The *Tick* Programmable Low-Latency SDR System

Haoyang Wu[†], Tao Wang[†], Zengwen Yuan[‡], Chunyi Peng[¶], Zhiwei Li[†], Zhaowei Tan[‡],

Boyan Ding[†], Xiaoguang Li[†], Yuanjie Li[‡], Jun Liu[†], Songwu Lu[‡]

[†]Peking University, [‡]University of California, Los Angeles, [¶]Purdue University

ABSTRACT

Tick is a new SDR system that provides programmability and ensures low latency at both PHY and MAC. It supports modular design and element-based programming, similar to the Click router framework [23]. It uses an accelerator-rich architecture, where an embedded processor executes control flows and handles various MAC events. User-defined accelerators offload those tasks, which are either computation-intensive or communication-heavy, or require fine-grained timing control, from the processor, and accelerate them in hardware. *Tick* applies a number of hardware and software co-design techniques to ensure low latency, including multi-clock-domain pipelining, field-based processing pipeline, separation of data and control flows, etc. We have implemented *Tick* and validated its effectiveness through extensive evaluations as well as two prototypes of 802.11ac SISO/MIMO and 802.11a/g full-duplex.

1 INTRODUCTION

Software-defined radio (SDR) allows users to build flexible and configurable wireless communication systems. A number of SDR platforms are already available for use for months or even years [2, 27, 32–35, 40]. However, today's requirements on SDR are different, given the latest technology push (e.g., 5G, wireless edge computing) and user demand pull (e.g., low-latency VR/AR applications, runtime control on drones and robotics). Consequently, the new challenge is to both offer sufficient programmability and ensure high performance; it has to be for both physical (PHY) and medium access control (MAC) layers. Unfortunately, we have found it hard to achieve both in the existing SDR systems.

The domain-specific challenge is that, wireless PHY and MAC differ in their data-flow and control-flow operations: (a) PHY is heavy in data-flow processing but lightweight on control flows; MAC needs complex control but little on data processing. (b) PHY works with simple pipeline stages for processing, whereas MAC needs complex branch instructions to perform event handling and control functions. (c) PHY is dominated by processing latency, whereas MAC is sensitive to timing. No existing software architecture nor hardware design solely addresses *all* these challenges to ensure both low latency and good programmability. We thus make a case for exploring software and hardware co-design solutions.

In this work, we describe the design and prototype of *Tick*. *Tick* is a new SDR system that seeks to the best of both worlds. On one hand, it provides a simple programming abstraction via graph of processing elements. Users write individual elements, which implement simple data processing or control-flow functions at PHY and MAC. Complete SDR system is built by connecting elements into the graph. This is made possible by adopting a novel accelerator-rich architecture, which consists of an embedded processor plus a number of user-defined accelerators. The processor focuses on control flows and various MAC event handling. Accelerators offload intensive computation, massive data transfer, fine-grained timing control from the processor, and accelerate these tasks in hardware.

Tick delivers high performance in low latency and high processing throughput using a number of software and hardware techniques. They include multi-clock-domain pipelining, field-based (rather than frame-based) processing pipeline, separation of data and control flows, and shadow registers for control flow and status report.

We have prototyped *Tick* on both PHY and MAC at a medium-priced FPGA development board. The implementation of *Tick* takes three-year efforts and includes PHY and MAC libraries offering 28 elements for PHY and 12 accelerators for MAC. We prototyped an 802.11ac SISO over an 80 MHz channel in *Tick* on Xilinx Kintex-7 FPGA kit. The measured latencies are 1.86 µs for PHY transmitter (Tx), 21.62 µs for PHY receiver (Rx), 17 µs for MAC Tx and 3 µs for MAC Rx. Compared with a simple reference design, *Tick* reduces latency by as much as 1.12 µs for PHY Tx (1.6× reduction, from 2.98 µs to 1.86 µs[1]), 26.39 µs for PHY Rx (2.2× reduction, from 48.01 µs to 21.62 µs), 8920 µs for MAC Tx (497× reduction, from 8938 µs to 18 µs) and 8926 µs for MAC Rx (2976×, from 8929 µs to 3 µs). While achieving low latency, *Tick* consumes moderate amount of FPGA resources (less than 30%). Our case studies of 802.11ac 2×2 MIMO and 802.11a/g full-duplex prototypes further confirm the programmability and latency performance of *Tick*.

2 A REFERENCE DESIGN FOR SDR

In this section, after introducing SDR, we describe a simple reference design. This reference adopts popular architectural choices at both PHY and MAC [22, 26, 33, 40]. We show that this popular design cannot meet the latency requirements of high-end wireless networking systems, e.g., 802.11ac [6].

[1]The latency reduction factor is defined as the reference design latency divided by the *Tick* latency. For example, we denote a reduction factor of 1.6 (2.98 µs/1.86 µs) as a 1.6× reduction for simplicity.

MobiCom '17, October 16–20, 2017, Snowbird, UT, USA

ACM ISBN 978-1-4503-4916-1/17/10…$15.00
https://doi.org/10.1145/3117811.3117834

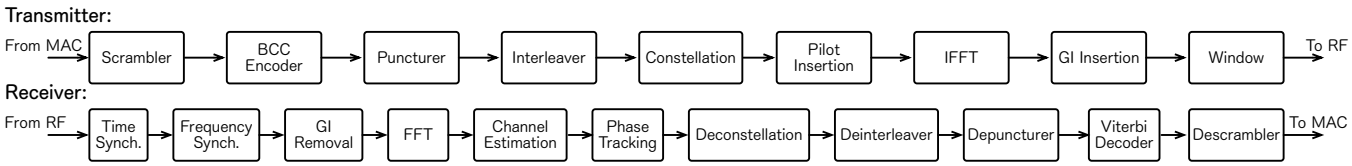

Figure 1: Transmitter and receiver modules for 802.11ac (SISO).

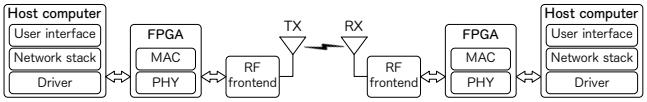

Figure 2: A typical SDR system: Transmitter and Receiver

2.1 SDR Primer

Shown in Figure 2, a typical SDR transmitter/receiver consists of four subsystems: a radio front-end (RF) that transmits/receives radio signals through antenna(s), the physical-layer (PHY) processing unit that uses PHY algorithms to convert the radio waveform into information bits or vice versa, the medium access control (MAC) processing unit that regulates transmissions over the shared wireless channel, and the interface to host computer that delivers higher-layer data packets to MAC or receives data from MAC.

We next briefly review both PHY and MAC. We use the 802.11ac SISO mode [6] as the reference for description. Note that practical wireless systems typically share similar design and algorithms, particularly at PHY.

PHY. PHY transforms information bits into a radio waveform, or vice versa. PHY has both control and data flows. The control flow is relatively simple. It sets configuration parameters. The data flow is more complex. It processes data using multiple functional blocks, which are pipelined together (illustrated in Figure 1). Data traverse these blocks. Each block performs certain computation on the transceived symbols. When the data rate is high (say, 433.3Mbps for 802.11ac SISO over 80 MHz channel, when using 256-QAM modulation type [6]), these PHY blocks require intensive processing power. Moreover, they operate on different types of data at different rates. In 802.11ac SISO, the scrambler works with one bit, while constellation maps each 6-bit block onto a complex symbol that uses two 16-bit numbers for 64-QAM.

From the latency standpoint, processing latency dominates the PHY unit. The required computation also increases with the communication speed.

MAC. MAC arbitrates channel access among all transceivers. Compared with PHY, MAC has more complex control flow but relatively simple data flow processing (e.g., CRC and (de)framing).

For its control flow, MAC needs to issue/accept various events and process each. Moreover, most MAC designs require timely response to critical events. Some even require accurate timing control at the granularity of microseconds. For example, in 802.11 CSMA/CA, short inter-frame spacing (SIFS) is needed between a DATA frame and an ACK frame. The current 802.11ac standard mandates SIFS to be 16 microseconds (µs) [6]. Fine-grained timing poses another challenge at MAC.

In summary, PHY and MAC differ in their data- and control-flow operations: (1) PHY is heavy in data flow processing but lightweight in control flow; in contrast, MAC needs complex control but is lightweight in data processing; (2) PHY typically uses simple pipeline stages for data processing, whereas MAC needs complex branch instructions to perform event handling; (3) PHY is dominated by processing latency, whereas MAC is sensitive to timing.

2.2 A Reference Design

Given the diverse requirements by PHY and MAC, people have explored different architectural techniques. In this section, we include them in a simple reference system. The goal is to (in)validate whether these existing ideas [22, 26, 33, 40] are effective to meet the latency requirements from 802.11ac.

Our reference design is for the 802.11ac SISO mode of Figure 1. At PHY, we use the pipelined modules, which all use a single global clock. Therefore, they form a single clock domain. This is the most popular technique used [22, 26, 40]. Moreover, processing is frame based. The entire frame traverses all pipeline stages. This is another popular technique reported by [22, 26, 33, 40].

At MAC, the reference uses the embedded processor based architecture. The processor handles all related event processing, including backoff, collision handling, timing control (SIFS, DIFS, etc.). More details on the reference are in §7.

2.3 Latency Performance

We next present the latency-related measurement results on the reference design. The 802.11ac standards [6] mandate the air transmission time for PHY, and SIFS between DATA and ACK frames. SIFS thus accounts for latency in both MAC and PHY. These timing requirements pose challenges to both PHY and MAC designs. Our experimental results show that, current reference SDR design cannot meet the stipulated timing requirements.

PHY. We test the 802.11ac prototype operating at 80 MHz channel width, using 64-QAM at 3/4 rate, for both the transmitter (Tx) and the receiver (Rx). Tx works at the maximum clock frequency 90 MHz, and Rx at the maximum clock frequency 45 MHz. We record the latency and the processing time for each frame. The latency is defined as the duration from the time the first incoming bit is received to the instant this first bit is processed and sent out. The processing delay is defined as the time it takes to process a whole frame.

