WHITE PAPER

Get Smart with NB-IoT

Efficient Low-Cost Implementation of NB-IoT for Smart Applications

Authors

Alexander Petrov-Savchenko
Staff Software Engineer, Synopsys Inc.

Pieter van der Wolf
Principal R&D Engineer, Synopsys Inc.

Introduction

Over the last decades, mobile communications technology has evolved to support ever higher data rates. This is exemplified by 2G, 3G, and 4G cellular technologies, with data rates going from a few 100 kbps to 100's of Mbps, and more to come with the enhanced Mobile Broadband (eMBB) features of the new 5G New Radio (NR) standard. However, these high data rates do not come for free as they demand a complex and costly baseband & radio subsystem that consumes a significant amount of power.

Many (future) applications of wide-area communications technology do not require high data rates, but instead demand low-cost devices that consume very little power and can be deployed in massive numbers. Examples of such applications are:

- **Smart cities**: Different types of sensors collect data so that assets and resources can be managed efficiently. Examples are water and energy metering, asset tracking (e.g. bicycles), waste management, intelligent streetlights, environmental monitoring (e.g. air quality), etc.
- **Smart industry**: Highly automated manufacturing plants in which extensive data communication is used to optimize factory throughput and provide life-cycle management.
- **Smart agriculture**: Farms that employ advanced communication technology to monitor and control for example soil quality, environmental conditions, crop yield and livestock, while enabling efficient use of water, energy, fertilizer, and other supplies.

To serve these applications, the 3GPP collaborative standards organization has defined a standard for low-power wide-area IoT networking called Narrowband Internet-of-Things (NB-IoT). The NB-IoT standard has been optimized for machine-type communications. The first version, referred to as Cat. NB1, was defined as part of 3GPP Release 13. An enhanced version, referred to as Cat. NB2, was published as part of 3GPP Release 14 and adds new features such as positioning and multicast.

The NB-IoT standard supports limited data rates and features to enable implementation of low-cost modules of just a few US dollars. Theoretical momentary peak data rates are only ~250 kbps, with sustainable data rates being significantly lower. In many cases, NB-IoT devices will be battery-operated and it may be costly to re-charge or exchange batteries. So very low power consumption is key. Further, to limit complexity, the NB-IoT standard specifically targets devices that are mostly stationary; it does not support seamless handover when moving a device from one cell to another. Also, communication by NB-IoT devices is assumed to be not very delay sensitive.

Unlike LTE, which supports a minimum transmission bandwidth of 1.08 MHz, NB-IoT uses a bandwidth of only 180 kHz [1]. However, NB-IoT can still be deployed in combination with legacy LTE, using single LTE resource blocks with 12 subcarriers and 15 kHz subcarrier spacing (hence 180 kHz). This can be either using in-band operation, by allocating a resource block within an LTE carrier, or using guard band operation, utilizing the guard-band of an LTE carrier. With guard band operation, NB-IoT does not consume valuable spectrum used for traditional mobile devices (e.g. smartphone traffic). Another option is to operate NB-IoT independently from LTE by utilizing a non-LTE wireless channel (e.g. a GSM band). This is also referred to as stand-alone operation [2].

NB-IoT provides extensive support for power savings. For example, it supports advanced power down modes like Extended Discontinuous Reception (eDRX), allowing a device to sleep for multiple periods of 10.24s, up to several hours of extended sleep. Another example is the Power Saving Mode (PSM) which enables a device to go dormant for even longer times, until it decides to wake up and transmit to the network. A dormant device remains registered with the network but does not respond to messages.

In addition to NB-IoT, 3GPP has also defined the LTE-M standard (Cat. M1 in Release 13 and Cat. M2 in Release 14). Cat. M1 uses a LTE-compliant transmission bandwidth of 1.08 MHz and supports higher data rates up to 1 Mbps, at the expense of increased system complexity and cost. Unlike NB-IoT, LTE-M supports seamless mobility so that connections do not get dropped on a base station switch. It also supports voice communication.

Rapid market growth is predicted for NB-IoT [3], with hundreds of millions of device shipments per year in the coming years. Large-scale deployment of NB-IoT is ongoing in China, while adoption in Europe has started as well. In the US, network operators have initially focused on LTE-M, but are now also working on nationwide deployment of NB-IoT networks. For example, Verizon is completing its NB-IoT network in 2018, supporting NB-IoT guard band operation.

Traditionally, the security mechanisms of cellular networks have been based on a physical SIM card attached to a device. The use of SIM cards to securely identify and authenticate subscribers has enabled roaming between network operators, which has been a key factor in the success of mobile networks. An important next step for the wide deployment of NB-IoT is the use of embedded SIM (eSIM) or integrated SIM (iSIM) to enable secure over-the-air SIM provisioning. Further benefits of eSIM and iSIM are reduced costs and reduced physical dimensions of devices.

