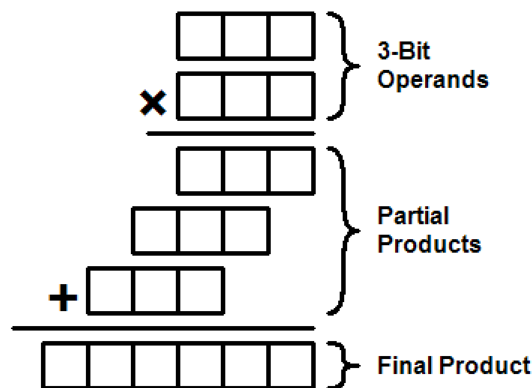
Lecture #27 – April 16, 2004

More Multiplication

Direct Hardware Multiplication

As we've seen before, the process of multiplication generates a series of partial products which are then added together. Performing these actions directly in hardware can make for complicated hardware. While there are parallel hardware adders, they use trees of sophisticated carry-look-ahead adders. We will next look at some alternative approaches to multiplication that do not require sophisticated hardware.



Multiplication by Table-Lookup

Often overlooked by many programmers is a purely software approach to multiplication, as well as other more complicated functions. The approach is called ***table lookup***, which is extremely fast but achieves its high speed at the expense of using a lot of memory. In essence, all the answers are pre-computed and stored in a table, and the "computation" process uses the input parameters to generate the address of the cell containing the right answer. In general, functions using table lookup run in O(1), or *constant* time.

On a processor such as the 6502, which does not contain a multiplication instruction, the table lookup approach is much faster than a general-purpose software multiplication subroutine. Unfortunately, the 6502 has extremely limited memory, so table lookup must be reserved for those problems where a small number of numbers must be multiplied together as fast as possible. Even on architectures which do support multiplication natively, the table lookup approach may be fast enough relative to the speed of the general-purpose multiplication instruction to justify their use, as long as the size of the table is not especially large.

Multiplying two 3-bit numbers (between 0 and 7) requires a table of 64 entries, containing all combinations of products between 0×0=0 and 7×7=49. We are pretending that the ARM chip has no built-in multiplication instruction, and will set up this problem

with two variables Op1 and Op2 containing the numbers we wish to multiply. Assume that Op1 and Op2 are always clipped to the correct range (i.e., only the rightmost three bits may contain non zero values). Since only three bits are valid, we need only logically OR one value with the other shifted left by three bits, forming a six bit offset into the table. As you can see from the code, the multiplication proceeds in constant time:

```
            LDR  R0,Op1          R0 = 000…000000000xxx
            LDR  R1,Op2          R1 = 000…000000000yyy
            MOV  R2,R0,LSL #3    R2 = 000…000000xxx000
            ORR  R2,R2,R1        R2 = 000…000000xxxyyy
            ADR  R5,Table
            LDR  Rx,[R5,R2,LSL #2]
            …

TABLE       DCD  0               Table[0]
            DCD  …               Table[1]
            DCD  …               Table[2]
            DCD  …               Table[3]
            …
            DCD  …               Table[60]
            DCD  …               Table[61]
            DCD  …               Table[62]
            DCD  49              Table[63]
```

Table lookup is a very nice mechanism for other tasks besides multiplication. Trigonometric functions SINE and COSINE are particularly well-suited for table lookup, as long as the input arguments are integer degrees. In this case, you need only 91 table entries to hold the sine values of all angles between 0° and 90°. Through reflection and rotation this same table can be used to extract the sine and the cosine of all angles between 0° and 360°. (The most obvious data type for a table of sine values is floating point, but I have used scaled integers as well.)

Multiplication by Shift / Add

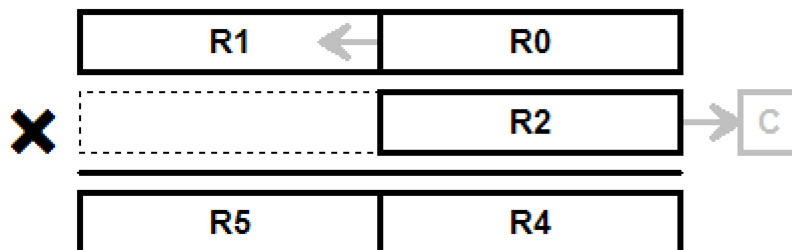
Continuing the pretense that our processor lacks a hardware multiplication instruction, we next look at a general-purpose software technique. When two 32-bit numbers are multiplied together the result is 64 bits in length, so we will need two registers to hold the result. As is turns out we will need to extend one of the operands to 64 bits as well. In a standard multiplication problem this will be the “top” operand.

