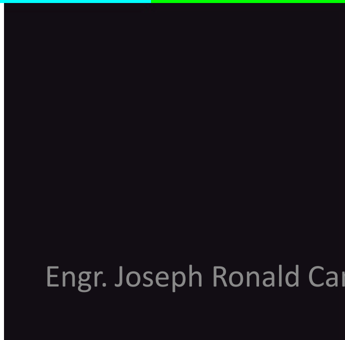
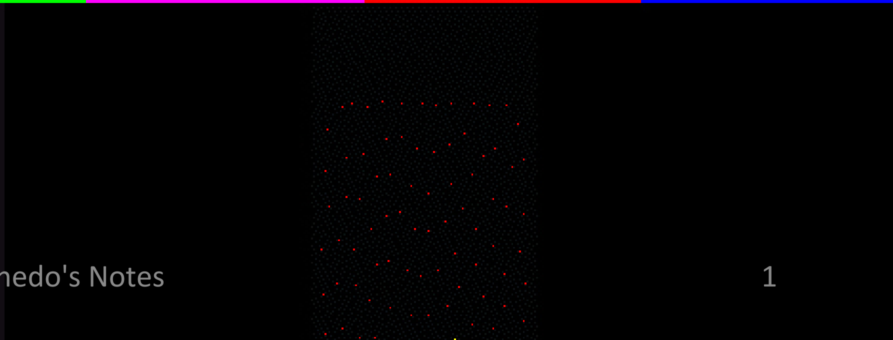



Assembly Language

by:

Joseph Ronald Cañedo

Instructor

Register

- Registers are the internal (working) storage for the processor.
- The number of registers varies significantly among processor architectures.
- Typically, the processor will have one or more accumulators. These are registers that may have arithmetic operations performed on them.
- In some architectures, all the registers function as accumulators, whereas in others, some registers are dedicated for storage only and have limited functionality.

Register

- Some processors have index registers that can function as pointers into the memory space. In some architectures, all general-purpose registers can act as index registers; in others, dedicated index registers exist.
- All processors have a program counter (also known as an instruction pointer) that tracks the location in memory of the next instruction to be fetched and executed. All processors have a status register (also known as a condition-code register, or CCR) that consists of various status bits (flags) that reflect the current operational state. Such flags might indicate whether the result of the last operation was zero or negative, whether a carry occurred, if an interrupt is being serviced, etc.

Machine Code

- Everything that a processor deals with is expressed as numbers in memory. That applies to data, and to software as well. Your compiled C program is converted to a sequence of numbers that is meaningful to the processor as a sequence of instructions.
- In this case, a 68HC11:

```
86 41 B7 01 00 BD 02 00
```

- This is known as **machine code**, and each numeric instruction is called an **opcode**.

Machine Code

- The 68HC11 instruction based on the previous machine code sequence:

```
Machine code    Assembly     Comment

86 41           LDAA #$41    ; anything after a semi-colon

B7 01 00        STAA $0100   ; is a comment

BD 02 00        JSR $0200    ;
```

Machine Code

Comparison of some different assembly languages

Processor	Instruction
Motorola 6800/68HC11	`LDAA #$41`
Intel x86	`moval, 41h`
Motorola 680x0	`move.b #$41,D0`
PIC16xx	`movlw 0x41`
Motorola 56000	`move #$0041,A`
Intel 80960	`lda 0x41,r4`

Signed Numbers

- How signed numbers are represented within the machine.
- Negative and positive numbers can be represented in binary (or hex) number systems by the state of the most significant bit.
- If the most significant bit is set, this indicates that the number is negative.

Signed Numbers

- An 8-bit accumulator can have (in decimal) 0 to 255 (unsigned) or -128 to +127 (signed).
- Given a positive number, we use two's complement to convert this to its negative equivalent.
- Taking the two's complement of a number is done by inverting the bits (a one's compliment) and adding one to the result.
- So, 0xFF is 1111 1111 (in binary) and is therefore negative. 0xFF is used to represent -1, 0xFE is -2, etc., all the way to 0x80, which is -128 (decimal).
- 16-bit, 32-bit, and 64-bit signed numbers work in the same way. The most significant bit represents the sign, and the remaining bits constitute the number

Addressing Modes

The different ways in which an instruction can reference a register or memory location are known as the **addressing modes** of the processor.

