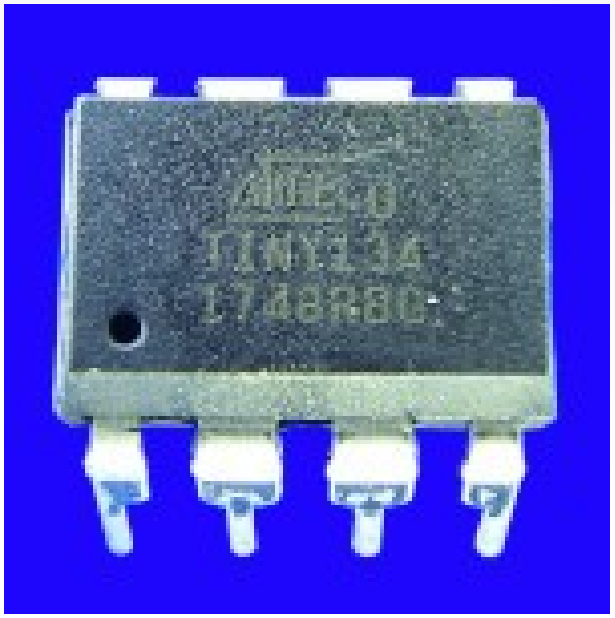


Lecture 3: Counting

Hardware, Internals and Programming of AVR Microcontrollers in Assembler

by
Gerhard Schmidt
Kastanienallee 20
D-64289 Darmstadt

The registers

- The AVRs have 32 registers, named R0 to R31. Use the .DEF directive to give them a more meaningful name and use that in the source code:

```
; Renaming a register
.DEF rMyReg = R16
      LDI rMyReg, ‘A‘ ; Load register MyReg with the ASCII character A
      INC rMyReg ; Register MyReg plus one, result is a ‘B‘
```

- Registers can hold any type of 8-bit information and data: numbers from 0 to 255 (hexadecimal: 0xFF), characters or just eight single bits. No type definition is required (if it’s ASCII: just keep this in mind for yourself), no restrictions apply when using this data for any purpose (copy, add, subtract, multiply etc.).
- Special functions of registers are:
  - ➔ R0 is the target of the Load from Program Memory instruction LPM (without any parameters).
  - ➔ R1:R0 is the 16-bit register pair where the result of hardware multiplications with MULT is written to (R1 is MSB, R0 is LSB).
  - ➔ R0 to R15 cannot be targets of instructions that require an 8-bit constant (LoaD Immediate LDI, Set or Clear Bit in Register SBR/CBR, SUBtract Immediate SUBI, AND Immediate ANDI, OR Immediate ORI). R16 to R31 can do that.

The registers

- **Special functions of registers (continued):**
  - ➔ **R27:R26 = XH:XL = X, R29:R28 = YH:YL = Y, R31:R30 = ZH:ZL = Z are 16-bit register pairs that can be used as address pointers to LoaD LD or STore ST registers to SRAM.**
  - ➔ **These three pairs, as well as the register pair R25:R24 (no special name), can be added 16-bit-wise with a constant ADd Immediate Word ADIW or SuBtracted Immediate Word SBIW.**
  - ➔ **R29:R28 = Y and R31:R30 = Z can temporarily add a displacement to LoaD Displaced LDD or STore Displaced STD bytes.**
- **Therefore and because of source code transparency and overview the placing of the data to the appropriate registers is vital for assembler. No other (higher level) language requires this thorough register planning. The result is more effective binary code and increased elegancy.**

The register landscape

- **The landscape of registers in AVRs therefore looks like this:**

Reg	+0	+1	+2	+3	+4	+5	+6	+7
R0	00	01	02	03	04	05	06	07
R8	08	09	0A	0B	0C	0D	0E	0F
R16	10	11	12	13	14	15	16	17
R24	18	19	1A	1B	1C	1D	1E	1F

#	Unrestricted	#	LPM	#	16-bit pointer
#	No immediate	#	MULT	#	ADIW SBIW

- **Keep this mind when placing data to registers. And always place them with a .DEF directive, so you can change the .DEF associations to registers easily.**
- **In PICs there is only one register, so this choice is unique for AVRs.**

