Embedded Systems Programming

Lecture 9 – Code optimization and Performance

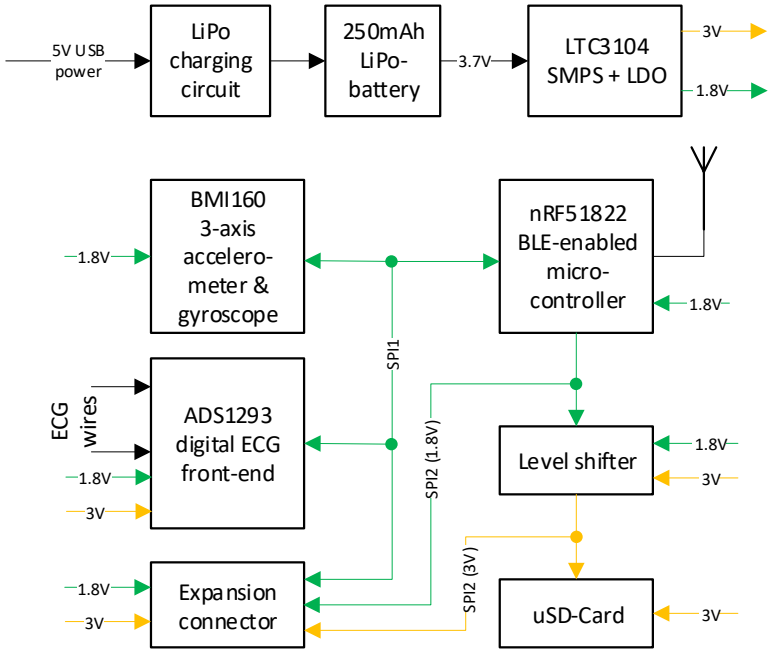
Jarno Tuominen

Lecture 9 – Code optimization and performance

- Review
- Code optimization and power consumption

Introduction – the sensor node

- Based on ultra low-power ARM Cortex M0 CPU core with Bluetooth radio, running at 16MHz. The device runs with a single coin-cell battery.
- One of the applications is ECG-monitoring/data acquisition, which is made possible by connecting an ECG-front-end IC (Texas Instruments ADS1293) to a CPU's SPI-bus.
- The device can log raw data from the sensors to an in-built microSD-Card (with time-stamping) – and the data can be further analyzed with Matlab, for example.

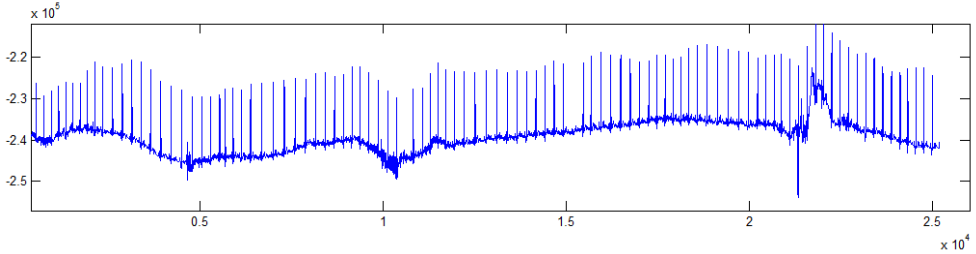


The problem setting

- The amount of data from the sensors correlates directly with power consumption of the system.
  - If data is stored to a memory card, higher data rate will yield to more frequent flash write cycles with high peak current
  - If data is transmitted over Bluetooth, high data rate will mean that the radio parts are powered on more often, to provide enough data channel bandwidth
- Thus, there's a high demand to perform data analysis in the sensor node – that is where DSP takes place.
- In this project we focus on ECG signal filtering, which enables further analysis and processing of ECG data.
- Power consumption is a trade-off between amount of data stored/trasferred and amount of CPU cycles used to preprocess/compress the data.

