Leveraging RISC-V for Flexible and Adaptive Real-Time Radar Sequencing

Michael Atzmüller[1] Rainer Findenig[1] Bernhard Greslehner-Nimmervoll[1]
Wolfgang Ecker[2] Daniel Große[3]

[1]*Infineon Technologies Austria AG*, Linz, Austria [2]*Infineon Technologies AG*, Munich, Germany
[3]*Johannes Kepler University Linz*, Linz, Austria

{michael.atzmueller, rainer.findenig, bernhard.greslehner-nimmervoll, wolfgang.ecker}@infineon.com daniel.grosse@jku.at

***Abstract*—*Frequency-Modulated Continuous-Wave* (FMCW) radar is essential for accurate measurements of distance, velocity, and angle in applications such as autonomous vehicles and industrial sensing. In FMCW radar, ramp scenarios involve the gradual change of transmitted signal frequency over time, repeated in sequences to enable precise object detection and tracking. Central to these systems is the sequencer, a specialized unit responsible for generating and distributing control signals with precise timing to synchronize hardware components during ramp generation. Traditional implementations, such as the *Domain-Specific Sequencer* (DSS), rely on custom *Instruction Set Architecture*s (ISAs) optimized for radar operations but suffer from limitations in flexibility. This paper introduces the *RISC-V Sequencer* (RVS), a novel approach leveraging the modular and extensible RISC-V ISA to overcome these challenges. By extending a RISC-V processor with custom *Control and Status Register*s (CSRs) and providing a software library, the RVS enables high-level and adaptive ramp scenarios, offering a flexible and advanced alternative to traditional radar sequencers such as DSS, which lack the adaptability required for real-time scenario changes.**

I. Introduction

The *Frequency-Modulated Continuous-Wave* (FMCW) radar principle is a widely used technology to measure distance, velocity, and angle by transmitting frequency-modulated signals and analyzing their reflections [1]. Its high precision and robustness make it essential in applications such as automotive driver-assistance systems, autonomous vehicles, industrial sensing, and environmental monitoring. By leveraging frequency sweeps, FMCW radar enables accurate object detection and tracking in complex and dynamic environments. These frequency sweeps, or ramps, involve a gradual change in frequency over a set period of time. In practice, multiple ramps are combined and repeated in sequences, forming what is known as a *ramp scenario*.

In FMCW radar systems, precise timing is critical to ensure accurate control of various involved hardware components, such as transmitters, receivers, monitoring components, and power amplifiers. This requires generating and distributing control signals with highly accurate timing to synchronize all hardware operations. A *sequencer* is a specialized unit responsible for orchestrating the precise timing and coordination of control signals, thereby enabling the accurate execution of ramp scenarios [2], [3]. Crucially, this coordination must occur in real-time – referred to as *real-time radar sequencing* – meaning the sequencer must deliver configuration data with deterministic timing during ongoing radar operation. Simply speaking, the sequencer ensures that each hardware unit receives the correct control values at exactly the right time.

State-of-the-art FMCW radar chips implement the sequencer as a domain-specific unit, equipped with a custom *Instruction Set Architecture* (ISA) optimized for radar operations [3]–[5]. Another prominent example is the *Domain-Specific Sequencer* (DSS) from [2], [6]. In essence, the DSS reads a sequencer program which describes the ramps and defines control parameters for the hardware. Its execution generates control values paired with timestamps. A global timer ensures that these values are delivered to the hardware at precisely the right times. A key benefit of this DSS is, that its programming model allows users to adapt the ramps to their needs regarding frequency bands, bandwidth, and many more configurations. This is particularly important in a competitive landscape, as the parameters used for ramp generation often represent intellectual property. Therefore, programmability is essential to enable users – or customers – to differentiate their radar systems.

However, the primary drawback of implementing the sequencer as a domain-specific hardware unit is its lack of flexibility. Specifically: (a) even minor changes to the programming model of the DSS require a hardware redesign, as the behavior of the instructions must be modified; (b) there is no existing ecosystem, meaning tools such as compilers must be developed alongside the radar chip; (c) the custom ISA of the DSS lacks the computational expressiveness needed for sequencer programs to react dynamically to environmental changes; and (d) programming is less accessible, as it requires writing low-level code in the DSS ISA.

