and the result is too large to fit into one byte.

There is a special register in the HC11, called the *condition code register*, or CCR, where this information is kept. Each condition is represented by a one-bit *flag* in the CCR; if the flag is 1, then the condition is true. The CCR has eight flags in all; four more in addition to the four mentioned.

Each flag has a name: the zero flag is called Z; the overflow flag is V, the negative flag is N, and the carry flag is C.

The usefulness of these flags is that programs may branch depending on the value of a particular flag or combination of flags. For example, the following fragment of code will repeatedly decrement the A register until it is zero. This code fragment uses the "branch if not equal to zero" instruction (`bne`) to loop until the A register equals zero.

```
loop:   deca            ; decrement A register
        bne loop        ; if not zero, jump back to "Loop"
        ...             ; program execution continues here
        ...             ;     after A is zero
```

An entire set of these conditional branching instructions allows the programmer to test if the result of an operation was equal to zero, not equal to zero, greater than zero, less than zero, etc.

Some of the conditional branching instructions are designed for testing results of two's complement operations, while others expect to test results of unsigned operations. As mentioned earlier, the same arithmetic operations can be used on both signed and unsigned data. This is true, but the way that one must interpret the condition codes of the result is different. Fortunately, the HC11 branch instructions will perform this interpretation properly, provided the correct instruction is used for the type of data the programmer has in mind.

Here is a list of some of the conditional branching instructions supported by the HC11:

`beq`: **Branch if Equal to Zero** Branch is made if Z flag 1 (indicating a zero result).

`bne`: **Branch if Not Equal to zero:** Branch is made if Z flag is 0 (indicating a non-zero result).

`bcc`: **Branch if Carry is Clear** Branch is made if C flag is 0, indicating that a carry *did not* result from the last operation.

`bcs`: **Branch if Carry is Set** Branch is made if C flag is 1, indicating carry occurred.

`blo`: **Branch if Lower** Branch is made if result of subtraction was less than zero. This instruction works correctly when using *unsigned* data.

`bge`: **Branch if Greater Than or Equal** Branch is made if result of subtraction is greater than or equal to zero. This instruction works correctly only when using *unsigned* data.

Other branching instructions work with signed data and check the proper combination of flags to tell if results are greater or less than zero, etc.

One important thing to remember about branching instructions is that they use the *relative addressing mode*, which means that the destination of a branch is specified by a one-byte offset from the location of the branch instruction. As such, branches may only jump forward or backward a maximum of about 128 bytes from the location of the branch instruction.

If it is necessary to branch to a location further away, the JuMP instruction (`jmp`) should be used, which takes an absolute two-byte address for the destination. The destination of a `jmp` instruction thus may be anywhere in memory.

If necessary, use a conditional branch instruction to jump to a `jmp` instruction that jumps to far-away locations.

3.7 Stack Pointer and Subroutine Calls

All microprocessors support a special type of data structure called the *stack*. A stack stores data in a last-in, first-out (LIFO) method.

To visualize the stack, one may imagine a dishwasher who is washing a sink full of dishes. After each dish is washed, it is placed on top of a pile of already-washed dishes. When the chef removes dishes from the pile, the dish that's removed is the last dish that the dishwasher placed on the pile. In this way, the stack of dishes stores the dishes using a last-in, first-out algorithm.

The stack on the HC11 works the same way. Instead of a stack of dishes, the HC11 stores bytes in a contiguous area of memory. Instead of a dishwasher and a chef, the HC11 uses a special register, called the *stack pointer* or *SP*, to keep track of the location of the stack in memory.

When a number is placed on the stack (called a *stack push*), the number is stored in memory at the current address of the stack pointer. Then the stack pointer is advanced to the next position in memory.

When a number is taken off the stack (called a *stack pull*), the stack pointer is regressed to the last location stored, and then the number at that memory location is retrieved.

The stack has many different uses. One use is temporary storage of data. Suppose there is a number in the A register to be stored and then retrieved a few instructions later. One could push it on the stack (`psha`) to save it, and later pull it off the stack (`pula`) to restore it.

The data in several different registers may be temporarily stored and retrieved in this way. It's important to remember that data goes on and comes off the stack in a particular order. If data is stored with a `psha` and then a `pshb` (push A register, push B register), it must restored with the sequence `pulb`, `pula` (pull B register, pull A register).

The most important use of the stack is involved with *subroutines*. Subroutines are pieces of code that may be "called," or executed, by your main program. In this way, they are like utility routines that your software uses.

For example, suppose a program often has need to execute a delay, simply waiting $\frac{1}{10}$ of a second. Rather than repeatedly writing the code to perform the delay, it can be written just once, as a subroutine. Then, whenever the main code needs execute the delay, it can just call the subroutine.

