AVR200: Multiply and Divide Routines

8-Bit AVR® MCU with Downloadable Flash

Application Note

Features

- **8 and 16-bit Implementations**
- **Signed & Unsigned Routines**
- **Speed & Code Size Optimized Routines**
- **Runable Example Programs**
- **Speed is Comparable with HW Multiplicators/Dividers**
  Example: 8 x 8 Mul in 2.8 ms, 16 x 16 Mul in 8.7 µs (12 MHz)
- **Extremely Compact Code**

Introduction

This application note lists subroutines for multiplication and division of 8 and 16-bit signed and unsigned numbers. A listing of all implementations with key performance specifications is given in Table 1.

Table 1. Performance Figures Summary

Application	Code Size (Words)	Execution Time (Cycles)
8 x 8 = 16 bit unsigned (Code Optimized)	9	58
8 x 8 = 16 bit unsigned (Speed Optimized)	34	34
8 x 8 = 16 bit signed (Code Optimized)	10	73
16 x 16 = 32 bit unsigned (Code Optimized)	14	153
16 x 16 = 32 bit unsigned (Speed Optimized)	105	105
16 x 16 = 32 bit signed (Code Optimized)	16	218
8 / 8 = 8 + 8 bit unsigned (Code Optimized)	14	97
8 / 8 = 8 + 8 bit unsigned (Speed Optimized)	66	58
8 / 8 = 8 + 8 bit signed (Code Optimized)	22	103
16 / 16 = 16 + 16 bit unsigned (Code Optimized)	19	243
16 / 16 = 16 + 16 bit unsigned (Speed Optimized)	196	173
16 / 16 = 16 + 16 bit signed (Code Optimized)	39	255

The application note listing consists of two files:

“avr200.asm”: Code size optimized multiplied and divide routines.

“avr200b.asm”: Speed optimized multiply and divide routines.

8 x 8 = 16 Unsigned Multiplication - “mpy8u”

Both program files contain a routine called “mpy8u” which performs unsigned 8-bit multiplication. Both implementations are based on the same algorithm. The code size optimized implementation, however, uses looped code, whereas the speed optimized code is a straight-line code implementation. Figure 1 shows the flow chart for the code size optimized version.

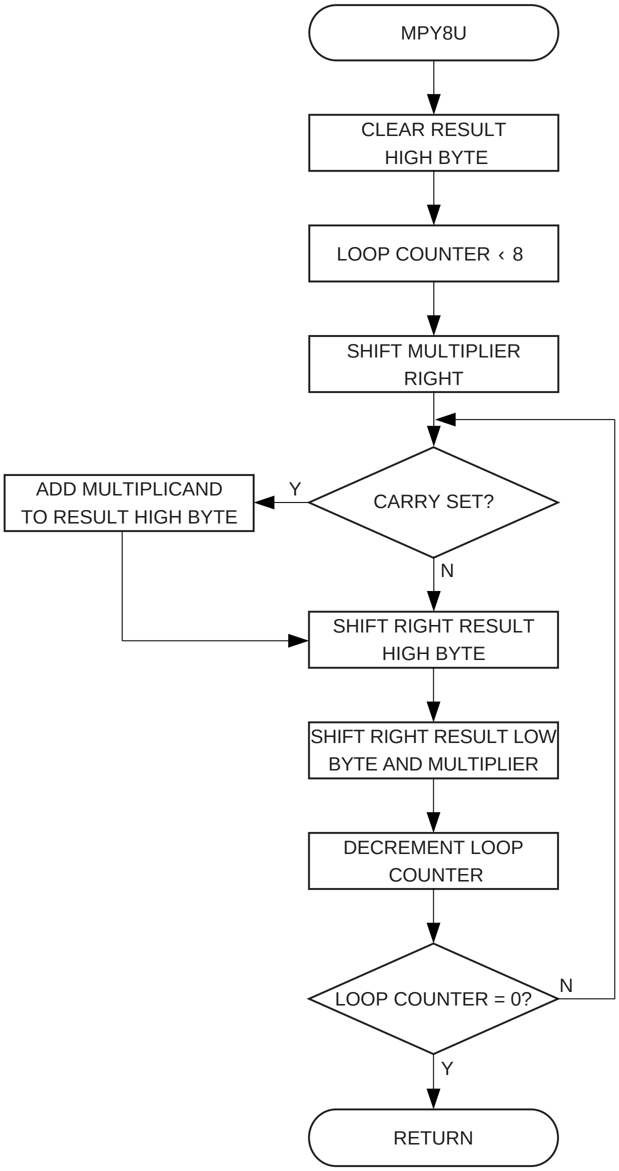
Rev. 0936B–10/98

Algorithm Description

The algorithm for the Code Size optimized version is as follows:

1. Clear result High byte.
2. Load loop counter with 8.
3. Shift right multiplier
4. If carry (previous bit 0 of multiplier) set, add multiplicand to result High byte.
5. Shift right result High byte into result Low byte/multiplier.
6. Shift right result Low byte/multiplier.
7. Decrement Loop counter.
8. If loop counter not zero, goto Step 4.

Figure 1. "mpy8u" Flow Chart (Code Size Optimized Implementation)



Usage

The usage of “mpy8u” is the same for both versions:

1. Load register variables “mp8u” and “mc8u” with the multiplier and multiplicand, respectively.
2. Call “mpy8u”
3. The 16 -bit result is found in the two register variables “m8uH” (High byte) and “m8uL” (Low byte)

Observe that to minimize register usage, code and execution time, the multiplier and result Low byte share the same register.

