Report: Towards a RISC-V platform with integrated radio IP

Timo Joas Paul Rigge Borivoje Nikolić

***Abstract*—Modern protocols for wireless communication are carefully designed towards their application to meet growing demand with regard to metrics like throughput, latency or reliability. Developing the hardware to implement such protocols is complicated and thus cost-intensive, since it typically requires optimization and design space exploration with respect to many variables. Thus, speeding up development cycles for integrated signal processing chains which are basic to every wireless hardware implementation is highly desirable. Recently, new generator-based design methodologies have been established [1] [2] as a first step to achieve this goal by re-usage of similar analog and digital intellectual property (IP) blocks. To further mitigate the resources spent on hardware design, we purpose a system on a chip (SoC) architecture in which flexible signal processing IPs are directly integrated with a general purpose RISC-V core. Our architecture allows for software reconfiguration of protocol parameters while minimizing the need for redesigning hardware. As an example, we demonstrate the implementation of parts of a transceiver chain for a low latency wireless protocol. We analyze the performance of software, which runs on top of an open-source real time operating system (RTOS) and show that typical workloads can realistically be served.**

1. Preface

This letter serves two purposes: It documents the progress made on running and benchmarking real-time, Zephyr RTOS en-powered software on a RISC-V core synthesized on the Xilinx ZC706 development board. Details of the implementation are available online [5] [6]. At the same time, it is meant as a blueprint for a future publication. Therefore, we may reference to the integration of real transceiver chain IP blocks. However, these weren't available at the time of the compilation of this document. All of the presented data is therefore gathered employing a benchmarking IP block.

2. Introduction

2.1. Background

Our approach aims to reduce hardware design complexity, by increasing re-usage of IP blocks. To this end, DSP IPs are directly controlled by a general purpose processor (*Com core*) as sketched in Figure 1a. This simplifies hardware adaption to a specific radio transceiver chain implementation, since configuration of the IPs can be conveniently done in software - the Com core simply writes parameters to control registers. At the same time, the explorable design-space is increased, since complex parameter changes among several IPs do not require hardware re-design. Additionally, the Com core can handle computationally in-intensive tasks previously implemented in hardware on it's own. Therefore, it reads status registers of the hardware, performs simple but completely software-defined calculations and writes the result back.

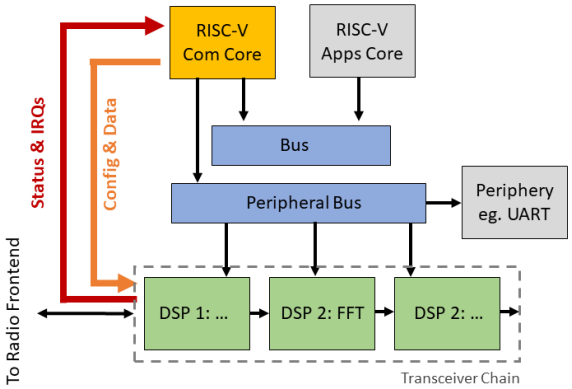

Figure 1. Overview SoC architecture. The Com core interacts with integrated DSP IPs in the transceiver chain via a peripheral bus. To this end, it may receive IRQs and read status registers of the hardware. In turn, it writes back configuration data and results of calculations done in software.

Our architecture is completely based on open-source hardware. In particular, the Com core is implemented as an open ISA RISC-V core. In comparison to commercial, mostly ARM-based solutions, this both reduces costs and simplifies legal considerations connected to licensing. The employed hardware generators enable fast workflows, which are easy to parametrize for specifc needs. For demonstrating the feasibility of our approach, we implement a modular transceiver chain that will finally run OccupyCow - a wireless protocol currently researched in our group.

2.2. Protocol under investigation

OccupyCow [3] is designed for industrial applications with strict requirements on low latency and high reliability. Since its parameters are still under investigation, our goal is to provide a flexible rapid prototyping platform. For the scope of this work, we employ a simplified version as shown in Figure 2a since it is sufficient to gain insights into hard- and software requirements. The protocol exploits multi-user diversity by using simultaneous relaying to enable reliable two-way communication between a central controller and a slave node within a cycle of length T. In each of the distinct phases of a cycle (Figure 2c), timeslots are assigned to transmission from one or multiple nodes. In the following, we usually take the perspective of a single node, which must serve different tasks for every timeslot. Typically, timeslots within the same phase of the cycle require execution of similar tasks and thus also feature similar interaction patterns between software running on a processor and hardware registers.

