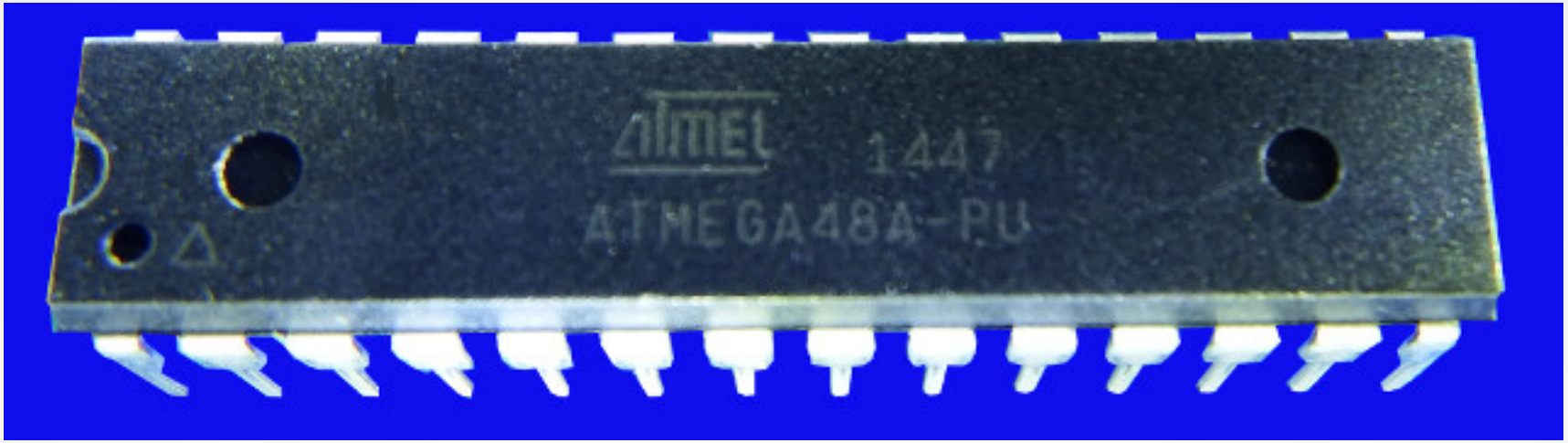


Lecture 10: Assembler math

Hardware, Internals and Programming of AVR Microcontrollers in Assembler

by
Gerhard Schmidt
Kastanienallee 20
D-64289 Darmstadt

Assembler math

- **Assembler math is special because it requires writing all routines by yourself using the available instructions that the CPU understands. While in higher-level languages this is already implemented as part of the language, assembler doesn't have this. The advantage is that your routines are tailored exactly to your needs, you do not waste memory space and unnecessary execution time.**
- **Assembler math is binary and therefore simpler than decimal math: only zeros and ones have to be considered.**
- **I'll discuss 8 and 16 bit addition and subtraction here as well as 8x8 and 16*8 bit multiplication (in software and in hardware versions) and 8/8 and 16/8 bit division. If you need other math, see if this website here provides examples.**
- **At the end I'll demonstrate 8- and 16-bit binary-to-decimal conversion.**

8-by-8 bit addition and subtraction

- **Adding two 8 bit numbers is simple: the mnemonic for that is ADD.**

```
; Adding two registers
    LDI R16, 128 ; Load 128 to a register
    LDI R17, 64 ; Load 64 to a second register
    ADD R16, R17 ; Add the second register, result in the first register
```

- **If the result is larger than 255, the carry flag C in SREG is set. If the result is zero (= 256) the Z flag in SREG is additionally set.**
- **Similiarly subtraction is also simple: the mnemonic for that is SUB.**

```
; Subtracting two registers
    LDI R16, 128 ; Load 128 to a register
    LDI R17, 64 ; Load 64 to a second register
    SUB R16, R17 ; Subtract the second register, result in the first register
```

- **The carry flag is set if the second number is larger than the first, and the Z flag if both are equal.**

16-by-8 bit addition and subtraction

- Adding an 8 bit number to a 16 bit number involves the carry flag. The mnemonics for that are ADC and SBC (add or subtract with carry).

```
; Adding an 8-bit register to a 16 bit register
      LDI R16, 128 ; Load 128 to a register as LSB
      LDI R17, 0 ; Load 0 to the MSB
      LDI R18, 240 ; Load 240 to the 8 bit register
      ADD R16, R18 ; First add the 8 bit register to the LSB
      BRCC NoCarry ; Branch if carry flag is clear
      INC R17 ; Add the carry to the MSB
NoCarry:
```

- An alternative to branching is to write zero to the 8-bit register and to use the ADC instruction. This adds zero plus the carry flag to the MSB.
- Do not use CLR for writing zero to the 8-bit register, that would clear the carry flag, too.