In this paper, we present the *RISC-V Sequencer* (RVS), a novel approach based on the RISC-V ISA [7], an open standard known for its modularity, extensibility, and suitability for custom hardware designs. The RVS extends a RISC-V processor with custom *Control and Status Register*s (CSRs) [8] specifically designed for radar operations, offering a flexible and advanced alternative to the traditional DSS unit. Building on the modularity and extensibility of the RISC-V architecture, the RVS not only overcomes the inherent limitations of the

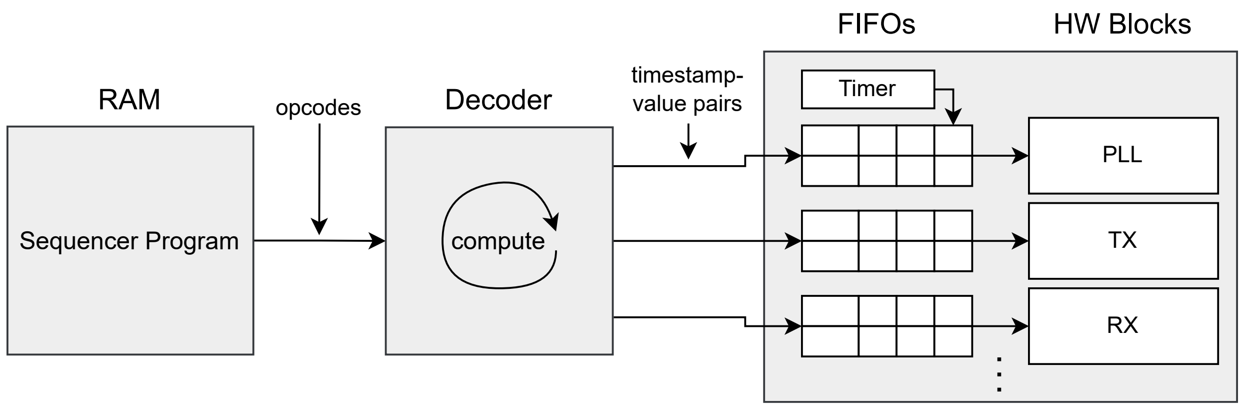

Fig. 1: DSS for generation of a ramp scenario; decoder implements the custom ISA

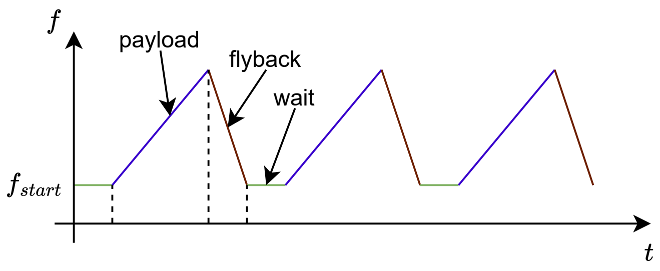

Fig. 2: Simple ramp scenario

DSS but also opens up new possibilities. Our solution also provides a software library to intuitively code ramp scenarios at a high level. In our experiments, we show the simulation of a standard ramp scenario using this library. For that, we have integrated the proposed RVS into an industrial full radar chip simulation to demonstrate that we achieve the same behavior in comparison to the DSS while still maintaining timing requirements and area constraints. Furthermore, we have extended the standard ramp scenario with event handling. This enables the radar system to react to environmental events during ramp generation, a capability we refer to as *adaptive ramp scenarios* – a feature not possible with the DSS. Finally, the flexibility of RVS is evidenced through a qualitative comparison of key design characteristics with those of the DSS.

II. Domain-Specific Sequencer

In this section the state-of-the-art DSS is reviewed. First, the FMCW radar principle is described in more detail (Section II-A). Thereafter, an overview on the DSS hardware implementation (Section II-B) and the DSS ISA (Section II-C) are provided, respectively. Finally, the benefits and drawbacks of this DSS design are discussed (Section II-D).

A. *Frequency-Modulated Continuous-Wave* (FMCW)

The FMCW radar principle relies on modulating frequency over time to perform ramps [9]. As shown in Fig. 2, a ramp can usually be split into at least three parts, a *payload segment*, in which the actual measurement is done, a *flyback segment*, to ramp back to the right starting frequency and a *wait segment* between two consecutive ramps. Typically, users define these ramps by a start frequency, a duration and either a stop frequency, a bandwidth or a slope. A ramp scenario usually involves the concatenation of several hundred to several thousand ramps. However, to address issues like interference [10], integrating configurable parameters becomes essential. Rather than simply repeating the same frequency ramp, parameters such as the time delays between ramps, starting frequency, output power, output phase offset, exact timing for initiating sampling in the receiver chain, and the receiver chain's settings including the gain, filter configurations, and the sampling rate should ideally be adjustable on a per-ramp basis [2], [6].

Central to FMCW radar systems is the sequencer, which synchronizes hardware through precise control signals. Traditional solutions like the DSS rely on custom ISAs. Before discussing programmability via the ISA, we first examine the DSS hardware implementation.

