University of Windsor

ELECTRICAL AND COMPUTER ENGINEERING
ELEC3270: MICROPROCESSORS
ASSIGNMENT 8: DECIMAL CONVERSIONS

OBJECTIVE: To understand how to convert to decimal numbers.

SECTIONS:

A.	Binary Coded Decimal (BCD)	(10%)
B.	Decimal Conversion using Division	(30%)
C.	Efficient Decimal Conversion (Two Digits)	(30%)
D.	Efficient Decimal Conversion (Four Digits)	(30%)

A. BINARY CODED DECIMAL (BCD)

Although semiconductor based CPUs operate in binary (base 2), it is often that we have to see their results in base 10 since we have been trained our whole lives on processing decimal digits. However, it is not easy to move between these two domains. If you recall, this process usually requires a number of divisions and remainder operations which can be quite costly as some low cost CPUs don't necessarily support those operations. To mitigate this problem, designers of these CPUs utilized an addition only approach to deliver decimal output in BCD with minimal computational cost. On the 68HC11, when any "ADD" type instruction (the ones that modify the H CCR flag) is **followed by** the "DAA" instruction, the separate 4-bit nibble result is corrected by adding 6 to either of them if their individual sum and carry is above 9 (A-F) therefore ensuring they remain BCD compliant; **it is not a conversion instruction but a correction one**. Each time an addition BCD addition is to be performed, the DAA instruction must follow to correct that addition. For this part of the assignment you will modify the "**SID_8a.asm**" file to output values from 0000 to 9999 (10000 lines) to the LCD in decimal. The code simply starts at 0, sends the value to the LCD using the PrtHex2 subroutine (prints the two-digit hexadecimal value of ACCA; remember that BCD is a subset of HEX), adds one to 1, and does it again until the value resets back to 0. In its original state however, it will output values from 0000 to FFFF (65536 lines) in hexadecimal; you must modify it by adding in the appropriate correction instructions. The BCD domain is highly inefficient as over 84% of its representation space is invalid, but it is easier to isolate the individual decimal digits and output them than doing a full conversion. It is suggested you understand how the PrtHex1 and PrtHex2 subroutines work since they used heavily in this assignment and others as well.

B. DECIMAL CONVERSION USING DIVISION

The method shown in part A only applies to increasing decimal values, however should the values be relatively random, a more traditional approach is required. The classic binary to decimal conversion process is quite simple: Set *Vx* to 0 for all *x*'s, set *x* to 0, and set *y* equal to the input. Divide *y* by 10, save the remainder in *Vx*, increment *x*, and *y* becomes the quotient. Repeat the last step until *y* is zero. We then output *Vx* from *x*-1 until it reaches 0 inclusively. We have to store the remainder values in *Vx* as we output our result from left to right, not right to left. For this part of the assignment you need to add code to the function "PrtDec2" (print decimal 2 bytes) in "**SID_8b.asm**" that will perform this process. It isn't necessary to create a loop as the maximum input given is 9999. The IDIV instruction (integer division on the 68HC11) divides ACCD by INDX, and stores the resulting quotient in INDX, and the remainder in ACCD. Immediately after this instruction, you can store the remainder into *Vx* using STAB (the remainder is between 0 and 9 so we can ignore the upper ACCA part as it will always be 0) and then use the XGDX to swap ACCD and INDX to prepare for the next division. Prior to the IDIV instruction, you will need to load INDX with 10 (the divisor) each time. It isn't necessary to check to see if the quotient is 0 as this will take longer and won't really speed anything up. Once you have computed *Vx*, you can output each value, in order, by calling the PrtHex1 subroutine which prints the ASCII value of the bottom 4-bits of ACCA.

C. EFFICIENT DECIMAL CONVERSION (TWO DIGITS)

The method in part B does work, but it can consume more time if the number of decimal digits gets larger. This

solution is what just about anyone would use, however given that resources in electronics are costly, a better engineered approach is available which takes advantage of the relationships we have already explored between decimal and BCD values. This approach is usually known as "add 3 and shift". See the table below as an example of this approach where we convert 255 into BCD which is two hexadecimal bytes: 02 and 55. 255 goes into the WORK register and the BCD digits (BCD-x) are cleared initially. The BCD column is the combined BCD-x digits in decimal for easier readability.