The process is to shift the “bottom” operand to the right so that its rightmost bit goes into the carry bit. If the carry bit is set, then it would generate a partial product with the top operand; the (64-bit) top operand is added to the 64-bit result when that happens. If the carry bit was 0, the partial product will also be zero, so no addition is necessary. After the shift and test, the top operand is shifted left (64-bits) so that it is aligned with the correct position in the result if an addition is necessary. This process repeats until the

bottom operand is zero (which may take as many as 32 passes through the loop). In a high-level language this would be accomplished by the following pseudocode:

```
Total := 0
Repeat
     C   := Op2 Mod 2
     Op2 := Op2 Div 2
     If C = 1 Then Total := Total + Op1
     Op1 := Op1 * 2
Until Op2 = 0
```

The `Mod` (remainder) and `Div` (integer divide) instructions are implemented in assembly language as a single right-shift, into the carry, of the register containing Op2, as in `MOVS R2,R2,LSR #1` (remember to set the flags in order to get the rightmost bit into the carry). Multiplying Op1 by 2 is a simple left-shift. If Op1 is in `R0` and Op2 is in `R2`, then we reserve `R1` for the upper half of the 64-bit extended version of Op1, and registers `R4` and `R5` for the 64-bit result, as shown below:



The complete ARM assembly code for this process (roughly equivalent to the `UMULL` instruction) is shown below:

```
            MOV   R4,#0
            MOV   R5,#0
            MOV   R1,#0
LOOP                                Repeat
            MOVS  R2,R2,LSR #1           R2 := R2 Div 2, C=remain
            BCC   NextBit                If C=1 Then
            ADDS  R4,R4,R0                    64-Bit Add (low)
            ADC   R5,R5,R1                    64-Bit Add (high)
NextBit     MOVS  R0,R0,LSL #1           64-Bit LSL (low)
            ADC   R1,R1,R1               64-Bit LSL (high)
            CMP   R2,#0             Until R2 = 0
            BNE   LOOP
```

This software process is quite easy to implement in hardware, but because it is ***serial multiplication*** it is much slower than the direct parallel hardware approaches we have examined earlier. Modern processors use much faster hardware approaches, but at a corresponding increase in circuit complexity.

Extended Precision Arithmetic

Techniques similar to those used in the software multiplication routine are also used to create ***extended precision*** arithmetic routines. In extended precision, software routines are written to create synthetic data types not available natively on the processor. For example, 64-bit or larger integer arithmetic routines can be created using 32-bit registers (on the ARM or 386/486/Pentium), 16-bit registers (on the 8088), or 8-bit registers (on the 6502). Indeed, on small processors such as the 6502 it is often critical to create such routines.

In each of the following code fragments, contiguous memory tables have been set up for the Op1, Op2, and Result variables. Each code fragment adds Op1 to Op2 and places the sum into Result. Each variable is three storage locations long, using the native size of the processor; thus the ARM variables are three 32-bit locations for a total of 96 bits, the 8088 variables are 48 bits, and the 6502 variables are 24 bits.

```
96-Bit ARM          48-Bit 8088          24-Bit 6502
LDR  R0,Op1         MOV AX,Op1           CLC
LDR  R1,Op2         ADD AX,Op2           LDA Op1
ADDS R0,R0,R1       MOV Result,AX        ADC Op2
STR  R0,Result      MOV AX,Op1+2         STA Result
LDR  R0,Op1+4       ADC AX,Op2+2         LDA Op1+1
LDR  R1,Op2+4       MOV Result+2,AX      ADC Op2+1
ADCS R0,R0,R1       MOV AX,Op1+4         STA Result+1
STR  R0,Result+4    ADC AX,Op2+4         LDA Op1+2
LDR  R0,Op1+8       MOV Result+4,AX      ADC Op2+2
LDR  R1,Op2+8                            STA Result+2
ADCS R0,R0,R1
STR  R0,Result+8
```

Subtraction is nearly identical. Multiplication is harder, but can be implemented using a combination of techniques that we have already covered. Division is much more difficult. It should be pretty obvious that there will be more code necessary for the 6502 than for the ARM to compute sums of equivalent lengths. It should also be obvious that the code is fairly regular, and can be implemented with fairly short loops.