Lecture #2 – January 30, 2004

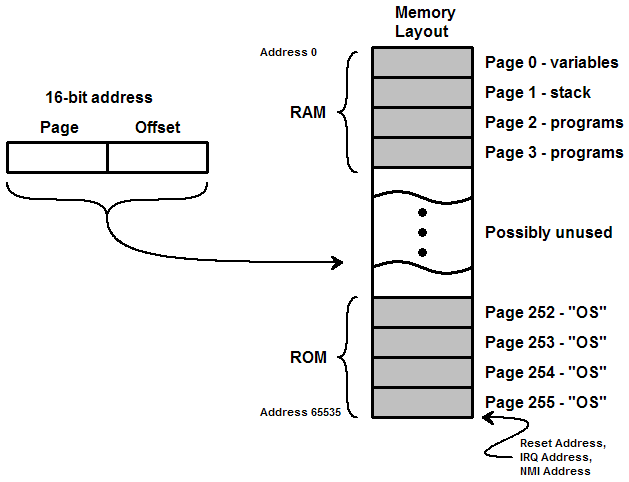
The 6502 Architecture

In order to understand the more modern computer architectures, it is helpful to examine an older but quite successful processor architecture, the MOS-6502. The 6502 was released in 1975, and was the main processor in the KIM-1, Apple][, AIM-65, and other microcomputers from the late 1970s into the 1980s. The 6502 is strictly an 8-bit architecture, with a single 8-bit ***accumulator register***, and can access a maximum of 64K bytes of random access memory. Memory addresses are exactly 16 bits in length, because $2^{16} = 65536 = 64K$. Valid addresses range from 0 up to 65535. The 6502 is called a ***one address*** machine because instructions can reference at most one address at a time, and most of those instructions use the accumulator implicitly (without the programmer specifying it directly).

As the 6502 is a byte-oriented machine, ***absolute addresses*** are two bytes in length, with the upper byte of the address representing the ***page*** of memory being accessed, and the lower byte the ***offset*** within that page. The entire 64K address space is treated as 256 pages of 256 bytes per page (256×256=65536). Instructions that reference memory typically use two-byte addresses to access any byte anywhere in the address space, but special one-byte addresses are also used to address the very first page by assuming the upper byte of the address to be zero (***page 0***). Page 0 is used to contain frequently referenced variables in order to save space in the program code. Page 1 is hardwired by the processor to be used as the stack, and is not used by programmers for any other purpose. The code for user programs often starts in page 2.

At the high end of memory, the last six bytes of the last page (page 255) of memory are used by the hardware to contain special addresses. Two of these addresses are for ***interrupts*** (called IRQ and NMI, which we will discuss later) and one is for the ***reset*** function. When the processor is first turned on and/or reset, the hardware uses the reset address to know where to start running code. Those last three addresses (six bytes) in memory, along with the program code to which they jump, must always contain valid information, even when the machine comes up from a "dead" state. Those areas of memory must be occupied by ROM (read only memory). ROM, unlike most RAM, does not lose its contents when the power is off.

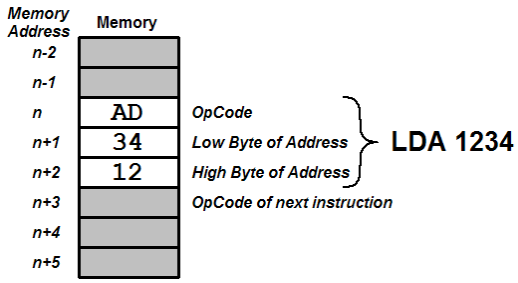
Therefore, in 6502-based systems the lower addresses (particularly pages 0 and 1) must contain RAM and the upper addresses (particularly page 255) must contain ROM. ROM code usually contains whatever passes for an ***operating system*** on that computer; often a very primitive OS. How much RAM and ROM is present beyond the basic requirements is a decision of the system designer, but many systems have "holes" in the address space where neither RAM nor ROM are present. Storing a value into one of these holes does nothing useful. Retrieving a value from one of these holes often results in garbage values being returned. (We have not yet talked about how input or output tasks are performed, but some holes in memory can be filled with special devices called ***ports***, which allow those functions.)