Performance

Table 2. “mpy8u” Register Usage (Code Size Optimized Implementation)

Register	Input	Internal	Output
R16	“mc8u” - multiplicand		
R17	“mp8u” - multiplier		“m8uL” - result Low byte
R18			“m8uH” - result High byte
R19		“mcnt8u” - loop counter	

Table 3. “mpy8u” Performance Figures (Code Size Optimized Implementation)

Parameter	Value
Code Size (Words)	9 + return
Execution Time (Cycles)	58 + return
Register Usage	• Low registers :None • High registers :4 • Pointers :None
Interrupts Usage	None
Peripherals Usage	None

Table 4. “mpy8u” Register Usage (Straight-line Implementation)

Register	Input	Internal	Output
R16	“mc8u” - multiplicand		
R17	“mp8u” - multiplier		“m8uL” - result Low byte
R18			“m8uH” - result High byte

Table 5. “mpy8u” Performance Figures (Straight-Line Implementation)

Parameter	Value
Code Size (Words)	34 + return
Execution Time (Cycles)	34 + return
Register Usage	• Low registers :None • High registers :3 • Pointers :None
Interrupts Usage	None
Peripherals Usage	None

8 x 8 = 16 Signed Multiplication - “mpy8s”

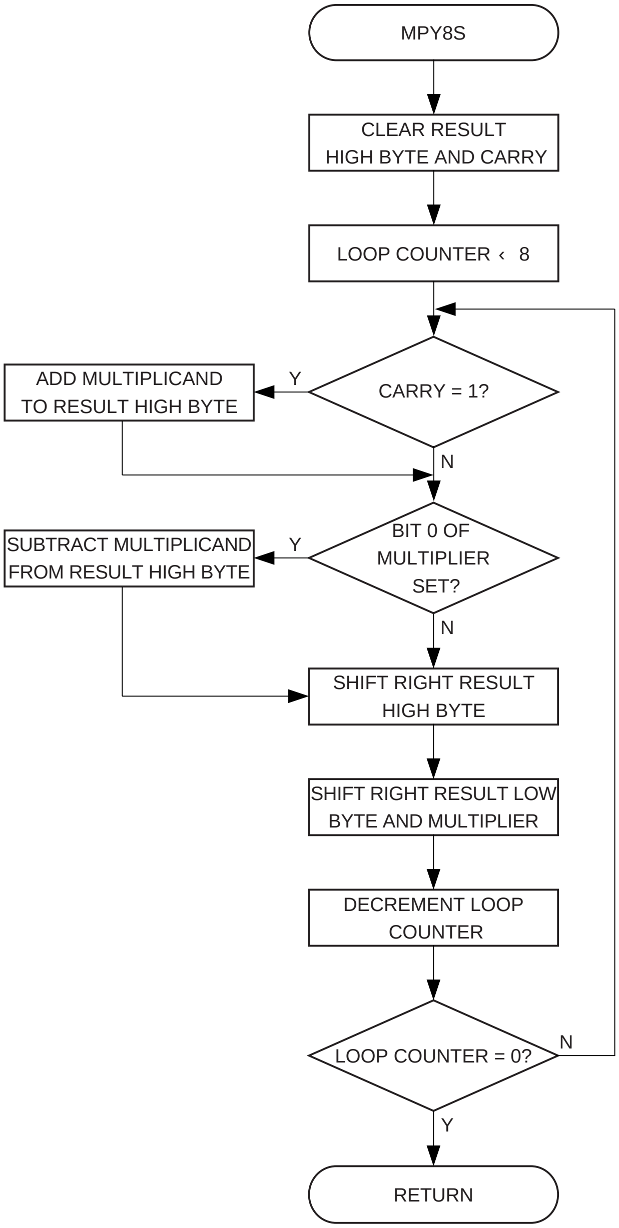
This subroutine, which is found in “avr200.asm” implements signed 8 x 8 multiplication. Negative numbers are represented as 2’s complement numbers. The application is an implementation of Booth's algorithm. The algorithm provides both small and fast code. However, it has one limitation that the user should bear in mind; If all 16 bits of the result is needed, the algorithm fails when used with the most negative number (-128) as the multiplicand.

Algorithm Description

The algorithm for signed 8 x 8 multiplication is as follows:

1. Clear result High byte and carry.
2. Load loop counter with 8.
3. If carry (previous bit 0 of multiplier) set, add multiplicand to result High byte.
4. If current bit 0 of multiplier set, subtract multiplicand from result High byte.
5. Shift right result High byte into result Low byte/multiplier.
6. Shift right result Low byte/multiplier.
7. Decrement loop counter.
8. If loop counter not zero, goto Step 3.

Figure 2. “mpy8s” Flow Chart


MPY8S
CLEAR RESULT HIGH BYTE AND CARRY
LOOP COUNTER ‹ 8
CARRY = 1?
Y
ADD MULTIPLICAND TO RESULT HIGH BYTE
N
BIT 0 OF MULTIPLIER SET?
Y
SUBTRACT MULTIPLICAND FROM RESULT HIGH BYTE
N
SHIFT RIGHT RESULT HIGH BYTE
SHIFT RIGHT RESULT LOW BYTE AND MULTIPLIER
DECREMENT LOOP COUNTER
LOOP COUNTER = 0?
N
Y
RETURN

Usage

The usage of “mpy8s” is as follows:

1. Load register variables “mp8s” and “mc8s” with the multiplier and multiplicand, respectively.
2. Call “mpy8s”
3. The 16 -bit result is found in the two register variables “m8sH” (High byte) and “m8sL” (Low byte)

Observe that to minimize register usage, code and execution time, the multiplier and result Low byte share the same register.