In the following sections of this white paper, we present further aspects of the NB-IoT standard with a focus on the user equipment (UE) side. Specifically, we discuss how an NB-IoT UE modem can be efficiently implemented on a small but very capable CPU/DSP processor that is supported by a few well-targeted accelerators. We detail the DSP capabilities of such processor and illustrate their effective use with efficient implementations of key NB-IoT software kernels. We conclude that a NB-IoT modem can be implemented in software on a small processor like the Synopsys' DesignWare® ARC® EM9D, thereby achieving low cost and low power consumption.

The NB-IoT Standard and Implementation Goals

The NB-IoT protocol may be viewed from two different perspectives. From one perspective, it is an evolution of the LTE standard and follows the same specification framework, so it is natural to look at it as a subset of its "bigger brother", LTE. From another perspective, NB-IoT is a completely new communication standard that has more in common with other narrowband communication technologies in terms of development effort, computational complexity, and required network and device resources.

The first wave of NB-IoT modems coming from major LTE market players approached the NB-IoT modem development task from the legacy LTE angle, leveraging their experience to strip down existing LTE designs. Recently, competing offerings have started to emerge from companies who realized that it is not only feasible, but even beneficial to create NB-IoT solutions from scratch with low power and low cost as primary design goals.

As mentioned previously, cost is a key driver of NB-IoT implementations. The cost of a modem is mostly determined by die area. On-chip memories and hardware accelerators are among the biggest contributors to IC silicon area. This defines two major vectors for system optimization: (1) code size and system memory footprint reduction and (2) minimization of signal processing hardware accelerators in favor of software only optimizations. Such accelerators should be added only when the performance gain outweighs the area increase. The applicability of such accelerators to other communication and IoT technologies should also be considered when making a decision about their inclusion so that a broad range of use cases can be targeted.

The typical application areas mentioned in the introduction section do not require continuous data transfers. For example, an NB-IoT-connected system could collect data from various sensors, process it and then transmit fused results based on a timer or data-driven event. Alternatively, or additionally, the system could monitor the communications channel in a conservative low-power mode and upon receiving a data packet from the network, perform some action, such as switching on a street light. In both cases, the active data transmission phase is only a small fraction of the overall system operation time. Thus, it is usually beneficial to focus on the power and area optimization while having just enough performance to meet the NB-IoT protocol timing constraints. For many applications, the low modem uptime also allows a single processor to be used for both modem and application code.

From the software design perspective, it is common in broadband LTE to have a real-time operating system (RTOS) handling multiple tasks, each doing small chunks of work asynchronously. A protocol scheduler resolves a multitude of possible states and conditions such as concurrent uplink and downlink data transmission and different radio resource control and power-saving states. In the NB-IoT case, thanks to the half-duplex operation mode and a single Hybrid Automatic Repeat reQuest (HARQ) process for Cat. NB1, the sequence of UE actions has fewer degrees of freedom and can be serialized, which allows the whole physical layer to be run in a single interrupt- or timer-driven thread. It would be a natural decision for an engineer with prior experience in designing broadband LTE modems to split the data processing into separate operating system tasks based on their logical function, but this would introduce unnecessary overhead for an NB-IoT implementation. Further complexity reduction versus broadband LTE is achieved by using a simpler channel coding. No Turbo coding is used in NB-IoT for the downlink, thereby avoiding the need for Turbo decoding in the UE modem.

NB-IoT is still an evolving standard, with 3GPP Release 14 bringing new functionality (such as positioning services and multicast) as well as increased throughput (with Cat. NB2 support) [4]. However, a well-designed and software-defined UE modem could support this and future changes through firmware upgrades, which could be managed over the air. This flexibility also favors implementations that do not rely heavily on application-specific hardware accelerators, but rather achieve the performance and footprint goals by employing an efficient processor with a rich instruction set and a set of flexible extensions. The further sections introduce a suitable hardware / software architecture and demonstrate the feasibility of a cost-effective, software-defined NB-IoT modem implementation.

Hardware / Software Architectures for NB-IoT Modems

Legacy LTE modems typically have rather complicated hardware / software architectures, including, for example:

- Multiple programmable DSP cores, specifically for executing PHY-layer functions.
- Hardware accelerators for off-loading compute intensive tasks that can be done more efficiently in specialized hardware. These hardware accelerators typically have local memories, e.g. for buffering inputs and outputs.
- One or more RISC cores, specifically for executing the L2 and L3 layers of the protocol stack.
- Advanced interconnects and DMA controllers for high-throughput communication between the different hardware blocks.
- External DRAM for code and data.
- Multiple software tasks executing on each processor, supported by a software infrastructure with real-time operating system(s), inter-processor communication, etc.