16-by-16 bit addition and subtraction

- **Adding a 16 bit number to a 16 bit number is also simple:**

```
; Adding a 16-bit number to a 16 bit number
      LDI R16, Low(1028) ; Load the LSB of 1028 to a register
      LDI R17, High(1028) ; Load the MSB
      LDI R18, Low(32700) ; Load the LSB of 32700 to a register
      LDI R19, High(32700) ; Load the MSB
      ADD R16, R18 ; First add the two LSB registers
      ADC R17, R19 ; Then add the MSBs and the carry
```

- **This can be extended to more bits: just use ADC for all higher bytes.**
- **Subtraction is similar: Use SUB for the LSB and SBC for all higher bytes to be subtracted with the carry.**
- **What if the numbers are signed? Signed numbers use bit 7 of the highest byte as sign bit: if it is zero, the number is positive, if it is one, the number is negative and the number has been subtracted from 256 (8 bit) or 65,536 (16 bit).**

Addition of negative numbers

- **As negative numbers are stored as their negative value (256 – number) adding two numbers, of which the second is negative, is simple:**

Register	Decimal	Signed	Binary
R16	100	0x64	0b0110.0100
R17	-20	0xEC	0b1110.1100
ADD R16, R17			
R16	80	0x50	0b0101.0000

- **Nothing has to be corrected.**
- **In assembler negative 16-bit numbers are handled like this:**

```
; A negative number in Z
      LDI ZH, High(-20) ; MSB of the negative number
      LDI ZL, Low(-20) ; LSB of the negative number
```

Register

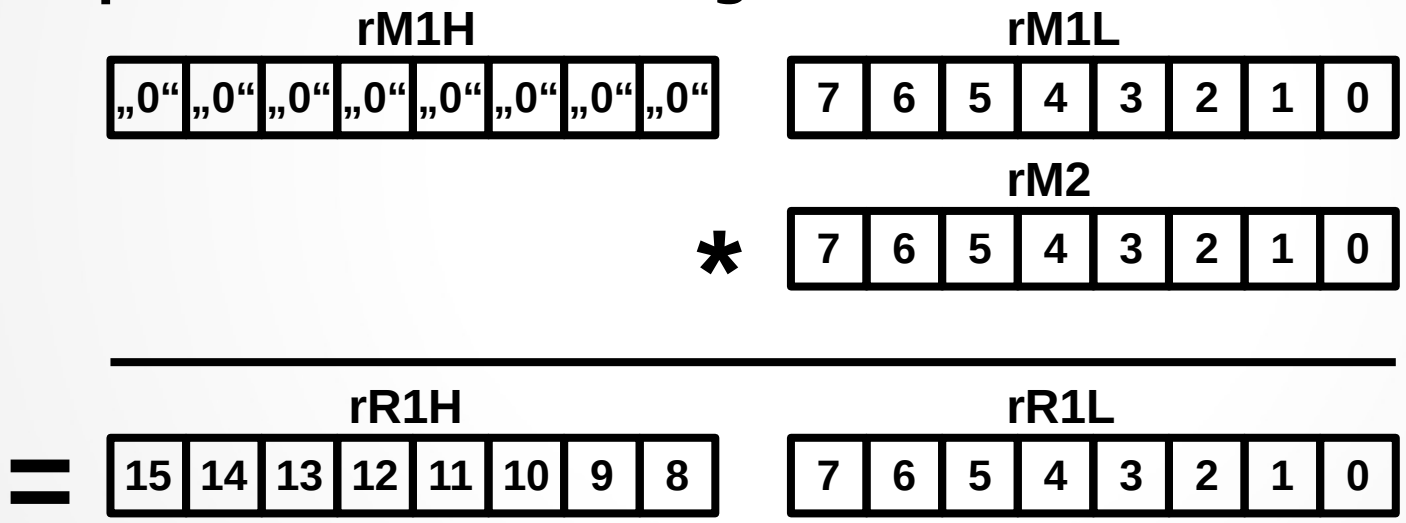
Reg	+0	+1	+2	+3	+4	+5	+6	+7
R0	00	00	00	00	00	00	00	00
R8	00	00	00	00	00	00	00	00
R16	00	00	00	00	00	00	00	00
R24	00	00	00	00	00	00	EC	FF

Multiplication 8 x 8

- **Multiplication of 8 bits by 8 bits can yield a 16 bit result. It can be done in two ways:**

  1. **by software,**
  2. **by hardware (in ATmega and in some advanced ATtiny).**

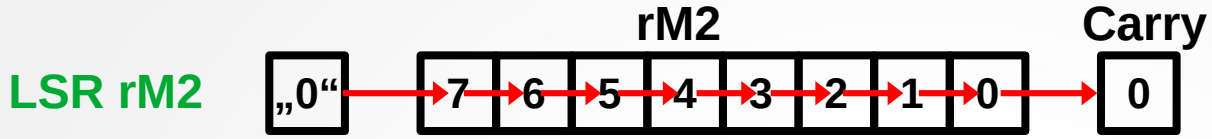
- **Multiplication in software goes like this:**