The results are shown in Table 1. We make two observations. First, long latency is observed at the Rx side. It takes 48.01 µs for the first bit to be processed. However, latency at Tx is much shorter. Second, PHY receiver cannot meet the air transmission time deadline. Under 80 MHz channel, the air transmission time for a 100 B frame takes 44 µs, while a 1500 B frame (the maximum transmission unit size, MTU) takes 88 µs. To continuously process the stream of incoming frames, the PHY Rx should process each frame at least faster than the air transmission time plus SIFS. However, neither (1500 B) MTU-sized frame nor larger aggregated frame can catch up with the incoming speed. The (100 B) frame reception seems fine.

Reference Design	PHY		MAC	
	Tx	Rx	Tx	Rx
100 B:				
Latency	2.98 µs	48.01 µs	613 µs	604 µs
Proc. time[a]	23.85 µs	57.91 µs	615 µs	605 µs
1500 B:				
Latency	2.98 µs	48.01 µs	8938 µs	8929 µs
Proc. time[b]	44.88 µs	195.52 µs	8954 µs	8938 µs

[a] PHY air transmission time for 100 B is 44 µs.
[b] PHY air transmission time for 1500 B is 88 µs.

Table 1: Timing performance for the reference design.

However, when sending ACK to the sender (that needs 2.98 µs to prepare the first bit at PHY), the receiver does not have any budget left for its MAC (57.91 µs + 2.98 µs > 44 µs + 16 µs).

MAC. We show that the reference MAC, which relies on the embedded processor for processing, cannot meet the timing deadline. We test the CSMA/CA MAC using the MicroBlaze processor. We record the latency and the processing time for each frame at the MAC layer (Table 1).

We first observe long latency at both Tx and Rx. It takes more than 613 µs for the first byte to be processed for the small frame, and nearly 8938 µs for the MTU-sized frame. Second, SIFS timing cannot be met. MAC and PHY together substantially exceed SIFS (18.68 µs + 604 µs + 2.98 µs + 1 µs = 626.66 µs > 16 µs, details in §8). Furthermore, MAC alone contributes at least 604 µs. We thus make a case for exploring new architectural ideas for low-latency SDR.

3 ARCHITECTURE

In this section, we describe the architecture of *Tick*. *Tick* seeks to achieve low latency in SDR by applying hardware and software co-design techniques.

3.1 Components

We next introduce the main components in *Tick*.

Hardware Components. *Tick* uses two hardware components in its architecture: an FPGA board for PHY and MAC, and a commodity wide-band RF board. The FPGA-based prototyping offers maximum flexibility to explore hardware and software co-design ideas. The RF front-end offers a well-defined interface between digital and analog. It contains A/D, D/A, etc. It further supports single or multiple antennas, which enables SISO and MIMO operations.

Tick has two communication interfaces. One is between the host computer and FGPA using USB 3.0 or PCIe. The other is from the vendor's FMC interface to connect PHY and RF in low latency.

Software Components. The software stack in *Tick* provides the programming support and systems services for implementing various PHY and MAC protocols (Figure 3). The communication module facilitates massive data exchanges with the upper-layer IP protocol and the RF unit. Moreover, *Tick* provides a number of techniques to greatly improve the programmability and latency performance of PHY and MAC processing. To this end, the PHY and MAC libraries offer commonly used functions. The PHY and MAC runtime support allows for users to code, compile, and run their customized functions at ease.

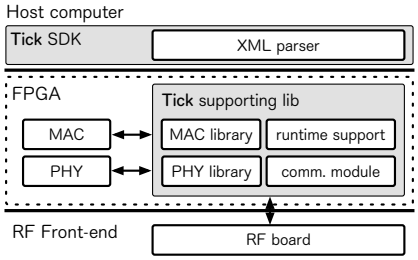

Figure 3: *Tick* software stack.

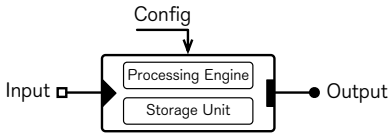

Figure 4: An *element* in *Tick*.

3.2 Programming *Tick*

To write a complete SDR system with both PHY and MAC, *Tick* provides a simple programming abstraction of "a graph of elements". A new wireless design is configured as a directed graph of elements. An element is the basic unit for modular programming in *Tick*. Each *element* represents a processing unit or function. All processing actions performed inside a function are encapsulated in an element. The edge, or connection, between two elements represents the route for message transfer from one element to the other. Therefore, the graph resembles the flowchart, with connections denoting message flow, and elements being actual objects that process messages.

Each element can have at most four components (Figure 4): the input and output ports, the processing engine (PE), the storage/memory unit, and the configurations. Ports are the endpoints of connections between elements, and each element can have a small number of ports for input and output. Messages are passed among elements via ports. PE implements the computation and processing needed by the element. The storage unit buffers the data or control information. Configurations enable us to set parameters and configurations for an element.

Elements are connected via ports. The connection is unidirectional. Each output port of an upstream element is thus connected to the input port of a downstream element. For simplicity, each port is only allowed for connect once. To offer flexibility, *Tick* provides special elements of multiplexer and demultiplexer for input merge and output split. In addition, an element without an input port is called a source, while an element without an outport port is a sink.

The embedded processor operates at the MAC layer only. It handles MAC event processing, offloads processing-heavy or communication-intensive tasks to MAC accelerators (a special class of elements to be elaborated next), and provides control and configurations on accelerators. However, it does not process MAC data frame directly.

Tick lets programmers use the standard XML language to perform several tasks for an element: connection of elements, memory/storage size, and parameters of interest. The coding of PE can be in high-level programming language HLS (High-Level Synthesis) C++ or in low-level language Verilog for FPGA.

Programming PHY is to write each PHY element in HLS C++ or Verilog, and interconnects them to form the pipeline. Users can use either XML file or our IDE (its GUI similar to GNU Radio) to specify

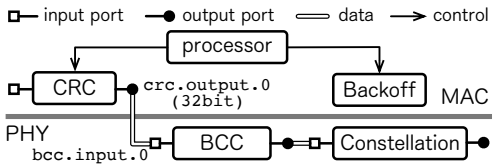

Figure 5: A toy example using *Tick*.

Program Snippet 1 Defining a BCC Encoder *element*

```
<name>bcc</name>
<input> <id>0</id> <width>32</width> </input>
<output> <id>0</id> <width>64</width> </output>
<configuration> <name>rate</name> <width>16</width> </configuration>
```

Program Snippet 2 Interconnecting MAC and PHY *elements*

```
<!-- interconnect elements in PHY -->
<connect>
    <port>bcc.output.0</port>
    <port>constellation.input.0</port>
</connect>
<!-- interconnection between MAC CRC and PHY BCC -->
<connect>
    <port>crc.output.0</port>
    <port>bcc.input.0</port>
</connect>
```

the configuration for interconnetion. Programming MAC needs to work with both the embedded processor and accelerators. Coding the embedded processor using API is straightforward (details in §7). Users write event-handling programs for MAC in C. An accelerator is also a special class of element. Programming it can be in HLS C++ or Verilog. Users can call common and systems accelerators (e.g., CRC, random backoff, timer). The configurations for accelerators are also via XML.

An Illustrative Example. We use a toy example (Figure 5) to show how to program the MAC and PHY chain. Assume the user builds a simple SDR transmitter, which has the following functions: if not in MAC backoff, the sender starts CRC computation and sends the frame to PHY, which has two pipeline stages of BCC encoding and constellation mapping.

The MAC layer in the resulting design consists of the embedded processor, a CRC accelerator, and a Backoff accelerator; The PHY layer has a BCC Encoder element and a Constellation element. Given this model, the user needs to take three steps.

First, the user populates *element*s using the template provided by *Tick*. It is done by defining `input`, `output` and `configuration` for the *element*. Program Snippet 1 shows the definition for the BCC Encoder. It defines the *element* name, input, output and one parameter (`rate`). The data widths are defined accordingly. *Tick* will automatically generate the corresponding Verilog module code, while the user uses either Verilog or HLS C++ to code the PE. The Constellation, CRC, and Backoff elements are defined similarly.

Second, the user interconnects the *elements* to make a working pipeline. It is also done using XML, as shown in Program Snippet 2. Note that all MAC accelerators are connected to the embedded processor by default.

Finally, the user implements proper MAC event-handling branch instructions within the processor. It thus interacts with each accelerator via reading and writing to corresponding registers (called CSR) through API, as shown in Program Snippet 3.

Program Snippet 3 MAC processor control logic

```
Reg_interrupt(INT_BackoffDone, handle_backoff); // register a Backoff
Write_CSR(backoffReqAddr, 1); // start Backoff
Write_CSR(crcReqAddr, 1); // calculate CRC

void handle_backoff() { // backoff interruption handler routine
  Write_CSR(backoffReqAddr, 0); // reset Backoff
  Write_CSR(sendToPHYReqAddr, 1); // send out CRC
}
```

4 DESIGN OVERVIEW

In this section, we provide a design overview for *Tick*.

4.1 Design for Programmability

Tick supports high programmability at both PHY and MAC. The overall idea is to carefully instrument an element, so that it balances latency performance and modular design.

PHY. Proper design of PHY elements enables a user to readily program PHY via the pipelined operation of such elements.

A PHY element uses its input/out ports to separate data flow (message passing) from control flow (configurations, or dynamic parameters such as frame size for the current frame). Each port needs to specify its bit-width (i.e., how much bits are needed). Control flow uses a small number of bits, and can be stored internally using the shadow control and status registers (CSRs). They *shadow* the CSRs from a special PHY accelerator, which passes information between MAC and PHY; see Figure 6 for an illustration.

Data flow needs a large number of bits. They are transferred between two connected elements via asynchronous FIFO (aFIFO). aFIFO provides the data queue for an upstream element and a downstream one; see Figure 7 for an illustration. It can also be viewed as a special element, which the user does not need to be aware of. It is automatically generated between elements by *Tick*'s runtime support.