BCD	BCD-2	BCD-1	BCD-0	WORK	Details
000	0000	0000	0000	11111111	No digit is 5 or above, continue with left shift
001	0000	0000	0001	11111110	No digit is 5 or above, continue with left shift
003	0000	0000	0011	11111100	No digit is 5 or above, continue with left shift
007	0000	0000	0111	11111000	BCD-0 is 7, add 3 to it
	0000	0000	1010	11111000	No other digit is 5 or above, continue with left shift
015	0000	0001	0101	11110000	BCD-0 is 5, add 3 to it
	0000	0001	1000	11110000	No other digit is 5 or above, continue with left shift
031	0000	0011	0001	11100000	No digit is 5 or above, continue with left shift
063	0000	0110	0011	11000000	BCD-1 is 6, add 3 to it
	0000	1001	0011	11000000	No other digit is 5 or above, continue with left shift
127	0001	0010	0111	10000000	BCD-0 is 7, add 3 to it
	0001	0010	1010	10000000	No other digit is 5 or above, continue with left shift
255	0010	0101	0101	00000000	Conversion complete, results in BCD digits

The steps are simple: check each BCD digit, if the value is 5 or above add 3 to it. Next, treat BCD-2, BCD-1, BCD-0, and WORK as a single value and do a left shift (which is a multiply by 2). The upper bit of WORK shifts into BCD-0 and so on; the incoming bit into WORK isn't important, but in this example it is set to 0. These operations are repeated for the number of bits in WORK, 8 in this case. When done, the BCD-x values contain the decimal digits. This is essentially a bit shifter with a correction after each shift to ensure BCD compliance. It is important to note that the order in which each BCD-x digit is checked for 5 and above is not important since there will never be a carry over to the next digit. Recall the DAA instruction checks if each nibble is 10 or above and if so, it adds 6 to either. The 5 and 3 is just a multiple of 2^x of the 10 and 6 approach the DAA instruction uses, but by using 5 and 3 the corrections order is not important and it avoid the carry propagating issues. This algorithm works very well in a hardware circuit as it can be easily parallelised.
It shouldn't be too difficult to code this in the 68HC11, however the inspection of each of the BCD digits will be a little tedious (bit shifts, etc.). Fortunately we can take advantage of the existing DAA instruction, as it does the same process as the 5 and 3 method, but we will have to perform the operation in order from BCD-0 and up. The H flag in the CCR is only altered by certain addition instructions so a logical shift left can't be used. But we can multiple by two by simply adding numbers to themselves. The BCD correction is done simply by issuing the DAA instruction right after the addition. So for the 68HC11 the process is quite simple: Take the WORK value and multiply it by 2 (either with a left shift or ADD); the resulting carry will be preserved in the carry flag. We then add the BCD digits to themselves (from low to high) using any "add-with-carry" operation (so that the carry from WORK is included in the sum). We do this 8 times since WORK is 8 bits. The resulting BCD digits can be easily printed using the supplied PrtHex2 subroutine.
Your task for this part is to complete the "PrtDec1" subroutine (**one**-byte conversion or two-decimal digit) in "**SID_8c.asm**"; follow the hints. The main loop will cycle through a "table" of 100 one-byte values in the code printing out their hexadecimal and decimal value.

D. Efficient Decimal Conversion (Four Digits)

Since the algorithm discussed in part C is easily scalable, this part will require you to expand the algorithm into the "PrtDec2" subroutine (**two**-byte conversion or four-decimal digit); WORK and BCD are now 16-bits. Modify the "**SID_8d.asm**" code; again, follow the hints. The main loop for this part will print out a "table" of 100 two-byte values in both hexadecimal and decimal.
ZIP all your ASM files into "SID_8.zip" and upload it to Blackboard.