An example for advantageous software-defined configuration of the hardware is switching between different protocol phases. Assume that a dedicated hardware block performs a FFT operation for decoding OFDM symbol on received data. As FFT length varies among different phases, software must be able to track the position in time within the protocol cycle and write updated configuration parameters into a hardware register of the FFT block. In this way, only software changes are required when FFT parameters change during evolution of the protocol.

2.3. Design goals

Typically, radio applications pose strict limitations on execution time. For our case, all software-handled tasks must be finished within a protocol cycle of time $T \sim 1$ ms. Failing this soft real-time constraint leads most likely to package loss, which shall be limited to a probability $p_{crit} < 10^{-9}$. Consequently, the main share of our execution time distribution should reside well left of the vertical red line in Figure 2b. Events within the tail of the distribution are hard to capture in verification. Thus, it is beneficial to minimize statistical influences on the execution time. Therefore, our key design goal is predictability at reasonably high performance. Obviously, this imposes several restrictions for both hard- and software running time-critical code. The employed Rocket chip generator for RISC-V processors feature high configurability allowing to parametrize a core that matches exactly our application's requirements. Furthermore, integration with custom periphery is rather simple. Our custom hardware and is described in the next Figure 3. The employed software design is investigated in greater detail in subsection 6.2.

3. Platform Description

Our platform is compromised out of 2 functional hardware parts: A transceiver chain and a general purpose RISCV-V processor (Com core). Figure 1 shows the block diagram of the SoC. All presented data is obtained by mapping the SoC onto a FPGA as described in detail in subsection 6.1.

The transceiver chain is formed by several DSP IP blocks performing the necessary signal processing from the radio to digital domain and vice versa. The IP blocks are integrated with the Com core through a TileLink peripheral bus. Although not strictly necessary, we advocate for a flexible IP design to improve re-usability, such that parameters can be altered by writing to configuration registers. Similar to the transceiver chain, additional periphery is easily added. For instance, we optionally adapted an open source UART controller (SiFive).

This work focuses on the Com core, which performs two time-critical tasks. In addition to the mandatory configuration of the hardware IPs, it may also handle computationally in-intensive tasks. We choose to run a real time operating system (RTOS) underneath our application. Recently, Zephyr was ported for the RISC-V RV32 ISA [7]. After adding support for our FPGA board, Zephyr provides us with a convenient developing environment. C applications can easily access registers of any periphery device via memory mapping while synchronization and scheduling is mostly abstracted away. The high granularity of kernel configuration options allows to match our real-time requirements with the IRQ and software design described in greater detail in subsection 6.2.

In order to reach predictability in execution time, we must avoid two major origins of undesirable time uncertainty (jitter) - cache misses at various parts of the memory hierarchy and dynamic branch (mis-) prediction. Addressing cache issues, we configure the D-Cache as a scratchpad memory where kernel and application data symbols are located (4.1 kB and 15.8 kB, respectively). A scratchpad size of 64 kB leaves margin for further application extensions.

Every additional state in the state machine (see subsection 6.2) takes up $k_s = 88$ bytes of memory in the scratchpad data segment. The total scratchpad memory footprint m_{sc} thus can be given as $m_{sc} = 18.6$ kB $+M_{states}k_s$. For saving memory, substates are introduced which do not require data memory. On the other hand, they are limited to repeat the same actions for multiple substates, which can be useful eg. to serve the same tasks for different nodes.

Additionally, we choose an I-Cache size (256 kB) that is strictly bigger than the instruction segment of our applications code of 39 kB. For verification of our real-time requirements, we access the performance registers (available through the RISC-V ISA) and determine an I-cache miss count of 0 for our critical loop. To avoid dynamic branch mis-prediction as a root of jitter, we configure our core without branch target buffer (BTB). In turn, our critical software loop makes intensive use of compiler annotations such that its common path is taking the branches buffered in the execution pipeline (known as static branch prediction). Turning these optimizations on, lowers the amount of branch mis-predictions in a single run of our critical loop for $\sim 20\%$ (~ 900 vs. ~ 720). In comparison, the