- Inherent
  - The instruction deals purely with registers. CLRB clears the B accumulator, for example.
- Immediate/Literal
  - The instruction has a literal number as an operand.
- Direct
  - The instruction accesses a memory location, specified by a short address. In other words, direct addressing provides access to a subset of the total address space.

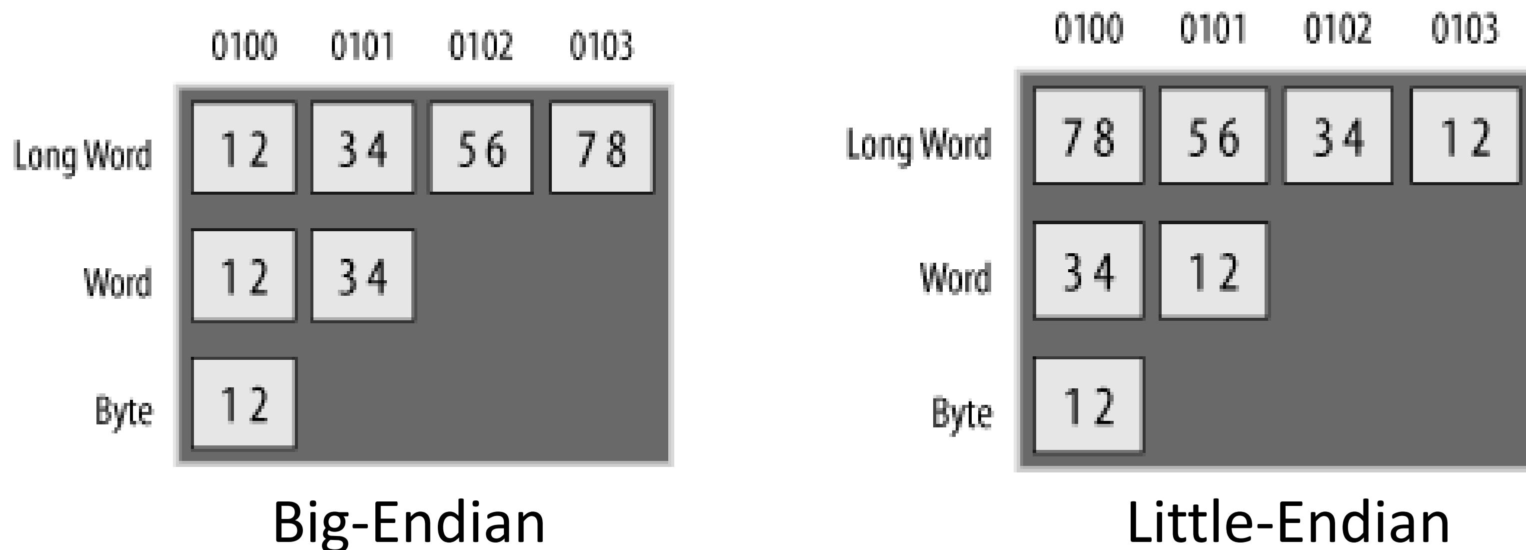
Addressing Modes

- Extended
  - The instruction accesses a memory location, specified by the full address.
- Indexed
  - The instruction uses the contents of a register as a pointer into memory.
- Relative
  - An offset is specified as part of the addressing. A branch instruction uses relative addressing to add (or subtract) a value from the program counter.