Raw ECG Signal

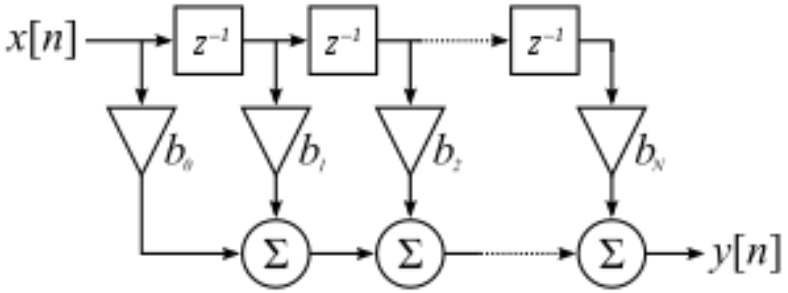
- ECG signal sampling frequency is 267Hz, which has already gone through a downsampling by 5, so effective signal BW is 55Hz. In practice, there is not much information above 40Hz frequencies.
- ECG signal has strong DC baseline wandering
- -> Bandpass filter required, with corner frequencies at 1Hz and 40Hz



RAW ECG, sampled and stored to a memory card (Test Subject: J. Tuominen)

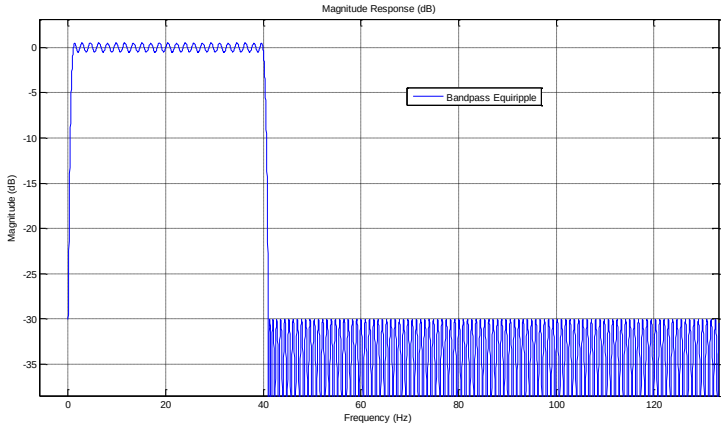
ECG Filter design

- The Cortex M0 CPU core is a single threaded CPU, which does not have single cycle MAC operation, so we can expect challenges with high filter tap count.
- At this point, the focus was mainly on fixed point, finite word length filter impelementation and the practical performance of the filter with the available limited processing capability.
- In final application, the filter type will be IIR, but in this project work FIR was implemented (in C), as the tap count will considerably higher, giving us a better view on processing capacity limitations



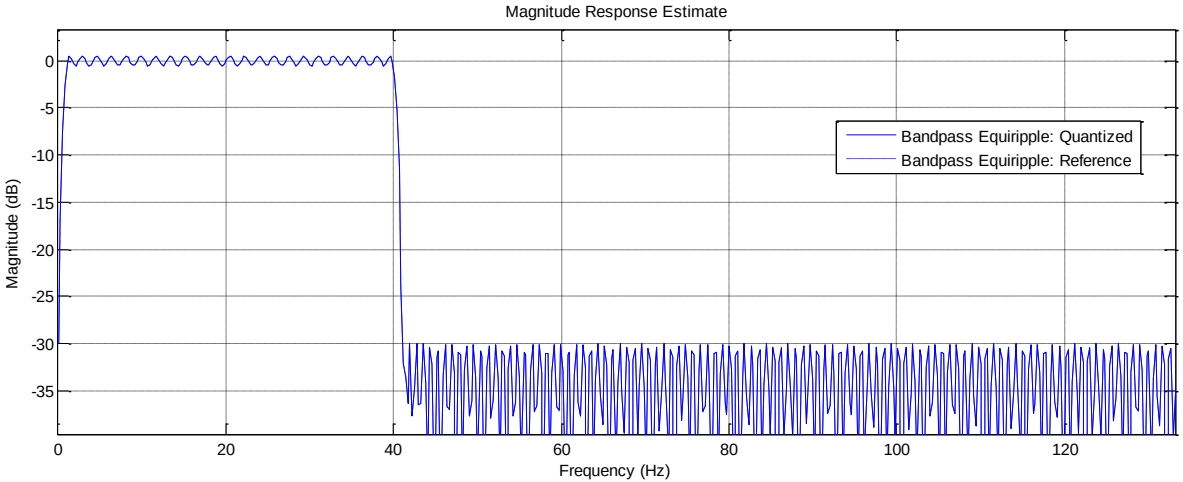
ECG Filter design (2)

- The specs for the filter are:
  - Fs: 267Hz, Fstop1; 0.01Hz, Fpass1: 1Hz, Fstop1: 40Hz, Fstop2: 41Hz, Apass 1dB, Astop 30dB.
- Using equiripple FIR implementation, with 308 taps we get:



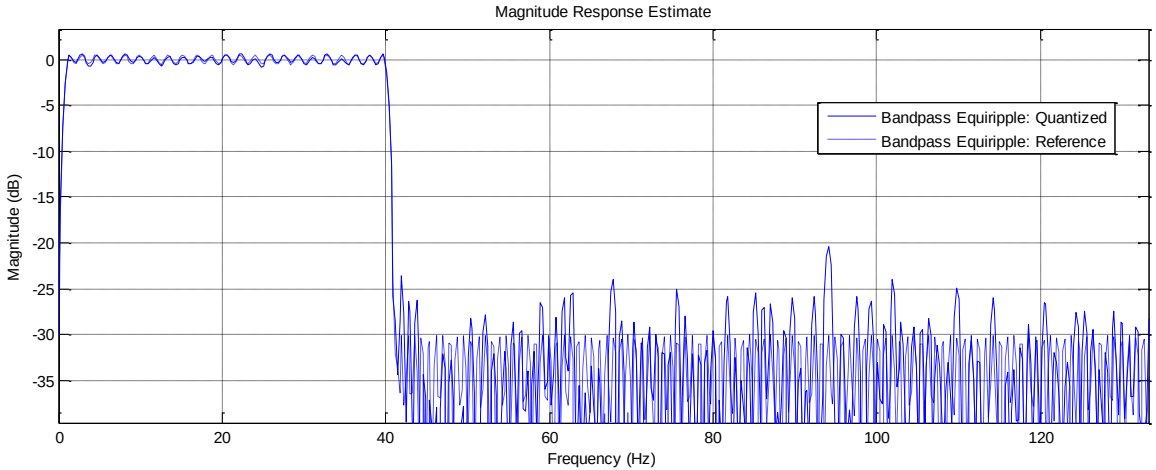
ECG Filter design (3)

- Even though ARM Cortex M0 has 32x32bit single-cycle multiplier, the coefficients were quantized to 16-bit to save run-time memory. As expected, FIR tolerates quantization very well:

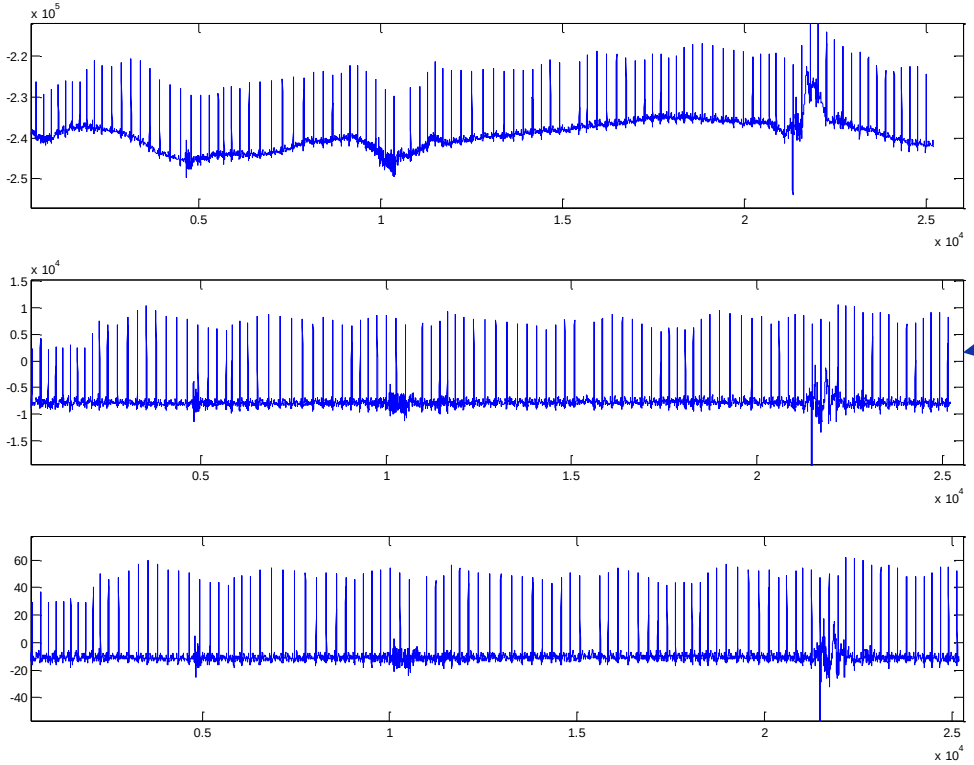


ECG Filter design (4)