An example hardware block diagram of a wideband LTE modem is shown in Figure 1, including the use of hardware accelerators in the transmit and receive pipelines.

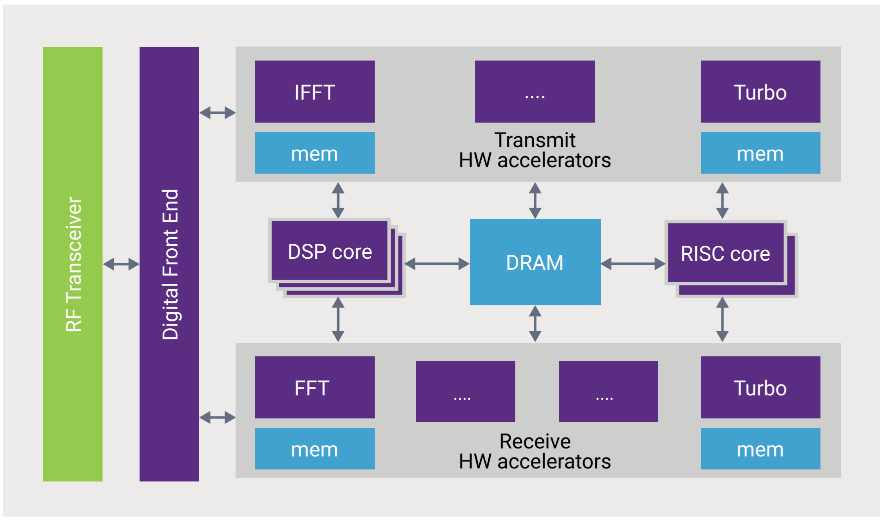

Figure 1. Example block diagram of wideband LTE modem

Such architectures are required to deal with the high data rates and advanced features of wideband LTE modems. For example, the high data rates not only demand extensive compute resources but also require large memories for buffering intermediate results. Taking such hardware / software architecture as starting point for an NB-IoT modem will not yield an efficient low-cost implementation. With data rates that are a few orders of magnitude lower than state-of-the-art LTE and a significantly simplified feature set, as described in the previous section, a rigorous re-design is required for NB-IoT.

A flexible NB-IoT modem can be based on a single small and efficient CPU/DSP processor, complemented with a few well-targeted hardware accelerators. By using a processor with the right DSP capabilities for efficiently executing the NB-IoT modem functions, the MHz requirements can be lowered. Consequently, the processor can be clocked at a low frequency, which helps to save power. In addition, a low frequency enables the use of a sub-nominal voltage to further reduce power consumption. The low MHz requirements also help to save area as a shallow processor pipeline suffices to meet the targeted frequency.

On the software side, the processor must offer excellent code density and an efficient implementation of the software stack, so that memory sizes can be small. The use of off-chip DRAM can then be avoided, which helps keep system costs down. Small code size is also important for achieving low power consumption, specifically by reducing accesses to the instruction memory.

An example hardware / software architecture for NB-IoT based on Synopsys' DesignWare ARC EM9D processor is depicted in Figure 2. The diagram shows the ARC EM9D CPU/DSP processor with its local memories and hardware accelerators. The software stack running on the processor implements the NB-IoT protocol layers and an application on top of that. A digital radio front-end implements the I/Q data interface between the protocol stack and the RF transceiver. This RF transceiver can be relatively simple due to the limited bandwidth, allowing for on-chip integration together with the baseband processing.

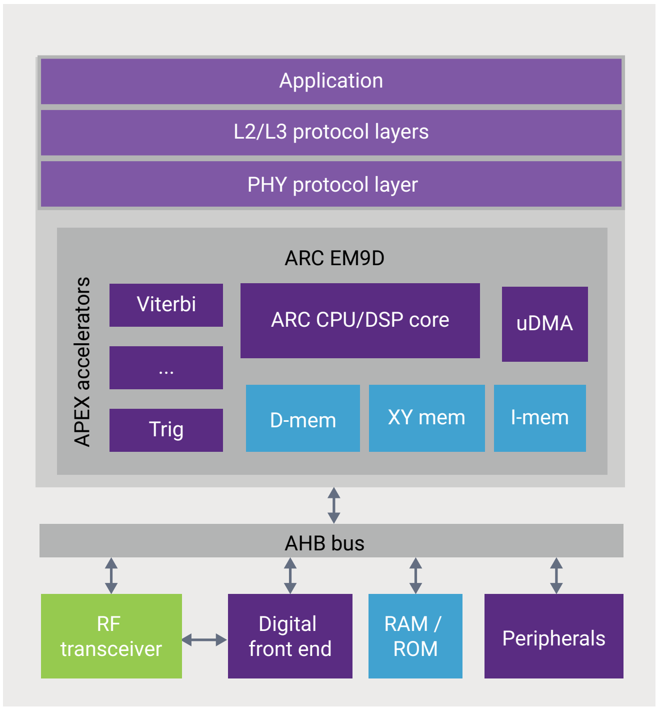

Figure 2. Hardware / software architecture for NB-IoT

The ARC EM9D processor is extremely well-suited for building an NB-IoT modem. It can efficiently execute both control and DSP code, thereby eliminating the need for separate CPU and DSP processors for executing the application and the NB-IoT protocol stack. As explained in the previous section, many applications do not require continuous data transfers. Hence, in many cases the processor used for NB-IoT can also be used to perform application tasks, like collecting and processing sensor data. A separate host processor is not required.