Each PHY element has a special configuration parameter, i.e., its operating clock. This parameter enables multi-clock-domain pipeline, as elaborated in §5.

MAC. The embedded processor commands the control flow and configurations for accelerators. However, it does not process data messages directly. Accelerators offload computing, communication, and timing tasks from the processor. They directly work with data messages. Data are passed among involved accelerators via DMA, thus bypassing the processor.

An accelerator is a special class of element. It has all four components of an element. Each accelerator can also run on its own local clock. The embedded processor is a general-purpose processor, and we use MicroBlaze in our prototype. By default configuration, all accelerators are connected to the processor.

Passing Information between MAC and PHY. MAC and PHY need to pass critical information at runtime for efficient processing, e.g., current frame size. This is done by implementing an accelerator element in *Tick*. Therefore, the processor operates at the MAC layer only, but uses the special accelerator to pass information between MAC and PHY.

4.2 Design for Low Latency

Tick explores several software and hardware co-design techniques, to reduce latency and sustain high processing throughput. At PHY,

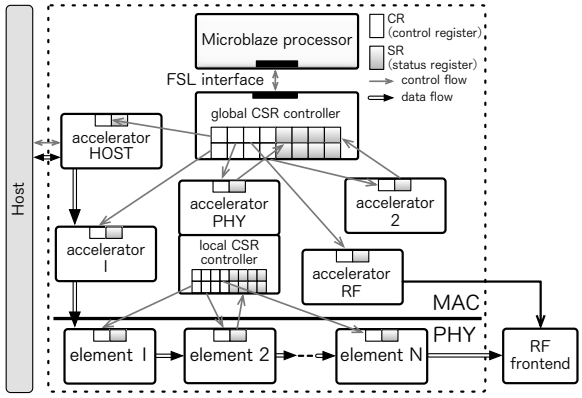

Figure 6: Control flow for both MAC and PHY in *Tick*.

two ideas are applied. First, multi-clock-domain pipelining (MCDP) lets each element on the pipeline operates with its separately generated clock. This greatly simplifies the global clock distribution, which is quite challenging to realize for ultra-high-speed PHY processing. Second, we apply field-based, rather than frame-based, processing to further reduce latency. It can customize the processing pipeline for different fields of the PHY frame. No all fields need to traverse the entire pipeline, thus reducing overall latency.

At MAC, we propose the accelerator-rich, embedded processor-aided design. It efficiently handles various events while meeting the timing deadline defined in tens of microseconds (e.g., SIFS) in modern wireless systems.

4.3 Aiming at the Best of Both Worlds

Tick is designed to be easily programmable and readily extendable, while simultaneously ensuring low latency. It differs from all existing SDR platforms, which do not aim to achieve both goals. Take the popular WARP system as an example. On one hand, the hardware-centric design by WARP (say, 802.11n) can meet the SIFS/DIFS timing requirements [36]. However, such fine-tuned, carefully instrumented designs tightly couple the PHY function blocks at the cost of easy programmability. Modification would not be easy, if impossible at all, without precise estimation on the entire signal processing flow. On the other hand, WARP offers a highly programmable option for PHY via WARPLab. However, WARPLab cannot meet the stringent 802.11 timing requirement, as concurred by several independent studies [21, 37, 37, 39, 41]. Consequently, it is even more challenging, if not feasible at all, to extend the current 802.11n design to the next-generation 802.11ac family.

Tick overcomes this dilemma by aiming at both goals. It introduces aFIFO to enable multiple-clock-domain pipeline (details in §5). Each aFIFO does incur small latency, which is, however, offset by the benefits gained from loosely-coupled modules and flexibilities to update anyone. Moreover, field-based processing significantly reduces the efforts to bypass bottleneck modules, e.g., IFFT. In contrast, a hardwired SDR design saves little on the aFIFO-incurred latency between modules, but results in high cost on diminished flexibility, extendability, and programmability.

We next elaborate on the design details in §5 and §6.

Figure 7: Multi-clock-domain pipeline (MCDP) in PHY.

5 MINIMIZING LATENCY AT PHY

We now describe the two techniques at PHY: multi-clock-domain pipeline and field-based pipeline processing.

5.1 Multi-Clock-Domain Pipeline (MCDP)

The first technique is multi-clock-domain pipeline (MCDP) [28, 30] of elements (Figure 7). In a nutshell, MCDP uses a globally-asynchronous, locally-synchronous clocking style [20]. Each element operates with its clock. Multiple adjacent elements can use the same clock to form a domain. Synchronous design can be used in each domain. Finally, all elements are interconnected to form the cross-domain PHY pipeline.

The choice of MCDP is motivated by the SDR PHY. First, it accommodates the diversity in computing workload of different elements at PHY. Second, it enables to customize the PHY pipeline stages to the operation demands of individual elements. Third, it offers better modular design at PHY. The design of each domain is no longer constrained. It can optimize the tradeoffs among clock speed, latency, and exploitation of element-level parallelism.

Our experience shows that, MCDP is not very effective at the transmitter of 802.11ac (Figure 1). This is because the IFFT module poses as the single processing bottleneck; adjusting clocks at other modules does not lead to sizable latency reduction. However, MCDP is quite effective at the receiver. The elements of Viterbi Decoder, FFT, and Time Synchronization all pose as processing bottlenecks. Using local clocks helps to adjust and match the bottleneck speeds.

We need to address three issues for MCDP: (1) How to enable message passing among elements across domains? (2) How to further reduce the latency overhead due to too many domains? (3) How to reduce latency when MAC exchanges information with PHY elements in different clock domains? We next elaborate on our solution to each.

Asynchronous FIFO for Inter-Domain Communication. Inter-domain communications in MCDP use explicit message-passing channels to communicate messages between elements in different domains.

In this work, we leverage the low-latency aFIFO for asynchronous communication across domains [10]. The design uses full and empty signals to indicate the occupancy of the FIFO. The empty and full signals are generated by FIFO itself. The empty signal is synchronized to the consumer's clock, while the full signal is synchronized to the producer's clock.

Grouping to Reduce Clock Domains. Inter-domain synchronization via aFIFO increases the number of clock cycles in MCDP. The issue becomes severe, if each element runs at its own clock and many aFIFOs are used.

Our solution gathers multiple adjacent elements together to form a group. Elements in a group operate at the same clock. This way, we can reduce the number of aFIFOs. Grouping balances the tradeoff among processing capability of each element, the communication delay due to aFIFO, and the number of pipeline bottlenecks.

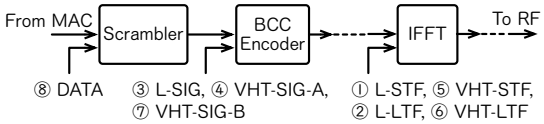

Figure 8: Field-based processing in 802.11ac PHY.

We automate the process of grouping elements. We compute the overall latency with grouping (i.e., reduced aFIFO but increased latency at elements in a group) and without grouping (minimal latency at each element but increased latency due to more aFIFOs). Whenever the grouped latency is smaller, we merge the element to the group; otherwise, we start a new group for the element.

Latency on PHY and MAC Interactions. We exploit shadow registers for efficient information exchange between PHY and MAC. The shadow register design provides the ability to load or shadow the contents of a primary CSR into a shadow CSR at the completion of a stored instruction. This is accomplished in 6 to 8 clock cycles (far below 1 µs given our clock frequency), with all registers being shadowed. We defer the discussion of CSRs to §6.

5.2 Field-Based Pipeline Processing

The next technique is to exploit the field-based, rather than frame-based, pipeline at PHY. Given the different fields of PHY data, we customize the pipeline (both which stages and the number of stages) to speed up computation. Therefore, only the data will go through the entire pipeline, while other PHY fields (headers and signals) only traverse a portion of the pipeline stages. We apply this technique at both the transmitter and the receiver of software radio.

The field-based pipelining is based on the observation that different fields play different roles at PHY processing. For example, the long/short training fields are used for channel estimation and training purpose only, and the content of these bits is not needed. Therefore, they do not need to traverse all stages of the entire pipeline. Instead, they can bypass certain stages, thus reducing latency. Moreover, this further enables concurrent processing along different parallel paths for different fields, contributing to further latency optimization. Figure 8 illustrates an example.

Customized Pipeline Stages for Fields. We apply domain knowledge to customize the pipeline at both Tx and Rx.

Take 802.11ac SISO of Figure 8 as an example. Note that PHY in many other wireless technologies share similar features. At the transmitter, both long and short training sequences enter the pipeline from IFFT, signals enter from the BCC encoder, but data will traverse the entire pipeline starting from the Scrambler. At the receiver, the short training sequence exits the pipeline after the Frequency-offset element, while the long training sequence exits after Channel-estimation. The signals exit after the Viterbi-decoder element, while the data go through the entire pipeline.

Control Flow for Fields. Compared with frame-based processing, field-based pipeline reduces processing latency but at the cost of finer-grained control flow at the field (but not the frame) level. The specific control information to enable field-based pipelining includes: how many fields in the frame, the order of fields, the starting point of a field and the field length, and the modulation type for each field.

To facilitate field-based processing, *Tick* automates the generation of control flow for fields. To this end, a user only needs to fill in the configuration table for PHY fields. Such configuration parameters are passed to each related element. The customized pipeline can then be constructed for each field of the PHY frame.

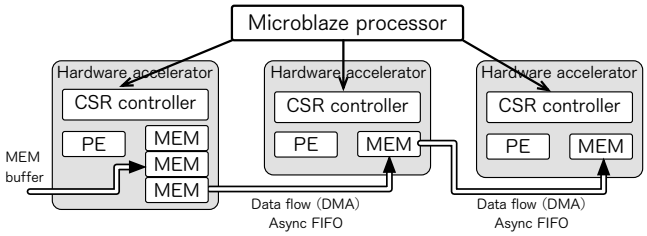

Figure 9: MAC layer accelerators architecture.