- When quantizing the coefficients even more, to 8 bits, we start to see degradation, specifically the stop band attenuation gets much worse:



Filter test results on ECG signal

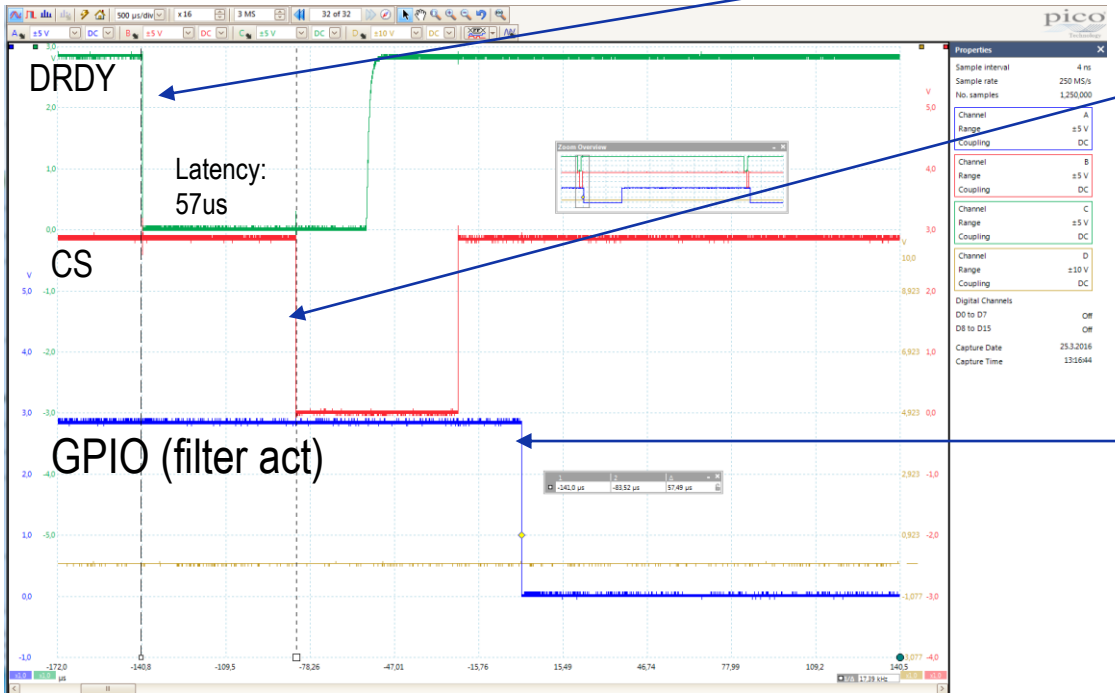


Top: original captured signal, 24-bit data

Mid: original signal filtered with Matlab, 24-bit data, double float precision coefficients

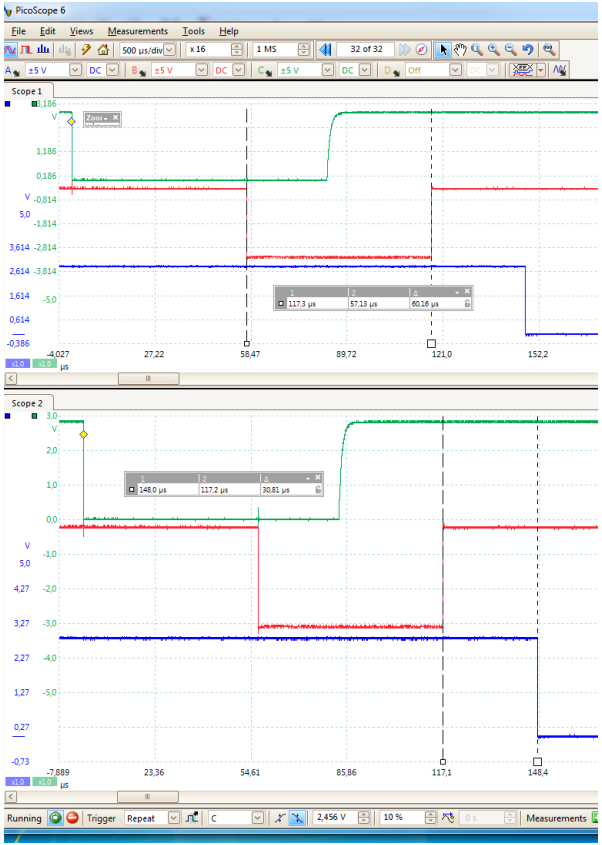
Bottom: original signal filtered real-time with ARM Cortex M0, 16-bit coefficients & data