Performance

Table 6. “mpy8s” Register Usage

Register	Input	Internal	Output
R16	“mc8s” - multiplicand		
R17	“mp8s” - multiplier		“m8sL” - result Low byte
R18			“m8sH” - result High byte
R19		“mcnt8s” - loop counter	

Table 7. “mpy8s” Performance Figures

Parameter	Value
Code Size (Words)	10 + return
Execution Time (Cycles)	73 + return
Register Usage	• Low registers :None • High registers :4 • Pointers :None
Interrupts Usage	None
Peripherals Usage	None

16 x 16 = 32 Unsigned Multiplication - “mpy16u”

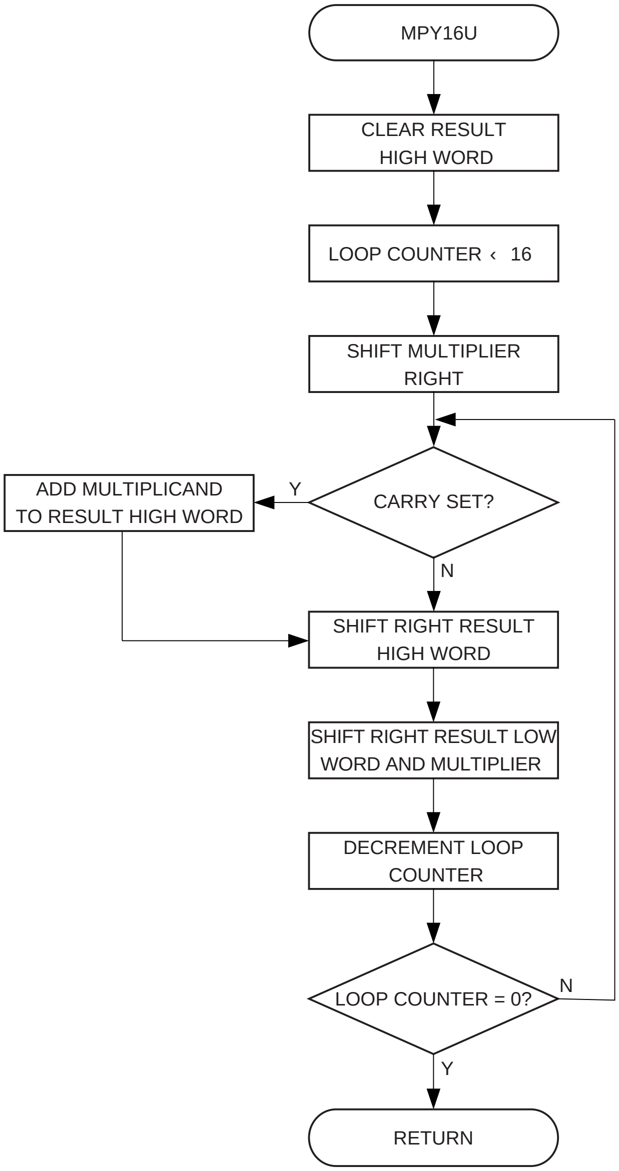
Both program files contain a routine called “mpy16u” which performs unsigned 16-bit multiplication. Both implementations are based on the same algorithm. The code size optimized implementation, however, uses looped code, whereas the speed optimized code is a straight-line code implementation. Figure 3 shows the flow chart for the Code Size optimized (looped) version.

Algorithm Description

The algorithm for the Code Size optimized version is as follows:

1. Clear result high word (Bytes 2 and 3)
2. Load loop counter with 16.
3. Shift multiplier right
4. If carry (previous bit 0 of multiplier Low byte) set, add multiplicand to result High word.
5. Shift right result High word into result Low word/multiplier.
6. Shift right Low word/multiplier.
7. Decrement Loop counter.
8. 8. If loop counter not zero, goto Step 4.

Figure 3. “mpy16u” Flow Chart (Code Size Optimized Implementation)



Usage

The usage of “mpy16u” is the same for both versions:

1. Load register variables “mp16uL”/”mp16uH” with multiplier Low and High byte, respectively.
2. Load register variables “mc16uH”/”mc16uH” with multiplicand Low and High byte, respectively.
3. Call “mpy16u”
4. The 32-bit result is found in the four-byte register variable “m16u3:m16u2:m16u1:m16u0”.

Observe that to minimize register usage, code and execution time, the multiplier and result Low word share the same registers.