The instruction set of the ARC EM9D enables efficient execution of NB-IoT modem functions. Some of the key instruction set features relevant to NB-IoT are:

- Instruction format designed for excellent code density
- Extensibility with customer-defined instructions, through ARC Processor EXtensions (APEX)
- Zero-overhead loops
- Fixed-point arithmetic with saturation and rounding
- Wide accumulators with guard bits
- 2x16 and 4x8 vector support
- 16+16 complex arithmetic & butterfly support
- Single-cycle MAC, including single-cycle complex MAC
- Circular and bit-reversed addressing
- Divide and square root
- XY memory with advanced address generation

The ARC EM9D processor comes with an advanced optimizing DSP C compiler that enables efficient development of DSP software, with effective use of the above instruction set features. For example, it supports data types and primitives for fixed-point programming, including vector data types and complex data types.

A feature particularly relevant to the efficient implementation of NB-IoT is the XY memory with advanced address generation. Basically, XY memory provides up to three logical memories that the processor can access concurrently, as illustrated in Figure 3. The processor can access memory through a regular Load or Store instruction or it can enable a functional unit to perform memory accesses through address generation units (AGUs). An AGU can be set up with an address pointer to data in one of the memories and a prescription to update this address pointer in a particular way when a data access is performed through the AGU. After set-up, the AGUs can be used in instructions for accessing operands and storing results.

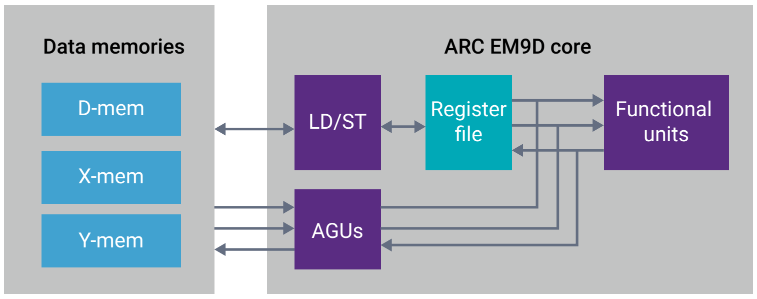

Figure 3. ARC EM9D with XY memory and address generation units

The effectiveness of the XY memory and AGUs can be illustrated with a small code example, as in Figure 4. The C-code example on the left shows a dot-product style vector multiplication. The __xy qualifiers tell the compiler to use the AGUs for accessing the arrays pointed to by q31_t *b and q31_t *c. The generated assembly code on the right shows that after setting up the AGUs, the loop executes at one multiply-accumulate (MAC) per cycle with the two operands fetched through the AGUs.

C source code

```
q31_t foo(__xy q31_t *b, __xy q31_t *c) {
   q31_t s = 0;
   for (i = 0; i <  N; i++)
      s += b[i] * c[i]
   return s;
}
```

Compiler generated assembly code

```
          SR ...    // set-up agu 0
          SR ...    // set-up agu 1
          LP lpend
          MAC 0, %agu_u0, %agu_u1
lpend:
          …
```

Figure 4. Small C-code example for the use of XY memory with associated assembly code

As the example shows, the memory accesses are performed implicitly through the AGUs; no explicit Load instructions have to be fetched, decoded, and executed for accessing the data memories, which saves power and yields a small code size. Also, the AGUs update their address pointers as prescribed at set-up, so that the next vector elements are accessed upon the next MAC operation. Hence, the single MAC instruction performs two memory accesses, two address pointer updates, a multiply and an addition.

If we compare this to the assembly code of a VLIW processor that achieves the same throughput of one MAC per cycle, we see some striking differences.