Timing & Performance analysis



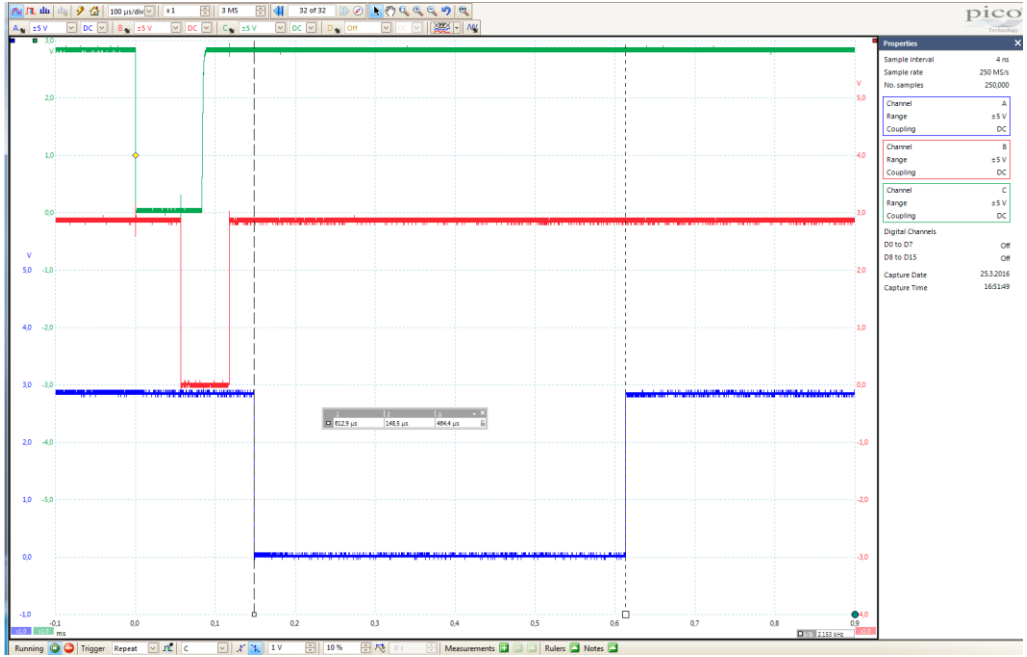
- The ECG FE-chip issues an interrupt (data ready, active low) when a new sample is available at the output register.
- CPU reacts to this interrupt by reading the sample from the chip over the SPI-bus.
- After the sample has been captured, it will stored to a ECG buffer (a FIFO)
- Additionally, each sample is filtered and stored
- Just before the filter function call, a GPIO-pin is deasserted and right after returning from the filter function, it is asserted – this way we can see the time used to execute the filter
- This cycle repeats at ECG sampling rate, every 3.7ms

SPI bus transaction, filter prep & function call



- Time consumed by SPI bus transfer (6 bytes of data): 60us.
- Time consumed for storing received data to local FIFO and executing filter function call (stack push etc). 31us
- Total time from SPI transfer start to filter execution start: 91us

Filter execution time



- Execution time for 308-tap filter (inner MAC loop only) : 464us
- Total time for processing one sample: 612us
- Where does the time go inside filter? See next slides.

SW performance analysis (FIR loop only)

```
void firFixed( const int16_t *fir_coeffs, int16_t *input, int16_t *output, int length, int filterLength )
...
  for ( uint16_t k = 0; k < filterLength; k++ ) {
    acc += *coeffp++ * *inputp--;
  }
...
```

Using ARM C Compiler, optimization level 0 (no optimization)