Performance

Table 8. "mpy16u" Register Usage (Code Size Optimized Implementation)

Register	Input	Internal	Output
R16	"mc16uL" - multiplicand low byte		
R17	"mc16uH" - multiplicand high byte		
R18	"mp16uL" - multiplier low byte		"m16u0" - result byte 0
R19	"mp16uH" - multiplier high byte		"m16u1" - result byte 1
R20			"m16u2" - result byte 2
R21			"m16u3" - result byte 3
R22		"mcnt16u" - loop counter	

Table 9. "mpy16u" Performance Figures (Code Size Optimized Implementation)

Parameter	Value
Code Size (Words)	14 + return
Execution Time (Cycles)	153 + return
Register Usage	• Low registers :None • High registers :7 • Pointers :None
Interrupts Usage	None
Peripherals Usage	None

Table 10. "mpy16u" Register Usage (Straight-line Implementation)

Register	Input	Internal	Output
R16	"mc16uL" - multiplicand low byte		
R17	"mc16uH" - multiplicand high byte		
R18	"mp16uL" - multiplier low byte		"m16u0" - result byte 0
R19	"mp16uH" - multiplier high byte		"m16u1" - result byte 1
R20			"m16u2" - result byte 2
R21			"m16u3" - result byte 3

Table 11. "mpy16u" Performance Figures (Straight-Line Implementation)

Parameter	Value
Code Size (Words)	105 + return
Execution Time (Cycles)	105 + return
Register Usage	• Low registers :None • High registers :6 • Pointers :None
Interrupts Usage	None
Peripherals Usage	None

16 x 16 = 32 Signed Multiplication - “mpy16s”

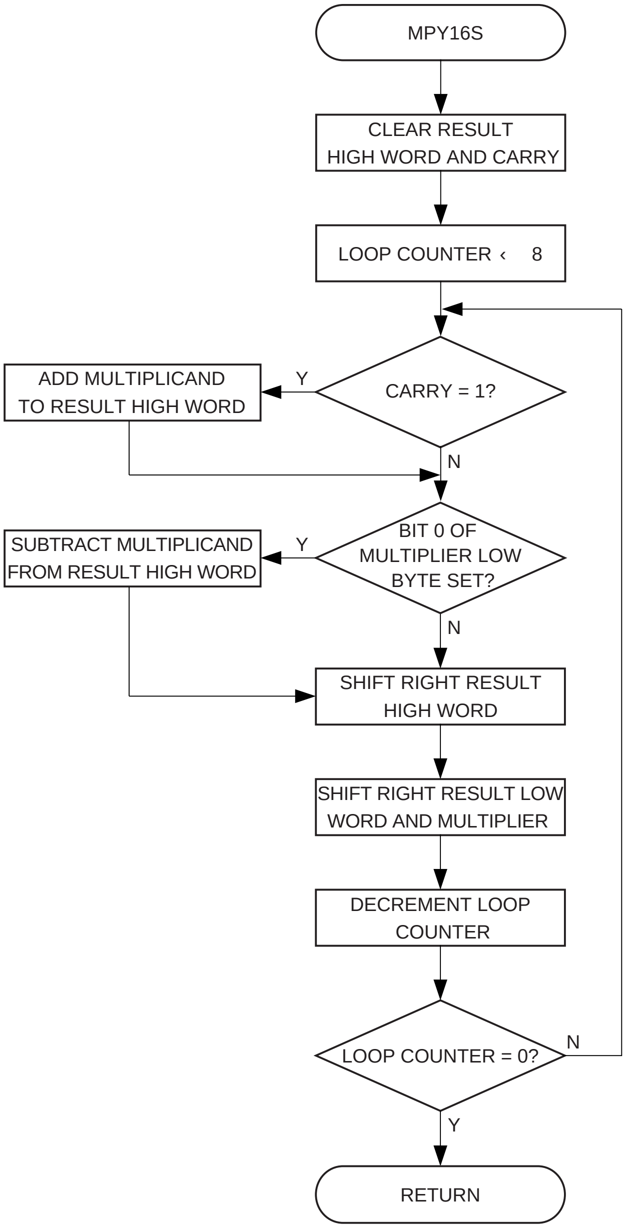
This subroutine, which is found in “avr200.asm” implements signed 16 x 16 multiplication. Negative numbers are represented as 2’s complement numbers. The application is an implementation of Booth’s algorithm. The algorithm provides both small and fast code. However, it has one limitation that the user should bear in mind; If all 32 bits of the result is needed, the algorithm fails when used with the most negative number (-32768) as the multiplicand.

Algorithm Description

The algorithm for signed 16 x 16 multiplication is as follows:

1. Clear result High word (Bytes 2&3) and carry.
2. Load loop counter with 16.
3. If carry (previous bit 0 of multiplier Low byte) set, add multiplicand to result High word.
4. If current bit 0 of multiplier Low byte set, subtract multiplicand from result High word.
5. Shift right result High word into result Low word/multiplier.
6. Shift right Low word/multiplier.
7. Decrement Loop counter.
8. If loop counter not zero, goto Step 3.

Figure 4. “mpy16s” Flow Chart


MPY16S
CLEAR RESULT HIGH WORD AND CARRY
LOOP COUNTER ‹ 8
CARRY = 1?
Y
ADD MULTIPLICAND TO RESULT HIGH WORD
N
BIT 0 OF MULTIPLIER LOW BYTE SET?
Y
SUBTRACT MULTIPLICAND FROM RESULT HIGH WORD
N
SHIFT RIGHT RESULT HIGH WORD
SHIFT RIGHT RESULT LOW WORD AND MULTIPLIER
DECREMENT LOOP COUNTER
LOOP COUNTER = 0?
N
Y
RETURN

Usage

The usage of “mpy16s” is as follows:

1. Load register variables “mp16sL”/”mp16sH” with multiplier Low and High byte, respectively.
2. Load register variables “mc16sH”/”mc16sH” with multiplicand Low and High byte, respectively.
3. Call “mpy16s”.
4. The 32-bit result is found in the four-byte register variable “m16s3:m16s2:m16s1:m16s0”.

Observe that to minimize register usage, code and execution time, the multiplier and result Low byte share the same register.