Addressing Modes

Big-Endian and Little-Endian Addressing

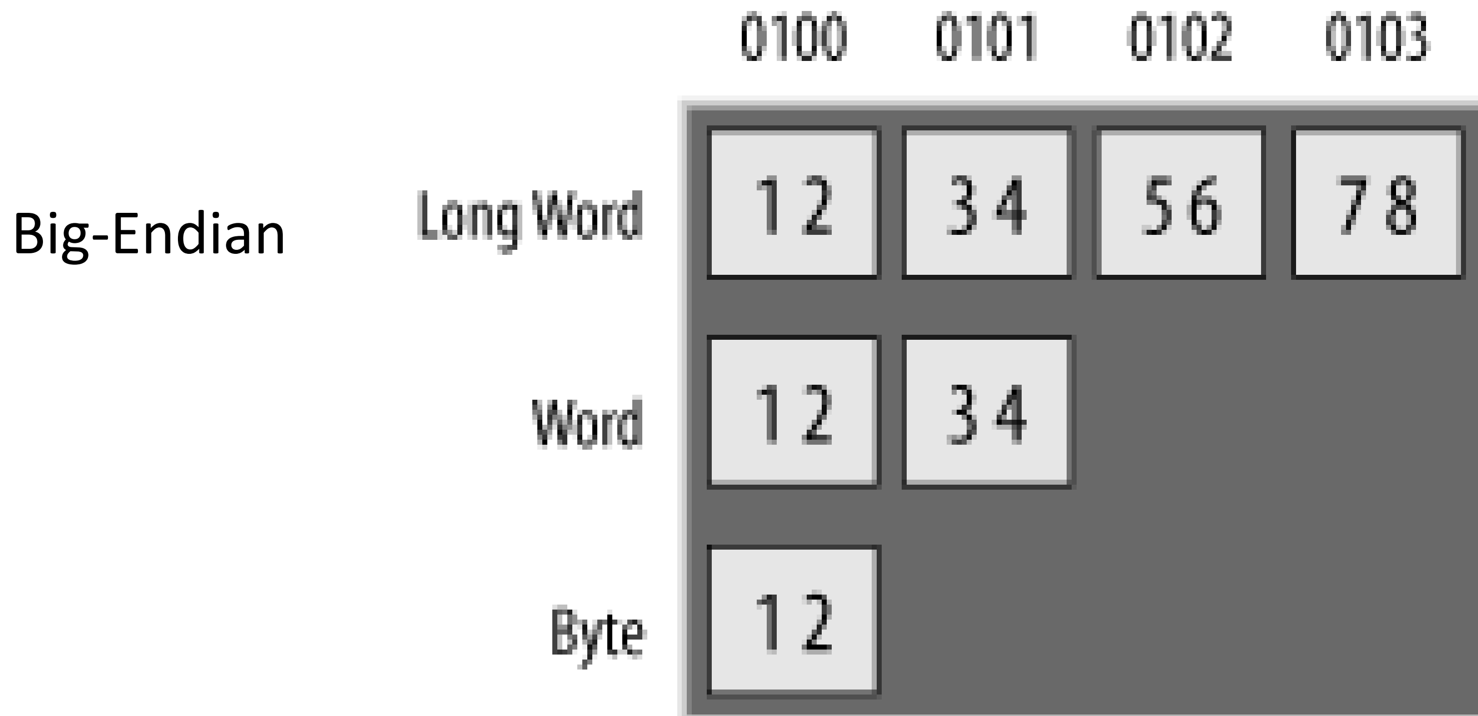
– Microprocessors are either big endian or little endian in their architecture. This refers to the way in which the processor stores data (16 bits or greater) to memory.