Counting with the controller

- **The plenty registers of an AVR can be used to count up or down.**

```
; Counting a register up
      CLR R16 ; Set register number 16 to zero
CountUp:
      INC R16 ; Count the register content up
      RJMP CountUp ; Jump back to the label CountUp
```

- **We'll simulate this to see what is going on inside. Start a new project in the simulator named „Count", select an ATtiny24.**
- **Add the above lines behind the „Main:" line and assemble.**
- **Either step through your program, until you reach R16=255. Or:**

With Run/Go or Ctrl-G see how the time advances.

- **CLR needs 1 µs.**
- **Each INC needs 1 µs.**
- **Each RJMP needs 2 µs.**

To increase execution speed remove one zero from the 10,000.

Execution times of the loop

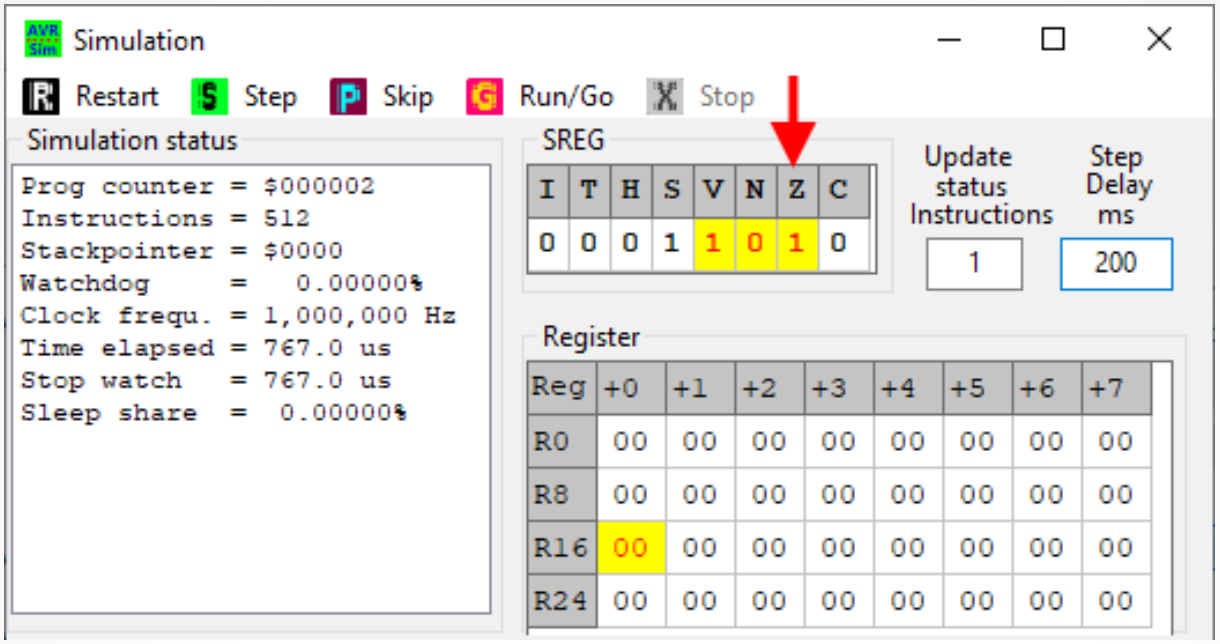
- If R16 reaches 0xF8 press „Stop“ or Ctrl-X.
- Then press „Step“ until R16 reaches 0xFF.
- The initial CLR needed one clock cycle, all the INCs needed 255 and all the RJMPs needed 2*255 = 510 clock cycles. Alltogether 766 clock cycles. At 1 MHz that means 766 µs.
- What we learn from that:

  1) All execution times for all instructions have exactly predictable durations.
  2) No hidden delay times or unknown factors affect this.
  3) That‘s what you get with assembler language exclusively.

- Where to know these cycles from? See the instruction list here, column „Clk“ or ATMEL’s Instruction Set Manual.

Status register flags

- With the next „Step“ the counter reaches zero and restarts at zero.



