DEVELOPING DISTRIBUTED HARD-REAL TIME SOFTWARE SYSTEMS USING FPGAS AND SOFT CORES

T. Włostowski, J. Serrano, CERN, Geneva, Switzerland
F. Vaga, University of Pavia, Italy

Abstract

Hard real-time systems guarantee by design that no deadline is ever missed. In a distributed environment such as particle accelerators, there is often the extra requirement of having diverse real-time systems synchronize to each other. Implementations on top of general purpose multitasking operating systems such as Linux generally suffer from lack of full control of the platform. On the other hand, solutions based on logic inside FPGAs can result in long development cycles. A mid-way approach is presented which allows fast software development yet guarantees full control of the timing of the execution. The solution involves using soft cores inside FPGAs, running single tasks without interrupts and without an operating system underneath. Two CERN developments are presented, both based on a unique free and open source HDL core comprising a parameterizable number of CPUs, logic to synchronize them and message queues to communicate with the local host and with remote systems. This development environment is being offered as a service to fill the gap between Linux-based solutions and full-hardware implementations.

BACKGROUND

Real-time controls in particle accelerators cover a broad range of applications that have different requirements for processing power, latency and determinism. For the purpose of this article, we divided these into the three following categories:

1. less than a microsecond of latency, such as beam injection and extraction, low level RF, interlocks and machine safety systems,
2. from dozens of microseconds to one millisecond, examples, power converter controls, orbit feedback or a number of timing and event distribution systems,
3. slow controls, such as cryogenics, vacuum, positioning or radiation monitoring.

Usually the parameter that has the highest impact on the architecture of the control system electronics is the worst case (never to be missed) processing latency. The sub-microsecond controls are the exclusive domain of FPGAs, ASICs and dedicated analog circuitry. On the contrary, the systems from point 3 are usually implemented on Programmable Logic Controllers (PLCs), or in many cases on industrial PCs running non-real time operating systems.

The widest variety of controls applications lies in between these two extremes. While the actions executed by these systems usually have strictly specified execution times, the data processing that triggers these actions can be done ahead in time, provided that the processing latency is guaranteed to

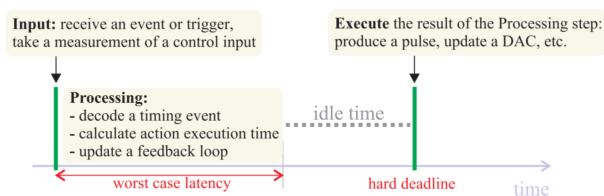

Figure 1: A typical hard-real time control system.

never exceed a certain upper limit, as shown in Figure 1. For example, the receiver of the CERN General Machine Timing (CTR) can produce pulses with 40 nanosecond resolution, but the minimum time between reception of a timing event and the generation of a corresponding pulse is almost one millisecond.

The traditional design approach in such systems, using an FPGA and developing custom HDL cores, brings several disadvantages:

- Only a small part of the HDL code implements the strictly timed part. In case of the CTR, these are the counters which drive the outputs and constitute around 20 % of the total HDL code. Event reception, filtering, logging and counter configuration logic has much slower execution time constraints.
- HDL development and testing take much more time compared to software development.
- Custom HDL cores need custom drivers, requiring additional development manpower.

More recent designs rely on soft processor cores, such as Xilinx MicroBlaze [1] or Altera Nios [2], which are responsible for the “slow” part, connected to custom logic implementing the “fast” part. This reduces the development time, but still leaves the following issues:

- No standard way of low-latency communication and synchronization between the devices (if they form a distributed system).
- Vendor lock-in - the cores provided by the FPGA vendors are not portable to other FPGAs,
- Issues with portability of drivers and low level host software, which are different for each FPGA manufacturer and each application.

In this article, we present the *Mock Turtle* (MT) - an HDL and software framework targeted at hard real-time distributed applications, offered as a ready to use service. *MT* provides:

- A deterministic multi-core CPU system that can be freely interfaced to user logic.
- A standardized way of communication with the host machine and (if needed) other devices in a distributed network.