The key thing about executing a subroutine properly is knowing where to return when it finishes. This is where the stack comes in. When a subroutine is called, the HC11 automatically "pushes" the *return address*—the place to continue after the subroutine is done—onto the stack. Then, it branches to begin executing the subroutine.

When the subroutine finished, the HC11 pulls the return address directly off the stack, and branches to that location.

One may think, "Well, we don't need a stack for this; we could just have one particular location where the return address is stored. We could just look there when returning from a subroutine."

Actually, that is not a bad solution, but using the stack gives us a special power: it enables *nested subroutine calls*. What happens when a subroutine calls a subroutine? If a stack is being used, the second return address simply gets pushed on top of the first, so that the first return address remains intact. In the other method, the first return address would be overwritten and destroyed when the second subroutine call occurred.

It can be seen that advanced computer science ideas like recursion are based in this principle of a stack.

One detail worth mentioning about the stack's implementation on the HC11 is that the stack builds *downwards* in memory. That is, when a number is pushed on the stack, the stack pointer is actually *decremented* to point to the next available memory location. This is somewhat counter-intuitive, but it doesn't matter in how the stack functions.

Since the stack is a dynamic structure, it must be located somewhere in HC11 RAM (read/write memory). As shown in Figure 2, the RAM is located from addresses 0x0000 to 0x00FF. It's customary to initialize the stack at the *top of RAM*—address 0x1ff. Then, as the stack grows, it moves downwards towards location 0x0.

A good way to crash the processor is to repeatedly push a value on to the stack and forget to pull it off. If this mistake is made inside a program loop, all of RAM will easily be filled with garbage. When a subroutine attempts to return to its caller, the return address will be nowhere in sight.

Just remember: each stack push must be matched with a stack pull. Each subroutine call must be matched with a return from subroutine. And don't try to execute too many nested subroutine calls, since the HC11 has only 256 bytes of RAM for stack space.

4 Architecture of the HC11

The HC11 chip includes many features that often must be implemented with hardware external to the microprocessor itself. Some of these features include:

- serial line input and output
- analog to digital converters
- programmable timers
- counters

This section explains how to use these advanced features of the HC11.

4.1 Register Block

The HC11 uses a particular area of memory to interface with the special functions. This area of memory is called the *register block*, and is located from addresses 0x1000 to 0x103f.

The general method of controlling the various features of the chip is by reading and writing data to the different registers in the register block. Since the register block is mapped into memory, the typical HC11 instructions for reading and writing data to any area of memory are used into interact with these registers.

4.2 Block Diagram of HC11

Figure 4 shows a simplified block diagram of the HC11 architecture. A more complete block diagram may be found in the green booklet, *Motorola HC11 Programming Reference Guide.*

When scanning the diagram, notice that some of the ports have arrows running in both directions (ports A, C, and D). These ports are *bidirectional*, meaning that they can be used for either input or output.

Each port contains eight data bits, making it equivalent to one byte of data. *Each data bit is mapped to a physical pin on the microprocessor package.* This means that

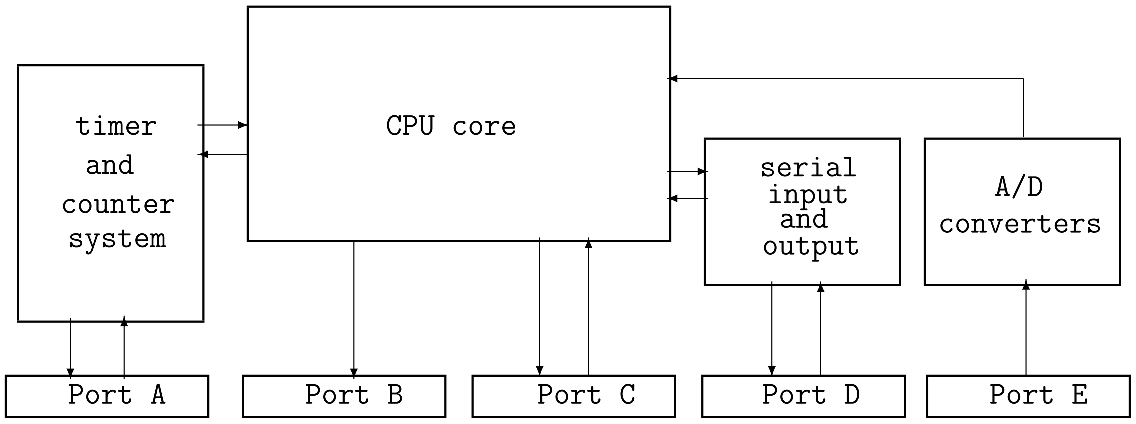

Figure 4: Simplified Block Diagram of HC11

when data is written to a particular output port, *that data appears as voltage levels on the real pins connected to that port.* In this way, the HC11 can interface with external devices, such as the motor chips, or off-board sensor devices.

