The joys of Composition

Henk Muller

Principal Technologist, XMOS

Composition

A very familiar concept in programming and hardware design

- Suppose I have two building blocks `F()` and `G()`
- I can compose them into a building block `F(G())` or `F()+G()`

Composition enables us to build systems that are large and complex

- In software, composition covers functionality only; no time
- In hardware, composition covers both, but easier said than done

This talk covers:

- Multi-core real-time
- The XMOS take on multicore to enable *composition of timing*
- What this means for ease-of-design and time-to-market.

Multi-core for Real-time systems

Embedded real-time designs revolve around plethora of independent activities, often implemented in on separate devices:

- Microcontrollers, FPGAs, DSPs, 555s, or blocks of an SoC
- Multicore has been at the heart of embedded design for decades.

The advantage of using a set of devices

- Allows independent tasks to be implemented on independent units
- Guaranteed no interference; unlike using an RTOS.

Disadvantage of using a set of devices

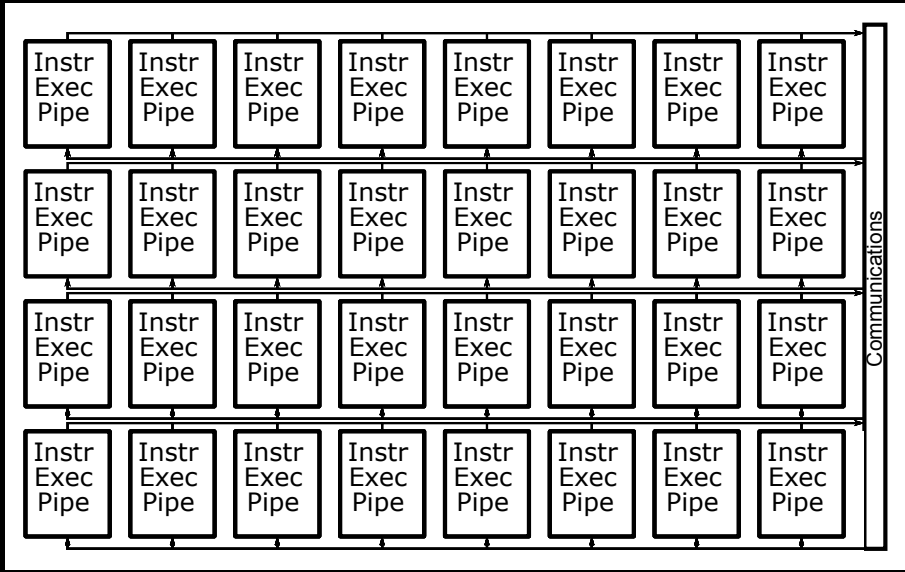
- Lacks communication infrastructure, abuse of I2C, UART, or memory busses
- High integration cost

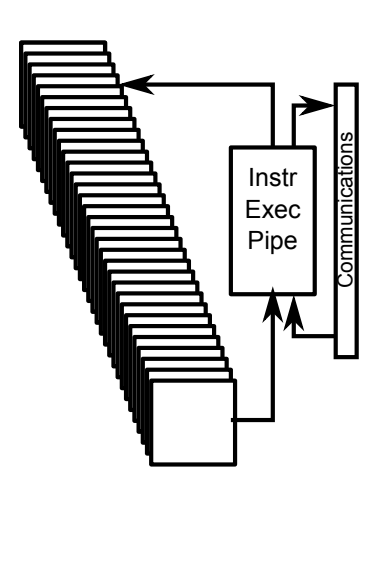
Make the microcontroller multi-core

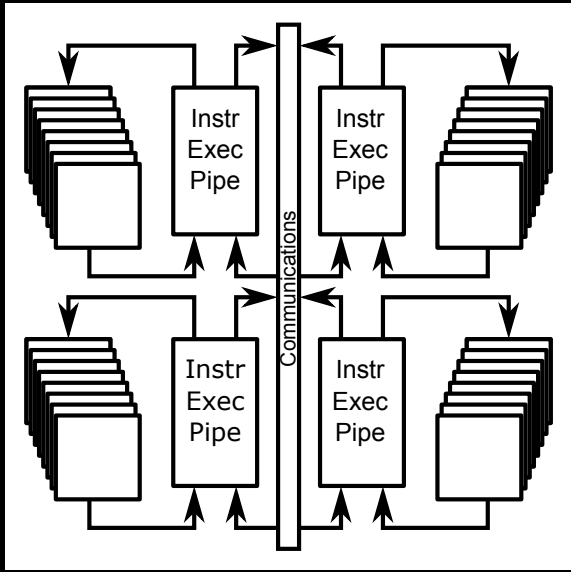
Provide communications (and integration)

