# SoC opportunities for boosting SDR GNSS performance

**A.A. Kumarin[1], I.A. Kudryavtsev[1]**

[1]Samara National Research University, Moskovskoe Shosse 34A, Samara, Russia, 443086

**Abstract.** Software-defined-radio (SDR) becomes an attractive technique for the development of GNSS receivers due to universal hardware and high flexibility. However, the performance of signal processing can be a challenging task. Real-time mode implementation requires fast floating point calculations in several threads, not available for most part of embedded systems. This paper describes the system-on-chip based device drastically increasing computational performance. A summary of computational complexity of each stage of GNSS receiver is provided and several particular solutions are proposed.

## 1. Introduction

Modern global navigation satellite systems (GNSS) such as GPS, GLONASS and Galileo allow user to obtain positioning accuracy about several meters using relatively simple receiver. However, there are many areas, where either better accuracy or special measurements are required [1]. In addition, GNSS signal structure is being modified permanently. A software defined radio (SDR) based receiver may be a solution, providing flexibility and advantages of combined usage of several systems. SDR usually has a universal RF-front-end capable of handling different types of signals in different bands, however, this simplicity requires complex software signal processing. One of the problems is the necessity to execute all the computations in real time mode.

In a typical SDR-based GNSS receiver signal acquisition and tracking are performed independently for each available satellite and each processed signal type e.g. C/A code, L2C, L5 etc. It implies significant computational burden, motivating engineers to seek efficient methods of hardware and/or software implementations.

Maintaining real-time computations in software is a challenging task. Using general-purpose microprocessor is a relatively expensive and power consuming solution. A less power consuming option is using digital signal processor (DSP). However, both aforementioned solutions are limited by the number of parallel threads. Field programmable Gate Arrays (FPGA) allow boosting computational performance and implementation of many parallel channels.

Thus, this paper presents an attempt of using system-on-chip (SoC) to build a GNSS receiver. A typical SoC is a combination of FPGA and a hardware processor system (HPS), allowing parallel acquisition and tracking and sequential computation of coordinates.

## 2. Algorithm overview

GNSS signal acquisition usually involves search of a correlation peak in a 2D array, containing time delay and Doppler frequency. The frequency dimension is usually about 40 cells for rough acquisition. In every frequency cell it is necessary to perform several FFT and IFFT calculations of several thousand points length. Is is a heavy computational load and we need a powerful floating point computational unit to execute it in an appropriate time period [2-3].

Signal tracking algorithm includes scalar product calculation for more than 20 thousand complex elements-long arrays. It can take over 80% of operational time and it is important to note that we have to repeat it for every period of PRN sequence in order to maintain continuous tracking.
Position calculation usually uses least squares method to assess position vector. It also involves some data corrections and auxiliary conversions.

## 3. Possible architectures and details of implementation

There are several possible architectures in this study. Using only FPGA or only HPS is not considered. The simplest solution is creating custom FPGA-based operation modules to expand the main assembler instruction set (figure 1a). For example, matrix operations can be implemented. As mentioned, position calculation uses least squares method which involves linearized equation system solving. It's described by the formula:

$$A^T Ax = A^T b \tag{1}$$

This task requires LU, QR or similar algorithms requiring much CPU time. Executing such algorithm using FPGA frees the HPS for performing other tasks.