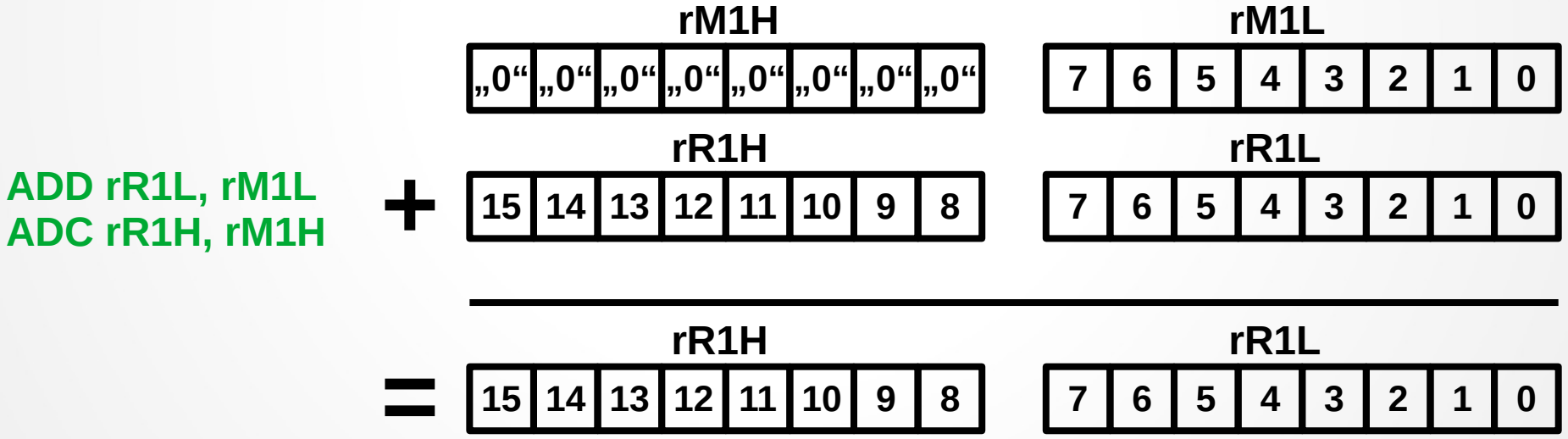
First, rM1L and rM2 are loaded with the numbers to be multiplied. rM1H as well as the result registers rR1H:rR1L are cleared.

Software multiplication 8 x 8

- **In the first step rM2 is shifted right.**



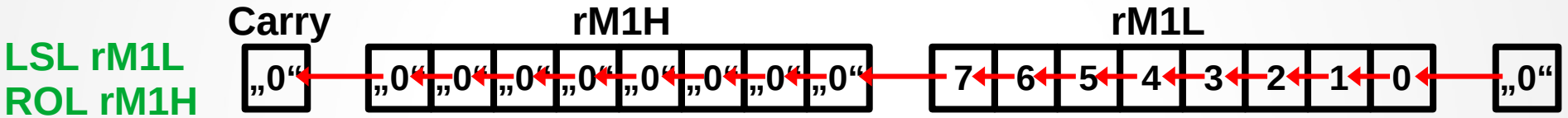
- **That shifts a zero to its bit 7, shifts its bits 7 to 1 one position to the right and its former bit 0 to the carry flag.**
- **If the carry flag is one, it adds rM1H:rM1L to the result registers:**



If not (BRCC), no addition is done and it is jumped over.

Software multiplication 8 x 8

- Now it is tested (TST rM2) if the multiplicator is already clear. If that is the case, multiplication is done and the result is already final and correct.
- If not the multiplicator rM1H:rM1L is multiplied by two, shifting rM1L one left and rotating the carry flag into rM1H.