B. *DSS Hardware Implementation*

We review the state-of-the-art DSS based on [2], [6]. The architecture of the DSS is depicted in Fig. 1. The left side features a memory block that stores the *Sequencer Program*, which defines ramp profiles and parameter settings for all hardware modules requiring cycle-accurate control. Next, there is a *Decoder* (center of Fig. 1) which reads the sequencer program from the memory and generates control values for all hardware blocks (see right side of Fig. 1). Every control value is paired with a timestamp and then pushed into a FIFO in front of the particular hardware unit. The FIFOs are connected to a global timer (*Timer* in Fig. 1), which acts as the heartbeat by providing a cycle-accurate time base. Whenever the timer value and a timestamp value of the FIFOs match, the particular control value is provided to the hardware unit.

The decoder comes with a custom ISA, which is highly optimized for radar operations. The instructions provided by this ISA allow users do define ramp scenarios by creating sequencer programs as described in the next section.

C. *DSS Custom ISA*

To write a sequencer program for the DSS, its custom ISA provides *opcodes* – each opcode corresponds to a control operation for components in the radar device [2], [6]. One such opcode is **SEG** for creating a single ramp. In addition,

```
1 LOOP 1024
2
3 SEG f_start, f_diff, t_chirp     ; payload segment
4 SEG f_stop, -f_diff, t_flyback   ; flyback segment
5 SEG f_start, 0, ARRAY1[t_idx]    ; wait segment
6
7 MODIFY_IDX t_idx, 1
8
9 LOOP END
```

Listing 1: Sequencer program on DSS

there are also opcodes for performing loops and accessing hardware arrays.

Example 1. *Listing 1 shows a sequencer program for the DSS. Line 1 uses the* ***LOOP*** *opcode to iterate over the enclosed ramp* 1024 *times, while Line 9 marks the loop's end. Lines 3–5 show the* ***SEG*** *opcode for the payload segment, the flyback segment and the wait segment, respectively. Using this opcode, a ramp can be defined by a starting frequency, a bandwidth and a duration. The parameters remain the same for all 1024 ramps, however the wait times between two consecutive ramps are obtained from arrays (see Line 5). The opcode* ***MODIFY_IDX*** *in Line 7 is used to increase the array index by one in each loop iteration.*

Defining ramp scenarios in this manner requires the user to manually compose a sequencer program using the presented low-level opcodes. This process is analogous to assembly programming on a general-purpose CPU and demands significant expertise due to its inherent complexity. The following section discusses the advantages and limitations from a general perspective.

D. Advantages and Limitations

The main advantage of this DSS design is its fast execution speed, which is achieved through the use of domain-specific opcodes. However, this fast execution speed might not be utilized whilst performing radar measurements due to a physical limit. In theory the minimum duration of one ramp is given by $\tau = 2 \times d_i/c_0$, where τ is the round-trip delay time, which a radar signal requires to travel from the radar sensor to an object and back, d_i is the distance between the radar sensor and the object, and c_0 is the propagation speed of the radar signal, which is the speed of light [11]. For automotive radar sensors the maximum distance in which objects need to be detected is up to 250 meters [12]. Therefore, radar measurements require at least $1.67\mu s$, which marks a physical limit. In other words, this means that the minimum duration of one payload segment is $1.67\mu s$. In practice the hardware blocks controlled by the DSS are slower and not able to achieve this physical limit. Therefore, it is assumed that one ramp requires around $25\mu s$.

As mentioned in the beginning of this section, the DSS excels with its fast execution speed, which would allow ramps even two magnitudes faster than the physical limit. However, due to the fact, that the controlled hardware blocks are much slower than the DSS, this execution speed can not be utilized during radar operation.

The following discussion outlines several drawbacks of the DSS that hinder radar sensor development and limit users when creating ramp scenarios.

First, even minor modifications to the programming model or signal structure require adjustments to the DSS, and due to the tight coupling between opcode behavior and hardware, these modifications often necessitate a hardware redesign. Thus, reusing the DSS in future radar sensor generations is challenging because anticipated changes in control signals will likely require hardware modifications, thereby increasing both development time and costs. Moreover, existing sequencer programs may become incompatible with new opcode configurations.

Second, the fully customized design of the DSS also means that no supportive software ecosystem is available, and building one adds complexity and expense to the development process. Furthermore, the specific nature of the DSS ISA means that users may be unfamiliar with it, further extending development time and expenses.

Third, the limited expressiveness of the custom ISA further restrict flexibility and accessibility. Since the DSS is implemented with highly specialized opcodes tailored for radar operations, users cannot run custom algorithms. This limitation makes it difficult to integrate additional functions, such as random number generation for randomized wait times, without adapting the ISA and modifying hardware.