6 DESIGN FOR LOW-LATENCY MAC

We now describe our efforts to reduce MAC latency. Following the industry practice, we first split MAC functions into two portions. The high MAC implements functions of accepting frames from higher-layer IP modules, adding frame headers, and handling management frames (beacons, etc.). It is non-time-critical, thus being implemented in the device driver on the host computer. The low MAC copes with timing-critical functions; this is the focus here.

We adopt the accelerator-rich design [11, 24], aided by an embedded processor (Figure 9). The processor is definitely needed, to preserve programmability on event-based handling at MAC. However, as we have seen in §2.3, the embedded processor only design cannot ensure low latency. Offloading computation-intensive tasks from the processor does help to reduce latency. However, it is not sufficient. We further explore the idea of decoupling control and data flows as elaborated next.

Decoupling Control and Data Flows. We separate the control and data flows at the MAC layer. The control-flow communication between the processor and an accelerator is done by read and write operations on CSRs. The embedded processor works with the global CSRs only via the FSL (Fast Simplex Link) interface, while each accelerator retains a copy of shadow CSRs locally. See Figure 6 for an illustration. For a control register (CR) within CSRs, the embedded processor performs write operation only, whereas the accelerator executes read operation only. In contrast, given a status register (SR), the processor performs read operation only, while the accelerator runs write operation only. Given the separation of CRs and SRs, no read/write conflicts are possible for the control flow between the processor and each accelerator. Given the low volume of control flow, no sizable latency is observed.

Figure 6 further shows that, we have three levels of CSRs. The local CSRs at each accelerator shadow the global CSRs, and the CSRs at any PHY element further shadows the CSRs at the PHY accelerator (that handles information passing between MAC and PHY layers).

As illustrated in Figure 9, data flow among accelerators leverages the memory unit in each accelerator. Message exchanges are pipelined between communicating accelerators. While the data is read from the memory of the previous accelerator, data are concurrently written into the next accelerator. Therefore, the accelerator is not blocked in processing by the massive data exchanges via DMA. When accelerators use multi-clock domains, data exchange can also be done via the asynchronous FIFOs.

Program Snippet 4 *Tick* API for low MAC

```
// write value to the control register
void Write_CSR(unsigned int addr, unsigned int val);

// return value stored in the status register
int Read_CSR(unsigned int addr);
```

7 IMPLEMENTATION

We first present *Tick*'s baseline implementation, and then provide more details on PHY, MAC and other issues.

Prototype Environment. We prototype *Tick* with both PHY and low MAC, with the 802.11ac [6] SISO as our base implementation. It is done on the Xilinx Kintex-7 FPGA KC705 Evaluation Kit [38]. The embedded processor is MicroBlaze. The interfaces between host computer and the FPGA board include both USB 3.0 (converted by CYUSB3KIT-003 explorer kit [14, 15] to FMC interface) and PCIe [17, 18]. We select a commodity RF board using AD9371 [1] with the FMC interface. The RF front-end is capable of transmitting and receiving two 80 MHz channels within the frequency range from 300 MHz to 6 GHz. Figure 10 shows the *Tick* system (without the host computer).

The *Tick* library is programmed on Ubuntu 14.04. Our implementation results in 18 098 lines-of-code (LoC) in Verilog for PHY (8267 LoC for PHY Tx, 9831 LoC for PHY Rx), and 214 LoC in XML (96 LoC for connection and 118 for module definition. The MAC consists of 7412 LoC in Verilog HDL for accelerators, and another 2034 LoC in C for the embedded processor logic. To support connectivity and programmability, the *Tick* library has 4226 lines of driver code in C to communicate with the host driver and 1253 LoC in Python for the XML parser.

Runtime Support. *Tick* is offered as a standalone library to users. In PHY modules, *Tick* provides 28 standard elements written in Verilog to support 802.11a/g [5], 802.11n [5] and 802.11ac [6] protocols. They follow algorithms/specifications from the wireless standards. For example, we provide FFT-64, FFT-128, FFT-256, Viterbi Decoder, and Channel Estimation elements. In MAC modules, *Tick* provides 12 accelerators to support programmable MAC, including CRC-32, Backoff, Global Timer, communication accelerators that interact with PHY, RF and host computer, etc.

As a result, programmers only need to use XML configuration files to call PHY elements/MAC accelerators from the *Tick* library and interconnect them, similar to Program Snippet 2. We implement an XML parser in Python to convert user-defined configurations into Verilog HDL code for the elements/accelerators. Users can use Vivado Logic Analyzer provided by Xilinx to debug their implementation.

API. *Tick* provides two levels of APIs, one for low MAC and the other for host control logic. The API for low MAC controls element/accelerator via CSR, as Program Snippet 4 shows two example cases. The API for host-level control has three aspects, including API for data (e.g., send data frames from host to MAC using DMA), API for interrupts (e.g., register an interrupt at host), and API for control (e.g., get/set config parameters).

7.1 PHY Issues

Implementing an Element. In addition to the default PHY elements provided by current *Tick*, users can readily customize their own ones. They can create a template in XML, fill in the various parameters, and write PE in Verilog or HLS C++.

(MHz)	Scrambler	BCC	Puncture	Interleaver	Constellation	Insert	IFFT	GI
Max	500	600	600	430	650	460	90	430
Tick	500	500	500	400	500	400	90	400

(a) PHY Tx

(MHz)	Time sync.	Freq. offset	GI	FFT	Channel est.	Phase	Decons-tell.	Deint-leaver	Depunc-turer	Viterbi	Descr-ambler
Max	45	120	710	100	340	460	420	460	510	170	710
Tick	45	100	500	100	333	333	333	333	500	167	510

(b) PHY Rx

Table 2: Working frequencies for *Tick*-PHY using MCDP.

Choosing Clock Frequencies for MCDP. We implement MCDP by providing maximum clock frequency that each element can run on the given FPGA board. Due to FPGA resource and layout constraints, the working clock frequency may not reach the ideal maximum value. The clock frequencies *Tick* used on our FPGA board are listed in Table 2.

Eliminating Jitter Incurred by aFIFO. Another challenge comes from the strict timing requirement to send a frame over the air. For example, full-duplex system must be aligned for each I/Q sample [4, 8, 12, 21]. However, the aFIFO between elements introduces an uncertain latency jitter, which can be up to ± 0.15 µs. We eliminate this jitter in *Tick* by leveraging the global clock timer provided by MAC. Jitter-sensitive aFIFOs will fetch a timestamp from the global clock when the data can be sent out. The jitter is thus cancelled since the aFIFO's sending times are aligned.

Grouping Elements to Avoid Resource Overuse. Due to the clock resource constraint, the user may not use a separate clock for every element. *Tick* automatically analyzes adjacent elements to determine whether they should be grouped together. The elements in the same group use the same clock.

Handling Field-Based Processing. The field-based processing follows the configuration table specified by users. Such a table is two-dimensional, specifying field length and how it should be processed by each element. Next, our XML parser automatically generates a corresponding table in FPGA and creates shadow registers in each element. The element works immediately, once the corresponding field of incoming data is ready.

A challenge is to guarantee correct ordering. We adopt the "stop-and-wait" strategy in processing fields. The configuration table also regulates the field order. The later fields must wait until the early fields have been processed.

7.2 MAC Issues

Global Controller via CSRs. In our accelerator-rich architecture, the embedded processor controls each accelerator via CSRs. To provide scalable control over multiple accelerators, we define shared control registers in the global CSRs. The shared control registers can be read by multiple accelerators, so that global parameters can be distributed efficiently.

The control-flow latency mainly comes from the delay of reading/writing a register, which costs 5 to 6 clock cycles. We reduce such latency between embedded processor and CSR to 2 to 3 clock cycles, by reading/writing to consecutive addresses. Taking the latency caused by aFIFO between global CSR and shadow CSR into consideration (6 to 8 clock cycles). The entire control-flow latency

between embedded processor and CSR in *Tick* is 11 to 14 clock cycles.

Implementing Accelerators. We implement four types of accelerators in MAC: computation-intensive accelerators (e.g., CRC checksum), timing-critical accelerators (e.g., Backoff), interface processing accelerators (e.g., interact with Host/PHY layer), and accelerators for grouped small modules (e.g., assemble ACK).

1. Computation-intensive accelerators. We offload computationally intensive CRC checksum in *Tick* to accelerators, so that the processor does not get stuck in fetching large data.

2. Timing-critical accelerators. We provide timing-related accelerators with access to a shared global clock and timestamp. For the processor, *Tick* uses a dedicated bus to reduce the latency of reading the timer value. Such a timing control mechanism avoids the significant delay and timing errors if reading the clock from the register.

The timeout notification from one accelerator to another is done through timestamp passing. The source accelerator first increments the global timer value it reads to the expected timeout value, and passes it to the processor as a timestamp. The processor then passes the timestamp to the timeout accelerator. Finally, when the timeout accelerator counts to the expected timestamp, it starts to alarm an event. The clock calibration unit in each time-related accelerator is also implemented using timestamp passing. It computes the latency offset based on the value it reads.

3. Interface processing accelerators. We next implement the accelerator for communicating with the host and with PHY. The host accelerator *prefetches* data from the host computer. It thus needs large memory to buffer data, and *Tick* sets eight frames as its buffer memory size. We also implement the PHY accelerator to provide isolation and control at the MAC layer. It communicates with the CSRs in each PHY element.

4. Accelerator for grouped small elements. Finally we use an accelerator for grouped small elements to reduce control overhead. Due to their relatively simple logic, one accelerator suffices to handle. We put assemble ACK, frame check, set retry bit into this accelerator in *Tick*.

7.3 Other Issues

Choosing Interface between RF and PHY Boards. Different RF boards introduce different latencies because they use different interfaces. For example, USRP N210 [16] uses the Ethernet interface, which incurs over 10 µs delay due to the Ethernet protocol. Therefore, we choose AD9371 [1] that adopts the FMC interface (a low-latency parallel data bus).