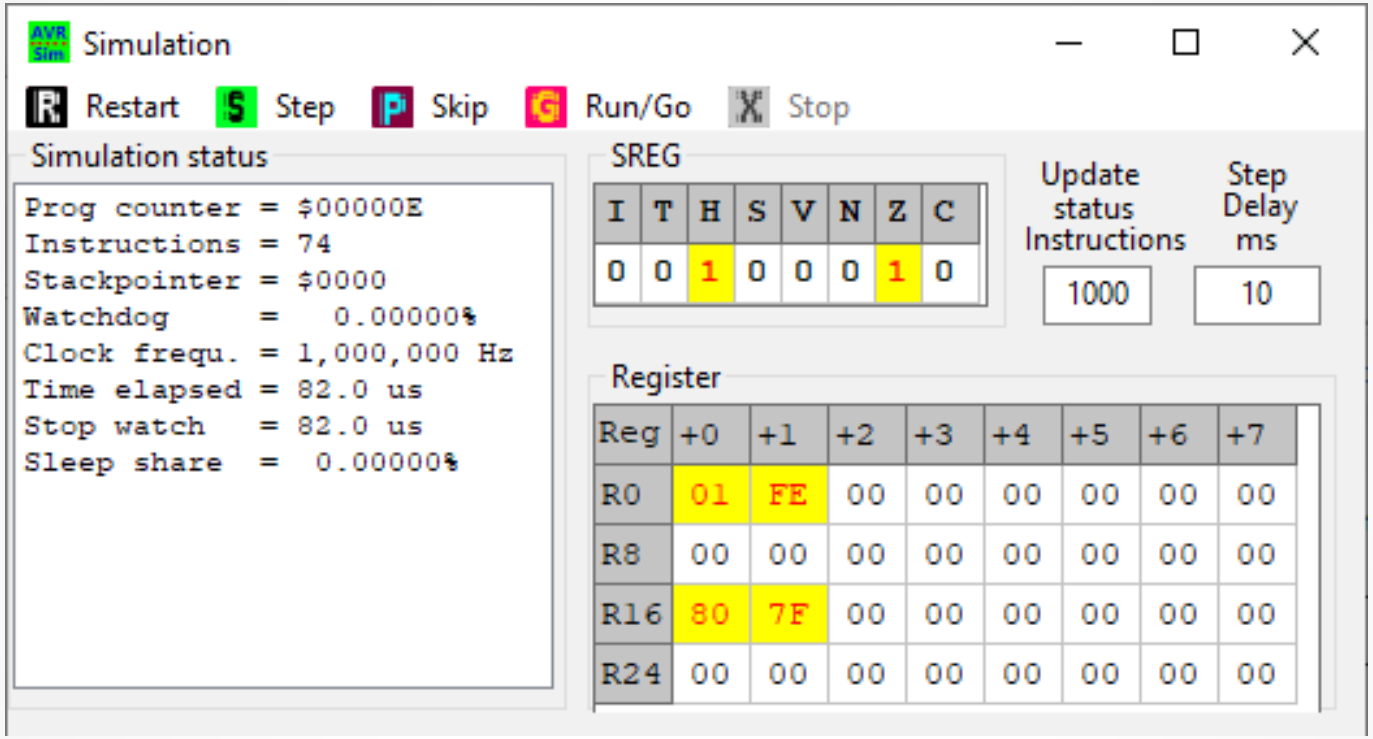


rM1H:rM1L now look like this:

- These steps are repeated until all ones in rM2 are treated and rM2 is empty.

Multiplication 8 x 8

- **The process, as simulated with the maximum numbers 0xFF, consumes 82 µs and the result in R1:R0 is correct:**



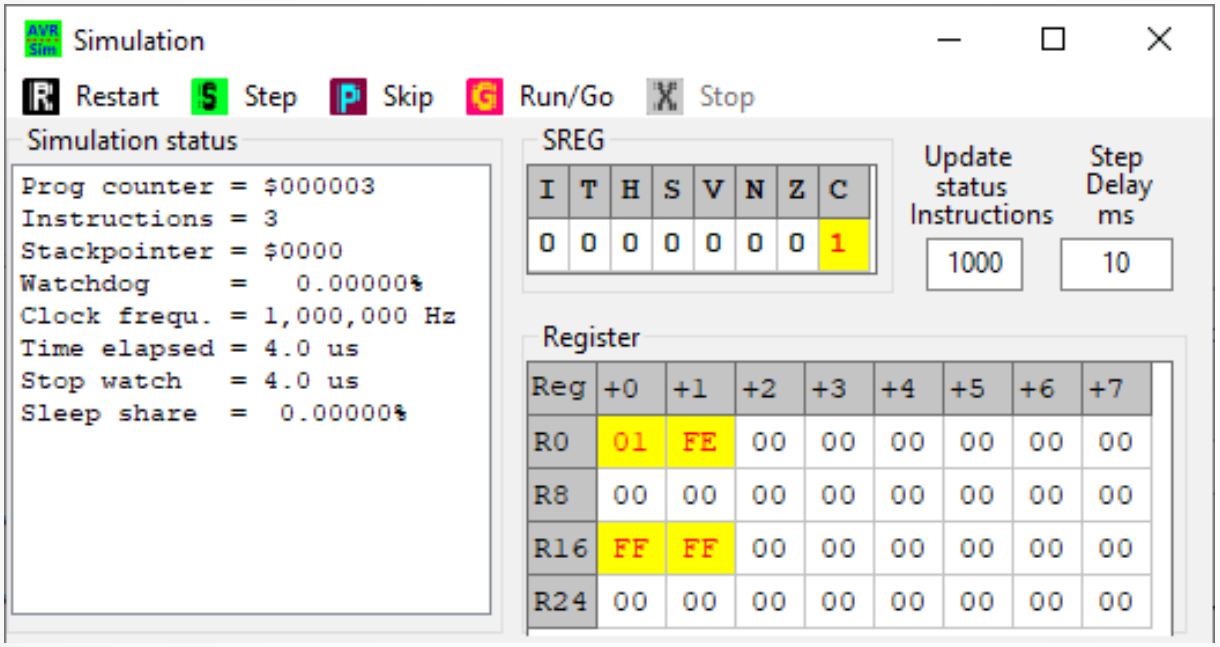
- **Therefore, hardware multiplication is unnecessary unless you cannot affort this short period. So the ATtiny24, that has no hardware multiplication on bord, is sufficient doing software multiplication.**