ISBN 978-3-95450-148-9

Copyright © 2015 CC-BY-3.0 and by the respective authors

- Optional built-in White Rabbit (WR) [3] synchronization.
- A generic Linux device driver, user space library and a set of GNU-based development tools.

The following chapters present the architecture of the HDL and software stack of the MT, the project status and future goals. We also describe the architecture of the *uRV* - an open source soft CPU core, being developed at CERN and targeted at hard-real time embedded applications.

HDL ARCHITECTURE

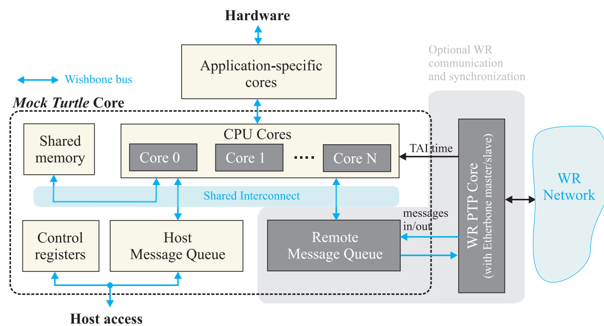

Figure 2: *Mock Turtle* Core Architecture.

Figure 2 shows a simplified block diagram of a *Mock Turtle*-based system. The key ingredients of the *MT* design, described in the following subsections, are:

1. Up to 8 32-bit CPU cores with a Shared Memory block.
2. Message queues, enabling communication with the host system and optionally, remote MT nodes.
3. User cores, controlled by the *MT* CPUs.
4. Control and debug logic.

All these modules are interconnected using Wishbone [4] buses, with a central, multiport crossbar SI (Shared Interconnect).

The CPUs and Shared Memory

MT provides a user-selectable number (1 to 8) of 32-bit RISC processor cores, organized into Core Blocks (CBs) as depicted in Figure 3. Each CPU has a private program and data space, located in the FPGA internal memory, whose contents can be uploaded at any time by the host software. This approach allowed to maximize the determinism of code execution by avoiding unpredictable wait states caused by arbitration of accesses to a shared memory block. For the same reason, the CPUs do not support interrupts - every asynchronous request must be serviced by means of polling.

The CPU core we used was initially a modified version of the *Lattice Mico32* (LM32) [5], due to its portability, speed and low footprint. Recent versions of the *MT* use the *uRV* RISC-V CPU core (described in a separate section of this article). We decided to switch to RISC-V as its architecture looks more viable in the long term than the proprietary architecture of LM32 (the LM32 license, despite being open source, contains export restrictions).

In order for the CPUs to communicate with each other, the *MT* core provides a Shared Memory (SMEM) block of user-configurable size, accessible by all the CPUs in an atomic way. The SMEM facilitates implementation of common inter-process communication primitives such as semaphores, mutexes, locks and queues, by providing a set of atomic operations: add, subtract, bit set, bit clear, bit flip and test-and-set. These operations are executed by reading or writing to the SMEM with the high address bits selecting the desired atomic operation (example in Listing 1). The SMEM can be also accessed by the host software, with the same atomic features as those available for the CPU cores.

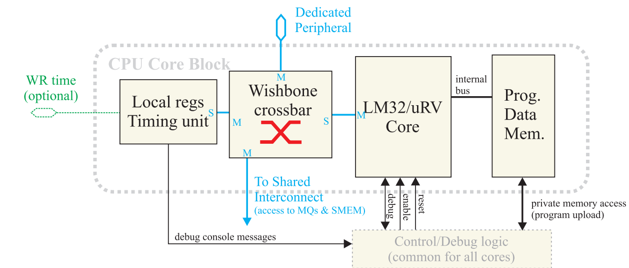

Figure 3: Design of the CPU Core Block (CB) in the *MT*.

Each CB also contains a set of Local Registers, accessible exclusively from the CB's own CPU. These registers let the CPU perform the following operations without disturbing other CPUs (i.e. without using shared resources):

- Get the current time (provided by WR or a local counter).
- Execute accurate delays (through self-decrementing delay generation registers).
- Poll the presence of received messages in the Message Queues.
- Send debugging messages to the host system.