- The CPU has set the Z flag (Z = Zero) in its status register (SREG) to signal that the INC ended with Zero in the register.
- The Z flag can be used to branch conditionally: branching if the Z flag is set or clear.
- There are eight flags in SREG, six of them are effected by the CPU. Which? See the instruction list here or in ATMEL’s Instruction Set Manual.

Conditional branching

- The conditional branching depending from the Z flag is done with BREQ or BRNE (Mnemonics for BRanch if EQal and BRanch if Not Equal). See other branches in the device‘s databook or here.
- These can be used to construct loops: repeating counting until zero will be reached.

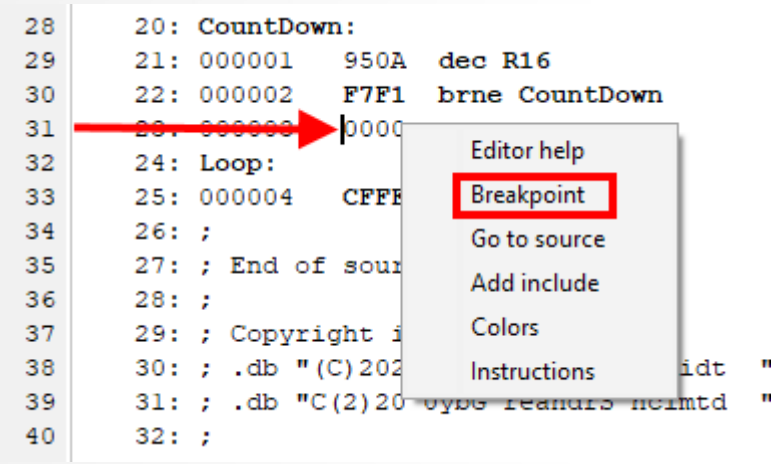
```
; Counting a register down until it is zero
      LDI R16,10 ; Set register number 16 to ten
CountDown:
      DEC R16 ; Count the register content one down (DECrease)
      BRNE CountDown ; Jump back to the label CountDown if not zero
      NOP ; Do nothing
```

- The method of repeating „Step“ until the counter reaches zero can be simplified by setting a Breakpoint.
- Breakpoints are executable lines of code, where execution stops before the code in this line is executed.

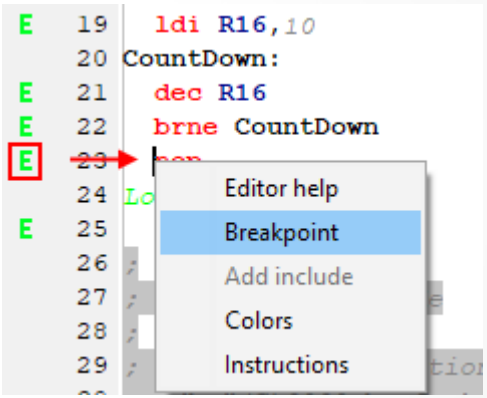
Breakpoints

- **To set a breakpoint, first assemble the source code.**

In the listing place the cursor into the line where execution should stop. Then right-click and select „Breakpoint“ from the context menu.



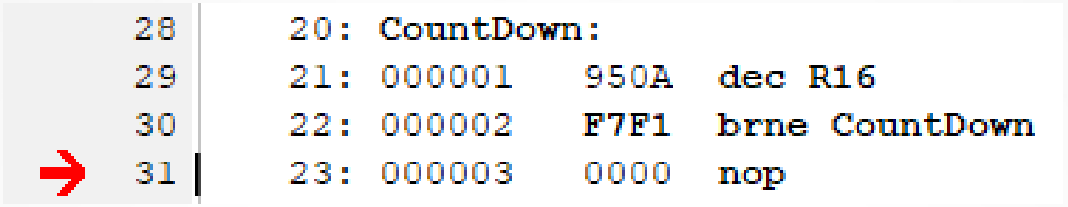
In the source code place the cursor into the line where execution should stop. Make sure that this line has a green E. Then right-click and select „Breakpoint“ from the context menu.

