Technical Note

Multiply Routines

TN000701-0603

General Overview

Many microcontroller applications require arithmetic functions beyond the instructions directly supported by the device. The most common among these applications are simple multiply and divide routines for scaling and filtering purposes. The Z8 register file architecture provides a significant advantage for implementing efficient routines for these functions. This Technical Note shows efficient implementations of 8-bit by 8-bit and 16-bit by 16-bit unsigned multiply routines. A companion Technical Note, Divide Routines (TN0006), describes an 8-bit by 16-bit and 16-bit by 16-bit divide routine.

Discussion

The first software module, **multiply_8_8** (see “Multiply_8_8” on page 2), illustrates an efficient algorithm for the division of two unsigned 8-bit values, resulting in a 16-bit unsigned product. The second software module, **multiply_16_16** (“Multiply_16_16” on page 3), multiplies an unsigned 16-bit multiplier by an unsigned 16-bit multiplicand resulting in an unsigned 32-bit product. Both routines use a similar algorithm.

In the **multiply_8_8** routine, the multiplicand is repetitively shifted right (via the Rotate Right through Carry instruction), with the low-order bits being shifted out into the carry flag. If the low-order bit shifted out is a 1, the multiplier is added to the high-order byte of the partial product. As the high-order bits of the multiplicand are vacated by the Shift Right instruction, the resulting partial product bits are shifted in. This routine takes 22 bytes of code and 4 registers.

In the **multiply_16_16** routine, the same basic algorithm is used as in the **multiply_8_8** routine above, except that four shifts are required to result in the 32-bit product, and a 16-bit Add is performed to make up the partial product. A final test is performed before exiting the routine to determine whether the result fits in 16 bits or requires 32 bits. If the final result fits into a 16-bit register, the zero flag is set; otherwise the zero flag is cleared to indicate that the result occupies 32 bits. The routine is implemented in 34 bytes of code and uses 7 registers.

The routines are implemented in a modular fashion such that they may be easily integrated into a target application. The registers used in the routines can be located in any working register group by defining the value for the *math_group* equate. The `.r(symbol_name)` designation causes the assembler to use a working register for the variable as opposed to an absolute address location. The variable name could be used directly, but this method uses less code space.

The execution time of these multiply routines depends on the values being used in the multiplier, the multiplicand, and the operating frequency of the MCU. In the following 8-bit by 8-bit multiply example, a Z8Plus device operating at a frequency of 10MHz takes 47µs to complete the multiply function (including entry and exit overhead).

0x55h x 0xAAh = 0x3872h

In the following 16-bit by 16-bit divide example, a Z8Plus device operating at a frequency of 10MHz takes 128µs to complete the multiply function.

0x5555h x 0xAAAAh = 0x38E31C72

Sample Code

Multiply_8_8

The following code illustrates the **multiply_8_8** software module.

```
;*************************************************************************
; Multiplies two 8-bit values resulting in a 16-bit product.
; Result is returned in product_hi,product_lo.
;*************************************************************************
math_group    equ     00h       ;Defines the WRG for this routine

  segment     data
  align               16
mult_len      ds      1
multiplier    ds      1
product_hi    ds      1
product_lo    ds      1

;-------------------------------------------------------------------------
  segment     code

multiply_8_8:
  push        rp                ;Save the current register pointer
  srp         #math_group       ;and use our own working register group
  ld          .r(mult_len),#9   ;multiplier is 8 bits (+1)

  clr         .r(product_hi)    ;Initialize MSB of result
  rcf
$loop_8:
  rrc         .r(product_hi)
  rrc         .r(product_lo)
  jr          nc,$next_8
  add         .r(product_hi),.r(multiplier)
$next_8:
```

```
  djnz        .r(mult_len),$loop_8

  pop         rp                ;Restore register pointer
  ret
```

Multiply_16_16

The following code illustrates the **multiply_16_16** software module.

```
;*******************************************************************************
; Multiplies two unsigned 16-bit values resulting in a 32-bit product.
; Input: multiplier_hi, multiplier_lo
; multiplicand_hi, multiplicand_lo
; Output: Result returned in:
; product_hi, product_lo, multiplicand_hi, multiplicand_lo (MSB...LSB).
;*******************************************************************************
math_group          equ     00h       ;Defines the WRG for this routine

  segment           data
  align                     16
mult_len            ds      1
multiplier_hi       ds      1
multiplier_lo       ds      1
product_hi          ds      1
product_lo          ds      1
multiplicand_hi     ds      1
multiplicand_lo     ds      1
;-------------------------------------------------------------------------------
  segment  code

multiply_16_16:
  push    rp                  ;Save the current register pointer
  srp     #math_group         ;and use our own working register group
  ld      .r(mult_len),#17    ;multiplier is 16 bits (+1)

  clr     .r(product_hi)      ;Initialize most significant
  clr     .r(product_lo)      ;word of 32 bit result
  rcf
$loop_16:
  rrc     .r(product_hi)
  rrc     .r(product_lo)
  rrc     .r(multiplicand_hi)
  rrc     .r(multiplicand_lo)
  jr      nc,$next_16
  add     .r(product_lo),.r(multiplier_lo)
  adc     .r(product_hi),.r(multiplier_hi)
$next_16:
  djnz    .r(mult_len),$loop_16

  ld      .r(mult_len),.r(product_hi);Z flag = 0 if result is > 16 bits
  or      .r(mult_len),.r(product_lo);Z flag = 1 if result fits in 16 bits
  pop     rp                  ;Restore register pointer
  ret
```

This publication is subject to replacement by a later edition. To determine whether a later edition exists, or to request copies of publications, contact:

ZiLOG Worldwide Headquarters
532 Race Street
San Jose, CA 95126
Telephone: 408.558.8500
Fax: 408.558.8300
www.zilog.com

Information Integrity

The information contained within this document has been verified according to the general principles of electrical and mechanical engineering. Any applicable source code illustrated in the document was either written by an authorized ZiLOG employee or licensed consultant. Permission to use these codes in any form, besides the intended application, must be approved through a license agreement between both parties. ZiLOG will not be responsible for any code(s) used beyond the intended application. Contact the local ZiLOG Sales Office to obtain necessary license agreements.

Document Disclaimer