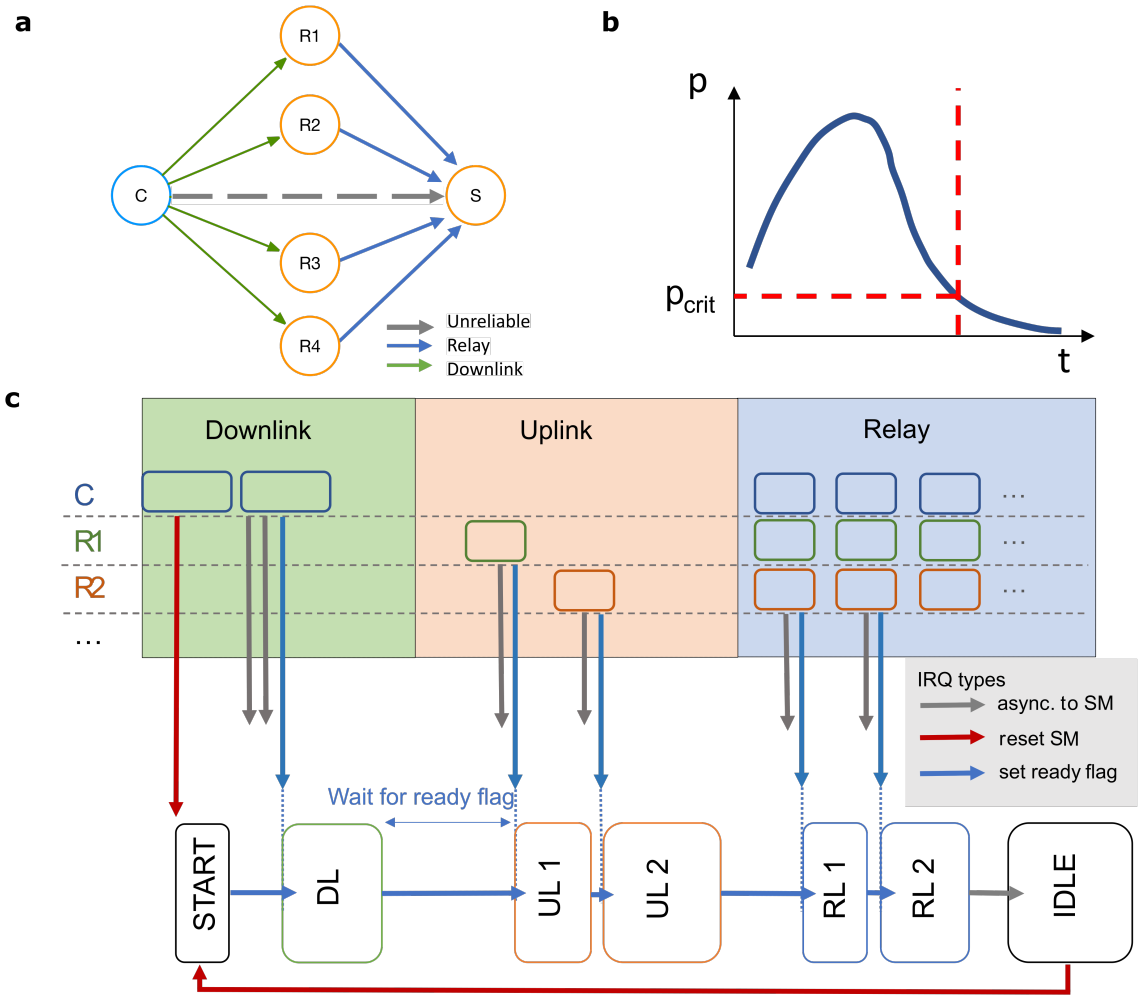

Figure 2. Wireless Protocol. (a) Illustration of relaying in OccupyCow. A controller node (C) sends data to a slave node (S) in a 2 step process. After downlink (green) to a set of nodes (R1...R4), they simultaneously relay (blue) to establish a more robust connection between controller and slave. The uplink from S to C is not shown, but works accordingly. (b) Sketched probability distribution of the execution time of processing a wireless package. (c) Simplified cycle of Occupycow. IRQs thrown by the DSP IPs drive the Com core's state machine.

absolute amount of mis-predictions is substantially reduced to $\sim$ 210 if employing a BTB. This is expected, since due to limitations of the Rocket branch predictor, backward branches will always be mis-predicted when using no BTB and cause a 5 cycle penalty. However, without BTB we avoid statistical variations due to buffering of the target addresses. The effectiveness of our measures is verified in Figure 3, where the execution time of our critical loop is nearly entirely concentrated into a single peak of the probability distribution.