Implementation does not require physical multi-core

- Time division multiplexing at instruction level
- Provide an economical number of physical cores, with an economical number of virtual cores on each (we do 1-4 physical cores, supporting 8-32 threads)







Programming model: *Sea of Threads*

Each real-time task is implemented using a thread

- Threads are interleaved by hardware thread scheduler
- Physical cores may be added in order to increase performance

Each thread is a self contained unit, with timing closure.

- Threads are trivially composed - no interference in timing
- If two modules work, then the composition works

You can embed multiple tasks in a single thread - if timing allows it

- We will use events for this

Threads communicate by exchanging data over channels

Composition of tasks in a multi-core

Assume that tasks B computes a result in time T_B, an task C computes a results in time T_C, then:

- $B;C;$ takes time $T_B + T_C$, and
- $par\{B;C;\}$ takes time $max(T_B, T_C)$

For this to work, we need to be careful, in particular avoid:

- Operating system schedulers
- Virtual memory
- Caches
- Locks
- Shared busses
- Interrupts

XCore timing composition

All resources are shared inside a physical core

- Each thread (logical core) can read or write to memory in turn
- All IO resources (ports, timers, comms) are also accessed in turn
- There is no contention.

The scheduler is implemented in hardware

- Scheduler works on a clock-cycle level
- Threads are woken up in a single clock cycle
- Threads are put to sleep in a single clock cycle
- N threads on an M MHz core are guaranteed $\frac{M}{N}$ MIPS.
- New threads threads can be started in a few instructions.

Interrupts are not used to program; only for exceptional situations

Worst case execution time

Time composition enables a tight WCET estimation

- WCET only depends on the software structure
- Software may take a data dependent branch
- Software may execute a data dependent loop

No engineering margins to deal with avoidable unpredictability

- Enables a tight design
- Minimising the resource required to complete a task

As the computations are worst case, software may be early

- Wait for the real-time to catch up before committing to time-sensitive IO

Composition of events inside a task

It is very rare for a task to have a single predictable thread of control

- Tasks respond to events outside the machine, or in other tasks
- For example, a request to change the control or a status request.

Given that we can compute a tight WCET for an event

- Compute worst case response times for sequences of events
- Compare with application requirements
- Take a compile-time decision whether composition of events will work

XCore and events

Resources (Comms, IO ports, timers) support events natively:

- Each resource has an event vector
- Resources events can be enabled individually
- Thread can be told to wait for next event; stop the thread and
  - Vector to the appropriate code when the resource is ready

Like interrupts, but

- In a compile-time known location
- With a known set of register values
- No need to save/restore state, respond within clock cycles.
- User and tools can reason how segments of code are interleaved.

Also enables us to cooperatively executes two tasks in a single thread; *whilst maintaining timing closure*

Composable thread structure

Typically a thread comprises the following steps:

1. Select a piece of work to do
2. Complete said piece of work
3. Optionally wait for the right time for IO
4. Optionally perform IO

The hard part of real time programming has been contained in step 4

- Programmer must argue that IO will not unduly block the task
- Specify contracts on the edge of tasks, non blocking nature should be a consequence of contracts agreeing

Step 2 only has to be *fast enough*, Steps 1 and 3 are trivial

Case study: Audio

I took two existing designs:

- Audio-2.0 over USB, comprising 16 tasks
- Audio over Ethernet (AVB), comprising 20 tasks

I reused those to create an USB-Audio-2.0 to AVB bridge

- Removed half of the USB design (no Audio CODECs)
- Removed half of the AVB design (no Audio CODECs)
- Stuck a buffer task in the middle

Design

The time to prototype this was three hours

- An hour: Study AVB code, draw thread diagrams
- An hour: Edit source code
- An hour: Making it compile, and remove trivial errors

Admittedly, I was familiar with the USB software ...

- ... but I had never seen the AVB-Ethernet software.

Of course one needs more than three hours to make it production ready...

- ... but it is good to have a prototype out quickly

Conclusions

Composition is fundamental to designing systems

- Has to incorporate timing
- Architecture has to support this

Composition of threads and events means

- Fast design of new systems
- Reuse and adaptation of existing components.