Choosing Interface between Host and MAC Board. Different interfaces between host and MAC introduce different latency. We have considered both PCIe and USB 3.0; they both provide sufficient throughput to 802.11. Our experiment shows that, the average latency of the PCIe interface (15.76 µs) is lower than that of USB 3.0 (53.25 µs). Since we implement a host accelerator which prefetches data from the host, the latency effect is offset. We thus choose USB 3.0, which is more accessible for laptops or even tablet PCs.

Dual-Interface Driver. We implement two interfaces when developing the *Tick* driver at the host, one to the user space and the other to the TCP/IP stack. The user-space interface provides direct access to FPGA at the user space for debugging. The network interface connects to the host's mac80211 interface (link-layer driver) in the kernel, so that the *Tick* can be used as an 802.11 device.

	Tick	**PHY Tx**	**PHY Rx**
SISO:			
	Latency	1.86 µs	21.62 µs
	Proc. time	21.26 µs	24.32 µs
MIMO:			
	Latency	1.85 µs	23.57 µs
	Proc. time	23.14 µs	26.28 µs

(a) 100 B

	Tick	**PHY Tx**	**PHY Rx**
SISO:			
	Latency	1.86 µs	21.62 µs
	Proc. time	42.07 µs	61.85 µs
MIMO:			
	Latency	1.85 µs	23.57 µs
	Proc. time	32.56 µs	43.55 µs

(b) 1500 B

Table 3: PHY latency and processing time of *Tick* 802.11ac.

8 EVALUATION

We first demonstrate that *Tick* achieves low-latency using benchmarks on PHY (§8.1), MAC (§8.2) and software and hardware interfaces (§8.3). Next, we assess the overall performance of *Tick* with latency, throughput, correctness, resource consumption, coding effort and comparison with state-of-the-art SDRs (§8.4). In this section, we mainly use 802.11ac SISO. More assessment of *Tick*'s extension for two case studies of 802.11ac MIMO and full duplex is presented in §9.

8.1 PHY

We implement and evaluate *Tick*-PHY design using the same configuration as the reference 802.11ac SISO design (§2). We have 9 elements in PHY Tx and 11 elements in PHY Rx. Our PHY implementation uses 64-QAM at 3/4 rate operating over 80 MHz channel for both Tx and Rx. We test two frame sizes: 100 B and 1500 B.

MCDP Performance. We first gauge MDCP performance at PHY. With MCDP, the overall latency and the processing time are reduced for both Tx and Rx at PHY (see SISO in Table 3).

For PHY Rx, the latency decreases from 48.01 µs (reference design in §2) to 21.62 µs, about 2.2× reduction. Note that, this latency remains identical for both frame sizes, because it takes the same amount of time for the first bit to be received and processed. *Tick* reduces processing time using MCDP as well. For small frame (100 B), the processing time decreases from 57.91 µs to 24.32 µs, 2.4× reduction. The reduction is even greater for the MTU frame (1500 B), from 195.52 µs to 61.85 µs (3.2× reduction).

For PHY Tx, *Tick* drops the latency from 2.98 µs to 2.49 µs (1.2× reduction). For small frames, the processing time changes from 23.85 µs to 21.89 µs (1.1× reduction). For MTU-sized frames, the processing time reduces from 44.88 µs to 42.70 µs (1.1× reduction).

MCDP Latency Reduction Analysis. The latency reduction achieved by MCDP is mainly through clock frequency speed up on elements. Specifically, there are three bottleneck elements in PHY Rx: time synchronization, FFT and Viterbi decoder, whose processing clock cycles are an-order-of-magnitude more than other elements. In the reference design, all three work at the lowest frequency (45 MHz). Using *Tick*, clock frequency for FFT is 100 MHz (2.2× speedup), and Viterbi decoder uses 167 MHz (3.7× speedup), as shown in Table 2b. For PHY Tx, latency reduction is also achieved through clock frequency speed up. However, PHY TX only has a single bottleneck in IFFT. Therefore, latency reduction at Tx is not as much as at Rx.

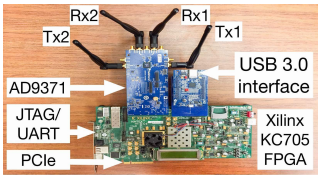

Figure 10: *Tick* system (w/o host computer).

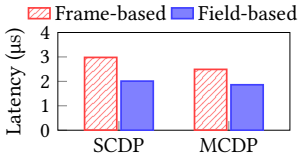

Figure 11: Field-based processing reduces latency (PHY Tx).

	Tick	MAC Tx	MAC Rx
Latency		17 µs	3 µs
Proc. time		18 µs	4 µs

(a) 100 B

	Tick	MAC Tx	MAC Rx
Latency		18 µs	3 µs
Proc. time		34 µs	19 µs

(b) 1500 B

Table 4: MAC latency and processing time w/ accelerators.

Field-Based Processing. Field-based processing further reduces latency over frame-based processing. We test both SCDP and MCDP in *Tick*. Figure 11 shows that, using field-based processing, the latency reduction is 0.97 µs (2.98 µs to 2.01 µs) for SCDP design. For MCDP, it reduces 0.63 µs (2.49 µs to 1.86 µs) for latency. The latency reduction is the same for both frame sizes.

Field-Based Processing Latency Reduction Analysis. In 802.11ac, field-based processing can start only as early as IFFT. The latency of the upstream elements is thus reduced. In 802.11ac, the reduced latency is visible but not significant. This is because IFFT, being the main latency contributor, is not skipped. Other 802.11 variants, e.g., 802.11a/g, may observe larger reduction by applying field-based processing, if the latency bottleneck element can be skipped. To meet the stringent SIFS timing requirement, every latency reduction factor matters. As an optimization technique, field-based processing further pushes latency reduction at PHY to the limit.

8.2 MAC

Accelerator-Based Latency Reduction. With the accelerator-based MAC architecture, *Tick* successfully reduces latency by more than *two orders of magnitude*. Table 4 summarizes the results.

For the Rx chain, the overall latency drops from 604 µs to 3 µs (201× reduction) for a small frame (100 B). *Tick* achieves 151× reduction (from 605 µs to 4 µs) in processing a complete frame. For MTU frame (1500 B), the overall latency reduction is 2976×, from 8929 µs to 3 µs. The processing time decreases from 8938 µs to 19 µs (470× reduction) in processing a full frame.

On the Tx chain for small frame (100 B), the overall latency drops from 613 µs to 17 µs (36× reduction); the processing time for a full frame drops from 615 µs to 18 µs (34× reduction). For MTU frame (1500 B), the overall latency decreases 497×, from 8938 µs to 18 µs. The processing delay for a full frame changes from 8954 µs to 34 µs (263× reduction).

Latency Reduction Analysis. The accelerator-rich architecture of *Tick*, which supports and coordinates multiple accelerators, is also the key enabler for low latency. Table 5 shows the latency breakdown for the reference design (§2). For both Tx and Rx, the latency has five factors: (L1) control flow latency at the interface between MAC and host (Tx)/PHY (Rx); (L2) processing time on the MAC frame header in the processor; (L3) the time to load data to the processor via DMA; (L4) the time to compute CRC checksum in the processor; and (L5) the time to send the frame to PHY (Tx), or to the host (Rx) . Among them, L1 and L5 are necessary overheads to control MAC and send data to the next stage. *Tick* does not effectively reduce these two factors. However, the two accelerators we implemented in *Tick* reduce L2, L3, and L4, effectively. The frame header accelerator modifies/checkes the header for data frame and reduces L2. The CRC accelerator offloads calculation from the processor, thus reducing the latency for L3 and L4.

Latency Breakdown	Tx		Rx	
	100 B	1500 B	100 B	1500 B
L1. Control information to MAC	12 µs	12 µs	<1 µs	<1 µs
L2. Frame header processing	10 µs	10 µs	12 µs	12 µs
L3. DMA data to processor	31 µs	447 µs	32 µs	448 µs
L4. CRC checksum calculation	560 µs	8469 µs	560 µs	8469 µs
L5. Transfer to PHY(Tx)/Host (Rx)	2 µs	16 µs	1 µs	15 µs

Table 5: MAC latency breakdown of the reference design.

In the reference design, L2, L3, and L4 together form the major latency bottleneck (at least 600 µs for the small frame and 8900 µs for the MTU-sized frame). Using accelerators, *Tick* reduces them to <6 µs for Tx and <3 µs for Rx (compared with Table 4). Note that, ad hoc design practice which accelerates CRC or frame header alone, still cannot meet the SIFS timing requirement.

8.3 Software and Hardware Co-design

Latency Overhead. The interfaces between MAC and PHY, PHY and RF front-end are two factors contributing to potentially large latency overhead. However, neither is an issue in *Tick*. The latency between PHY and MAC comes from aFIFO, which takes 6 to 8 clock cycles to feed the data from MAC to PHY. Under the maximum clock frequency (100 MHz) that aFIFO runs, it takes less than 0.08 µs for PHY to receive a data bit sent from MAC. For the AD9371 RF board, the latency at the RF front-end is less than 1 µs.

Note that, the interface between the host and FPGA also incurs communication latency. However, it does not affect timing-critical functions at MAC (e.g., DIFS, backoff, SIFS), because we implement a host accelerator, which prefetches data from the host. To support sufficient throughput for 802.11, both PCIe (15.76 µs latency on average) and USB 3.0 (53.25 µs latency on average) are viable choices.

SIFS Timing Requirement. The time interval between the DATA frame and an ACK frame should be smaller than SIFS. This interval consists of four components: (a) the delay at PHY Rx, from receiving the last frame sample at RF, to delivering the last byte of the frame to MAC; (b) the delay at MAC, from receiving the last byte from PHY Rx, to delivering the first byte of ACK to PHY Tx, after MAC finishes all Rx and Tx processing; (c) the delay at PHY Tx, from receiving the first byte of ACK from MAC to sending its first sample to RF; and (d) the communication delay, including PHY-MAC interface delay and the RF delay.