VLIW assembly code

```
          // Prologue SW pipelining
          LDD %r2, [%r0, 8]  ; // 64b vector load
          LDD %r4, [%r1, 8]  ; // 64b vector load
          LDD %r6, [%r0, 8]  ; // 64b vector load
          // 4x unrolled loop
          LP lpend
          {MAC 0, %r2, %r4; LDD %r8, [%r1, 8]}  ; // 32b MAC and 64b vector load
          {MAC 0, %r3, %r5; LDD %r2, [%r0, 8]}  ; // 32b MAC and 64b vector load
          {MAC 0, %r6, %r8; LDD %r4, [%r1, 8]}  ; // 32b MAC and 64b vector load
          {MAC 0, %r7, %r9; LDD %r6, [%r0, 8]}  ; // 32b MAC and 64b vector load
lpend:    // epilogue if loop count not multiple of 4
          …
```

Figure 5. VLIW assembly code for dot-product style vector multiplication

As Figure 5 illustrates, the VLIW processor must issue explicit Load instructions to perform the memory accesses. These must be 64-bit Load instructions to sustain the one MAC per cycle throughput in the loop. A prologue is needed to load initial values and the loop must be unrolled to achieve proper software pipelining. Finally, an epilogue is needed if the loop count is not a multiple of four. The example illustrates that the code size of the VLIW is significantly larger and that more instructions must be fetched from memory per cycle, which increases power consumption.

Using the instruction set features of the ARC EM9D core, including the XY memory, many NB-IoT modem functions can execute efficiently on the processor without additional hardware accelerators. An example is the complex FFT function that is executed frequently in the NB-IoT protocol. Using features like 16+16 complex arithmetic & butterfly support, bit-reversed addressing, XY memory with AGUs, zero-overhead loops, excellent performance is achieved for the complex FFT function. Consequently, it consumes only a small fraction of the targeted MHz requirements for executing the NB-IoT protocol stack. A further small performance improvement does not warrant the extra cost and power consumption of a special FFT hardware accelerator.

There are, however, a few functions that have rather specific characteristics and cannot be implemented efficiently on a generic CPU/DSP processor. One such example is Viterbi decoding, which is a prominent function in the NB-IoT protocol stack for performing forward error correction in the receiver. For these functions, the extensibility of the ARC EM9D processor can be used to add custom instructions that significantly accelerate the execution of the functions. These custom instructions, also called APEX instructions, can then be called as intrinsics in the C-code of such a modem function. In the next section we present a detailed code example for the Viterbi decoder.

We note that the ability to extend the processor with APEX instructions is radically different from adding an external hardware accelerator on a bus. Using an external bus-based hardware accelerator requires data to be moved over a bus and buffered with additional synchronization requirements (e.g. through interrupts), thereby impacting area, cycles, power consumption and code size. The extensibility of the ARC EM9D processor avoids such overheads, providing an efficient NB-IoT implementation.

Implementation of the NB-IoT PHY Layer

In this section we further analyze the computational complexity of the NB-IoT protocol and illustrate its efficient implementation on the ARC EM9D processor. For this purpose, we use a reference implementation of a complete NB-IoT software stack, including a PHY / L1 layer as well as L2 and L3 layers.

When analyzing the computational complexity of the NB-IoT protocol, one must consider the different phases that the NB-IoT protocol stack goes through during operation:

1. Cell search
2. Cell attach
3. Control data reception
4. Downlink data reception
5. Uplink data transmission

The same functions may be used across several phases, but their configuration and parameters as well as timing constraints may be different. Such constraints may emerge from the implementation, e.g. from the available depth of FIFO sample buffers, and from NB-IoT deadlines imposed by the protocol specification, such as the RX-TX scheduling delay.

For example, Viterbi decoding is part of the cell attach, control data reception, and downlink data reception phases. Performance-critical Viterbi decoding occurs during the initial cell attach procedure when some of the channel resource allocation parameters are not yet identified and blind decoding must be performed. That is, the UE receiver must try different repetition counts for the control data coming from the eNodeB (base station in LTE networks), performing Viterbi decoding for each of these.

After extensive profiling of the UE stack using worst-case throughput and resource allocation scenarios, we identified the list of processing kernels that are key to meeting all the deadlines across all phases and thus are the first candidates for optimization. This list is presented in alphabetic order in Table 1. In the following sections, the implementation details for several of these functions are presented to illustrate the efficiency of the ARC EM9D processor.

Function	Principal operation	Algorithm
Channel interleaver	Data re-shuffling	Standard
Convolutional decoder	Viterbi decoding	Standard
Convolutional encoder	Turbo encoding	Standard
FFT	FFT butterfly	Standard
Modulation mapper	Look-up table addressing	Standard
NPSS search	Auto- and cross-correlations	Vendor-specific
NRS timing recovery	Complex multiplication	Vendor-specific
NSSS decoder	Cross-correlation with Zadoff-Chu sequences	Vendor-specific
Phase & CFO correction	Multiplication with complex exponentials	Standard
PN sequence generator	Linear feedback shift register	Vendor-specific
Resampler / interpolator	Complex convolution	Standard
Soft-bit normalization	Bit shift with rounding	Standard
Sub-block de-interleaver	Data re-shuffling	Standard