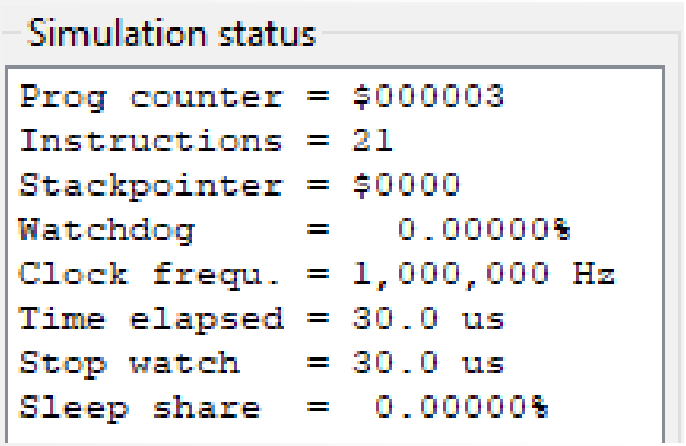


- **The breakpoint is now marked with a red „B“ in the listing as well as in the source code.**
- **Whenever this breakpoint will be executed as next instruction, it will stop the running simulation.**

Loop execution ended

- **When simulating with Run/Go the simulator stops exactly at the breakpoint and the color of the execution arrow turns from blue to red.**





- **The time required for that loop can be read from the lines „Time Elapsed“ or „Stop watch“ in the simulation status window.**
- **It is**
  - **one clock cycle for LDI, plus**
  - **nine loop executions ending with the Z flag not set = one for DEC and two for BRNE, plus**
  - **the last loop with one for DEC and BRNE.**
- **Together 30 clock cycles have been elapsed.**
- **The formula is:**
  Clock cycles = 1 + 3 * (N – 1) + 2 = 3 * N

Instruction duration and flags

- **The durations of the instructions can be read from three documents:**

  a) **From the instruction set manual: Link**

  b) **From the datasheet for each controller (Chapter Instruction Set Summary): Link ATtiny24**

23. Instruction Set Summary

Mnemonics	Operands	Description	Operation	Flags	#Clocks
ARITHMETIC AND LOGIC INSTRUCTIONS					
ADD	Rd, Rr	Add two Registers	Rd ← Rd + Rr	Z,C,N,V,H	1
ADC	Rd, Rr	Add with Carry two Registers	Rd ← Rd + Rr + C	Z,C,N,V,H	1
• • •					
INC	Rd	Increment	Rd ← Rd + 1	Z,N,V	1
DEC	Rd	Decrement	Rd ← Rd – 1	Z,N,V	1
TST	Rd	Test for Zero or Minus	Rd ← Rd • Rd	Z,N,V	1
CLR	Rd	Clear Register	Rd ← Rd ⊕ Rd	Z,N,V	1
SER	Rd	Set Register	Rd ← 0xFF	None	1
BRANCH INSTRUCTIONS					
RJMP	k	Relative Jump	PC ← PC + k + 1	None	2

c) **The instruction list here also provides this information.**

Nested loops

- **If you need longer times for loops: loops can be nested.**

```
; Defining loop counts
.equ cOuter = 200 ; the outer loop count
.equ cInner = 100 ; the inner loop count
;
; Defining register names
.def rOuter = R16 ; defining the register for the outer loop
.def rInner = R17 ; defining the register for the inner loop
;
; Starting the outer loop
      LDI rOuter,cOuter ; Set register rOuter to the cOuter count value
LoopOuter:
      LDI rInner,cInner ; Set register rInner to the cInner count value
LoopInner:
      DEC rInner ; Decrease inner loop counter
      BRNE LoopInner ; Jump back to the inner label if not zero
      DEC rOuter ; Decrease outer loop counter
      BRNE LoopOuter ; Jump back to the outer label if not zero
; Long loop done
      NOP ; For setting breakpoints
```

Assembler directives

- The source code now uses so-called directives:
  - All directives start with a dot character „.“.
  - They are meant for the assembler only and produce no executable code.
  - .EQU name = value defines a symbol constant with a name and sets it to the value given or calulated (calculation rules here).
  - .DEF name = Rn defines a symbol name for the register Rn (n = 0 .. 31).
- The advantage of using symbols are:
  - The names are easier to remember than register numbers (rInner and rOuter remind yourselve of the purpose they are used for).
  - Changing the two constants or placing the registers somewhere else is easier than having to go through the complete source code.