Multiplication 16 x 8

- **Extending this scheme to multiplying 16 bits by 8 bits is simple: just add another rM1 register to the left, such as rM1S for super, and add another result register to the left. Now addition requires an additional ADC and shifting an additional ROL. That is it.**
- **The same with 24 by 8 or 32 by 8 or 64 by 8. Add registers that the AVR has plenty of.**
- **Extending the scheme to 16 by 16 is also simple: add another rM2H to the left and add two super registers to rM1 and rR1 (the result can now have 16 + 16 = 32 bits). Shifting the lowest bit to carry involves an additional LSR and the previous LSR is a ROL now. Addition adds two ADCs, multiplication of rM1 by two now has then two additional ROLs.**
- **If you have understood how the 8-by-8 multiplication works, you can do whatever multiplication.**

Hardware multiplication 8 x 8

- **The hardware multiplication is even simpler and faster: just add the instruction MUL rM1, rM2 (if your AVR type has that) and the result is in R1:R0.**



- **Setting the two registers to 0xFF and MUL needs only 4 µs, if the hardware multiplicator is used.**

Hardware multiplication 16 x 8

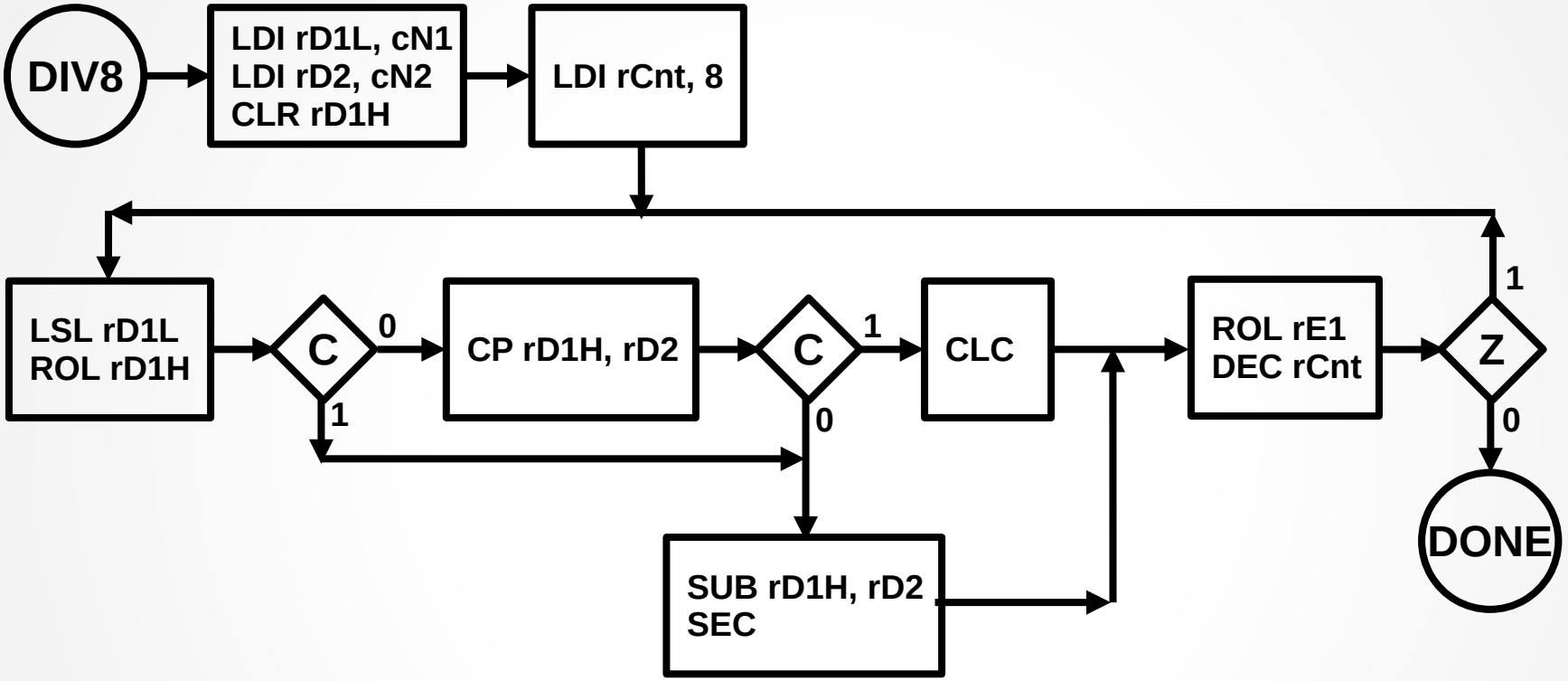
- **Extending the hardware multiplication to 16 by 8 bits is not that simple, as it requires not only one step but two.**
- **The 16 bit registers rM1H:rM1L * rM2 have to be multiplied using the following scheme:**