4. Results

4.1. Execution Time Estimation

As stated above, the Com core allows to handle computationally in-intensive tasks additionally to its main configuration workload. While it is obvious that such tasks should not exceed a certain degree of complexity, giving an exact threshold is not trivial. Eventually, it is necessary to specify, which tasks can completely run in software and which require dedicated hardware accelerators. Facilitating this design decision, we create a model for the execution time of the software T_{sw} as a tool for optimization. Then, a necessary condition that needs to hold true up to the probability p_{crit} for the execution time T_{sw} of all software tasks is

$$T_{sw} = T_{sm} + T_{irq} + T_{os} < T_{cyc} \quad (1)$$

Where T_{sw} has been decomposed into the time T_{irq} spent in handling interrupt requests (IRQ), the duration T_{sm} it takes our finite state machine (FSM) to handle all work and the time T_{os} the OS requires for its operation. The latter is usually dominated by thread scheduling, which is also considered a major source for jitter. However, since our kernel configuration (see subsection 6.2) avoids rescheduling

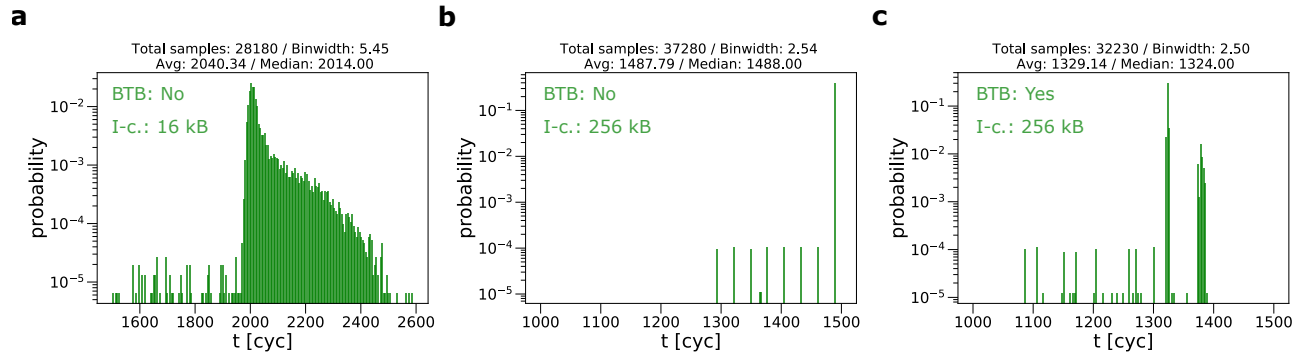

Figure 3. Execution time of the FSM loop (no load) with (a&c) and without (b) BTB and different sizes of I-cache. BTB or small I-cache size both lead to undesired jitter.