Listing 1: SMEM access semantics example

```
#define OFFSET_ADD       0x00010000
#define OFFSET_TEST_SET  0x00020000

// atomically adds x to the *var
void atomic_add(uint32_t *var, uint32_t x)
{
  *(uint32_t *) (var + OFFSET_ADD) = x;
}

// atomically sets to *varto 1, returns
// the value of *var before set
uint32_T atomic_test_set(uint32_t *var)
{
  return *(uint32_t *) (var +
      OFFSET_TEST_SET);
}
```

The Communication System

The communication system consists of two message queues, shared by the CPUs. The Host Message Queue (HMQ) passes messages between the CPUs and the host system. The HMQ is the primary means of communication

Copyright © 2015 CC-BY-3.0 and by the respective authors

ISBN 978-3-95450-148-9

between the node and the host software (e.g. FESA [6]). Presence of an outgoing message is indicated to the host by raising an interrupt.

The optional Remote Message Queue (RMQ) exchanges messages with remote nodes in the WR network. The Etherbone protocol [7], developed by GSI, is used as the transport layer. Writing a message in the RMQ automatically sends out a UDP packet containing the message.

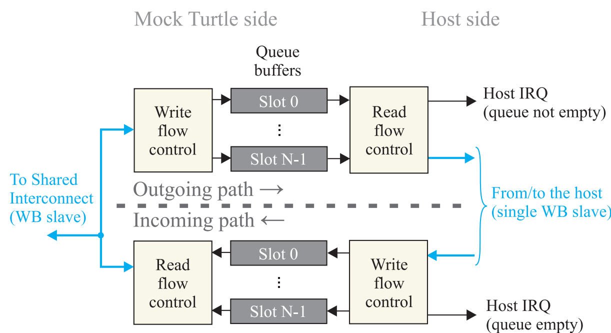

Figure 4: Design of the Host Message Queue.

The structure of the HMQ is depicted in Figure 4. The MQs are multi-word FIFOs, containing a number of unidirectional communication channels called *slots*. Outgoing slots transfer messages from *MT* CPUs to the Host or other MT nodes. Conversely, incoming slots let the CPUs receive messages sent by the host or the remote nodes. The number of slots is configured by the user. Each slot can buffer a user-configurable number of messages of configurable maximum size. Multiple slots allow handling independent message streams from different sources - for example, the Trigger Distribution application [8] uses a pair of incoming/outgoing slots for the control commands and a separate outgoing slot to stream the log of executed triggers.

In order to send a message, the transmitter writes a number of words to an MQ outgoing slot and marks it as ready to send. The receiver side gets an indication that its MQ incoming slot is not empty, reads out its contents and indicates to the MQ that it has processed the message. MQs ensure integrity of the messages: if the message is not received completely (i.e. the Etherbone core connected to the RMQ reported an error), it is not received at all.

Since *MT* is designed for hard real-time applications, the MQs do not have any flow control: if an MQ slot becomes full, it starts to drop the incoming messages. Users may implement flow control in software if needed, although in all *MT* applications we foresaw, any buffer contention is considered erroneous, as it may break the deterministic behavior of the system. We are developing a Forward Error Correction core to improve the robustness of message delivery without using retransmission.

Connectivity

The *MT* can interface with the user HDL design in two ways:

- Each CPU Block has a Dedicated Peripheral (DP) Wishbone master port that can be connected to the core's private peripherals. For example, in the WR Trigger Distribution [8] node, the CPU core responsible for reading out timestamps has a Time-To-Digital Converter Core connected to its DP port.
- The Shared Interconnect provides a Share Peripheral (SP) Wishbone master, that all cores can access concurrently. A typical use case of the SP is connecting large memories for storing auxiliary data or diagnostics info.

From the host side, all *MT* features are accessible through a Wishbone slave port and an interrupt line. The Self Describing Bus (SDB) [9] is used to automatically discover all the *MT* cores in the system.