Performance

Table 12. “mpy16s” Register Usage

Register	Input	Internal	Output
R16	“mc16uL” - multiplicand low byte		
R17	“mc16uH” - multiplicand high byte		
R18	“mp16uL” - multiplier low byte		“m16u0” - result byte 0
R19	“mp16uH” - multiplier high byte		“m16u1” - result byte 1
R20			“m16u2” - result byte 2
R21			“m16u3” - result byte 3
R22		“mcnt16u” - loop counter	

Table 13. “mpy16s” Performance Figures

Parameter	Value
Code Size (Words)	16 + return
Execution Time (Cycles)	218 + return
Register Usage	• Low registers :None • High registers :7 • Pointers :None
Interrupts Usage	None
Peripherals Usage	None

8 / 8 = 8 + 8 Unsigned Division - “div8u”

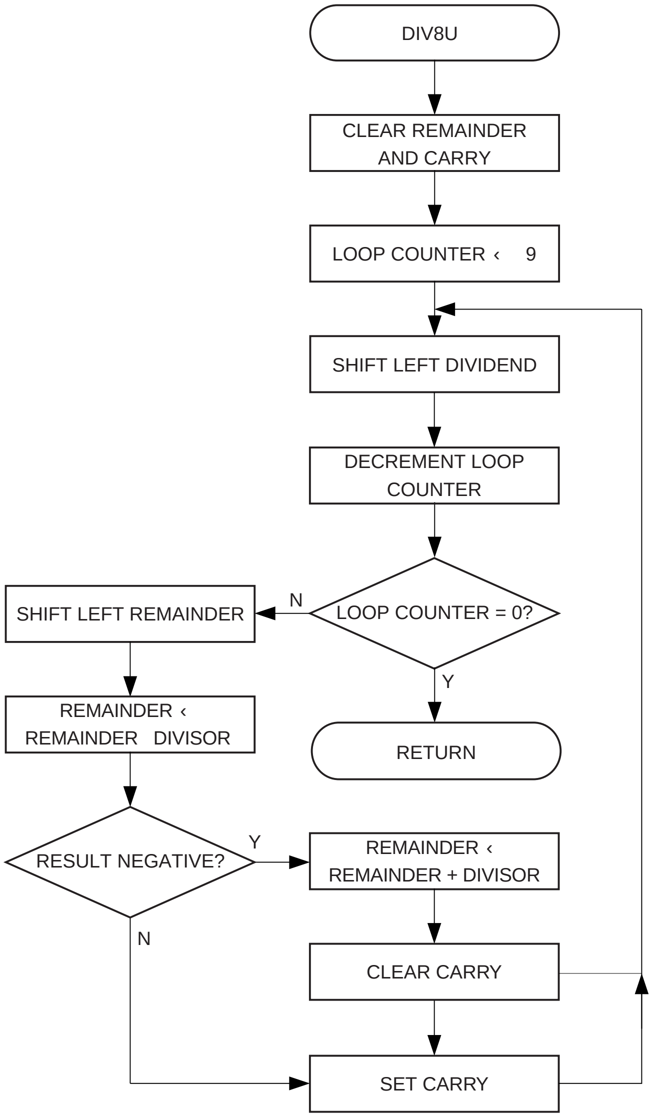
Both program files contain a routine called “div8u” which performs unsigned 8-bit division. Both implementations are based on the same algorithm. The code size optimized implementation, however, uses looped code, whereas the speed optimized code is a straight-line code implementation. Figure 5 shows the flow chart for the code size optimized version.

Algorithm Description

The algorithm for unsigned 8 / 8 division (Code Size optimized code) is as follows:

1. Clear remainder and carry.
2. Load loop counter with 9.
3. Shift left dividend into carry.
4. Decrement loop counter.
5. If loop counter = 0, return.
6. Shift left carry (from dividend/result) into remainder
7. Subtract divisor from remainder.
8. If result negative, add back divisor, clear carry and goto Step 3.
9. Set carry and goto Step 3.

Figure 5. "div8u" Flow Chart (Code Size Optimized Implementation)



Usage

The usage of "div8u" is the same for both implementations and is described in the following procedure:

1. Load register variable "dd8u" with the dividend (the number to be divided).
2. Load register variable "dv8u" with the divisor (the dividing number).
3. Call "div8u".
4. The result is found in "dres8u" and the remainder in "drem8u".

Performance

Table 14. "div8u" Register Usage (Code Size Optimized Version)

Register	Input	Internal	Output
R15			"drem8u" - remainder
R16	"dd8u" - dividend		"dres8u" - result
R17	"dv8u" - divisor"		
R18		"dcnt8u" - loop counter	

Table 15. "div8u" Performance Figures (Code Size Optimized Version)

Parameter	Value
Code Size (Words)	14
Execution Time (Cycles)	97
Register Usage	• Low registers :1 • High registers :3 • Pointers :None
Interrupts Usage	None
Peripherals Usage	None

Table 16. "div8u" Register Usage (Speed Optimized Version)

Register	Input	Internal	Output
R15			"drem8u" - remainder
R16	"dd8u" - dividend		"dres8u" - result
R17	"dv8u" - divisor"		

Table 17. "div8u" Performance Figures (Speed Optimized Version)

Parameter	Value
Code Size (Words)	66
Execution Time (Cycles)	58
Register Usage	• Low registers :1 • High registers :2 • Pointers :None
Interrupts Usage	None
Peripherals Usage	None

8 / 8 = 8 + 8 Signed Division - "div8s"