  1. **Multiply rM1L with rM2 and copy the result in R1:R0 to the result registers in rR1S:rR1H:rR1L to rR1H:rR1L and clear rR1S.**
  2. **Multiply rM1H with rM2 and add the result in R1:R0 to the result registers rR1S:rR1H (the result registers by that are multiplied by 256).**

- **Extending that scheme to 16-by-16 involves four multiplications:**
  rM1H:rM1L * rM2H:rM2L = rM1L*rM2L + 256*rM1H*rM2L +
  256* rM1L*rM2H + 65536*rM1H*rM2H
  The multiplication with 256 and 65536 is done by just adding R1:R0 to the respective and correct upper registers.

Division 8 x 8

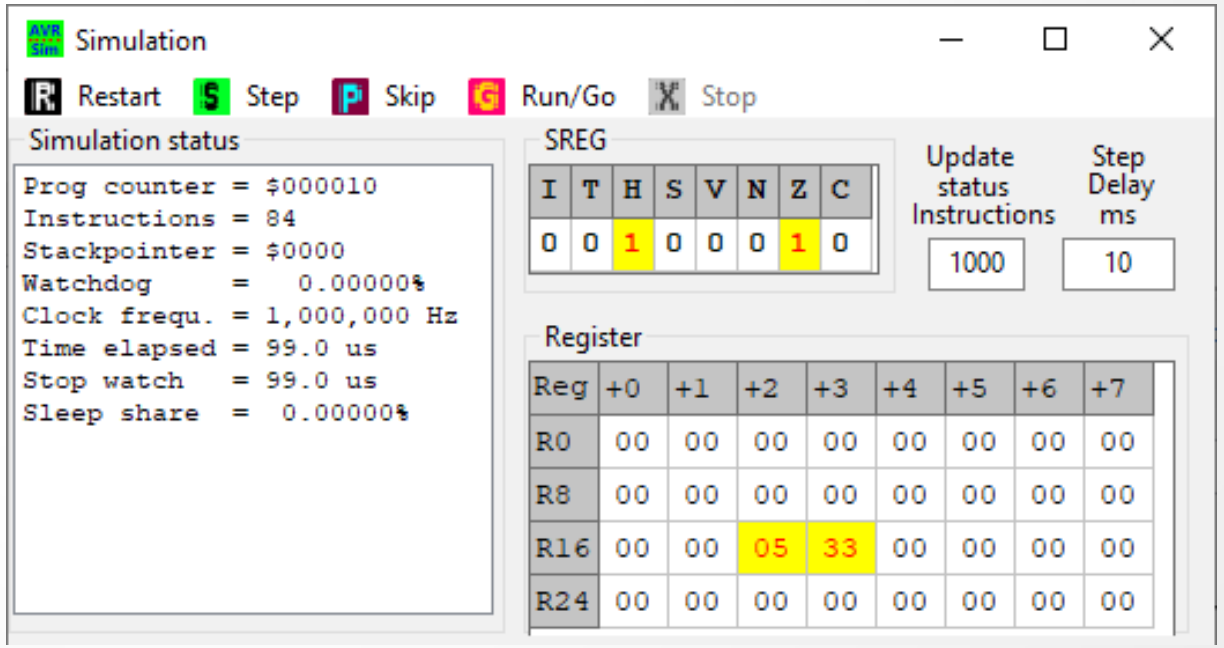
- **Division is also rather simple. It starts with loading the registers and setting a counter to 8.**



Then the divider registers are multiplied by 2. If the carry is one, the high byte of the divider is surely larger than the divisor: the divisor is subtracted from the high byte and a one is left-rotated into the result. If not, the high byte is compared with the divisor.

Division 8 x 8

- If the divisor is larger than the high byte (carry after compare is set), the carry is cleared and left-shifted into the result. If not, the divisor is subtracted from the high byte and a one is left-shifted into the result.
- In all cases the counter is decremented. If he reaches zero, the division is done. If not is the cycle repeated.
- Dividing 255 by 5 requires 99 µs, slightly longer than multiplication. The result in R19 is 0x33, which is decimal 51 and obviously correct.



- Note that the result is always an integer, not a fractioned number.