Execution time of nested loop

- The outer loop executes one inner loop.
- So first the execution time of the inner loop is:

```
; Inner loop
        LDI rInner,cInner ; One clock cycle
LoopInner:
        DEC rInner ; One clock cycle
        BRNE LoopInner ; Two clock cycles when jumping, one if not
```

- The formula for the inner loop is:

  Clock cycles = 1 + 3 * (cInner – 1) + 2 = 3 * cInner

- The outer loop is like that:

```
; Outer loop
        LDI rOuter,cOuter ; One clock cycle
LoopOuter:
        ; (Inner loop), 3 * cInner clock cycles
        DEC rOuter ; One clock cycle
        BRNE LoopOuter ; Two clock cycles when jumping, one if not
```

- Its formula is:

  Clock cycles = 1 + (cOuter - 1) * (3 * cInner + 3) + 3 * cInner + 2

The nested loop, continued

- **Clock cycles = 1 + (cOuter - 1) * (3 * cInner + 3) + 3 * cInner + 2**
- **Clock cycles = 1 + cOuter * (3 * cInner + 3) – (3 * cInner + 3) + 3 * cInner + 2**
- **Clock cycles = 1 + 3 * cOuter * cInner + 3 * cOuter – 3 * cInner - 3 + 3 * cInner + 2**
- **Clock cycles = 1 + 3 * cOuter * cInner + 3 * cOuter - 1**
- **Clock cycles = 3 * cOuter * cInner + 3 * cOuter**
- **For cOuter = 200 and cInner = 100:**
  Clock cycles = 3 * 200 * 100 + 3 * 200 = 60,600
  @1 MHz: 60.6 ms
- **At maximum: cOuter = 256, cInner = 256**
  Clock cycles = 3 * 256 * 256 + 3 * 256 = 197,376
  @1 MHz: 197 ms
  (Do not expect that LDI rOuter,cOuter assembles correct if you .EQU cOuter=256. What else to correct that error?).
- **With that, we can expand the two-stage nested loop to a three-stage nested loop and achieve the desired 500,000 clock cycles delay.**

The next lecture: the seconds blink

- Now with one ATtiny24 (14-pin) we have constructed the following:
  - An oscillator with 1 MHz (with the internal RC oscillator of 8 MHz, by default divided by 8),
  - Two 8 bit counters counting down from selectible initial values down to zero.
- One ATtiny24 therefore replaces the following CMOS ICs here:
  - one 4093 as oscillator, 14 pins,
  - four 4516 4-bit presetable binary down-counters, 4 * 16 = 64 pins.
- A good argument why a controller is the better choice. And:
- Why learning assembler is so much better than C: you'll keep anything under your own control and no compiler interferes with your calculations and places unknown and unnecessary instructions into your execution code.

Questions and Tasks for Lecture 3

Question 3-1: What happens if your program does not jump back

a) in the simulator, or

b) inside the controller?

(Hint for b: The flash memory CANNOT be empty, it is always 0xFFFF if cells are not programmed.)

Bonus question: What makes the difference between a hardware reset to 0x0000 and a simple jump to address 0x0000? (Hint: Try to simulate that by setting some register values in between. A hardware reset in the simulator is forced with the Restart menu entry! A jump to 0x0000 can be done with RJMP 0).

Questions and Tasks for Lecture 3, Continued

Task 3-2: Go through the instruction set summary and find out if there are INC and DEC instructions for 16 bits. What are their limitations or use conditions?

Questions and Tasks for Lecture 3, Continued

Task 3-3: Find the differences between DEC R16 and SUBI R16,1 in respect to

a) their execution time, and

b) the affected flags in the status register.

Bonus question: There is no ADDI instruction listed in the instruction set. How can you then increase a register‘s content with the same flag settings like in SUBI? (Try to simulate your solution and see if the flags Z and C are set/cleared correct.)