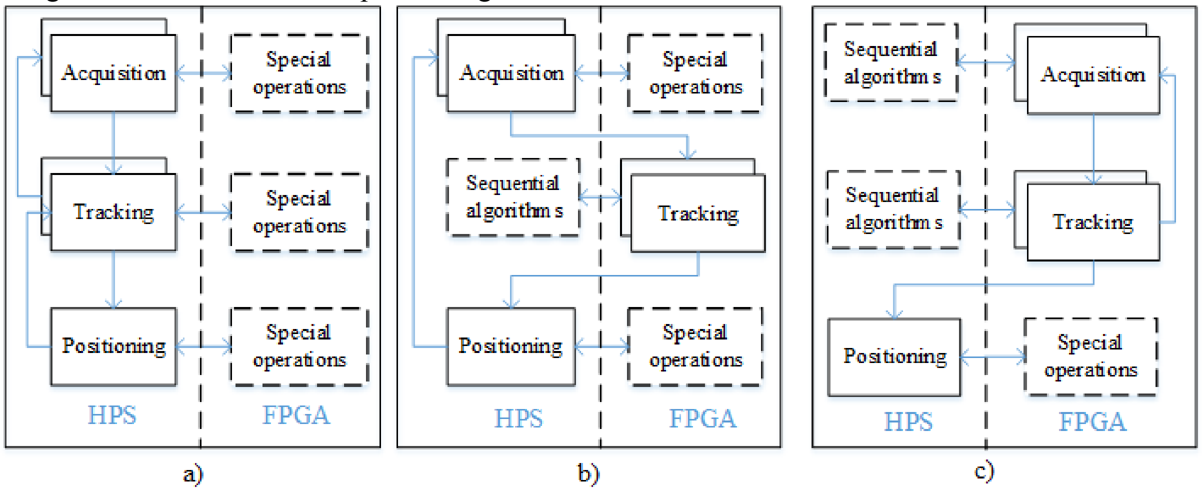

**Figure 1.** Possible SoC GNSS receiver architecture.

In the second architecture (figure 1b) the entire tracking stage is implemented in an FPGA. In this case HPS executes signal acquisition and position computation. After the signal is acquired, its parameters are transferred into a tracking module following signal changes as long as it is available. If the signal is lost, the module sets a flag making the HPS to try reacquisition. As soon as enough navigation data is available, the HPS starts position calculating. It may be assisted by FPGA-executed instructions.
The third architecture (figure 1c) performs both acquisition and tracking in FPGA. The acquisition module constantly tries to acquire all available signals, which are not found yet. Acquired signals are automatically tracked by the tracking modules. These modules may use assistance of the HPS to perform serial algorithms, though this may slow down the process. When there is enough data, the position calculation algorithm is performed by the HPS. It can be assisted by FPGA-based operations.

## 4. Discussion

The first architecture is the simplest to implement by a software engineer. It can be as close as needed to processor-only version and has the same logic. It does not require a very sophisticated SoC. The performance gain of this architecture depends on FPGA-based operations set and their implementation. For example, calculating the sum of more than 20 thousand complex numbers elements for each signal in every 1ms period of the signal is a heavy computational load and takes about 80% of time. This task could be executed in parallel by FPGA, drastically reducing operational time. The number of adders, that can be synthesized is limited only by the quantity of available logic cells. However, the HPS is still heavily loaded by signal gathering which have to be done with relatively low latency due to the limited input data buffers' capacity.

The second architecture allows much more freedom for HPS because tracking is executed without involving processor core. It allows the HPS to perform other important tasks, not necessarily navigational. The position calculation routine is typically called once per second. Reacquisition and search of new signals are not frequent tasks either. Thus, this architecture allows using HPS as a system CPU, not just for positioning.
The third architecture allows executing heavy computational loads other than navigation. Moreover, the HPS can postpone its navigational tasks in favor of more urgent tasks without substantial loss of performance. However, such architecture requires much more logic cells, which implies using high-consuming FPGAs, which could be inacceptable sometimes.
In each architecture it should be taken into account, that FPGA is limited in using floating point arithmetic. Some FPGA provide DSP latches, which contain hardware adders and multipliers. However, the number of such latches is limited. In addition, operations like root, sine, cosine etc. should be implemented in logic cell usage-efficient way. For example, CORDIC can be used.
Thus, the developer should choose the architecture based on the amount of available logic cells, channel number and latency requirements.
Since the choice of the most appropriate architecture depends on the number of channels, complexity of the signal processing algorithms and the power of used HPS and programmable logic, it looks reasonable to compare software and hardware implementations of the critical computational blocks. The most time-consuming part, which has to be executed cyclically to avoid data loss, is tracking. In fact, the main task of tracking is to monitor Doppler frequency and code delay changes with subsequent determination of bit boundaries. In most cases, position calculation is usually performed in the moments when a new data word is received and decoded. The idea of the algorithm, used for GPS, is the following:

1. Calculation of IP/QP, IL/QL, IE/QE for a chunk of data, equal at least to one signal period (1ms);
2. Correction of the phase and frequency of local code and carrier generators;
3. Determination of a phase change on the bit boundary (if any) on every 20th signal period;
4. Collection of the received bits with grouping them into words with parity check;
5. Collection of the words in data blocks with subsequent data extraction;
6. Preparing data for a pseudorange calculation.

These steps are performed every signal period, however, steps 3-6 are performed relatively rare, thus the main computational burden is confined in the steps 1 and 2. To demonstrate C++ implementation, used by the authors it is reasonable to look only at the most "heavy" loop, shown in Fig. 2.

```
for (unsigned int i = 0; i < Size; i++)
{
         Tm += Mult;                    // Elementary phase
         cosT = cos(Tm);
         sinT = sin(Tm);
         V = Datt[i].I;                 // Data[] is an array of data samples
         W = -Datt[i].Q;                // I and Q are real and imaginary parts
         ICON = cosT*V - W*sinT;
         QCON = sinT*V + W*cosT;
         TauCode += CodeTau;
         Tx = TauCode - Spacing;        // Data spacing for the code dicriminator
         if (Tx < 0) Tx = 1023.0 + Tx;
         if (C_A[(unsigned int)Tx] == 1){
                  IE += ICON;
                  QE += QCON;
         }else{
                  IE -= ICON;
                  QE -= QCON;
         }
         … To save space here, similar operations with IP/QP and IL/QL are omitted
}
```