The 6502 has one main 8-bit ***accumulator register*** and two auxiliary 8-bit ***index registers*** called `X` and `Y`. The index registers are often used as array subscripts when referencing memory.

Instructions are one, two, or three bytes in length. The first byte in memory of any instruction is a number called the operation code, or ***OpCode***, which tells the processor what kind of instruction is being loaded. Once the processor has the OpCode it knows whether or not it must fetch the next byte or the next two bytes from memory in order to complete the instruction, or whether it has all the information it needs already. Instructions are laid out linearly in memory one after the other; the first byte after any instruction contains the OpCode of the next instruction to be executed.

For example, the instruction `LDA 1234` means load the accumulator from the byte in memory at absolute address 1234 (expressed in base 16, or hexadecimal). The numerical code for `LDA` when used with a complete 16-bit address is the hexadecimal value `AD`, so the instruction will be laid out in memory as follows:



The order in which addresses (or multibyte numbers) are split across individual bytes in memory is called the ***endian***. In ***little endian*** the lowest byte of a number is stored at the lowest address in memory and the highest byte is at the highest address. In ***big endian*** the order is reversed, with the highest byte of the number at the lowest address. As you can see, the 6502 processor stores addresses in little endian format. All Intel processors (the 8088, the 486, the Pentium, etc.) use little endian format. Motorola processors (such as the 68000 used in the original Mac) use big endian format.

Little endian format makes the most sense for the 6502 because instructions that reference bytes in page 0 use only the offset into page 0 and can omit the page number. For example, the `LDA 34` instruction references address 0034 (page 0, offset 34), and is only two bytes in length instead of three. The 6502 tells the difference between similar instructions by their OpCode numbers: `LDA` from a 16-bit absolute address has OpCode `AD`, while `LDA` from page 0 has OpCode `A5`. In both cases the byte after the OpCode is the low-order byte of the address (little endian). Page 0 instructions stop there, but if the instruction uses a 16-bit address the processor must make one more fetch to get the high byte (page number) of the address before it can execute the instruction.

In a high-level language we might write a statement such as **`C = A + B`** to add two 32-bit (4 byte) numbers together. For an 8-bit machine such as the 6502 this means adding only one byte of each pair of operands at a time, storing the result in the correct byte of the answer, and repeating the process a total of four times.

If we know that the storage for variable `A` will start at absolute memory location 1234 (hexadecimal), then all four bytes will occupy locations 1234, 1235, 1236, and 1237. Similarly, variable `B` will occupy locations 1238, 1239, 123A, and 123B, and variable `C` will occupy locations 123C, 123D, 123E, and 123F. So, the 6502 code to add the two numbers together in little endian format will be:

```
CLC             Clear carry bit
LDA  1234       Load lowest byte of A
ADC  1238       Add lowest byte of B
STA  123C       Store into lowest byte of C
LDA  1235       Load next byte of A
ADC  1239       Add next byte of B
STA  123D       Store into next byte of C
LDA  1236       Load next byte of A
ADC  123A       Add next byte of B
STA  123E       Store into next byte of C
LDA  1237       Load highest byte of A
ADC  123B       Add highest byte of B
STA  123F       Store into highest byte of C
```

There is a special bit called the ***carry bit*** which allows multibyte additions such as these; the `ADC` instruction says "add with carry" so that a carry out of the sum of one pair of bytes is added to the sum of the next pair. The carry bit must be zero (no carry) at the start of the process; hence the inclusion of the `CLC` (clear carry) instruction.