Finally, the inability to use high-level programming languages compels the use of low-level code, which requires special expertise, and delays the development of sequencer programs.

III. RISC-V Sequencer

In this section, we present the proposed RVS. First, in Section III-A a comparison of the DSS and RVS from an architectural perspective is given. Next, Section III-B discusses the extension of the RVS with custom CSRs, along with the resulting hardware/software interface. Finally, in Section III-C high-level functions for defining ramp scenarios are introduced as well as an example to program an adaptive ramp scenario is provided.

A. From DSS to RVS

To overcome the limitations of the DSS and support additional functionality, we propose the RVS – an architecture based on a RISC-V processor extended with custom CSRs. A comparison of the block diagrams of the DSS (Fig. 1) and the RVS (Fig. 3) reveals structural similarities, with both architectures featuring a memory block on the left and hardware units coupled with FIFOs on the right. However, in the RVS we replace the domain-specific decoder and custom instructions with a RISC-V and custom CSRs, denoted as *Sequencer Subsystem* in Fig. 3. This marks a fundamental shift in how ramp scenarios are programmed, transitioning from low-level opcode-based control to the use of high-level programming languages. Although this design serves as a more general solution for cycle-accurate control, which is not limited to radar, it still works as a drop-in replacement for the DSS, because the interfaces to the FIFOs can be easily constructed using the custom CSRs.

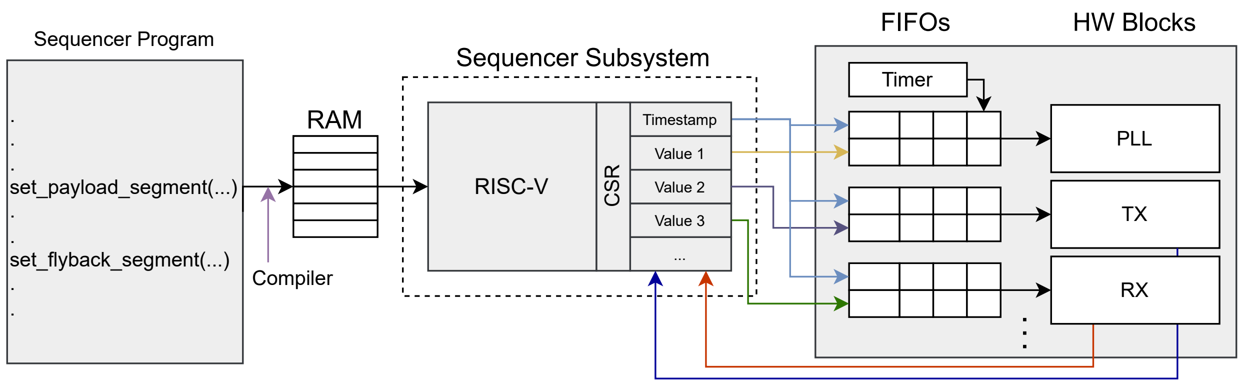

Fig. 3: Proposed RVS for generation of ramp scenario

```
1  #define F_START   ((uint32_t) 0x9C0)
2  #define SLOPE     ((uint32_t) 0x9C1)
3  #define DURATION  ((uint32_t) 0x9C2)
4  #define TIMESTAMP ((uint32_t) 0x9C3)
5  #define HW_READY  ((uint32_t) 0x9C4)
6
7  template <uint32_t address>
8  inline void write_csr(uint32_t const value) {
9    __asm__ volatile ("csrw␣%0,␣%1" : : "i" (address), "r"
           (value));  // input operand
10 }
11
12 template <uint32_t address>
13 inline uint32_t read_csr(){
14   uint32_t value;
15   __asm__ volatile ("csrr␣%0,␣%1"
16   : "=r" (value)       // output operand
17   : "i" (address));    // input operand
18   return value;
19 }
```

Listing 2: Low-level CSR code

B. *Control and Status Registers* (CSRs)

Besides the base instruction set specified in the RISC-V instruction set manual [7], also some extensions for various tasks are defined. One extension specifies the CSRs [8]. This extension comes with an extra address space for 4096 registers, accessible through specialized instructions (e.g. `csrw` for writing and `csrr` for reading a CSR). It is noted, that not all registers for the whole address space must be present in every implementation. The majority of the address space is reserved for standard use like for the debug system, timers, counters or other peripherals. However, there are also some registers, which can be used for custom applications. For RVS, we realize radar specific CSRs and demonstrate how to access them in the following example.