Table 1. Key processing functions in NB-IoT protocol stack (in alphabetical order)

From our profiling activities we have drawn several conclusions. First, the NB-IoT stack complexity is dominated by the Layer 1 processing, with the L2/L3 layers consuming only a small fraction of the total processing power. Even the more resource-demanding functions of Layer 2, such as cryptography (AES, SNOW-3G and, optionally, ZUC) are not visible in the bigger picture of the whole stack, and thus can be implemented purely in software on a processor that offers the right capabilities for their efficient execution. Second, a quite large and diverse set of functions dominates the processing demands of the NB-IoT stack. This implies that there is not a single silver bullet for optimizing the MHz requirements and that the CPU/DSP processor must be able to efficiently handle quite diverse processing requirements. Third, the top Layer 1 processing kernels are not only the conventional DSP functions (such as FFT and resampling). These top contributors also include functions used in error correction, such as Viterbi decoding, and PN sequence generation.

The 3GPP NB-IoT protocol specification defines the external behavior of the system, but leaves many implementation aspects to the modem maker, who may follow common design patterns or come up with novel solutions. For example, only the transmitter path is defined in the standard, while it is up to the developer to design a receiver. As shown in the right-hand column of Table 1, we distinguish two kinds of functions: "standard" and "vendor-specific". The first category uses standard algorithms that are strictly defined by the 3GPP specification or have an optimal algorithmic implementation already established in the industry. An example is the decoder for the convolutional codes used in NB-IoT, for which Viterbi decoding is the de-facto standard. There is limited flexibility in implementing these functions algorithmically, so the processor capabilities, compiler efficiency and processor-specific optimization become the main factors in determining their performance. This also explains the relevance of selecting the proper processor for implementing the NB-IoT stack. The algorithms that constitute the second group are vendor-specific because they are not defined by the 3GPP specification, and thus can be optimized at the algorithm level. Examples of such algorithms are the synchronization signal search and decoding (NPSS, NSSS – Narrowband Primary Synchronization Signal and Narrowband Secondary Synchronization Signal) [5].

The following sub-sections focus on the optimization of several key processing kernels. Some of the most computationally-intensive "standard" functions are chosen for this deep dive. We discuss implementation on the ARC EM9D processor and show how its capabilities are used effectively.

Viterbi decoder

NB-IoT uses convolutional coding for forward error correction (FEC) in the downlink transmissions [6]. The received data is decoded on the UE side using the Viterbi algorithm. From the profiling data, we concluded that this kernel is one of the most computationally intensive parts of the NB-IoT modem. Viterbi or similar FEC schemes are used in many other communication technologies, especially in the IoT field, and often are a bottleneck in modem design.

A software implementation of the Viterbi algorithm on a generic CPU/DSP processor can provide only limited speed-ups. For this reason, we decided to apply hardware acceleration using APEX instructions to significantly reduce the MHz requirements of Viterbi decoding. We used a flexible Viterbi decoder APEX module for the ARC EMxD family of processors. It has a stellar performance and a simple API which allows it to be used as a drop-in replacement for software implementations.

An example of the Viterbi decoder C-style pseudocode powered by the Viterbi APEX extension is presented in Figure 6. The APEX-related intrinsics are shown in **bold**. Two APEX instructions (`vitacc0` and `vitacc1`) are used for the calculation of the path metrics. The APEX instruction `vittb` is used for accelerating the traceback. Note that `__xy` qualifiers are used to instruct the compiler to use XY memory with advanced address generation for accessing the input samples, path metrics and decision bits as well as for storing the decoded result.

```
void viterbi_decode(__xy int32_vec3x8b sample[], __xy int32 result[], int frame)
{
  // sample: array of 3*8b components of input sample
  // result: decoded result
  // frame:  number of bits in the frame (multiple of 32)
  // store path metric decision bits
  __xy int32 decisions[frame*2];  // infer XY address generation

  // path metric reset, single cycle
  vitrst();

  // compute path metrics and decision bits, two cycles per bit
  for (i = 0; i < frame; i++) {
    decisions[2*i]   = vitacc0(sample[i]);
    decisions[2*i+1] = vitacc1(0);
  }

  // traceback, one cycle per bit
  i = frame-1; j = frame/32-1;
  while (i > 0) {
    for(k = 0; k < 31; k++) {
      vittb(decisions[2*i], decisions[2*i+1]);
      i--;
    }
    result[j--] = vittb(decisions[2*i], decisions[2*i+1]);
    i--;
  }
}
```