Address	Opcode	Parameters	Cycles	Description
0x0001E220	MOVS	r3,#0x00	1	Load 0 to r3 (i.e. Reset the multiplier argument 1 register for each loop round)
0x0001E222	LDR	r2,[sp,#0x08]	2	stack pointer relative indirect addressing,, loads the coefficient address to r2
0x0001E224	LDRSH	r3,[r2,r3]	2	The CPU is 32-bit and we are dealing with 16-bit data - so handling halfwords (16-bit) units of data, copy coeff from r2 to r3
0x0001E226	ADDS	r2,r2,#2	1	Add 2 to contents of r2 (in C: *coeffp++ , so we are increasing coeff pointer by two bytes)
0x0001E228	STR	r2,[sp,#0x08]	2	And store the coeff pointer value to address "stack pointer+8". (That's the input argument for C filter function)
0x0001E22A	MOVS	r7,#0x00	1	Load 0 to r7 (i.e. Reset the multiplier argument 2 register for each loop round)
0x0001E22C	LDR	r2,[sp,#0x04]	2	Indirect addressing (we use pointers in C!), loads the data sample to r2
0x0001E22E	LDRSH	r7,[r2,r7]	2	The CPU is 32-bit and we are dealing with 16-bit data - so handling halfwords (16-bit) units of data, copy sample from r2 to r7
0x0001E230	SUBS	r2,r2,#2	1	Subtract 2 from contents of r2 (in C: *inputp-- , so we are decreasing sample data pointer by two bytes)
0x0001E232	MULS	r3,r7,r3	1	Perform the multiplication: r3*r7, store result to r3
0x0001E234	ADDS	r6,r3,r6	1	Perform the addition: r6+r3, store result to r6 (accumulator in this case)
0x0001E236	STR	r2,[sp,#0x04]	2	Output of the filter function is pointer as well (16-bit output data)
0x0001E238	ADDS	r2,r1,#1	1	Increase the loop counter (in r1) by 1 and store it's value to r2
0x0001E23A	UXTH	r1,r2	1	zero-extend the unsigned halfword (16-bit) to 32-bit
0x0001E23C	CMP	r1,r5	1	r5 holds the loop length , check if there are still rounds to go, set T-flag if true (in C: k < filterLength)
0x0001E23E	BLT	0x0001E220	1	If "T", jump to beginning of loop, otherwise, move forward
Clock cycles			22	
Execution time @ 16MHz [us]			1,375	
Total cycles for 308 taps			6776	
Execution time for 308 taps [us]			423,5	
Measured filter execution time			465	Includes approx 40us overhead used for GPIO function calls etc.

From the disassembly, we see that only 2 clock cycles are used for MAC, the rest 20 cycles are used for data fetching and loop control

http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0432c/CHDCICDF.html

SW performance optimization (FIR loop only)

- First, let us use C-compiler optimization level 3 (highest). This makes the code very difficult to debug (in C-domain) and and the assembler can get quite messy – although in this case the compiler does excellent job:

0x0001E204	MOVS	r6,#0x00	1	Load zero offset (to be used later)
0x0001E206	MOVS	r7,#0x00	1	Load zero offset (to be used later)
0x0001E208	LDRSH	r6,[r5,r6]	2	load coefficient to r6 from memory address contained in r5 (coeff pointer)
0x0001E20A	LDRSH	r7,[r3,r7]	2	load input data sample to r7 from memory address contained in r3 (input data pointer)
0x0001E20C	ADDS	r2,r2,#1	1	increment loop counter by 1
0x0001E20E	MULS	r6,r7,r6	1	multiply coeff with data sample, result in r6
0x0001E210	ADDS	r0,r6,r0	1	accumulate: add r6 to r0 and store value to r0
0x0001E212	SUBS	r3,r3,#2	1	decrease input data pointer by 2 bytes (16-bit data!)
0x0001E214	UXTH	r2,r2	1	zero-extend the unsigned halfword (16-bit) to 32-bit
0x0001E216	ADDS	r5,r5,#2	1	increase coeff pointer by 2 bytes (16-bit coeffs!)
0x0001E218	CMP	r2,r4	1	r4 holds the loop length , check if there are still rounds to go, set T-flag if true (in C: k < filterLength)
0x0001E21A	BLT	0x0001E204	1	If "T", jump to beginning of loop, otherwise, move forward
Clock cycles			14	
Execution time @ 16MHz [us]			0,875	
Total cycles for 308 taps			4312	
Execution time for 308 taps [us]			269,5	
Measured filter execution time			309	

- The trick here is that the compiler has copied pointers to coefficient table and data buffer from stack to separate registers r3 and r5. This way all the hassling with stack pointer offsets and storing them back can be avoided.

SW perf optimization (FIR loop only) (cont.)