THE *URV* SOFT PROCESSOR

The *uRV* core (expanded as micro RISC-V) is a small, robust and open source soft CPU core, targeted at deeply embedded FPGA applications, such as the *MT* or the WR PTP Core [10]. We took the following assumptions when designing the *uRV*:

- Fully open (no patent/trademark limitations) and standard ISA (Instruction Set Architecture).
- Optimize for modern FPGA resources and executing applications primarily from the FPGA's internal memory. Balance optimization effort between clock speed and FPGA area.
- Design with portability in mind. Separate the FPGA-specific features from the common CPU code.
- No high level operating system facilities, such as an MMU (Memory Management Unit). We intend the core to run low-level deterministic applications.

We decided to use the RISC-V [11] ISA with Multiply/-Division extension (RV32IM) [12], developed at Berkeley University. The primary reasons were simplicity (27 base integer instructions with clearly defined extensions), clear legal status of the ISA and availability of high quality development tools (recent GCC and Clang ports). Additional reason to support RISC-V was the vivid and diverse developer community, with ongoing high-performance silicon implementations and backing of several universities and companies.

Core Architecture

uRV employs a modified Harvard architecture: code and data reside in a shared 32-bit memory space, but are accessed through separate memory interfaces. Instructions are executed by a four-stage, single-issue pipeline, shown in Figure 5 and consisting of the following stages:

- Fetch (F), calculating the address of the next instruction and requesting it from the memory,
- Decode (D), which also computes the immediate values (sign-extensions and bit reordering), pre-computes the operand values for the Execute 1/Memory (X1/M)

ISBN 978-3-95450-148-9

Copyright © 2015 CC-BY-3.0 and by the respective authors

stage and manages the hazards by inserting empty instructions into the Execute stage.

- Execute 1/Memory (X1/M), which executes most of the instructions, generates memory addresses and issues memory read/write requests.
- Execute 2/Writeback (X2/W), which completes execution of Load, Multiply and Shift instructions and writes the result back to the CPU registers.

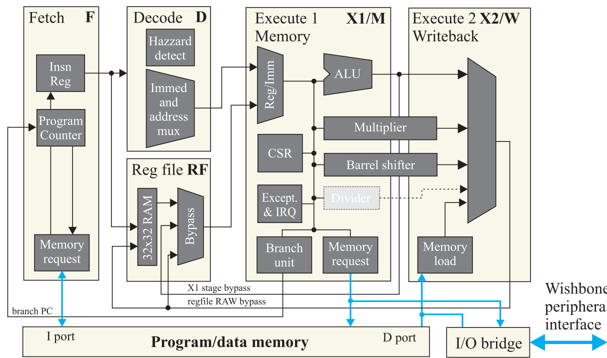

Figure 5: Design of the *uRV* pipeline.

In parallel to the D stage, there is a Register File (RF), hosting the 32 architectural registers (`x0 - x31`). The RF is built on top of two FPGA RAM blocks, providing two read ports and one write port, with a single clock cycle latency.

The core includes a platform-optimized barrel shifter and multiplier (both with 2 cycle latency) and an optional, multicycle division/modulo unit. Most of the instruction results are bypassed to achieve interlock-free execution of dependent instructions, either by the Read-After-Writer (RAW) bypass path in the RF or after the X2/W stage.

uRV supports the RISC-V Control and Status Registers (CSRs), interrupts and a limited set of exceptions, although the interrupt CSR layout has been simplified (with respect to the RISC-V specification) to minimize the occupied FPGA area. Exceptions are used to implement loads and stores at non-aligned addresses, as well as to emulate instructions not supported in hardware (division, modulus, upper 32 bits of multiplication).

Design Trade-offs

In order to achieve a relatively high clock frequency, combined with a reasonable area, we have applied several optimizations:

- The most timing-critical instructions, such as Multiply and Shift are distributed over two pipeline stages (X1/X2).
- Multiply, Shift and Load instructions are not bypassed to minimize the long combinatorial X2/W stage bypass path. Instead, the D stage raises an interlock in case of a RAW hazard. Surprisingly, with the instruction scheduling done by modern optimizing compilers, there are very few situations where the result of the current instruction is immediately needed by the next one.
- Multiplexing and bypassing of the ALU operands is carefully split between D and X1 stages.