Division of larger numbers

- **Extending division to 16 by 8 requires**
  1. **extension of the divider to three bytes,**
  2. **extension of the result to two bytes, and**
  3. **16 division steps.**
- **Extension to 16 by 16 requires additionally the extension of the divisor to two bytes.**
- **As general rules: the divider always has to be extended by the length of the divisor in bytes, the counter always has to be set to the length of the divisor in bits.**
- **Note that in division the first step is alway the multiplication of the divider by two, while in multiplication this step is done as last step.**
- **If you need rounding: if the next division step yields a one, add 1.**

Conversion of a binary to hexadecimal

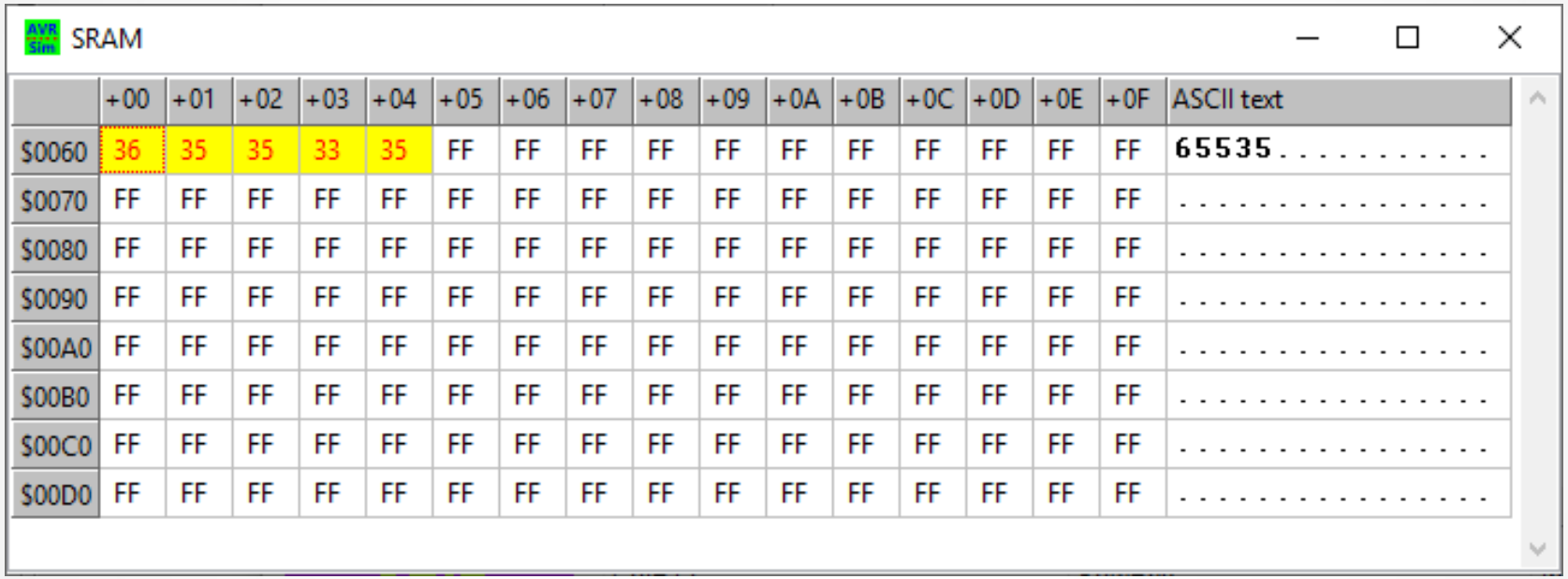
- If you need conversion of a byte to a hexadecimal format to display those (e.g. on a LCD) first convert the upper four bits (upper nibble), then the lower four bits. Use the instructions
  - SWAP R16 to exchange both nibbles in R16,
  - ANDI R16, 0x0F to isolate the lower nibble in R16,
  - SUBI R16, -‘0‘ to convert the binary to a hexadecimal character, and
  - SUBI R16, -7 in case that the binary nibble is larger than ‘9‘.
- It is convenient to PUSH R16 to the stack, then to SWAP and RCALL the nibble conversion, then to POP the original and doing nibble conversion.
- If two or more bytes have to be displayed, copy those to R16 and RCALL the byte conversion. (See the hex conversion here).

Conversion of a binary to decimal

- If you need conversion of a byte to decimal format to display those (e.g. on a LCD) the following algorithm can be used.
- First subtract binary 100 (0x64) until a carry occurs. These are the hundreds. Add 100 to undo the last subtraction.
- Then subtract binary 10 (0x0A) until a carry occurs, which are the tens. Add 10 to undo the last subtraction.
- The remaining rest of the number is the last digit.
- If you need zero suppression to blanks set a flag (e.g. T in SREG) and exchange zeroes by blanks, as long as the flag is set. If a non-zero digit occurs clear the flag. Do not use the flag with the last digit.
- Extending the conversion to 16 bit numbers: start with binary 10,000, then with binary 1,000 and the lower ones and use two bytes for subtraction.