Figure 6 . Example of the APEX-accelerated Viterbi decoder source code

Figure 7 and Figure 8 show snippets of execution traces of the path metrics computation and traceback loops. As can be seen from Figure 7, the path metrics computation executes an APEX instruction every cycle while accessing data through the AGUs, effectively performing the path metrics computation in two cycles per bit. As shown in Figure 8, the traceback loop also executes an APEX instruction every cycle, thereby achieving a throughput of one bit per cycle for the traceback. Consequently, this gives a tremendous speed-up of the Viterbi decoding, reducing the worst-case MHz requirements for this function to *less than 1 MHz* on the ARC EM9D with the Viterbi APEX accelerator. Also there is no overhead of moving data to/from the accelerator or for performing some form of explicit synchronization (e.g. through interrupts).

```
…
721924:       000002a8 386f4001|vitacc1  agu_u0,0
721925:       000002a4 382f4840|vitacc0  agu_u0,agu_u1
721926:       000002a8 386f4001|vitacc1  agu_u0,0
721927:       000002a4 382f4840|vitacc0  agu_u0,agu_u1
721928:       000002a8 386f4001|vitacc1  agu_u0,0
…
```

Figure 7. Example of resulting instruction trace for the APEX-accelerated Viterbi path metrics calculation

```
…
928719:       000002ec 3900483e|vittb    0,agu_u1,agu_u0
928720:       000002f0 3900483e|vittb    0,agu_u1,agu_u0
928721:       000002f4 3900483e|vittb    0,agu_u1,agu_u0
928722:       000002f8 3900483e|vittb    0,agu_u1,agu_u0
928723:       000002fc 3900483e|vittb    0,agu_u1,agu_u0
928724:       00000300 3900483e|vittb    0,agu_u1,agu_u0
928725:       00000304 3900483e|vittb    0,agu_u1,agu_u0
928726:       00000308 39004822|vittb    agu_u2,agu_u1,agu_u0
928727:       000002ec 3900483e|vittb    0,agu_u1,agu_u0
…
```

Figure 8. Example of resulting instruction trace for the APEX-accelerated Viterbi traceback

FFT

The Fast Fourier Transform (FFT) is an algorithm widely used in digital signal processing to transform the digital signal between the time and frequency domains. The orthogonal frequency division multiplexing (OFDM) modulation scheme used in the NB-IoT protocol relies heavily on the frequency domain processing which makes FFT a significant contributor to the overall processing complexity. FFT is used in the cell attach phase as well as in all data reception and transmission phases. As we explained in the previous section, the complex FFT function executes efficiently using the native capabilities of the ARC EM9D processor and no further acceleration is required. Performing the FFTs in the NB-IoT protocol stack requires just a few MHz on the ARC processor.

Phase and carrier frequency offset correction

In wireless communications, a receiver must correct for offsets in complex signal phase and carrier frequency that occur due to unknown properties and state of the remote transmitter. Left uncorrected, even a small difference in carrier frequency can cause a significant distortion in the received signal. These corrections must be performed in all modem operation phases after a first estimation of the offset parameters is done in the cell search phase. In the downlink communication these corrections are applied to the received signal based on the results of a continuous channel estimation procedure. In the uplink communication the same corrections are applied to the signal before transmission to improve the reception performance at the eNodeB receiver.

We consider phase and carrier frequency offset corrections together in this section since they are essentially the same algorithm but applied in different domains. Both use multiplication with a complex phasor to achieve their effect. For phase offset correction the multiplication is done in the frequency domain and for carrier frequency offset correction the multiplication takes place in the time domain. As shown in Figure 9, this kernel requires computation of complex exponentials, which can be expressed as the sine and cosine of the same argument, where φ_0 is the starting phase offset and φ is the phase offset increment. Note that input data (**src$_n$**) and output data (**dst$_n$**) are also complex data.

$$dst_n = src_n \cdot e^{j(\varphi_0 + \varphi \cdot n)} = src_n \cdot \left(cos(\varphi_0 + \varphi \cdot n) + j \cdot sin(\varphi_0 + \varphi \cdot n)\right),$$

Figure 9. Phase and frequency offset correction

The ARC EM9D processors offer several capabilities for optimization of this kernel.

- The complex multiplications are accelerated with a single-cycle complex multiply instruction, zero-overhead loop support and XY memory access with advanced address generation.
- The sine and cosine calculations are accelerated with trigonometry APEX instructions.

With these optimizations, the phase correction kernel takes less than 2 MHz on the ARC EM9D.

Resampler / Interpolator

The resampler / interpolator kernel serves several purposes.

- It matches the sampling rate of the radio frontend to the sampling rate used by the algorithms of the NB-IoT physical layer.
- It performs the final radio channel filtering if the filters of the radio frontend do not achieve the required stop-band attenuation.
- It interpolates the signal to adjust the sampling position with sub-sample accuracy, which is required for achieving the best demodulation performance.

This processing kernel is involved in all phases of the modem's activity and thus could become one of the performance bottlenecks if not designed properly.