Following the above breakdown analysis, we show that, software and hardware co-design of *Tick* meets the SIFS timing requirement. We make three observations. First, the latency in *Tick*-PHY alone does not exceed the SIFS timing bound. In *Tick* PHY, Rx latency for (a) (4.84 µs) + Tx latency for (c) (1.86 µs) < SIFS (16 µs for 802.11ac [6]). Second, the MAC latency alone in *Tick* does not exceed the SIFS timing bound. With accelerators, the MAC latency of

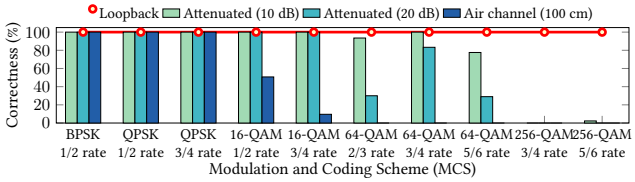

Figure 12: Correctness of *Tick* under different channels.

Frame Size	100 B	500 B	1000 B	1500 B	2000 B	3000 B	4000 B
Latency (µs)	334	356	382	404	415	449	489

Table 6: Overall latency in *Tick*.

processing the ACK frame is 7 µs (3 µs counts for MAC Rx, and 4 µs counts for ACK processing and MAC Tx). This latency corresponds to (2); it is smaller than the SIFS bound. Finally, the SIFS timing bound is satisfied by PHY and MAC together in *Tick*. This is valid because 6.70 µs + 7 µs + 1 µs (interface latency) < 16 µs.

8.4 Overall Performance

Overall Latency. We first gauge the overall latency of 802.11ac SISO on *Tick*. It is measured end-to-end, starting from the data frame generated by host computer, until it is received by the host computer. This latency consists of interface latency (USB 3.0 interface between host and Tx/Rx chain), Tx/Rx chain processing latency, air transmission time, and host processing time. We vary the frame length from 100B to 4000B and show the results in Table 6. *Tick* achieves overall latency at hundreds of µs, compared with 2500 µs latency in WARP [3]. We also notice that latency slightly grows with the frame length (334 µs for 100B, 489 µs for 4000B). This is because the air transmission time for a larger frame is longer.

Correctness. We also evaluate correctness under the following three scenarios: (1) ideal: we loopback the output waveform created by Tx into Rx, without traversing any channel. (2) controlled: we connect Tx and Rx through an attenuator (10 dB and 20 dB attenuation here) which emulates the wireless channel over the air. (3) real: we set RF frontends for Tx and Rx apart (1 m here) and run tests over the noisy radio channel; We also use a power amplifier (Mini-Circuits ZRL-2400LN+ [25]) to amplify radio signals. We send 1500 B frames consecutively at various fixed rates (aka, modulation and coding schemes (MCS)) and measure the percentage of correctly received frames at the Rx host. Figure 12 shows the results for 10 MCSes (from BPSK to 256-QAM) under different coding rates. We make three observations. First, our implementation is logically correct as it achieves 100% correctness in the ideal case. Second, correctness is sensitive to channel quality and MCS. *Tick* achieves 100% correctness for BPSK and QPSK, but turns difficult to support 256-QAM in both controlled and real-world experiments. This matches our expectation. Last, we also notice the gap between the controlled test and the real-world experiment. This is mainly caused by the used power amplifier; the radio signal is still too weak to support high MCS value. More efforts are clearly needed; this is part of our future work.

Consumed FPGA Resource. Table 7 shows the resource utilization on the Xilinx KC705 FPGA board. We count the amount of flip-flop (FF), lookup table (LUT) and block RAM (BRAM) used by PHY and MAC together. *Tick* consumes a modest amount of FPGA resources (less than 28% of each resource for SISO). We notice that

		***Tick*-Tx**			***Tick*-Rx**		
		FF	LUT	BRAM	FF	LUT	BRAM
SISO:	Total Available	407 600	203 800	445	407 600	203 800	445
	Used	24 869	37 020	69	29 407	56 890	75
	Utilization (%)	6.1	18.2	15.5	7.2	27.9	16.9
MIMO:							
	Used	38 359	43 457	133.5	59 501	133 173	136.5
	Utilization (%)	9.4	21.3	29.9	14.6	65.3	30.6

Table 7: FPGA resource usage of *Tick* 802.11ac.

	With *Tick*		**Without *Tick***	
	Language	LoC	Language	LoC
Element definition	XML	4	Verilog	151
Element connection	XML	8	Verilog	80
Algorithm implementation	Verilog	160	Verilog	160
Total LoC	XML & Verilog	172	Verilog	391

Table 8: Lines-of-code needed to program a BCC element.

Work	**Metric**	**WARP**	***Tick***	**Note**
WURC [3]	overall latency	2.5 ms	334 µs	
	bandwidth	10 MHz	80 MHz	
MIDU [4]	transmission delay	50 ms	100 µs	delay between
	bandwidth	500 kHz	80 MHz	consecutive frames
Duplex [21]	processing latency	11 µs	4.91 µs	Duplex modifies hardware
	ACK latency	75 µs	9.85 µs	code and cannot
	bandwidth	10 MHz	20 MHz	meet SIFS (>16 µs)

Table 9: Comparison of WARPLab and *Tick*.

LUT on the Rx side may become the main bottleneck, especially when it runs MIMO (details later in §9.1). This supplies a reference estimate for *Tick* implementation on other FPGA boards.

Coding Effort and Programmability. *Tick* requires less coding effort which implies nice programmability. Table 8 compares the LoC to program a BCC Encoder element (from the toy example) with/without *Tick*. With *Tick*, the LoC is merely half of that without *Tick*. Moreover, as the library usually implements the elements (the cost is inevitable and usually comparable across various SDR platforms), *Tick* users only need to define and configure elements in most cases. Since it uses XML, the cost is much lower (12 LoC vs. 231 LoC). In fact, the PHY implementation for the 802.11ac SISO in *Tick* has 18 312 LoC in total, including 18 098 LoC for the PHY library. Only 214 LoC is for element configuration and connection. Compared with the reference implementation without *Tick*, which uses 23 706 LoC in Verilog, the total coding cost reduces by 22.8%; Moreover, excluding the efforts for the basic PHY blocks, the programming efforts reduce from 5608 LoC to 214 LoC (26.2× reduction).

Comparison with the State-of-the-Art SDR. We compare *Tick* with WARP [22]. We consider both WARP's reference design and WARPLab, where the former is designed for low latency and the latter is designed for good programmability, as discussed in §4.3. For WARP's 802.11n reference design, its latency for data processing (2.48 µs, 802.11n) is comparable to *Tick* (2.48 µs, 802.11ac). However, it is hard to program over the WARP's 802.11n reference design. We further compare *Tick* with several representative studies using WARPLab [3, 4, 21] with latency results. We evaluate *Tick* under similar or even more stringent settings. Table 9 summarizes the key performance metrics. As a programmable SDR platform, *Tick* reduces latency by around one-order-of-magnitude (even two

	PHY Tx	PHY Rx	MAC Tx	MAC Rx
Latency	1.86 µs	21.62 µs	18 µs	3 µs
Proc. time[a]	15 101 µs	26 731 µs	10 504 µs	10 489 µs

[a] PHY air transmission time is 31 908 µs.

Table 10: Latency and processing time for A-MPDU in *Tick*.

orders of magnitude in [4]) than WARPLab, and supports higher bandwidth. Moreover, *Tick* is easy to program and extend. *Tick* supports 802.11ac and full duplex (see §9). The WARPLab's incapability of satisfying the timing constraints has been reported in the full-duplex [21], MIMO [37, 41] and 802.11ac [37, 39] cases.

9 CASE STUDIES

We use *Tick* to implement two case studies: 802.11ac (SISO, 2×2 MIMO) and full-duplex 802.11a/g, and assess their performances. The former is to demonstrate the performance merits of *Tick*, whereas the latter validates its programmability.

9.1 802.11ac SISO and MIMO

Implementation in *Tick*. For 802.11ac SISO, we follow the implementation for PHY and MAC in §7 and evaluate it in §8. Here we focus on 802.11ac 2×2 MIMO.

For 802.11ac 2×2 MIMO, we use the same MAC prototype as 802.11ac SISO. For MIMO, we have multiple spatial streams in PHY. We thus duplicate the data pipeline to implement multiple streams. Figure 13 shows the PHY block diagram. The PHY streams in MIMO will merge and split (via parser). We use Multiplexer and Demultiplexer elements to enable merge and parser operations. We also implement the multi-input multi-output element to process MIMO decoding.

Evaluation. We first assess the MAC performance. Since our 802.11ac MIMO and SISO prototypes use the same MAC, it achieves the same reduction in latency and processing time (Table 4).

We next look at PHY latency reduction (summarized in Table 3).. Note that our 802.11ac 2×2 MIMO prototype does not support frame aggregation and block ACK, but applies both MCDP and field-base processing as 802.11ac SISO. We can see that, 802.11ac 2×2 MIMO achieves latency and processing delay similar to 802.11ac SISO. In particular, it achieves microsecond-level latency at PHY Tx (1.85 µs). Regarding processing time, it is slightly faster than 802.11ac SISO for a large frame (say, 1500 B). Tx reduces latency from 42.07 µs to 32.56 µs, and Rx latency reduces from 61.85 µs to 43.55 µs. This is because MIMO supports two streams and certain processing functions are run in parallel. Based on the used air transmission time, the timing requirement for SIFS can also be met following the analysis similar to §7.

We also test the latency and processing delay of the maximum A-MPDU length frame (1 048 575 B) for the 802.11ac protocol. Table 10 shows the results. The PHY latency is the same as 802.11ac SISO, but processing time is significantly longer due to more data frames being sent out.

Since the MIMO implementation approximately doubles the used elements compared with SISO, the consumed resources also roughly double, as shown in Table 7.