Big-Endian Little-Endian

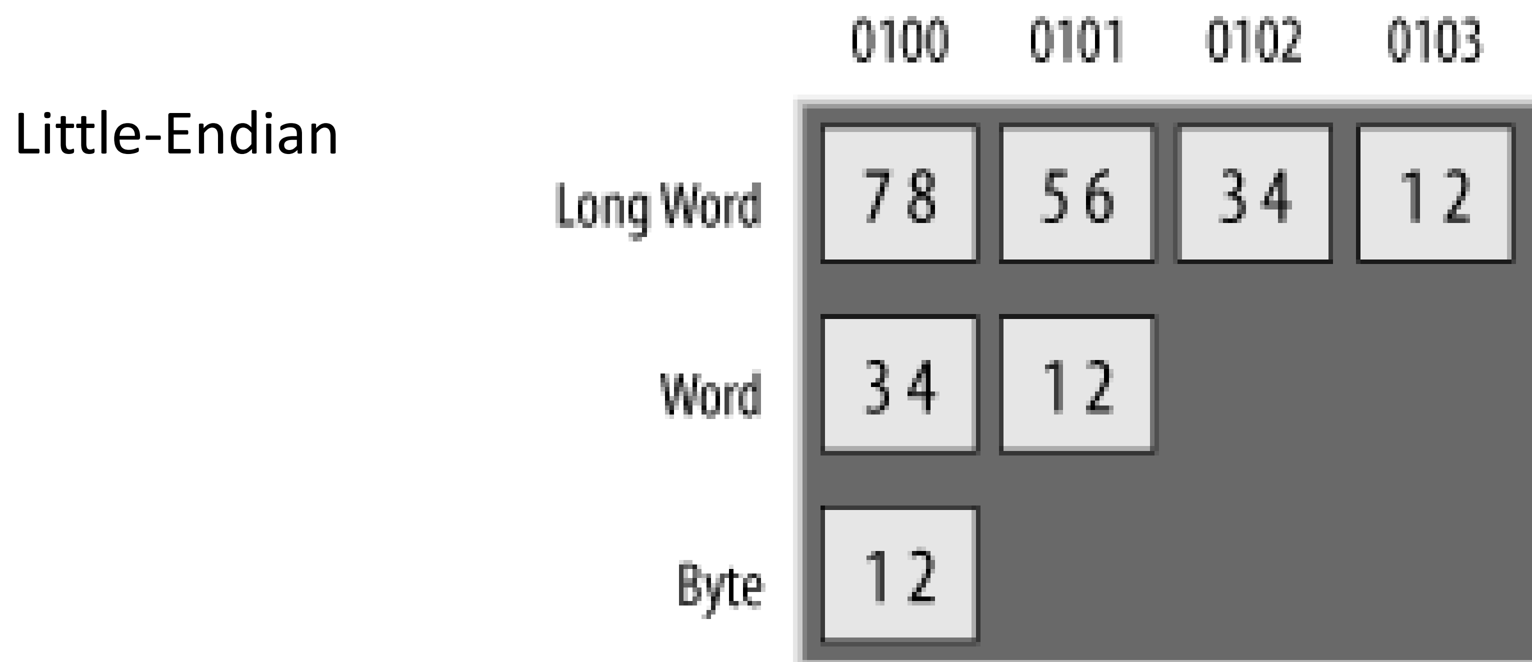
Addressing Modes

A big-endian processor stores the most significant byte at the least significant address.



Addressing Modes

A little-endian processor stores the most significant byte at the most significant address.



Coding in Assembly

- Assembly-language programming is not difficult, particularly with the smaller microcontroller and microprocessor architectures.
- The main difficulty many people seem to have is knowing how to tackle a given programming problem.
- As with any other language, there are always many ways in which a program may be written.
- There is rarely a single "right way" (although there may be a "most efficient way").
- It is simply a matter of breaking down the problem into a series of tiny steps

Disassembly

- Disassembly is the conversion from a sequence of machine code back to the mnemonics that represent that code.
- This is done when we have a machine code program (perhaps written by someone else) and we want to know what it does and how it works.

Position-Independent Code

- This method of programming avoids the use of absolute addresses (except where appropriate).
- It means that rather than jumping to the absolute address of our subroutine, we branch to the subroutine relative to the program counter's contents.

Loops

- It is often useful to repeat an instruction or series of instructions.
- Using branch instructions allows us to redirect the program counter and hence control the flow of the program.

Masking

- It is often useful to examine the state of certain bits (such as the status of a peripheral chip or a flag).
- We are interested in the state of a given bit, but the state of the other bits is unknown (and unimportant to us).

