EEAP 282

TABLE DRIVEN CRC CALCULATION:

Abstract
You wrote a program which computed the CRC for Lab #2. However, that is a very slow, inefficient way of calculating CRCs. A much faster algorithm uses table lookup.

Mathematical Background

If A, B and C are 16-bit quantities and we use the + to indicate the bitwise exclusive-or operation, then we know that A + B = B + A and (A + B) + C = A + (B + C). That is, the exclusive-or operation is commutative and associative. If we denote the left-shift operator by S (zero shifted into the low bit), then we have the following:

```
        (A + B)S = AS + BS
```

for any A, B. We will denote Sn to mean n applications of the left-shift operator. Now, let's look at the inner loop of the algorithm used in Lab #2.

```
          MASK = %0001000000100001   /* CCITT G */
          CRC = 0
  LOOP:   get next available byte (character) C
          CRC = CRC EOR C<<8
          for i=1 to 8
                  if (bit 15 of CRC == 1) then
                          CRC = (CRC<<1) EOR MASK
                  else
                          CRC = CRC<<1
                  endif
          goto LOOP
```

First, decompose CRC into crch and crcl where crcl is simply the high byte of CRC with eight zeros appended to make a 16-bit quantity. Similarly, crcl is the low byte of CRC left-extended with zeros to make a 16-bit quantity. Then we have

```
          CRC = crch + crcl
```

The first pass through the loop yields

```
          CRC = (crch + crcl)S + P1
```

where P1 is either 0 or MASK. After two iterations we have

```
          CRC = ((crch + crcl)S + P1)S + P2,  or
          CRC = crchS2 + crclS2 + P1S + P2
```

where P2 is either 0 or MASK. Using the properties of the S and exclusive-or operations we can write the CRC after eight iterations as:

```
          CRC = crchS8 + crclS8 + (P1S7 + P2S6 + ... + P8)
```

Again, the Ps are either 0 or MASK.

Note that the term crchS8 vanishes, so the only question is how can we calculate the last term? The observation we made earlier tells us that the third term depends only the initial value of crch. This means that we can compute the third term by computing all the possible values in the inner loop for CRC between $00 and $FF. Thus, the third term can be writted as

```
          (P1S7 + P2S6 + .. + P8) = CRC_TABLE[crch>>8]
```

(Note that the term on the right is written in a C style pesudocode where >>8 means shift to the right by 8 bits and CRC_TABLE[n] refers to the n-th element in the one dimensional array CRC_TABLE.) Remembering that + means exclusive-or, we obtain the relation

```
          CRC = CRC_TABLE[ crch>>8 ]  +  crcl<<8
```

Substituting this relation into the algorithm in Lab #2, we can derive the table driven CRC algorithm.

PSEUDOCODE:

1. Set up CRC_TABLE[i] for i ranging from $00 to $FF. You can use the algorithm in Lab #2.
2.

```
          CRC = 0
  LOOP:   get next available byte C from the data set
          i = CRC>>8 EOR C
          CRC = CRC_TABLE[i] EOR CRC<<8
          go to LOOP
```

MEASURING CLOCK CYLES:

There is a 68000 clock count pseudo-register in the debugger which can be reset and read when using the debugger. This lets you time and "fine-tune" code modules for fastest execution time during simulation.

Use the debugger command "Expression Monitor Values @cycles" to display the number of "cycles" on the screen. The "cycles" value is the cumulative number of clock cycles executed since this value was last reset. You can reset the value of cycles using the debugger command "Memory Registers @cycles=0").

To measure the number of clock cycles required by your program using the Lab #2 algorithm: reset the value of "cycles", run the program, and then read out the value of "cycles" from the Monitor window.

For the table look-up algorithm in this lab you need to first measure the number of clock cycles required for setting up the crc table. Then, you need to temporarily stop your program to reset "cycles" to zero. You will then execute your algorithm and determine the number of clock "cycles" used by your algorithm.

To temporarily stop your program you will need to set a break point. This can be done using the "Break Instruction addr" command in the debugger where "addr" is the address of the first instruction in your actual crc calculation routine, not the table setup routine. Once you have reset "cycles" use "Program Run" to re-start excuting your program after the break point. Hint: Place a NOP between setting up the crc table and your CRC algorithm. You can then use the address of this NOP as a break point.

For this lab, you should measure the number of clock "cycles" required by the algorithm used in Lab #2 and compare it to the number of clock cycles required by the table driven crc algorithm. Record this data for all 3 data sets given in Lab #2.

Here are the results we got. Your results may vary slightly, but you should be able to see a performance increase of 5 to 6 times.

Simple CRC Calculation

```
          Data Set #1 :  72227 "clock" cycles
          Data Set #2 :  72547
          Data Set #3 : 401873
```

Table Driven CRC

```
      CRC Table Setup :  74259 "clock" cycles
          Data Set #1 :  12081
          Data Set #2 :  12081
          Data Set #3 :  66695
```

DATA:

Use your program on the 3 data sets provided for Lab #2. Your lab writeup should include:

1. signed title page
2. program listing (assembler listing with symbol table)
3. short explanation of how your program works. Be sure to explain how you constructed CRC_TABLE and fetched data from it in your program. (Be sure to indicate what addressing mode you used and how it worked)
4. CRCs for the three blocks of data
5. Comparisons of clock cycles for normal CRC and table driven CRC calculations

BONUS:

If you can beat our numbers for the CRC table calculation, you will get a 10% bonus.