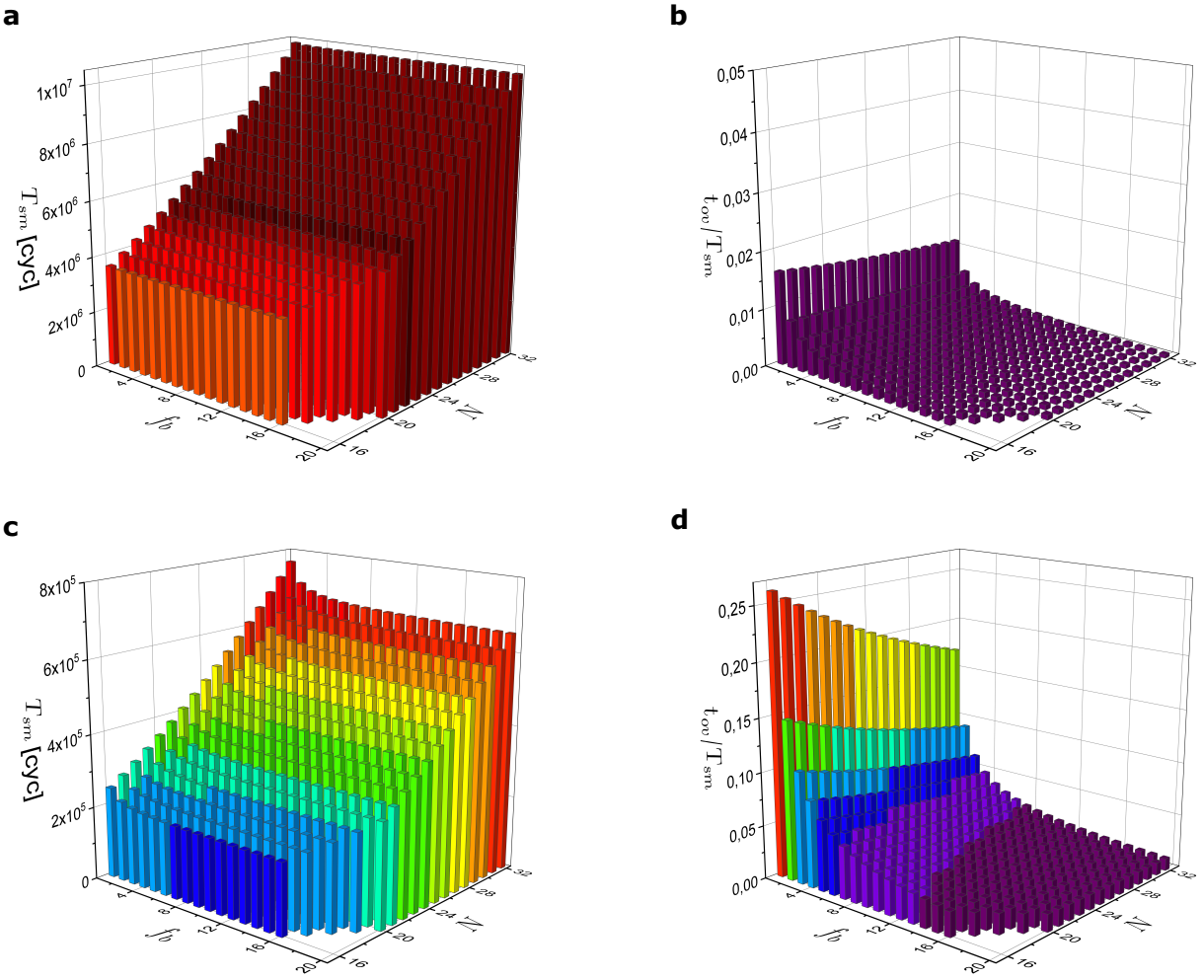

Figure 4. Analysis FSM. Estimated software execution time T_{sw} versus number of nodes N and batching factor f_b for scenario 1 (a) and 2 (c). Relative overhead t_{ov}/T_{sw} for scenario 1 (b) and 2 (d).

during execution of the FSM, we may safely neglect T_{os} here. T_{irq} generally dependens on implementation details of the DSP IP blocks. For our transceiver chain, we estimate for N wireless nodes $T_{irq} \sim 2NT_{isr}$. Here, T_{isr} is the average execution time of an interrupt service routine. Employing Xilinx chipscope for probing the PC of our core, we experimentally determine $T_{isr} = 409$ cpu cycles. We model T_{sm} applying the simplification that it depends solely on the number M_r of reads from memory, write to registers (M_w) and multiply accumulate operations (M_m) for software tasks. Each operation is assumed to be applied to 32 bit wide registers. Experimentally determined performance characteristics are listed in Table 1. We note the possibility for performance improvements by reading and writing to larger register and memory blocks at once. This is why the model should be understood as an upper limit for the software execution time. Finally, the model for T_{sm} can be given as

$$T_{sm} = k_s \frac{N}{f_b} \Big(t_{ov}(N, f_b) + f_b(M_m \kappa_m + M_r \kappa_r + M_w \kappa_w) \Big) \tag{2}$$

k_{sch} is a dimensionless parameter quantifying how well tasks can be arranged within the execution of the FSM. For instance, we assume a factor of $k_{sch} = 1.1$ meaning that an additional 10% overhead is spent waiting on input data before new tasks can be scheduled. Depending on the exact duration of waiting intervals, it might be favorable to send the core to an energy saving state.

Our analysis is based on different tasks that have to be served in a single protocol cycle of Occupycow. In Table 2 we list tasks that might realistically run on the Com core with their required number of operations. The main workload of configuring IPs is given in the first row. Computation-intensive signal processing which certainly requires hardware is neglected, here. We model the software execution time for two scenarios. In scenario 1, all listed tasks are assumed to be run by software. On the other hand, in scenario 2, a hardware accelerator for channel estimations reduces the amount of software operations. Note that we assume an accelerator to feature a result register to read and an input registers to write. This requires the typical inputs (source and destination address, some weight) to be compressed in a single, 32 bit wide register.