Despite these trade-offs, with proper instruction scheduling, *uRV* can execute all instructions except division and jumps in a single clock cycle. Division takes 37 clock cycles, taken jumps have a constant 3-cycle penalty and missed jumps consume one cycle.

Connectivity

The core provides two simple buses for accessing the memory space, with support for memory wait states. These buses can be connected directly to an FPGA RAM block. The I/O peripherals are accessed through a dedicated Wishbone master, which is controlled by a bridge connected to the data memory bus. The internal memory buses can also connect the CPU pipeline to the instruction/data cache (currently under development).

Performance

An example implementation of an *uRV*-based system, incorporating a GPIO port, UART and 64 kilobytes of RAM takes 1210 LUTs, 954 FFs, 34 Block RAMs and 3 DSP cells on a Spartan-6 series FPGA, achieving a clock speed of 100 MHz (toolchain set up to minimize area).

The core successfully passes the official RV32IM test suite as well as the Coremark 1.0 [13] benchmark. The comparison of Coremark scores against other popular 32-bit architectures[1] is presented in Table 1. Despite the low footprint and several design trade-offs, *uRV* combined with a modern GCC tool chain (version 5.2) performs remarkably well.

Table 1: Coremark 1.0 Results Comparison for *uRV* Against Popular 32-bit CPU Cores

Core and platform	Compiler	Score/MHz
uRV (Spartan-6)	GCC 5.2.0 `-O2`	2.14
Nios II/f (Altera FPGA)	GCC 4.9.2 `-O2`	1.87
Cortex-M3 (STM32F103)	GCC 4.4.1 `-O3`	1.80
LM32 (*MT*)	GCC 4.5.3 `-O3`	1.78

Status and Outlook

The *uRV* is an ongoing project. In the nearest future, we are planning to:

- Add a Debugging Unit and JTAG interface,
- Implement caches, enabling access to large external memories through a Wishbone.B4 [4] or AXI4 Lite bus.
- Add floating point support for more demanding controls applications (e.g. power converter control).

[1] Scores of the competing implementations have been taken from the official Coremark result repository [13]. Note that Coremark measures both the performance of the hardware and the efficiency of the C compiler. Different compiler versions may give different scores.

Copyright © 2015 CC-BY-3.0 and by the respective authors

ISBN 978-3-95450-148-9

The core can be already used within the *MT*, and the WR PTP core is being upgraded to use *uRV*. We expect significant savings in the FPGA area due to the lower footprint and the more compact code size of the RISC-V ISA.

SOFTWARE

The *MT* software stack, presented in Figure 6 is made of three components: a Linux driver, a Linux library and a dedicated library for real time development. All these components together make a framework ready to be used for the final application development. This framework helps the developers by hiding all the MT details so they can immediately start coding the user-space and the real-time applications running on the MT soft cores.

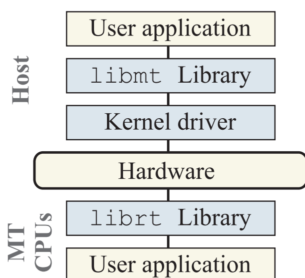

Figure 6: Architecture of *MT* software.

The driver exports all the *MT* features to user space:

- Cores management: load/dump the binary containing the real-time applications to/from each soft core's memory. Pause execution on selected cores, completely enable/disable or reset the core.
- MQ access: send and receive messages to/from the running real time applications. Both synchronous and asynchronous messaging APIs are available.
- SMEM access: access any location on the shared memory in concurrency with the running real time applications.
- Debug interface: read and dump all the debug messages coming from the running real time applications.

Except for the SMEM access, which uses `ioctl()`, all other features are usable directly from the shell through `sysfs` attributes or character devices.

To optimize driver usage the framework provides a library and a Python wrapper, with the same functionality. The library allows the developer to access directly all driver features, listed above. On top of this, developers can build application-specific libraries or directly write their applications. The Python wrapper can be also used for implementing tests or to easily decouple the development of the host application and the real time one.

The real time application library provides a set of tools to ease the communication with the host system. Mainly, it hides from the real time developer all the communication layer with the host system so that in the real time code it is reduced to what is really important for the real time tasks. The library also provides functions to allow the host application to read/write the memory locations explicitly exported by the real time application (variables, SMEM, hardware registers, structures).