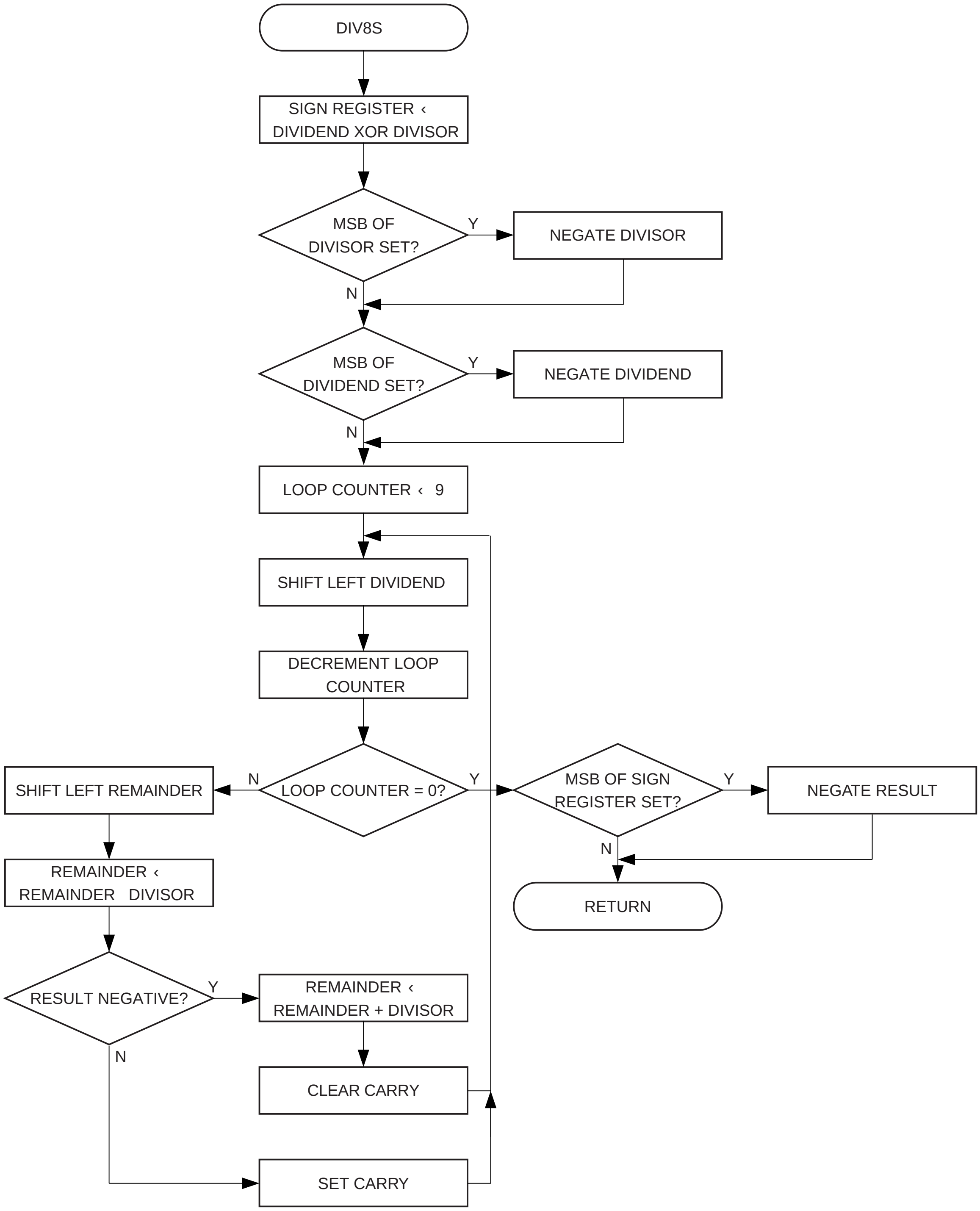
The subroutine "mpy8s" implements signed 8-bit division. The implementation is Code Size optimized. If negative, the input values shall be represented on 2's complement's form.

Algorithm Description

The algorithm for signed 8 / 8 division is as follows:

1. XOR dividend and divisor and store in a Sign register.
2. If MSB of dividend set, negate dividend.
3. If MSB if divisor set, negate dividend.
4. Clear remainder and carry.
5. Load loop counter with 9.
6. Shift left dividend into carry.
7. Decrement loop counter.
8. If loop counter ≠ 0, goto step 11.
9. If MSB of Sign register set, negate result.
10. Return
11. Shift left carry (from dividend/result) into remainder.
12. Subtract divisor from remainder.
13. If result negative, add back divisor, clear carry and goto Step 6.
14. Set carry and goto Step 6.

Figure 6. “div8s” Flow Chart



Usage

The usage of “div8s” follows the procedure below:

1. Load register variable “dd8s” with the dividend (the number to be divided).
2. Load register variable “dv8s” with the divisor (the dividing number).
3. Call “div8s”.
4. The result is found in “dres8s” and the remainder in “drem8s”.