Example 2. *Listing 2 gives an example how to access CSRs for defining a ramp in C++ code. In Lines 1–4, the addresses of the starting frequency, the slope, the duration and the timestamp are specified. The register defined in Line 5 contains a signal, which tells whether the FIFOs are ready to receive new control values or not. The two template functions shown in Lines 7–19 wrap the CSRs instructions for reading and writing those registers.*

As already implied with the register names in Listing 2, the RVS employs these registers and instructions as control values. These CSRs are directly connected to the FIFOs as shown in Fig. 3. This approach enables easy adaptation of the control signal structure, as all necessary control signals are wired to CSRs during hardware development, while subsequent modifications can be handled entirely in software.

C. *Software Library and High-Level Control*

A key advantage of the software-based approach is that, as long as the CSRs described in Section III-B are used, the underlying implementation details remain abstracted and can vary without affecting functionality. This means that we have an additional level of abstraction here. In the following, we present a simple yet illustrative example demonstrating this abstraction for defining ramp scenario.

Example 3. *Listing 3 contains a sequencer program, which allows to perform a ramp scenario similar to the one shown in Listing 1, however, this time completely written in C++. This sequencer program utilizes the CSR functions and addresses shown previously in Listing 2. Starting in Line 1, a struct is defined to group the required control values in one combined type. Next, in Line 3, the number of ramps is defined. In addition, the number of segments is defined in Line 4. Therefore, the number of ramps is multiplied by 3, in order to execute a payload segment, a flyback segment and a wait segment for every ramp. Now in Line 5, an array containing all the required ramp parameters is defined. As these parameters are highly dependent on the hardware blocks which are controlled, the initialization routine of this array is not shown here. The execution of the ramps is done in the for loop in Lines 7–13. This loop contains another while loop in Line 8, which stalls until the hardware is ready to receive control values. Finally, in Lines 9–12 the control values are written to the hardware, using the functions and addresses shown in Listing 2.*

A completely new feature, which is hard and costly to achieve using the DSS are adaptive ramp scenarios. Therefore, CSRs are used to report events from hardware units back to the RISC-V processor as shown in Fig. 3 (cf. red and blue arrows pointing from the hardware blocks to the sequencer subsystem). The RVS allows to react on events like temperature changes, drops in supply voltages or interference directly in software and adapt the generation of ramps. In the following, we give an example.

```cpp
struct TRamp{uint32_t f_start; uint32_t slope; uint32_t
     duration; uint32_t timestamp; };

constexpr size_t ramp_cnt = 1024;
constexpr size_t segment_cnt = ramp_cnt * 3;  // times 3
      for payload, flyback and wait segment
constexpr TRamp ramp_params[segment_cnt] = {...};

for (int i = 0; i < segment_cnt; ++i){
  while(!read_csr<HW_READY>());
  write_csr<F_START>(ramp_params[i].f_start);
  write_csr<SLOPE>(ramp_params[i].slope);
  write_csr<DURATION>(ramp_params[i].duration);
  write_csr<TIMESTAMP>(ramp_params[i].timestamp);
}
```

Listing 3: Sequencer program in C++ on RVS

```cpp
1  RampScenario rmp(f_start, f_diff, t_chirp);
2  constexpr size_t ramp_cnt = 1024;
3
4  for (int i = 0; i < ramp_cnt; ++i){
5    rmp.set_payload_segment(i);
6    rmp.set_flyback_segment(i);
7    rmp.set_wait_segment(i);
8
9    switch (rmp.get_event()){
10     case 0: break; // no event
11     case 1: rmp.frequency_hopping(); break;
12     case 2: rmp.additional_wait(); break;
13     case 3: // critical event
14     default: rmp.abort();
15   }
16 }
```

Listing 4: Adaptive sequencer program in C++ on RVS

Example 4. *Listing 4 shows a sequencer program written in C++, which uses a higher level of abstraction compared to Listing 3 and also implements event handling for adaptivity. In Line 1 of Listing 4, a class called RampScenario is initialized. The implementation of this* class *is not provided here, however, it may abstract the CSR functions from Listing 2 and allows to set ramp parameters at a higher level. Next, in Line 2 the number of ramps is defined. Executing ramps is also done within a* for *loop in Lines 4–16. Lines 5–7 show the already mentioned high-level functions to set ramp segments. In Lines 9–15, there is a* switch *statement, which is used to check for events reported by the hardware and handle them. This example handles four possible scenarios. First, shown in Line 10, no event was detected and the execution of the program is continued as planned. Second and third, shown in Lines 11–12, an event occurs and is tackled with either hopping to another starting frequency or adding an arbitrary wait time between two ramps. Finally, shown in Lines 13–14, a critical or unknown event was detected. This means continuing is no longer reasonable, thus the ramp scenario is aborted immediately.*

A major advantage of RVS compared to the DSS is that late changes to the programming model, even after tape out, are now possible. This can be either used to fix bugs, but also to add or adapt algorithms if required. Due to the fact, that now all algorithms for generating ramps are written in software, the RVS is much more flexible compared to the DSS. This flexibility arises for both the user of the radar chip, because high-level languages like C++ are already known and there is no need to cope with a custom ISA, as well as for the developer of the radar chip, because less hardware changes are required in future for chip variants.