In Figure 4c, we show the estimated total execution time T_{sw} for scenario 2 with respect to the number of wireless nodes N and the batching factor f_b (see details in subsection 6.2). There exists a benefit for batching which is on the other hand not a dominating influence. Since batching is a measure to reduce overhead, the observation is consistent with the moderate overhead shown in Figure 4d. As the relative time spent on executing code unrelated to the protocol tasks is mostly below 25%, lowering overhead has a limited effect on the total execution time. If we target a protocol cycle time $T_{cyc} = 1$ms, we obtain clock frequencies for the SoC which are required to handle the workload as shown in Table 3. For scenario 1, it can be concluded from Figure 4c&d that execution time is barely influenced by the overhead. However, required clock frequencies of > 3.5 GHz are clearly illusive. On the other hand, scenario 2 should be realistically run-able on an ASIC with 850 MHz clock frequency, even without batching.

Given the estimates on software execution time, two main results can be stated: First, our model is a viable tool to decide which workloads can be handled by software. If eg. due to a power budget a clock frequency goal is given, it enables optimization of the tasks preferably implemented by hardware accelerators. Moreover, we present strong indication that Occupycow as an example for a wireless protocol, can be implemented by the proposed integrated radio platform, if accelerating hardware for the channel estimation is provided.

We note important limitations to our argument which is based on fulfilling Equation 1. First, the assumed scheduling factor $k_s = 1.1$ might be higher. Detailed analysis of the task timing in the OccupyCow protocol is needed to find a more accurate estimate for k_s. More importantly, any prediction based on the execution time of the protocol cycle neglects possible inter-protocol real-time constraints. In particular, our analysis cannot assure that tasks calculating data for subsequent protocol phases are finished in time. However, such effects may be neglected for the case of Occupycow, since using data from the prior protocol cycle can be tolerated.

5. Conclusion

We proposed a SoC architecture, in which radio DSP IPs are directly integrated with a RISC-V core. Running benchmarking applications on top of Zephyr RTOS, we confirmed a negligible amount of jitter in a FSM loop and calibrated an execution time model. The latter allowed to estimate that moderate clock frequencies support running the Occupycow protocol, if an accelerator was used for channel estimation.

Our findings pave the way for a future integration of real radio IP within the presented hard- and software platform.

References

[1] Bailey, Wright, Mehta, Hochman, Jarnot, Milovanovic, Werthimer, Nikolic *A 28nm FDSOI 8192-Point Digital ASIC Spectrometer from a Chisel Generator* CICC, 2018

[2] Crossley, Puggelli, Le, Yang, Nancollas, Jung, Kong, Narevsky, Lu,Sutardja, et al. *BAG: A designer-oriented integrated framework for the development of AMS circuit generators*, Computer-Aided Design (ICCAD), 2013.

[3] Swamy, Suri, Rigge, Weiner, Ranade, Sahai, Nikolic *Cooperative Communication for High-Reliability Low-Latency Wireless Control*, IEEE ICC, 2015.

[4] Asanović, Avizienis, Bachrach, Beamer, Biancolin, Celio, Cook, Dabbelt, Hauser, Izraelevitz, Karandikar, Keller, Kim, Koenig, Lee, Love, Maas, Magyar, Mao, Moreto, Ou, Patterson, Richards, Schmidt, Twigg, Vo and Waterman *The Rocket Chip Generator*, Technical Report No. UCB/EECS-2016-17, 2016

[5] Online repository, Apps on Zephyr and fpga-zynq
https://github.com/timoML/zephyr-riscv/tree/cow

[6] Online repository, Rocket rv32 on Xilinx ZC706
https://github.com/timoML/fpga-zynq/tree/new-devices-cow

[7] Online repository, Jean-Paul Etienne, Zephyr for RISC-V
https://github.com/fractalclone/zephyr-riscv

[8] Online repository, Rocket for Xilinx Zynq
https://github.com/ucb-bar/fpga-zynq

6. Methods

6.1. Hardware

All parts of the SoC in Figure 1 are derived from Chisel based generators for digital or analog IP blocks. A top level instance of the Rocket chip generator integrates both RISC-V cores with the DSP IPs and generates synthesizable verilog RTL. The Com core, which is the one interacting with the DSPs, is a RV32 RISC-V core generated by the Rocket chip generator. In addition to the processing core, Rocket implements a on-chip DTIM memory, an instruction cache, an interrupt controller, a real-time clock, and some TileLink ports. As a key feature, Rocket allows for detailed parameterization of all generated hardware. Parameters are listed in Table 4. Additional FPGA logic is added by the fpga-zynq extension [8] of Rocket chip, in order to tether our SoC to a host ARM core providing console output and a programming interface for our SoC's memory. Xilinx tools are employed for synthesization and implementation onto a Xilinx ZC706 development board.