Performance

Table 18. “div8s” Register Usage

Register	Input	Internal	Output
R14		“d8s” - sign register	
R15			“drem8s” - remainder
R16	“dd8s” - dividend		“dres8s” - result
R17	“dv8s” - divisor”		
R18		“dcnt8s” - loop counter	

Table 19. “div8s” Performance Figures

Parameter	Value
Code Size (Words)	22
Execution Time (Cycles)	103
Register Usage	• Low registers :2 • High registers :3 • Pointers :None
Interrupts Usage	None
Peripherals Usage	None

16 / 16 = 16 + 16 Unsigned Division - “div16u”

Both program files contain a routine called “div16u” which performs unsigned 16-bit division

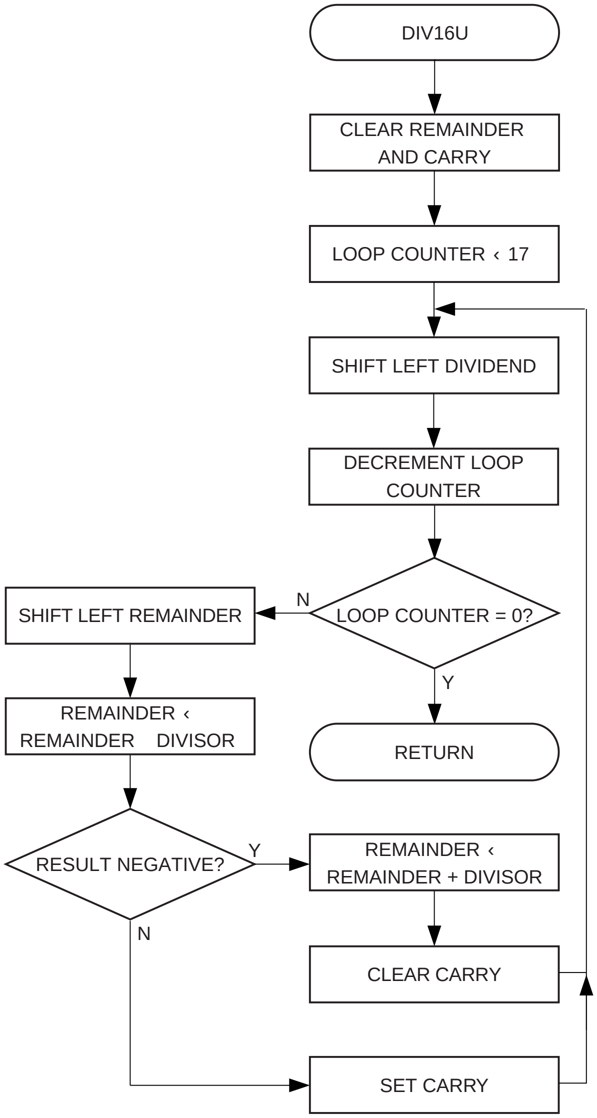
Both implementations are based on the same algorithm. The code size optimized implementation, however, uses looped code, whereas the speed optimized code is a straight-line code implementation. Figure 7 shows the flow chart for the code size optimized version.

Algorithm Description

The algorithm for unsigned 16 / 16 division (Code Size optimized code) is as follows:

1. Clear remainder and carry.
2. Load loop counter with 17.
3. Shift left dividend into carry
4. Decrement loop counter.
5. If loop counter = 0, return.
6. Shift left carry (from dividend/result) into remainder
7. Subtract divisor from remainder.
8. If result negative, add back divisor, clear carry and goto Step 3.
9. Set carry and goto Step 3.

Figure 7. "div16u" Flow Chart (Code Size Optimized Implementation)



Usage

The usage of "div16u" is the same for both implementations and is described in the following procedure:

1. Load the 16-bit register variable "dd16uH:dd16uL" with the dividend (the number to be divided).
2. Load the 16-bit register variable "dv16uH:dv16uL" with the divisor (the dividing number).
3. Call "div16u".
4. The result is found in "dres16u" and the remainder in "drem16u".

Performance

Table 20. “div16u” Register Usage (Code Size Optimized Version)

Register	Input	Internal	Output
R14			“drem16uL” - remainder low byte
R15			“drem16uH - remainder high byte
R16	“dd16uL” - dividend low byte		“dres16uL” - result low byte
R17	“dd16uH” - dividend high byte		“dres16uH” - result high byte
R18	“dv16uL” - divisor low byte		
R19	“dv16uH” - divisor high byte		
R20		“dcnt16u” - loop counter	

Table 21. “div16u” Performance Figures (Code Size Optimized Version)

Parameter	Value
Code Size (Words)	19
Execution Time (Cycles)	243
Register Usage	• Low registers :2 • High registers :5 • Pointers :None
Interrupts Usage	None
Peripherals Usage	None

Table 22. “div16u” Register Usage (Speed Optimized Version)

Register	Input	Internal	Output
R14			“drem16uL” - remainder low byte
R15			“drem16uH - remainder high byte
R16	“dd16uL” - dividend low byte		“dres16uL” - result low byte
R17	“dd16uH” - dividend high byte		“dres16uH” - result high byte
R18	“dv16uL” - divisor low byte		
R19	“dv16uH” - divisor high byte		

Table 23. “div16u” Performance Figures (Speed Optimized Version)

Parameter	Value
Code Size (Words)	196 + return
Average Execution Time (Cycles)	173
Register Usage	• Low registers :2 • High registers :4 • Pointers :None
Interrupts Usage	None
Peripherals Usage	None

16 / 16 = 16 + 16 Signed Division - “div16s”