IV. Results

In this section, we present our results and insights for the RVS. First, in Section IV-A, we provide implementation details and simulation results. Thereafter, in Section IV-B, we share insights about the performance of the RVS as well as an area estimation. Finally, in Section IV-C, we qualitatively assess and compare the flexibility of DSS and the RVS.

A. Full Radar Chip Simulation

Our implementation of the RVS is based on a proprietary RISC-V processor in Verilog, extended with integer multiplication and division and CSRs (RV32IMC). In our industrial flow, we perform the full radar chip simulation using a *Virtual Prototype* (VP) [13], [14] implemented in SystemC [15]–[17]. For this, we lifted our RVS implementation using Verilator [18] to a C++ model including SystemC interfaces. The result was embedded into the VP. As an initial result, the specification and simulation of various ramp scenarios using the DSS and the RVS produced consistent and comparable outcomes.

In addition to generating ramps by just executing a sequencer program from memory, the RVS allows to utilize reporting events from the hardware units to the RISC-V processor. These events are mapped to CSRs, as illustrated in Fig. 3. By executing an adaptive sequencer program similar to Listing 4, we achieved the simulation result depicted in Fig. 4. We only show two signals here, the transmit frequency and a signal which indicates events triggered by the hardware. As can be seen, there are three more ramps executed after the time point when the hardware event occurred (lowest bit of `HWEvent_rd_data` is 1) until frequency hopping is done and the ramp scenario is further executed at another starting frequency. These three ramps are present due to the fact, that the respective "old" control values, which were generated before the event occurred, are already present in the FIFOs. Overall, we have shown that the proposed RVS allows frequency hopping on the basis of events. In the presence of interference, this enables targeted exclusion of corrupted ramps, eliminating the need to discard the entire measurement and thereby preserving valuable data and execution time.

We now consider the impact on performance and area.

B. Performance and Area

In the software-based approach adopted by the RVS, execution speed primarily depends on two factors. First, on the number of control signals which need to be set and second, whether control values are pre-calculated and stored in the memory or calculated during radar operation. For a performance estimation of the RVS, we operated it under its worst operating conditions: always all control signals are set, even signals which do not change and furthermore, the RVS calculates all control values during radar operation.

At a clock speed of 200 MHz for the RISC-V processor, this would allow ramps which require around $7.5\mu s$ on average. This means, that with no optimizations and the worst operating conditions for the RVS, it is still fast enough to perform typical

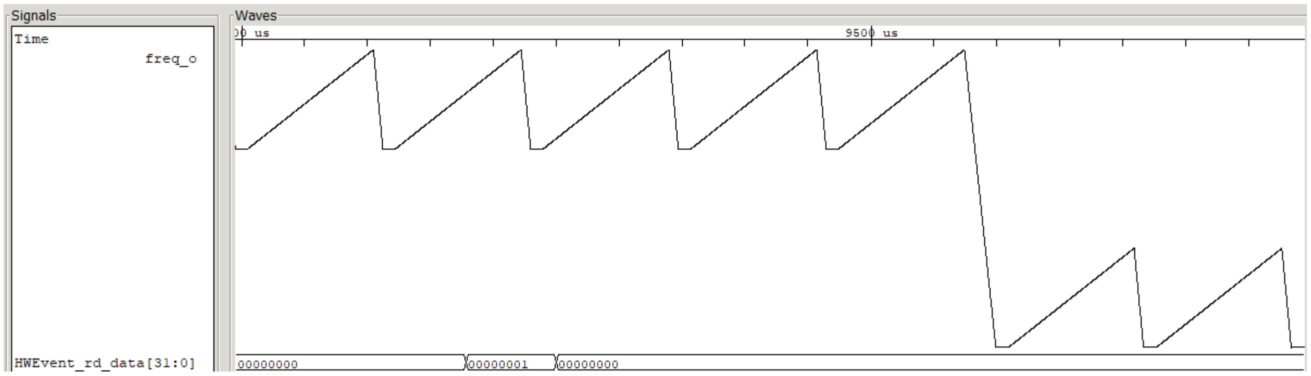

Fig. 4: Simulation showing frequency hopping after event occurs

ramps with a duration of around $25\mu s$. If faster ramping is required in future, there is still a broad range of optimizations possible which can be tuned for the particular application.