A simple request-response communication protocol has been defined for the communication over the HMQ between the host and the real time application. To really take advantage of the full software stack it is suggested to use this protocol, but developers are free to implement alternative solutions.

PROJECT STATUS

The *MT* framework is the base platform for several projects at CERN, such as:

- The Trigger Distribution system for the LHC Beam Instability Diagnostics [8], already operational in the LHC.
- The proof-of-concept RF over Ethernet distribution system [8].
- The WorldFIP Master controller card [14].

MT is supported on the SVEC [15] and SPEC [16] FPGA Mezzanine Card (FMC) carrier boards. The HDL core, software and the documentation can be found at [17]. The *uRV* CPU source code and test software is available at [18].

CONCLUSIONS

Writing C is faster and less error-prone than writing VHDL or Verilog. A whole family of problems can benefit from a cast into software problems while still guaranteeing real time behavior. *Mock Turtle* aims at providing a modular open source solution which can be the seed of a collaborative community effort. We expect this platform to provide a robust basis for the development of distributed hard real time controls and data acquisition systems, reducing risks and development time.

REFERENCES

[1] Xilinx Corporation, Microblaze soft CPU IP Core: http://www.xilinx.com/tools/microblaze.htm

[2] Altera Corporation, Nios II soft CPU IP Core: https://www.altera.com/products/processors/overview.html

[3] J. Serrano, M. Cattin, E. Gousiou, E. van der Bij, T. Włostowski, G. Daniluk, M. Lipiński, "The White Rabbit Project", IBIC2013, Oxford, UK (2013).

[4] Wishbone bus specification, version B.4: http://cdn.opencores.org/downloads/wbspec_b4.pdf

[5] Lattice Corporation, Mico32 soft CPU IP Core: http://www.latticesemi.com/

[6] FESA 3 - the new Front-End software framework at CERN and the FAIR facility. A. Schwinn, S. Matthies, D. Pfeiffer, M. Arruat, L. Fernandez, F. Locci, D. Saavedra, PCAPAC2010, Saskatoon, Canada (2010).

[7] M. Kreider, R. Baer, D. Beck, W. Terpstra, J. Davies, V. Grout, J. Lewis, J. Serrano, T. Włostowski, "Open borders for system-on-a-chip buses: A wire format for connecting large physics controls", Phys. Rev. ST Accel. Beams, vol. 15 (2012).

ISBN 978-3-95450-148-9

Copyright © 2015 CC-BY-3.0 and by the respective authors

[8] T. Włostowski, D. Cobas, F. Vaga, J. Serrano, G. Daniluk, "Trigger and RF Distribution Using White Rabbit", ICALEPCS2015, Melbourne, Australia (2015).

[9] A. Rubini, W. Terpstra, M. Vanga, Self-Describing Bus (SDB) Specification for Logic Cores (version 1.1), http://www.ohwr.org/documents/428

[10] G. Daniluk, "White Rabbit PTP Core: the sub-nanosecond time synchronization over Ethernet", *M.Sc. thesis*, Warsaw University of Technology (2012), http://www.ohwr.org/documents/174

[11] University of Berkeley, The RISC-V project, http://riscv.org

[12] A. Waterman, Y. Lee, D. Patterson, K. Asanovic, RISC-V ISA Specification (version 2.0), http://riscv.org/spec/riscv-spec-v2.0.pdf

[13] Coremark CPU benchmark homepage, https://www.eembc.org/

[14] *MasterFIP* project homepage:' http://www.ohwr.org/projects/masterfip/

[15] Simple VME64x FMC Carrier project homepage: http://www.ohwr.org/projects/svec/

[16] Simple PCI Express FMC Carrier project homepage: http://www.ohwr.org/projects/spec/

[17] *Mock Turtle* project homepage: http://www.ohwr.org/projects/wr-node-core/

[18] *uRV* RISC-V CPU Core: https://github.com/twlostow/urv-core

Copyright © 2015 CC-BY-3.0 and by the respective authors

ISBN 978-3-95450-148-9