Finally, we measure the maximum data processing capacity (measured in Mbps) that *Tick* PHY can support. SISO Tx supports up to 556.55Mbps and SISO Rx supports up to 314.09Mbps. Both exceed the throughput maximum stipulated by the standards [6] (263.3Mbps using 64-QAM[2]). MIMO Tx supports up to 1113.1Mbps and MIMO Rx supports up to 632.4Mbps. The throughput requirement from the standard (526.5Mbps using 64-QAM[2]) is also met.

Note that, to further boost performance, users do not need to change the SDR architecture. They just need to focus on optimizing the algorithm of a module or upgrade the FPGA capacity.

9.2 Full-Duplex 802.11a/g

Implementation in *Tick*. We implement a full-duplex 802.11a/g to demonstrate the programmability of *Tick*. Figure 14 shows its PHY function blocks. To support the channel estimation (self-interference channel and target channel) , we introduce a customized new preamble field to the 802.11a/g frame. This field is critical to fully realize full-duplex 802.11a/g. We observe that, using frame-based processing without *Tick*, one would have to rewrite every standard module to accommodate the newly added field. In contrast, with *Tick*, only lightweight effort is required to modify or customize some elements provided by the *Tick* library (marked in Figure 14). In particular, our implementation results in a full-duplex PHY with 22 elements, consisting of 25 740 lines of code. Among these elements, 15 elements are the standard ones *without* any modification (e.g., Interleaver, IFFT, Viterbi Decoder) provided by the Tick PHY library. We modified four elements from the library, i.e., Pilot Insertion, Channel Estimation, Timing Synchronization, and Phase Tracking. We customized three elements: Preamble Insertion, Tx-Rx Path and Self-Interference Cancellation. As a result, the time and invested human effort are reduced.

We further implement a simple MAC for full duplex with five accelerators. We use CRC-32 and RF communication accelerators from the *Tick* MAC library. We have customized accelerators of MAC-PHY configuration, Timing Control and Customized ACK.

Evaluation. We have also discovered the major gain by using field-based processing with full-duplex 802.11 a/g. *Tick* achieves the smallest latency (0.07 µs) for PHY Tx, which further reduces 1.79 µs (27× reduction) compared with 802.11ac. This latency is smaller than that for 802.11ac PHY, because the used preamble is fixed for every frame. Therefore, we precompute this field and bypass the bottleneck IFFT element in our field-based processing. The measured PHY latency and processing delay for the *Tick*-enabled full-duplex 802.11a/g are summarized in Table 11.

The maximum data processing capacity achieved by both half-duplex Tx and Rx are 122.73Mbps, following the previous definition. Therefore, our full-duplex 802.11a/g supports up to 245.46Mbps, which exceeds double throughput of the 802.11a/g protocol, which is 108Mbps.

Finally, Table 12 shows the FPGA resource utilization. Our implementation of full-duplex 802.11a/g consumes the amount of resources on KC705 FPGA with less than 10% on Tx and less than 20% on Rx.

[2]When using the long GI mode.

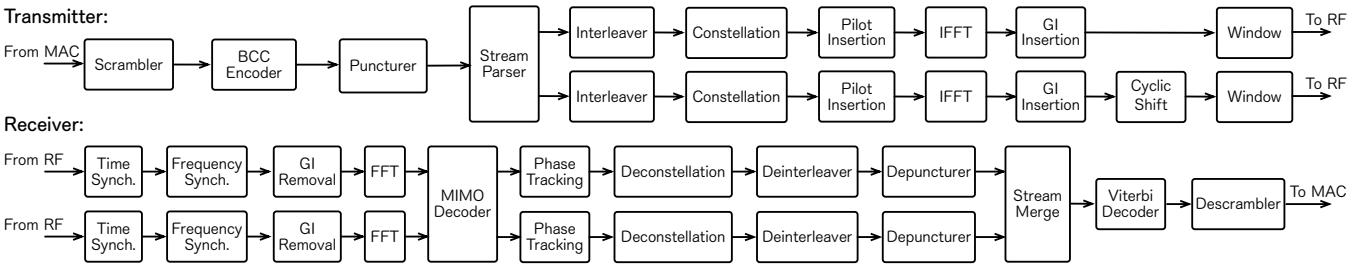

Figure 13: Transmitter and receiver modules for 802.11ac (MIMO).

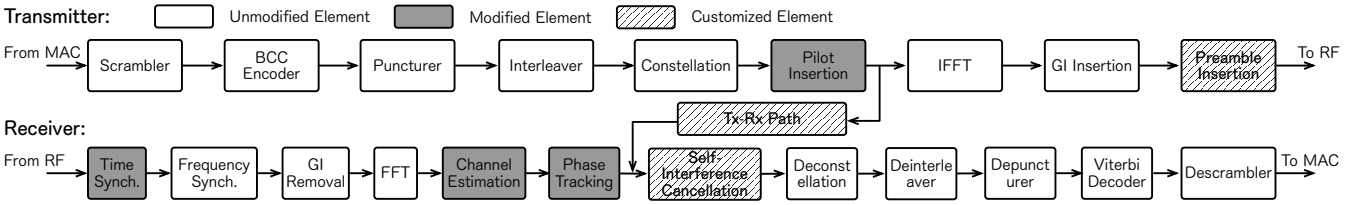

Figure 14: Transmitter and receiver modules for full-duplex 802.11a/g.

	PHY		MAC	
	Tx	Rx	Tx	Rx
100 B:				
Latency	0.07 µs	13.86 µs	0.25 µs	1.14 µs
Proc. time[a]	13.88 µs	49.79 µs	1.92 µs	2.17 µs
1500 B:				
Latency	0.07 µs	13.86 µs	0.25 µs	1.14 µs
Proc. time[b]	92.50 µs	115.57 µs	23.37 µs	28.18 µs

[a] PHY air transmission time for 100 B is 120 µs.
[b] PHY air transmission time for 1500 B is 328 µs.

Table 11: Latency and processing time of *Tick* full-duplex 802.11a/g.

	Tick-Tx			*Tick*-Rx		
	FF	LUT	BRAM	FF	LUT	BRAM
Total Available	407 600	203 800	445	407 600	203 800	445
Used	8111	19 552	41.5	22 773	40 820	93.5
Utilization (%)	2.0	9.6	9.3	5.6	20.0	21.0

Table 12: FPGA resource usage of *Tick* full-duplex 802.11a/g.

10 RELATED WORK

A number of software-defined radio (SDR) platforms have been well documented in recent years [2, 7, 22, 26, 27, 31–35, 40]. They achieve different goals following different designs. The software-centric design like GNU Radio [34] or Sora [33] offers good programmability at the cost of latency guarantee and fine-grained timing control, since they rely heavily on CPU computations. Atomix [7] proposes a modular software framework targeting DSP, which cannot meet the timing requirement of 802.11a. Ziria [31] designs a domain-specific programming language for SDR, but it is limited to PHY only. An alternative solution is to take the hardware-centric design approach, such as AirBlue [26], WARP [22] or OpenMili [40]. Proposals in this category ensure higher processing capacity and lower latency. However, they do not offer good programmability support at both MAC and PHY, due to the intricacy of hardware programming. None can achieve both low latency and good programmability.

Tick explores software and hardware codesign approaches. Its modular design is inspired by the Click router [23], but applies domain-specific optimizations at wireless PHY and MAC to boost performance. Our multi-clock-domain pipelining leverages design ideas from the architecture field [9, 29]. We adapt it to speed up the PHY pipeline. Accelerator-based architecture in FPGA has been applied for data centers [13, 19]. Our accelerator works on a different problem domain. It offloads computation and communication from the embedded processor at MAC.

11 CONCLUSION

Building highly programmable SDR systems with low latency performance can be challenging; no existing systems can achieve both. Of course, there are a number of technical challenges for both PHY and MAC. However, it is still conceptually feasible. In the SDR context, software techniques are great to promote programmability, while hardware can ensure high performance in terms of low latency and accurate timing control. This makes a case for us to explore software and hardware codesign techniques. The outcome is the *Tick* SDR platform reported in this paper.

The design and prototype of *Tick* is a three-year effort with many ups and downs. Many seemingly nice techniques turn out to be ineffective. They force us to explore new solution ideas, thus producing *Tick*. To date, it has been under internal use and tests at three university and research sites for over three months. On one hand, we hope it can eventually yield a reliable SDR platform with low latency and programmability for us and the broad research community. On the other hand, we are using it to explore domain-specific architecture designs for wireless networking systems in the long run. Along this general direction, we are just starting.

ACKNOWLEDGMENTS

The authors would like to thank their shepherd, Dr. Božidar Radunović, and the anonymous reviewers for their valuable comments and helpful suggestions. This work is also supported in part by NSF awards (CNS-1423576 and CNS-1526985) and in part by National Natural Science Foundation of China (NSFC) Grants 61370056 and 61531004.

REFERENCES

[1] Analog Devices. 2017. AD9371 Transceivers. https://www.digikey.com/en/product-highlight/a/analog-devices/ad9371-transceivers. (February 2017).

[2] Narendra Anand, Ehsan Aryafar, and Edward W Knightly. 2010. WARPlab: a flexible framework for rapid physical layer design. In *Proceedings of the 2010 ACM workshop on Wireless of the students, by the students, for the students (S3 '10)*. ACM, 53–56.

[3] Narendra Anand, Ryan E Guerra, and Edward W Knightly. 2014. The case for UHF-band MU-MIMO. In *Proceedings of the 20th Annual International Conference on Mobile Computing and Networking (MobiCom '14)*. ACM, 29–40.

[4] Ehsan Aryafar, Mohammad Amir Khojastepour, Karthikeyan Sundaresan, Sampath Rangarajan, and Mung Chiang. 2012. MIDU: Enabling MIMO full duplex. In *Proceedings of the 18th Annual International Conference on Mobile Computing and Networking (MobiCom '12)*. ACM, 257–268.

[5] IEEE Standards Association. 2012. IEEE Standard 802.11-2012: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications. *IEEE Std.* 802 (2012).