Indexed Addressing

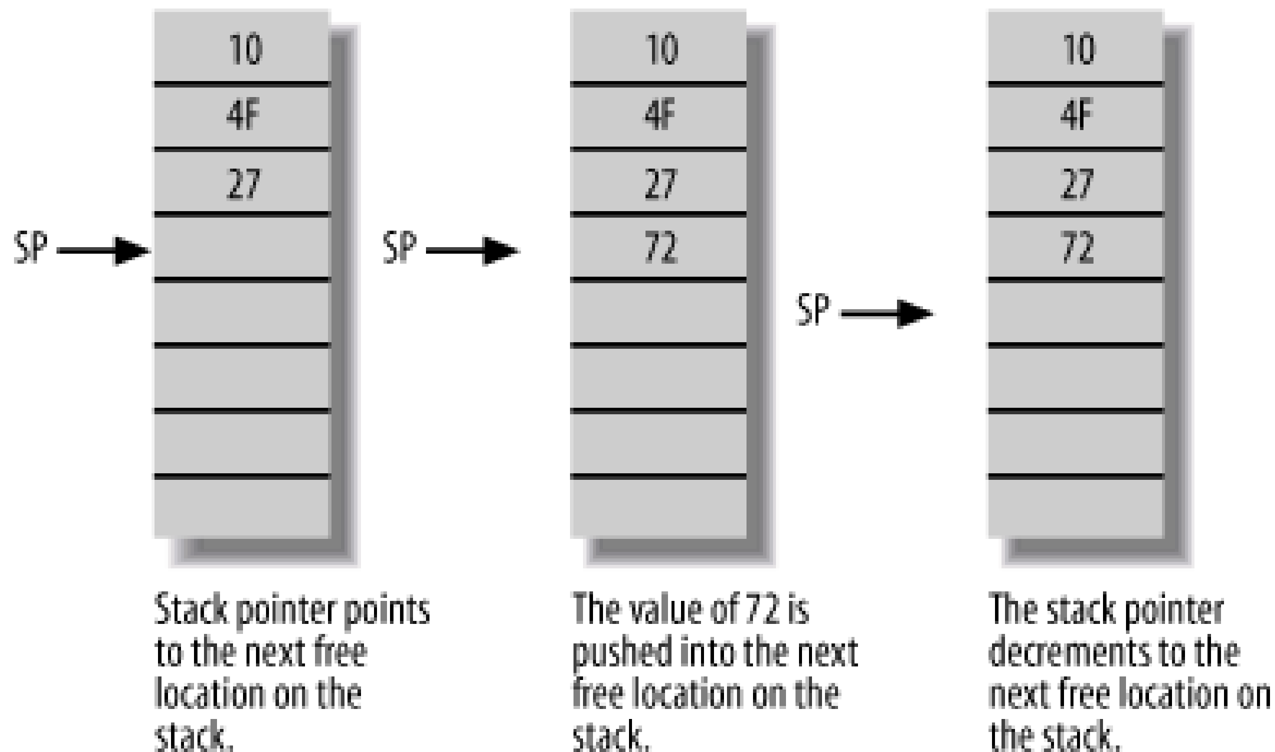
- It is often useful to use a pointer to reference a section of memory.
- The 68HC11 has two index registers, X and Y. These are the equivalent of pointers in C.

Stacks

- Many processors implement one or more stacks, which serve as temporary storage in external memory. The processor can push a value from a register on the stack to preserve it for later use. The processor retrieves this value by popping from the stack back into a register. In some processor architectures, popping is also known as pulling.
- Most processors have a special register known as a stack pointer, which references the next free location on the stack. Some processors implement more than one stack and so have more than one stack pointer. Most stacks grow down through memory. (Some processors have stacks that grow up as the stack is filled.) When the processor pushes or pops a value to or from the stack, the stack pointer automatically decrements or increments to point to the next free location.

Stacks

- Shows the steps that occur when the content of a register (in this case, 72) is pushed onto the stack.



Timing of Instructions

- On some processors, particularly CISC chips, different instructions take varying lengths of time to complete.
- Sometimes it is important to know how long a given section of code will take to execute, particularly if the software is interacting with a time-critical external system.
- The execution time can be calculated by looking up the number of cycles each instruction takes and adding them together.