Solutions to Optional questions 2.5 in Tutorial #2

2.5 Program Counter related Addressing Modes

Note to students: *Some important concepts that these question will bring across are:*
1. *Understand how PC relative addressing can support position independent code.*
2. *Understand how the modification of PC contents affects program flow.*

Complete the mnemonics M1 to M4 in Fig. 2.5. Except for M1, where absolute addressing is used, you should give solutions for M2 to M4 that support **position-independent code**.

Suggested solution:

Address	Mnemonics or Hex Data	Comments
:		; Other parts of the program
0x020	MOV PC,#0x030	; M1 – Absolute Jump to instruction at address 0x030
:		; Other parts of the program
0x030	ADD R0,[PC+0x02E]	; M2 – Add constant C1 to R0
:		; Other parts of the program
0x05D	MOV R0,PC	; M3 – Move start address of next instruction into R0.
0x05E	MOV PC,R0	; M4 – Create infinite loop to prevent execution beyond code space
0x060	0x005	; C1 – Data constant stored in code memory
:		

Figure 2.5 – Various VIP mnemonics and their respective start addresses in code memory

Comments on solutions:

M1 – By loading the absolute address of 0x030 into the PC, the next instruction to be executed will be at this address. This is effectively an implementation of the absolute jump instruction.

M2 – In order to make the code position independent, we must assume that we have no knowledge of where the actual address the data we want to access resides. We can only access the data with the knowledge of its offset from the current PC position. As such, we have to employ PC relative addressing. Position independent program (i.e. consisting of both executable code and its associated memory data) can be shifted anywhere in memory and still execute correctly.
Assuming the data is stored at a higher address, the offset is computed by subtracting the address where the data is stored with the start address of the instruction immediately following the ADD instruction. Since the ADD R0,[PC+offset] instruction is **2**-word long, the correct offset is given by (0x060-(0x030 +**2**)) = 0x02E.

M3 – We can retrieve the start address of the next instruction by moving the content of the PC into register R0. At the execution of the MOV R0,PC instruction, the PC would have advanced by 1 after fetching the instruction. Since the MOV R0,PC instruction is 1-word long, the value in the PC is actually the start address of the next instruction.

M4 – In order to get an instruction to execute in an infinite loop, it has to jump back to itself. Since we have instruction M3 load the start address of the next instruction into R0, we can make the next instruction jump back to itself by making it load the value in R0 into the PC. Both instruction M3 and M4 are position independent since no knowledge of the actual address where this instructions were stored is actually employed.