In many cases, ports may contain a mixture of pins used for either input or output. In other cases, particular pins in a port are dedicated to a specific function.

Following is a brief description of each port on the diagram. The rest of this section explains how to use each port in detail.

Port A. This is a digital, bidirectional port that implements special timer and counter circuitry. The timers can be used to generate waveforms of varying frequencies; the counters can be used to count certain events (like rising edges of signal) on the input lines.

Port B. This is a digital port that may be used for output only. In the Mini Board design, this port is used to control the motors.

Port C. This is a digital, bidirectional port. In the Mini Board design, this port is used for input or output. Its default state is for input (thus it is a digital sensor port).

Port D. This is a bidirectional port dedicated to serial input and output functions. Two of the Port D pins are used in the Mini Board board design for communications with a host computer. The other four pins can be used to implement a high speed link with another Mini Board.

Port E. This is the analog input port. In the Mini Board design, these eight pins are wired to a connector port.

The following section begins the in-depth explanation of these ports with Port B, the motor output port.

4.3 Port B

CCF		SCAN	MULT	CD	CC	CB	CA

Figure 5: Diagram of `ADCTL` Register

Port B is implemented as eight output pins on the HC11. Port B is controlled by a register located at address 0x1004. By writing to this register, the Port B pins are turned on and off.

When the HC11 is operated in expanded mode (that is, an external memory is attached), then Port B becomes the upper eight address bits.

4.4 Port C

Port C is implemented as eight bidirectional pins on the HC11. This port may be configured as an output port and used in a similar fashion to Port B using the `DDRC` (data direction for Port C) register, as detailed in the Motorola literature.

Port C is controlled by reading the value of the location 0x1003. Whatever input signals that are present on the Port C lines are then “latched” into Port C during the read operation.

The Port C register is referred to by the name `PORTC`.

4.5 Port E

Port E is the analog input port. This port is controlled by several registers, and may be configured in a few different ways.

In order to use the analog-to-digital (A/D) converter, the A/D system must first be powered-up (its default state is off).

The System Configuration Options register (`OPTION`) is used to turn on the A/D system. Bit 7 of this register must be set to “1” in order to turn on the A/D system:

```
LDAA    #0b10000000    ; bit 7 set to 1
STAA    0x1039         ; location of OPTION register
```

The A/D system is actually configured as two banks of four channels each. In one of its operating modes, it repeatedly samples values from *either* of these four-channel banks.

In another operating mode, the A/D system will repeatedly sample *only one of the eight input channels*. Because sampling takes a finite amount of time (about 17 μsec), this is a useful mode if one wants to look at one channel very closely.

The A/D Control Status Register (`ADCTL`) is used to select these different modes. Figure 5 is a pictorial of the `ADCTL` register.

Bit 7 of the `ADCTL` register, `CCF`, is the Conversions Complete Flag. It is set to “1” when the A/D system has finished converting a set of four values. It is important to wait for this flag to be set only when the *the mode is changed* of the A/D system. Then, the `CCF` will be set to zero, and one should explicitly wait for it to turn to one before trusting the converted values.

Bit 5 is `SCAN`, the Continuous Scan Control. If this bit is set to one, the A/D system will repeatedly convert values. If it is set to zero, the A/D will convert four values and

CD	CC	CB	CA	**Channel Signal**	**Result in `ADRx` if `MULT = 1`**
0	0	0	0	`AD0 port E bit 0`	`ADR1`
0	0	0	1	`AD1 port E bit 1`	`ADR2`
0	0	1	0	`AD2 port E bit 2`	`ADR3`
0	0	1	1	`AD3 port E bit 3`	`ADR4`
0	1	0	0	`AD4 port E bit 4`	`ADR1`
0	1	0	1	`AD5 port E bit 5`	`ADR2`
0	1	1	0	`AD6 port E bit 6`	`ADR3`
0	1	1	1	`AD7 port E bit 7`	`ADR4`

Figure 6: Settings of A/D Channel Select Bits

stop. For typical usage, it is probably simpler to set it to one and expect the A/D system to continuously convert values.

Bit 4 is `MULT`, the Multiple Channel/Single Channel Control. If this bit is set to one, the A/D will convert *banks of four channels*. If it is set to zero, the A/D will convert *one channel only*.

Bits 3 to 0 select the channel(s) to be converted. The results of the A/D conversion appear in four other registers, called `ADR1`, `ADR2`, `ADR3`, and `ADR4`.

Figure 6 is a table that maps the settings of the channel select bits to the readings that appear in the `ADRx` registers when `MULT` equals one. If `MULT` is zero, then the channel select bits select the channel that gets converted into all four `ADRx` registers.

`ADCTL` is located at address 0x1030; `ADR1` through `ADR4` are located at addresses 0x1031 through 0x1034.

Contents