For the analysis in subsection 4.1, we exclusively employ a benchmarking IP block. While it is connected to the Com core like the actual DSP hardware, functionality is limited to generate simple interrupts employed for benchmarking.

Any software task above the MAC level is served by an additional RISC-V core. This second, so far neglected, *Apps core* receives packets after decoding from the Com core and is responsible for all higher-level processing and interfacing with applications. Vice versa, it prepares a packet to be sent and delivers it to the Com core. The details of its implementation reside outside of the scope of this work.

6.2. Software

The software running on the Com core serves its two tasks (configuration and in-intensive workloads) by analyzing status registers or receive buffers of the DSPs and by taking appropriate action. We implement a finite state machine (FSM) as a natural choice for a real-time system. The design of our FSM and its relation to a cycle of the wireless protocol are sketched in Figure 2c. In the simplest case, every timeslot in the protocol corresponds to a state in the FSM. In every state, actions are invoked by the FSM to serve the corresponding tasks. State transitions are guarded by flags that have to be set by the hardware prior to entering a state. If needed, switching events can be used to trigger specific transitions. Otherwise, each state defines a default state which is switched to next.

A typical data flow (see Figure 5) would involve an IP to generate an IRQ to the Com core, informing the latter about a change in one of the IP's status registers. The invoked software ISR then transfers the updated state to its memory pool and sets flags for later handling by the FSM thread. After finishing the calculation in an action connected to one of the FSM states, the result is written-back to the IP.

Parameter	Value
Overhead per state of FSM:	
$t_{ov}(N, f_b)$	$t_1 + f_b t_2 + N_{act} t_3$
Calibration values:	
t_1	373 cyc
t_2	9 cyc
t_3	177 cyc
#actions:	
N_{act}	12
Duration MAC operation:	
κ_m	10 cyc
Duration read from memory:	
κ_r	42 cyc
Duration write to periphery reg:	
κ_w	51 cyc
Average duration of ISR:	
T_{isr}	409 cyc

TABLE 1. Parameters to model the execution time of the FSM according to Equation 2. Values are obtained by fitting the model to experimentally, software-measured execution times. For κ_x we set $N = f_b = 1$ and measure the execution time while increasing the respective M_x. t_1, t_2 and t_3 are calibration parameters obtained from a 2D fit to experimental data while setting $\kappa_x = 1$.

Crucially, the execution time of the FSMs critical loop must be predictable with high confidence. To this end, it is run inside a cooperative (non-preemptible) thread. All tasks within a single protocol cycle are implemented such that no thread rescheduling is required. We leverage a property of our use-case to reduce usage of computationally expensive synchronization objects: Since order and time of hardware-generated events are known a priori to good precision, most synchronization can be done by waiting on lightweight atomic flags. For instance, transitions in the FSM typically require certain flags to be set. Minimal usage of more complex synchronization objects inside ISRs reduces execution time and jitter of the interrupted context and avoids collisions between different IRQs. Note that our IRQs have equal priority and that the employed PLIC does not support preemption of ISRs. Some IRQs may even only transfer data from hardware registers to memory relying on synchronization done by subsequent events. In such cases, the protocol cycle must guarantee that event order is strictly satisfied.

There is overhead connected to the execution of the FSM logic. Especially, every state transition runs through code, eg. to look up the target state given the distinct event that triggered the transition. Therefore it is beneficial to reduce the amount of states, which was initially proportional to the number of timeslots in the protocol. Batching can mitigate the issue by compressing several timeslots into a single software state of the Com core's FSM. In turn, this typically requires additional hardware functionality to distinguish data from different wireless nodes in the same shared state. The batching factor f_b introduced in our analysis in subsection 4.1 gives the number of timeslots per state of the FSM.

Task	Scenario 1 M_m, M_w, M_r	Scenario 2 M_m, M_w, M_r
DSP Configuration	30 / 0 / 30	30 / 0 / 30
CFO + Time Sync.	6(N+2) / 3(N+2) / 4(N+2)	6(N+2) / 3(N+2) / 4(N+2)
Channel Estimation	2208 / 60N+1088 / 1088 + 60N	2(N+2) / N+2 / 3
Schedule Tx	10 / 0 / 5	10 / 0 / 5
Relay Decision	2N / 1 / 5	2N / 1 / 5
Relay Selection	20N / 0 / 5	20N / 0 / 5

TABLE 2. Assumed number of basic operations for handling tasks for N wireless nodes on the CPU. A Occupycow protocol cycle is separated in 3 phases (down- and uplink, relaying) and a single task may be repeated for multiple users in the uplink and relay phase. In comparison, scenario 1 performs additionally the channel estimation in software.