The DSS is already silicon proven, produced using a 28 nm CMOS process, however the RVS does just exist as model[1]. To obtain an approximate estimate regarding size difference, the RISC-V processor was synthesized using the same 28 nm process. This showed, that the RVS requires around 20-30% more silicon area compared to the DSS. Considering, the fact, that the RVS design is much more flexible which will save costs at other points in the project, this is an acceptable overhead. The flexibility aspect is detailed in the next section.

C. Flexibility

This section provides a qualitative assessment of carefully selected flexibility characteristics, as inspired by Hennessy and Patterson in [19], applied (and refined if necessary) to both the DSS and the RVS. The result is summarized in Table I. The first column gives the flexibility characteristic while the remaining two columns provide a score from --, -, o, +, ++ with ++ marking the highest for DSS and the RVS, respectively. In the following, each characteristic is explained.

Compatibility is expressed as having a stable ISA across generations of graphical processing units. The stability of the ISA can also be projected on the sequencing units discussed in this paper. The ISA of the DSS will most likely change for a new generation of radar chips or even for a new radar chip the current generation, because there is a high possibility that the signal structure changes or new features are added, and hence new instructions/opcodes are needed. In contrast, it is very unlikely that the RISC-V ISA will change.

Very close to compatibility, is *Reuseability*, which means reusing hardware or software in other products. For the DSS, the same as for compatibility applies, due to changes in the signal structure. The RVS allows reuse through abstracted interfaces implemented with CSRs [20].

The term *Scalability* is used when it comes to parallel computing, which means being able to expand memory and number of processors. However, in our case scalability can be described as the ability to extend the signal structure. Although the DSS requires hardware modifications, it imposes

TABLE I: Flexibility Characteristics

Characteristic	*Domain-Specific Sequencer*	*RISC-V Sequencer*
Compatibility	-	+
Reuseability	+	++
Scalability	+	o
Reconfigurability	--	++

no limitations on the number of supported signals. As the CSRs employed in the RVS have a limited address space and even less registers can be used for custom applications, there is a theoretical limit for the amount of control signals. Nevertheless, the RVS only utilizes a small fraction of the available CSRs, so this may not become a problem in future.

Reconfigurability is often referred to reconfiguring logic in hardware implementations, like on *Field-Programmable Gate Array*s (FPGAs), to enhance flexibility. For our work, reconfigurability means adding or changing features of the sequencer. Unlike the DSS and its ISA, which cannot be modified after tape-out, the RVS supports functional updates even in the radar chip's final application.

V. Conclusions

In this paper, we introduced the *RISC-V Sequencer* (RVS), a new RISC-V-based sequencer for *Frequency-Modulated Continuous-Wave* (FMCW) radar applications. With RVS, the user can create sequencer programs at a much higher level instead of using domain-specific low-level opcodes. In addition, adaptive ramp scenarios become possible, i.e. scenarios can react dynamically to the environment, for instance to mitigate interference. Overall, this shifts the whole paradigm how ramp scenarios are programmed. From the technical side, we integrate custom *Control and Status Register*s (CSRs) as part of RVS to enable the generation and distribution of control signals with precise timing for real-time radar sequencing. Ultimately, the proposed solution scales effectively to future radar systems, requiring only an extension of the CSR mapping in hardware, with all other adaptations managed in software.

In future work, we plan to investigate novel verification techniques [21], [22] and analysis techniques [23], [24].

Acknowledgment

This work has been partially supported by European Union's Chips Joint Undertaking (ChipsJU) under grant agreement No. 101139892 (EdgeAI-Trust), the Austrian Research Promotion Agency (FFG) under grant No. 909770, and by the LIT Secure and Correct Systems Lab funded by the State of Upper Austria.

[1]Verilog and verilated SystemC.

References

[1] M. A. Richards, J. A. Scheer, and W. A. Holm, *Principles of Modern Radar: Basic principles*. SciTech Publishing, 2010. [Online]. Available: https://digital-library.theiet.org/doi/abs/10.1049/SBRA021E

[2] R. Findenig and B. Greslehner-Nimmervoll, "Modular sequencer for radar applications," U.S. Patent US12 000 952B2, 2024.

[3] P. Ritter, M. Geyer, T. Gloekler, X. Gai, T. Schwarzenberger, G. Tretter, Y. Yu, and G. Vogel, "A fully integrated 78 ghz automotive radar system-an-chip in 22nm fd-soi cmos," in *2020 17th European Radar Conference (EuRAD)*, 2021, pp. 57–60.