Conversion of binaries to decimal

- As an example: here is the source code of the decimal conversion program that uses a table with the decimal binaries to write the binary to SRAM.
- The picture shows the conversion of hexadecimal FFFF to decimal.

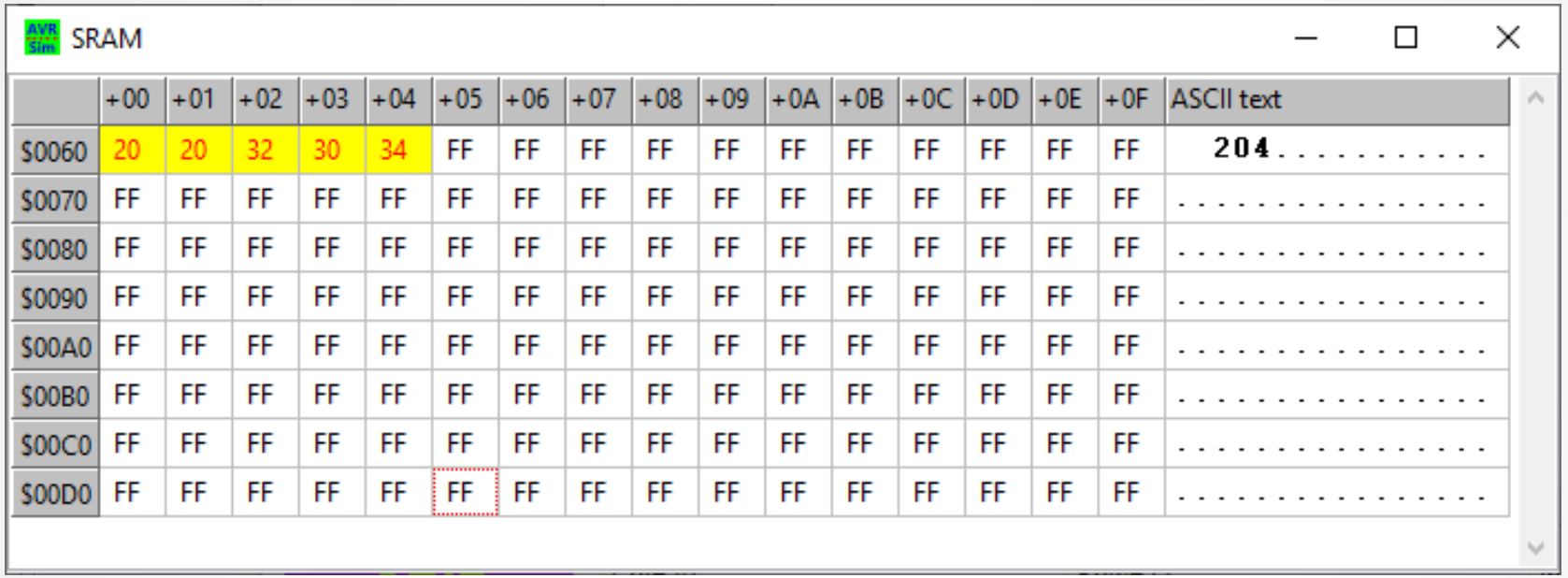


	+00	+01	+02	+03	+04	+05	+06	+07	+08	+09	+0A	+0B	+0C	+0D	+0E	+0F	ASCII text
$0060	36	35	35	33	35	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	65535. . . . .
$0070	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	. . . . .
$0080	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	. . . . .
$0090	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	. . . . .
$00A0	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	. . . . .
$00B0	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	. . . . .
$00C0	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	. . . . .
$00D0	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	. . . . .

- The execution lasted 206 µs.

Conversion of binaries to decimal

- **This is the result if 0x00CC is converted. Leading zeroes are blanked, later appearing zeros are not blanked.**



	+00	+01	+02	+03	+04	+05	+06	+07	+08	+09	+0A	+0B	+0C	+0D	+0E	+0F	ASCII text
$0060	20	20	32	30	34	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	204 . . . . .
$0070	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	. . . . .
$0080	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	. . . . .
$0090	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	. . . . .
$00A0	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	. . . . .
$00B0	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	. . . . .
$00C0	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	. . . . .
$00D0	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	. . . . .

- **In that case conversion needed 127 µs.**

Questions and tasks in Lecture 10

Task 10-1: Write a program that sets and then adds and subtracts two 24-bit numbers.

Bonus question: How many registers would a 64-by-64 bit addition need and what would be the largest numbers than can so be added/subtracted? Can all living individuals in the world be tagged with a unique 64 bit number? Who would then get number 0 in your view?

Questions and tasks in Lecture 10 - Continued

Task 10-2: Write a program that sets and then multiplies two 16-bit numbers.

Bonus question: How many registers would be needed to multiply two 64-bit numbers? Can that be handled in the 32 registers in AVRs?

Questions and tasks in Lecture 10 - Continued

Task 10-3: Convert the number 12,345,678 from binary to decimal with leading-zero-blanking.