[6] IEEE Standards Association. 2013. IEEE Standard 802.11ac-2013: Enhancements for Very High Throughput for Operation in Bands below 6 GHz. *IEEE Std.* 802 (2013).

[7] Manu Bansal, Aaron Schulman, and Sachin Katti. 2015. Atomix: A framework for deploying signal processing applications on wireless infrastructure. In *Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI '15)*. USENIX Association, Oakland, CA, 173–188.

[8] Dinesh Bharadia, Emily McMilin, and Sachin Katti. 2013. Full duplex radios. *ACM SIGCOMM Computer Communication Review* 43, 4 (2013), 375–386.

[9] S Casale Brunet, Endri Bezati, Claudio Alberti, Marco Mattavelli, Edoardo Amaldi, and Jörn W Janneck. 2013. Partitioning and optimization of high level stream applications for multi clock domain architectures. In *Proceedings of the 2013 IEEE Workshop on Signal Processing Systems (SiPS '13)*. IEEE, 177–182.

[10] Tiberiu Chelcea and Steven M. Nowick. 2000. Low-latency asynchronous FIFO's using token rings. In *Proceedings of the 6th International Symposium on Advanced Research in Asynchronous Circuits and Systems (ASYNC '00)*. 210–220.

[11] Tao Chen and G Edward Suh. 2016. Efficient data supply for hardware accelerators with prefetching and access/execute decoupling. In *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-49)*. IEEE, 1–12.

[12] Jung Il Choi, Mayank Jain, Kannan Srinivasan, Phil Levis, and Sachin Katti. 2010. Achieving single channel, full duplex wireless communication. In *Proceedings of the 16th Annual International Conference on Mobile Computing and Networking (MobiCom '10)*. ACM, 1–12.

[13] Jason Cong, Zhenman Fang, Yuchen Hao, and Reinman Glenn. 2017. Supporting address translation for accelerator-centric architectures. In *Proceedings of the 23rd IEEE Symposium on High Performance Computer Architecture (HPCA '17)*. IEEE, IEEE, Austin, TX, USA.

[14] Cypress. 2014. CYUSB3ACC-005 FMC Interconnect Board. http://www.cypress.com/documentation/development-kitsboards/cyusb3acc-005-fmc-interconnect-board-ez-usb-fx3-superspeed. (October 2014).

[15] Cypress. 2017. CYUSB3KIT-003 SuperSpeed Explorer Kit. http://www.cypress.com/documentation/development-kitsboards/cyusb3kit-003-ez-usb-fx3-superspeed-explorer-kit. (June 2017).

[16] Ettus. 2017. USRP Family of Products. https://www.ettus.com/product. (March 2017).

[17] Jian Gong, Jiahua Chen, Haoyang Wu, Fan Ye, Songwu Lu, Jason Cong, and Tao Wang. 2014. EPEE: An efficient PCIe communication library with easy-host-integration property for FPGA accelerators. In *Proceedings of the 2014 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '14)*. ACM, 255–255.

[18] Jian Gong, Tao Wang, Jiahua Chen, Haoyang Wu, Fan Ye, Songwu Lu, and Jason Cong. 2014. An efficient and flexible host-FPGA PCIe communication library. In *Proceedings of the 24th International Conference on Field Programmable Logic and Applications (FPL '14)*. IEEE, 1–6.

[19] Muhuan Huang, Di Wu, Cody Hao Yu, Zhenman Fang, Matteo Interlandi, Tyson Condie, and Jason Cong. 2016. Programming and runtime support to Blaze FPGA accelerator deployment at datacenter scale. In *Proceedings of the 7th ACM Symposium on Cloud Computing (SoCC '16)*. ACM, 456–469.

[20] Anoop Iyer and Diana Marculescu. 2002. Power and performance evaluation of globally asynchronous locally synchronous processors. In *Proceedings of the 29th Annual International Symposium on Computer Architecture (ISCA '02)*. IEEE, 158–168.

[21] Mayank Jain, Jung Il Choi, Taemin Kim, Dinesh Bharadia, Siddharth Seth, Kannan Srinivasan, Philip Levis, Sachin Katti, and Prasun Sinha. 2011. Practical, real-time, full duplex wireless. In *Proceedings of the 17th Annual International Conference on Mobile Computing and Networking (MobiCom '11)*. ACM, 301–312.

[22] Ahmed Khattab, Joseph Camp, Chris Hunter, Patrick Murphy, Ashutosh Sabharwal, and Edward W Knightly. 2008. WARP: a flexible platform for clean-slate wireless medium access protocol design. *ACM SIGMOBILE Mobile Computing and Communications Review* 12, 1 (2008), 56–58.

[23] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M Frans Kaashoek. 2000. The Click modular router. *ACM Transactions on Computer Systems (TOCS)* 18, 3 (2000), 263–297.

[24] Yunsup Lee, Rimas Avizienis, Alex Bishara, Richard Xia, Derek Lockhart, Christopher Batten, and Krste Asanović. 2011. Exploring the tradeoffs between programmability and efficiency in data-parallel accelerators. In *Proceedings of the 38th Annual International Symposium on Computer Architecture (ISCA '11)*, Vol. 39. ACM, San Jose, California, USA, 129–140.

[25] Mini-Circuits. 2017. ZRL-2400LN+. https://www.minicircuits.com/WebStore/dashboard.html?model=ZRL-2400LN%2B. (February 2017).

[26] Man Cheuk Ng, Kermin Elliott Fleming, Mythili Vutukuru, Samuel Gross, Arvind, and Hari Balakrishnan. 2010. Airblue: A system for cross-layer wireless protocol development. In *Proceedings of the 6th ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS '10)*. ACM, 4:1–4:11.

[27] Rishiyur Nikhil. 2004. Bluespec system verilog: Efficient, correct RTL from high-level specifications. In *Proceedings of the 2nd ACM and IEEE International Conference on Formal Methods and Models for Co-Design (MEMOCODE '04)*. IEEE, 69–70.

[28] John Oliver, Ravishankar Rao, Paul Sultana, Jedidiah Crandall, Erik Czernikowski, LW Jones, Diana Franklin, Venkatesh Akella, and Frederic T Chong. 2004. Synchroscalar: A multiple clock domain, power-aware, tile-based embedded processor. In *Proceedings of the 31st Annual International Symposium on Computer Architecture (ISCA '04)*. IEEE, 150–161.

[29] Greg Semeraro, David H Albonesi, Steven G Dropsho, Grigorios Magklis, Sandhya Dwarkadas, and Michael L Scott. 2002. Dynamic frequency and voltage control for a multiple clock domain microarchitecture. In *Proceedings of the 35th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-35)*. IEEE, 356–367.

[30] Greg Semeraro, Grigorios Magklis, Rajeev Balasubramonian, David H Albonesi, Sandhya Dwarkadas, and Michael L Scott. 2002. Energy-efficient processor design using multiple clock domains with dynamic voltage and frequency scaling. In *Proceedings of the 8th International Symposium on High-Performance Computer Architecture (HPCA '02)*. IEEE, 29–40.

[31] Gordon Stewart, Mahanth Gowda, Geoffrey Mainland, Bozidar Radunovic, Dimitrios Vytiniotis, and Cristina Luengo Agullo. 2015. Ziria: A DSL for wireless systems programming. In *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '15)*, Vol. 50. ACM, 415–428.

[32] Paul D Sutton, Jorg Lotze, Hicham Lahlou, Suhaib A Fahmy, Keith E Nolan, Baris Ozgul, Thomas W Rondeau, Juanjo Noguera, and Linda E Doyle. 2010. Iris: An architecture for cognitive radio networking testbeds. *IEEE communications magazine* 48, 9 (2010).

[33] Kun Tan, He Liu, Jiansong Zhang, Yongguang Zhang, Ji Fang, and Geoffrey M Voelker. 2011. Sora: high-performance software radio using general-purpose multi-core processors. *Commun. ACM* 54, 1 (2011), 99–107.

[34] The GNU Radio Foundation. 2017. GNU Radio. http://www.gnuradio.org/. (March 2017).

[35] Artem Tkachenko, Danijela Cabric, and Robert W Brodersen. 2007. Cyclostationary feature detector experiments using reconfigurable BEE2. In *Proceedings of the 2nd IEEE International Symposium on New Frontiers in Dynamic Spectrum Access Networks (DySPAN '07)*. IEEE, 216–219.

[36] WARP Project. 2013. IFS calibration and benchmarks. http://warpproject.org/trac/wiki/802.11/Benchmarks/IFS. (November 2013).

[37] Xiufeng Xie, Xinyu Zhang, and Karthikeyan Sundaresan. 2013. Adaptive feedback compression for MIMO networks. In *Proceedings of the 19th Annual International Conference on Mobile Computing and Networking (MobiCom '13)*. ACM, 477–488.

[38] Xilinx. 2017. Xilinx Kintex-7 FPGA KC705 Evaluation Kit. https://www.xilinx.com/products/boards-and-kits/ek-k7-kc705-g.html. (July 2017).

[39] Hang Yu, Oscar Bejarano, and Lin Zhong. 2014. Combating inter-cell interference in 802.11ac-based multi-user MIMO networks. In *Proceedings of the 20th Annual International Conference on Mobile Computing and Networking (MobiCom '14)*. ACM, 141–152.

[40] Jialiang Zhang, Xinyu Zhang, Pushkar Kulkarni, and Parameswaran Ramanathan. 2016. OpenMili: A 60 GHz software radio platform with a reconfigurable phased-array antenna. In *Proceedings of the 22nd Annual International Conference on Mobile Computing and Networking (MobiCom '16)*. ACM, 162–175.

[41] Xinyu Zhang, Karthikeyan Sundaresan, Mohammad A Amir Khojastepour, Sampath Rangarajan, and Kang G Shin. 2013. NEMOx: Scalable network MIMO for wireless networks. In *Proceedings of the 19th Annual International Conference on Mobile Computing and Networking (MobiCom '13)*. ACM, 453–464.