[4] B. P. Ginsburg, K. Subburaj, S. Samala, K. Ramasubramanian, J. Singh, S. Bhatara, S. Murali, D. Breen, M. Moallem, K. Dandu, S. Jalan, N. Nayak, R. Sachdev, I. Prathapan, K. Bhatia, T. Davis, E. Seok, H. Parthasarathy, R. Chatterjee, V. Srinivasan, V. Giannini, A. Kumar, R. Kulak, S. Ram, P. Gupta, Z. Parkar, S. Bhardwaj, Y. C. Rakesh, K. A. Rajagopal, A. Shrimali, and V. Rentala, "A multimode 76-to-81ghz automotive radar transceiver with autonomous monitoring," in *IEEE International Solid-State Circuits Conference - (ISSCC)*, 2018, pp. 158–160.

[5] C. Kohlberger, R. Feger, R. Hüttner, A. Haderer, and A. Stelzer, "A 77-ghz quasi-monopulse tracking radar with metamaterial lens and transceiver feeds," in *European Microwave Conference (EuMC)*, 2024, pp. 1086–1089.

[6] R. Findenig, B. Greslehner-Nimmervoll, G. Itkin, M. J. Lang, U. Moeller, and M. Wiessflecker, "Efficient programming model for real-time radar sequencing in a radar device," U.S. Patent US12 248 088B2, 2025.

[7] A. Waterman and K. Asanović, *The RISC-V Instruction Set Manual; Volume I: Unprivileged ISA*, SiFive Inc. and CS Division, EECS Department, University of California, Berkeley, 2019.

[8] ——, *The RISC-V Instruction Set Manual; Volume II: Privileged Architecture*, SiFive Inc. and CS Division, EECS Department, University of California, Berkeley, 2019.

[9] M. Jankiraman, *FMCW radar design*. Artech House, 2018.

[10] G. M. Brooker, "Mutual interference of millimeter-wave radar systems," *IEEE Transactions on Electromagnetic Compatibility*, vol. 49, no. 1, pp. 170–181, 2007.

[11] M. Gerstmair, A. Melzer, A. Onic, and M. Huemer, "On the safe road toward autonomous driving: Phase noise monitoring in radar sensors for functional safety compliance," *IEEE Signal Processing Magazine*, vol. 36, no. 5, pp. 60–70, 2019.

[12] E. NCAP, "Euro ncap assisted driving, highway & interurban assist systems, test and assessment protocol v2.2," https://www.euroncap.com/media/83320/euro-ncap-ad-test-and-assessment-protocol-v22.pdf, September 2024, accessed: 2025-06-25.

[13] T. De Schutter, *Better Software. Faster!: Best Practices in Virtual Prototyping*. Synopsys Press, March 2014.

[14] R. Leupers, G. Martin, R. Plyaskin, A. Herkersdorf, F. Schirrmeister, T. Kogel, and M. Vaupel, "Virtual platforms: Breaking new grounds," in *Design, Automation and Test in Europe Conference*, 2012, pp. 685–690.

[15] "IEEE 1666-2023 standard for standard SystemC language reference manual." [Online]. Available: https://doi.org/10.1109/IEEESTD.2023.10246125

[16] D. Große and R. Drechsler, *Quality-Driven SystemC Design*. Springer, 2010.

[17] V. Herdt, D. Große, and R. Drechsler, *Enhanced Virtual Prototyping: Featuring RISC-V Case Studies*. Springer, 2020.

[18] W. Snyder, P. Wasson, D. Galbi, and et al, "Verilator." [Online]. Available: https://github.com/verilator/verilator

[19] J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach*, ser. The Morgan Kaufmann Series in Computer Architecture and Design. Elsevier Science, 2017.

[20] M. Melik-Merkumians, M. Wenger, R. Hametner, and A. Zoitl, "Increasing portability and reuseability of distributed control programs by i/o access abstraction," in *2010 IEEE 15th Conference on Emerging Technologies & Factory Automation (ETFA 2010)*, 2010, pp. 1–4.

[21] C. Hazott, F. Stögmüller, and D. Große, "Verifying embedded graphics libraries leveraging virtual prototypes and metamorphic testing," in *Asia and South Pacific Design Automation Conference*, 2024, pp. 275–281.

[22] ——, "Using virtual prototypes and metamorphic testing to verify the hardware/software-stack of embedded graphics libraries," *Integr.*, vol. 101, 2025.

[23] L. Klemmer and D. Große, "WAL: a novel waveform analysis language for advanced design understanding and debugging," in *Asia and South Pacific Design Automation Conference*, 2022, pp. 358–364.

[24] ——, "WAVING goodbye to manual waveform analysis in HDL design with WAL," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 43, no. 10, pp. 3198–3211, 2024.