While it will work, it is pretty terrible code. It is hard to maintain and hard to understand. Most assemblers accept a ***pseudo-instruction*** to define symbols, so we can use more meaningful addresses than pure numbers:

```
A    EQU  1234
B    EQU  1238
C    EQU  123C
```

The `EQU` symbol looks like an OpCode, but it is strictly an instruction to the assembler to associate the value 1234 with symbol `A`. In most high level languages this would be considered a constant. Our code to add two multibyte numbers together is then rewritten as the following:

```
CLC             Clear carry bit
-------------------------------------------
LDA  A
ADC  B          Add lowest bytes
STA  C
-------------------------------------------
LDA  A+1
ADC  B+1        Add next bytes
STA  C+1
-------------------------------------------
LDA  A+2
ADC  B+2        Add next bytes
STA  C+2
-------------------------------------------
LDA  A+3
ADC  B+3        Add highest bytes
STA  C+3
```

The code is still not perfect, but it is a lot easier to tell what is going on than before. An instruction such as `LDA A+1` is evaluated *by the assembler* to the same machine code as in the original example: since `A` is defined to be equal to 1234, then the instruction `LDA A+1` is converted by the assembler into `LDA 1235`. Note that this happens at *assembly time*, not at run time. (The instruction `LDA A` is really the same as `LDA A+0`, but we would rarely write it that way.)

Now consider what happens when we wish to change the code to add 8-byte numbers instead of 4-byte (or any other size). We must change the addresses in the `EQU` pseudo-instructions, and we must also include enough `LDA–ADC–STA` groups to match the overall number of bytes being added. This is dreadfully wasteful of program space; with only 64K bytes of memory every byte saved is a benefit.

If, instead, you think of the variables `A`, `B`, and `C` as zero-based arrays of four bytes each, then the process that we really want to implement is similar to the following high level code (which ignores and omits the proper handling of any carries):

```
X := 0 ;
Repeat
     C[X] := A[X] + B[X] ;
     X := X + 1 ;
Until X = 4 ;
```

Changing the lengths of the array variables requires only that we change the 4 in the loop counter to the new length of the array. By defining a symbol `SIZE` (initially equal to 4) and using `SIZE` wherever the array size is needed, we change the precision of our multibyte numbers by editing a *single* definition and reassembling the program.

In the 6502, the index registers X and Y are used to simulate the actions of an array. For example, the instruction LDA A,X says to add the starting address of array A to the value in the X register, and use the result as the final address of the byte to load into the accumulator. This happens at *run time*, not at assembly time.

The X register must be initialized to zero, and must be compared to the index value which points one byte beyond the end of the arrays in order to know when to stop the loop. In both cases the LDX instruction (load X register) and CPX instruction (compare X register) use ***immediate values*** instead of addresses; the purpose of the # symbol in the program text is to indicate that the value is a constant, not an address.

The result of the CPX instruction is reflected in a ***status bit*** which is tested in the next instruction, which branches (jumps) to a location in the program if the test is met (i.e., if the contents of register X and the constant SIZE are not equal in value). In the 6502, branches cannot go to an arbitrary address in memory. Instead, the distance between the instruction and the target of the branch is computed by the assembler, and the difference is stored as one byte in the branch instruction. With one byte for this difference, the most distant destination can be no more than about ±127 bytes from the point of the branch. In our final code below, the destination is the label called Loop, and the assembler determines if it is within range of the branch instruction (and will generate an error message if it is not). The final code is as follows:

```
SIZE EQU  4          Change this to set array size
A    EQU  1234       Starting address of variables
B    EQU  A+SIZE     B is SIZE bytes later than A
C    EQU  B+SIZE     C is SIZE bytes later than B

     …               other stuff goes here

     CLC             Clear carry bit
     LDX  #0         Set X register to zero
Loop LDA  A,X        Load accumulator from A[X]
     ADC  B,X        Add accumulator from B[X]
     STA  C,X        Store accumulator into C[X]
     INX             Increment X (e.g., X++)
     CPX  #SIZE      Compare X register to SIZE
     BNE  Loop       Branch to Loop if not equal
```

The final code is pretty complicated, but it is very general. Changing our numbers A, B, and C from four bytes each to eight bytes each only requires that we change the SIZE constant and reassemble the program. While the number of bytes in the data area will change as SIZE changes, the executable portion of the code stays exactly the same size (17 bytes). Unfortunately, on the 6502 that's as good as we can do.

We will see with Intel and ARM chips that these tasks are simplified by using more (and larger) registers than those available on the 6502. By studying the 6502 we have introduced a number of core concepts that we will visit again in other architectures.