N,f_b	Scenario 1 f_{clock} (MHz)	Scenario 2 f_{clock} (MHz)
16 / 4	3930	225
16 / 1	3975	275
32 / 4	11230	745
32 / 1	11320	840

TABLE 3. Estimated CPU clock frequencies required to run software workloads for OccupyCow at a protocol cycle time of $T_{cyc} = 1$ms and $k_{sched} = 1.1$.

Parameter	Value
ISA extension	RV32 imacf
Core freq.	50 MHz
Virtual Memory support	No
BTB	No
D-Cache	64 kB scratchpad
I-Cache	256 kB

TABLE 4. Configuration details of the used RISC-V Com core.

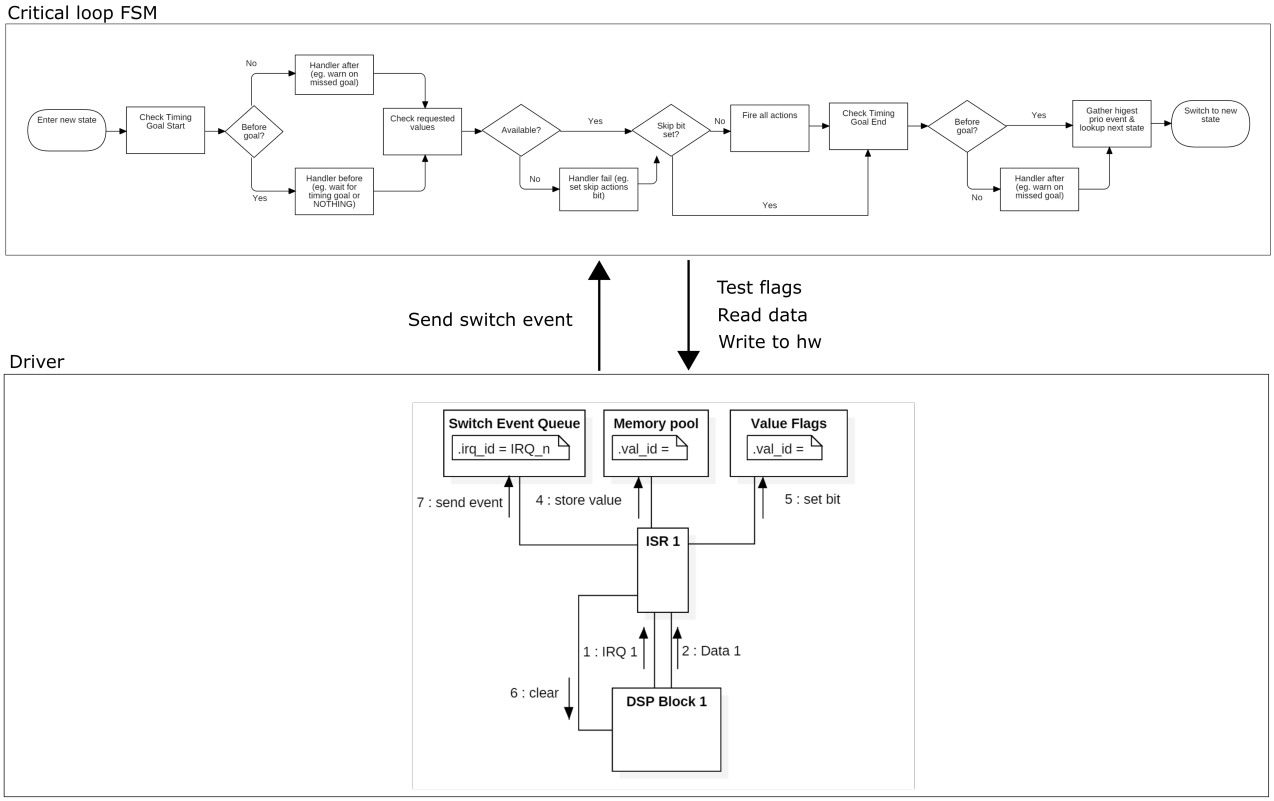

Figure 5. Interaction between FSM and driver. Top: Flowchart of the critical loop of the FSM. Different handlers can be defined to react on missed time goals or unavailable requested hardware values. The availability of the latter is checked by testing flags of the driver. Bottom: Data flow of the driver. The corresponding ISR of a DSP IP block stores data in a memory pool and sets flag bits. Additionally, switching events are sent to the FSM through a software queue.