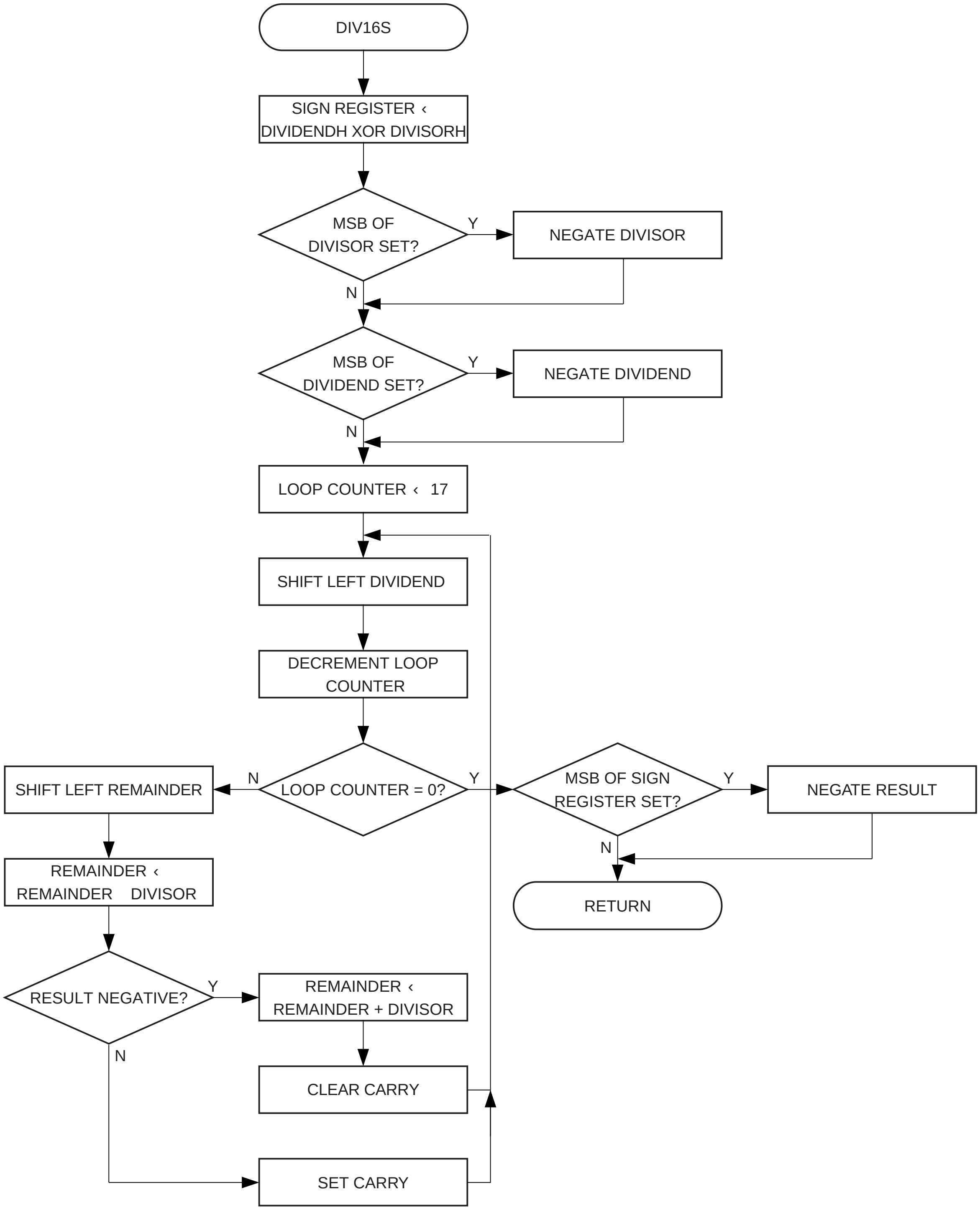
The subroutine “mpy16s” implements signed 16-bit division. The implementation is Code Size optimized. If negative, the input values shall be represented on 2’s complement’s form.

Algorithm Description

The algorithm for signed 16 / 16 division is as follows:

1. XOR dividend and divisor High bytes and store in a Sign register.
2. If MSB of dividend High byte set, negate dividend.
3. If MSB if divisor set High byte, negate dividend.
4. Clear remainder and carry.
5. Load loop counter with 17.
6. Shift left dividend into carry.
7. Decrement loop counter.
8. If loop counter ≠ 0, goto step 11.
9. If MSB of Sign register set, negate result.
10. Return
11. Shift left carry (from dividend/result) into remainder
12. Subtract divisor from remainder.
13. If result negative, add back divisor, clear carry and goto Step 6.
14. Set carry and goto Step 6.

Figure 8. “div16s” Flow Chart



Usage

The usage of “div16s” is described in the following procedure:

1. Load the 16-bit register variable “dd16sH:dd16sL” with the dividend (the number to be divided).
2. Load the 16-bit register variable “dv16sH:dv16sL” with the divisor (the dividing number).
3. Call “div16s”.
4. The result is found in “dres16s” and the remainder in “drem16s”.

Performance

Table 24. “div16s” Register Usage

Register	Input	Internal	Output
R14			“drem16sL” - remainder low byte
R15			“drem16sH - remainder high byte
R16	“dd16sL” - dividend low byte		“dres16sL” - result low byte
R17	“dd16sH” - dividend high byte		“dres16sH” - result high byte
R18	“dv16sL” - divisor low byte		
R19	“dv16sH” - divisor high byte		
R20		“dcnt16s” - loop counter	

Table 25. “div16s” Performance Figures

Parameter	Value
Code Size (Words)	39
Execution Time (Cycles)	255
Register Usage	• Low registers :2 • High registers :5 • Pointers :None
Interrupts Usage	None
Peripherals Usage	None