Fractional resamplers used in wireless communications are usually implemented with complex polyphase finite impulse response (FIR) filters with linear phase response. Depending on the ratio of the sampling rates at the input and the output of the resampler, the filtering step is preceded by zero padding and/or followed by decimation. The use of FIR filters with zero padding and decimation allows to reduce complexity without any reduction in the computational precision.

FIR filters are commonly implemented by a linear convolution of the input signal with the impulse response of the filter. A straightforward implementation on a generic processor would require 4*N multiplications, 4*N additions, and 4*N memory loads per complex sample at the least of the input and output sampling rates, several branching and/or address calculation operations to skip over the padded zero samples and samples subject to further decimation. Here N stands for the number of filter taps, which depends on the filter design criteria derived from the protocol specification. Depending on the filter length, input sample rate and register widths, it could also be required to perform some additional arithmetic and memory access operations to prevent register overflow.

The ARC EM9D processor can perform the complex convolution with just N complex MAC (multiply-accumulate) instructions per sample. This is achieved with the DSP instruction set supporting single-cycle complex MAC, wide accumulator registers with guard bits to prevent overflows, the XY memory allowing simultaneous access to both the input samples and the filter coefficients, the AGUs supporting advanced (circular) addressing patterns, and zero-overhead loops. Ultimately, the whole inner loop of the convolution algorithm is just a single instruction without any overhead. It is worth mentioning that this level of efficiency does not require any hand-written assembly code; such optimized loops are generated from C source code by the DesignWare ARC Metaware Development Toolchain [7].

Channel Interleaver

The channel interleaver kernel is used in the narrowband physical uplink shared channel (NPUSCH) to rearrange the data bits to improve the robustness to transmission errors. The rearrangement rule is defined in the 3GPP specification in a pseudo-code form [6]. However, using the features of the ARC EM9D processor, the whole algorithm is implemented as a single instruction in an AGU-driven zero-overhead loop. The code snippets for both the C source and the ARC-specific assembly are shown in Figure 10. Note that on the ARC EM9D processor, only a single loop is required instead of two nested loops thanks to the AGU capabilities supporting the advanced addressing pattern. This efficient implementation of the channel interleaver requires less than 0.2 MHz on the ARC processor.

Generic C source code derived from the 3GPP spec:

```
const uint32_t c_mux = 144 / n_sc_ul;
for(row = 0; row < n_sc_ul; row++) {
  for(col = 0; col < c_mux; col++) {
    dst_ptr[row + col * n_sc_ul] =
    src_ptr[row * c_mux + col];
  }
}
```

ARC-optimized assembly code:

```
  SR …                   ; set-up AGU pointer 0 (source)
  SR …                   ; set-up AGU pointer 1 (modulo destination)
  SR …                   ; set-up AGU pointer 2 (additional destination step)
  mov %lp_count,143      ; set-up zero-delay loop for 143 repetitions
  lp  .LB1               ; begin loop
  mov %agu_u1,%agu_u0    ; copy data and update the source and destination pointers
.LB1:                    ; end of loop
  modapp 0,%agu_u2       ; make the final increment of the destination pointer
  mov %agu_u2,%agu_u0    ; copy the last data (element number 144)
```

Figure 10. NPUSCH channel interleaver C source code and assembly examples

Conclusion

Low cost and low power consumption are critical success factors for deployment of NB-IoT in a broad range of emerging smart applications. In this white paper we presented how the NB-IoT standard reduces implementation complexity by supporting a limited data rate and feature set. Consequently, compared to legacy LTE modems, a simplified hardware/software architecture can be used which employs a single small CPU/DSP processor for executing a complete NB-IoT software stack including the PHY layer. Using the ARC EM9D as an example, we illustrated several processor capabilities that are key to efficient implementation of NB-IoT, such as a well-defined DSP instruction set, XY memory with advanced address generation, and extensibility with select custom instructions. For several key NB-IoT software kernels, we then detailed how they can be efficiently implemented using these processor capabilities. We conclude that an NB-IoT modem can be implemented in software on a small processor using small memories and running at a low frequency, thereby achieving low cost and low power consumption.

References

[1] TS 36.101 User Equipment (UE) radio transmission and reception, 3GPP, April 2018

[2] Narrowband Internet of Things, Whitepaper, Rohde & Schwarz, August 2016.

[3] Ericsson Mobility Report, June 2018.

[4] Overview of 3GPP Release 14 Enhanced NB-IoT, IEEE Network, vol. 31, issue 6, November 2017

[5] TS 36.211 Physical channels and modulation, 3GPP, January 2018

[6] TS 36.212 Multiplexing and channel coding, 3GPP, January 2018

[7] MetaWare DSP Programming Guide, Synopsys ARC, September 2018.