- Still, after optimizing compiler, there are two places that can be easily improved:

```
MOVS   r6,#0x00
MOVS   r7,#0x00
LDRSH  r6,[r3,r6]
LDRSH  r7,[r2,r7]
```

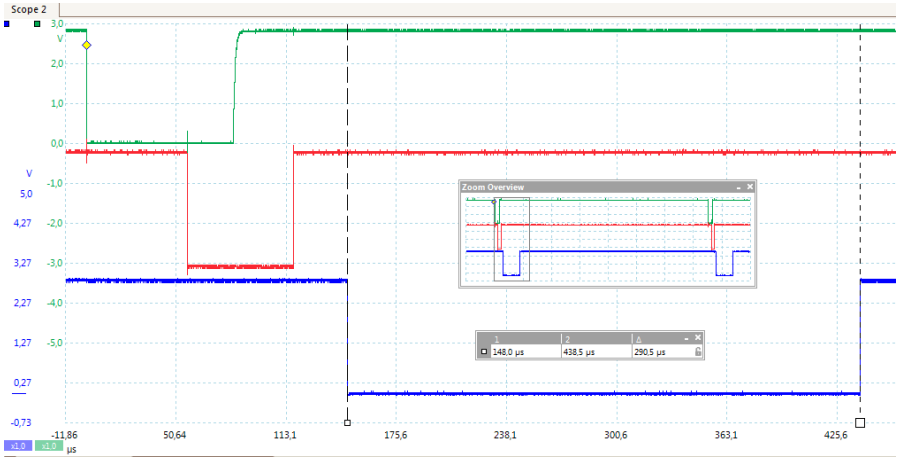
```
MOVS   r7,#0x00
LDRSH  r6,[r3,r7]
LDRSH  r7,[r2,r7]
```

- Left-side: knowing that LDRSH (load sign-extended halfword)-instructions third parameter is register containing offset – in this case zero, we see that it is unnecessary to reset both r6 and r7 – we just need value zero from some register. On right side this is done -> 1 clock cycle (out of 14) saved. (This was not implemented now, it would require inline assembler in C)

- Another place for optimization is to declare loop counter as 32-bit integer, which is the native wordlength for ARM Cortex M0. This way we can get rid of the zero-extension of the loop counter before comparison to loop limit, thus saving one clock cycle

```
void firFixed( const int16_t *fir_coeffs, int16_t *input, int16_t *output, int length, uint32_t filterLength )
...
for ( uint32_t k = 0; k < filterLength; k++ ) {
```

Final code & timing results



Address	Opcode	Parameters	Cycles
0x0001E204	MOVS	r6,#0x00	1
0x0001E206	MOVS	r7,#0x00	1
0x0001E208	LDRSH	r6,[r3,r6]	2
0x0001E20A	LDRSH	r7,[r2,r7]	2
0x0001E20C	SUBS	r2,r2,#2	1
0x0001E20E	MULS	r6,r7,r6	1
0x0001E210	ADDS	r0,r6,r0	1
0x0001E212	ADDS	r3,r3,#2	1
0x0001E214	ADDS	r5,r5,#1	1
0x0001E216	CMP	r5,r4	1
0x0001E218	BCC	0x0001E204	1
Clock cycles			13
Execution time @ 16MHz [us]			0,8125
Total cycles for 308 taps			4004
Execution time for 308 taps [us]			250,25
Measured filter execution time			291

The code optimization shortened the filter execution time by 173us = 41%.

With a simple trick (inline asm) additional 19us (~5%) could be saved.

What about power consumption?

- The average power consumption of the system was measured without filtering and with non-optimized and optimized filters. A 3V CR2032 with 220mAh capacity is used in battery lifetime calculation.

	I_tot(avg)[mA]	I_filter(avg)[mA]	Percentage of total power	Estimated battery lifetime [h]
System baseline (no filter)	3,46	--		63,6
Non-optimized filter	4,42	0,96	27,7	49,8
Optimized filter	4,19	0,73	16,5	52,5
Filter power saving [%]		23,96		

- As can be seen, the shorter is the filter execution time, the longer CPU can stay in the ”sleep” mode, waiting for the next event to be processed, leading to lower average power.
- FIR filtering of ECG has a big contribution to the overall system power consumption, so the optimization effort of the filter really pays off. Note that the optimization was done only for the inner loop (MAC), not for the complete filter, so there is potential for more savings.