**Figure 2**. Code fragment of the main part of tracking implementation.

It could be seen that the main computational load here is concentrated in sine and cosine calculation. It is worth to notice that we use floating-point variables of the double size. Intel did not modify the implementation of sine and cosine computation for a long time and the only acceleration, which we can achieve here, is a combined cosine/sine computation, using *fsincos* instruction [4]. This instruction takes about 100-200 CPU cycles depending on an architecture and argument type. Variable `Size` in the experiment varies in the range from 30000 to 50000 depending on sampling frequency of RF Front-End. Intel CPU implementation of the code above could involve some low-level improvements of the code, for example, using SIMD extensions allows parallel execution of arithmetic operations for `IP/QP` and other variables. We do not discuss here cash memory issues, like false sharing and other similar ones. Using debugging tools, one can see that these sine/cosine computations takes about 70-80% of the whole time, thus it is reasonable to select this operation and accelerate it using hardware implementation.

This paper considers using coordinate rotation digital computer (CORDIC) algorithm [5]. It allows simultaneous sine and cosine computation without explicit multiplication operations. Since the angle changing step remains constant during cycle (figure 2), an optimized algorithm can be used: the microrotation directions can be calculated once before executing main loop. Thus, proposed FPGA implementation scheme is shown in figure 3:

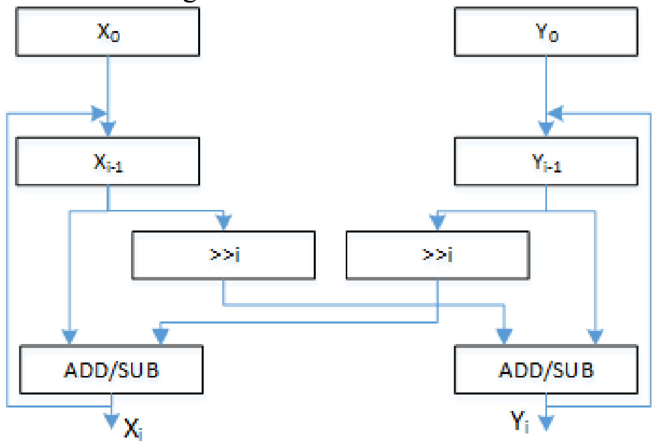

**Figure 3.** Possible SoC GNSS receiver architecture.

Each iteration increases the accuracy. The result is provided on the adder/subtractor output. The number of iterations usually equals to the desired result size *s*. Therefore, serial computation requires about *s* times *Size* cycles.

It has to be noted that FPGA allows pipeline implementation of this algorithm. The latency of the CORDIC would be about s + 2 due to the necessity of scaling the output data. The overall cycle iteration duration is estimated as 5 cycles in pipeline implementation. Serial computation uses about 20 macro operations or about 230 cycles. This makes the operational time of 50MHz FPGA execution comparable to 2.3 GHz CPU serial execution in a single thread. A CPU implementation typically uses sequential tracking channel-after-channel or using several processor cores in parallel. The number of available cores is usually less than required number of tracking channels, that reduces overall performance. FPGA, on the contrary, relatively easily provides any required number of processing modules even for multisystem GNSS receivers. This boosts computation performance drastically, allowing the HPS to perform other tasks.

Thus, FPGA implementation and partial execution of certain GNSS receiver algorithms assists HPS, allowing reducing computational load in comparison with a CPU-only solution. Multichannel pipeline architecture allows fast execution of the heaviest parts of the algorithm.

## 5. References

[1] Shafran, S.V. Snapshot technology in GNSS receivers / S.V. Shafran, E.A. Gizatulova, I.A. Kudryavtsev // 25th Saint Petersburg International Conference on Integrated Navigation Systems (ICINS), 2018. – P. 1-3.

[2] Psiaki, M.L. Block Acquisition of Weak GPS Signals in a Software Receiver // Proceedings of ION GPS. – Salt Lake City, UT, 2001. – P. 1-13.

[3] Guruprasad, S. Design and Implementation of a Low-Cost SoC-Based Software GNSS Receiver // IEEE Aerospace and Electronic Systems Magazine. – 2016. – Vol. 31(4). – P. 14-19.
[4] Agner Fog. Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs [Electronic resource]. – Access mode: www.agner.org/optimize/ (01.02.2019).
[5] Meher, P.K. CORDIC Designs for Fixed Angle of Rotation / P.K. Meher, S. Member, S.Y. Park // IEEE transactions on very large scale integration systems. – 2013. – Vol. 21(2). – P. 217-228.