Task Partitioning and Placement in Multicore Microcontrollers

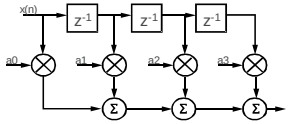
David Lacey
24th September 2012

Multicore Microcontrollers

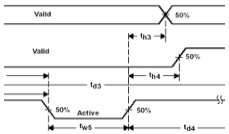
- Many embedded systems are now multicore
- Maybe lots of cores:
  - XMOS devices have 8, 16, 32 “logical cores”
- Programming techniques are varied…
- … but the application level concerns are similar
- This talk looks at one aspect: task partitioning and placement
- In other words the top level design of a parallel program

XMOS Multicore Microcontrollers

Signal processing

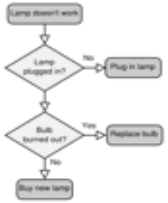


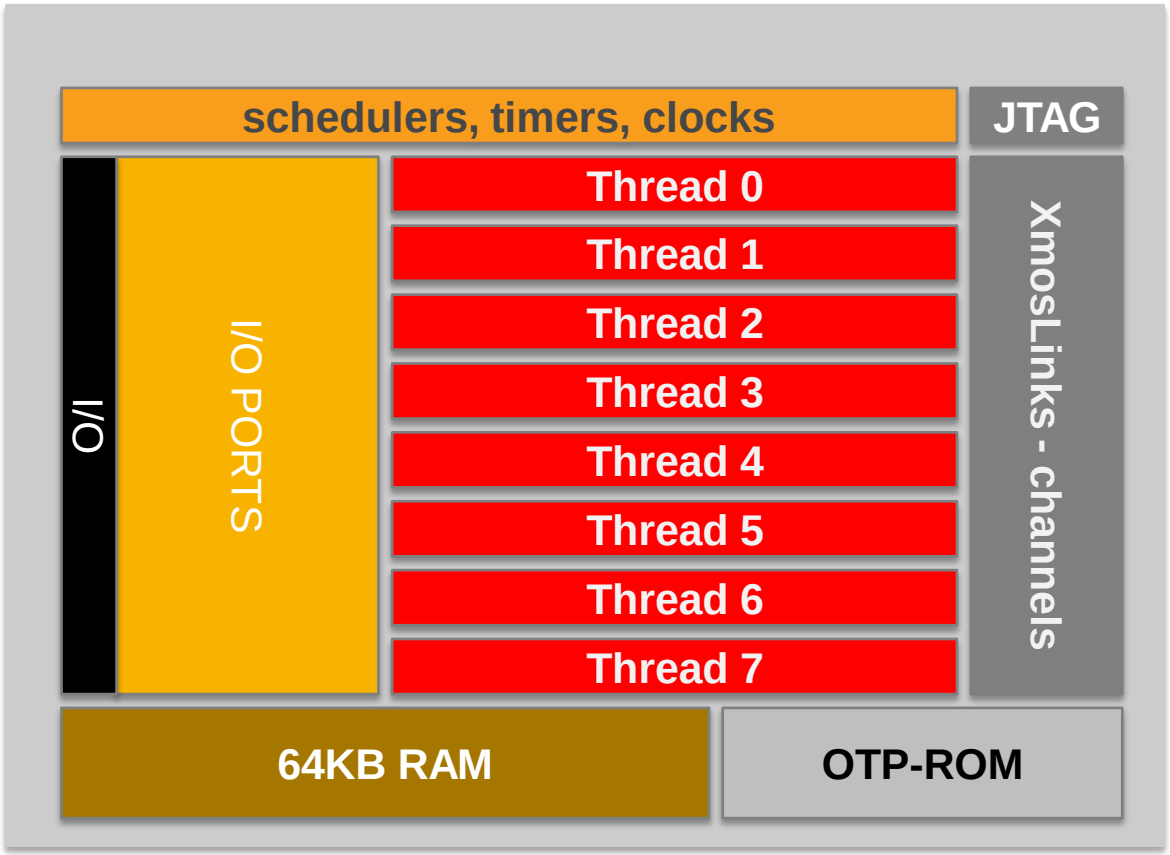
Hardware interfaces



XMOS devices

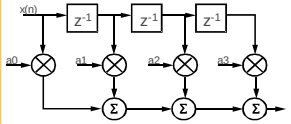
Software tasks



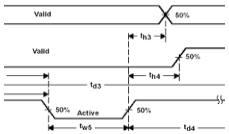


The Task Partitioning/Placement Problem

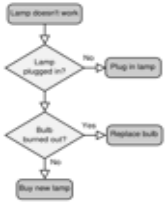
Signal processing



Hardware interfaces

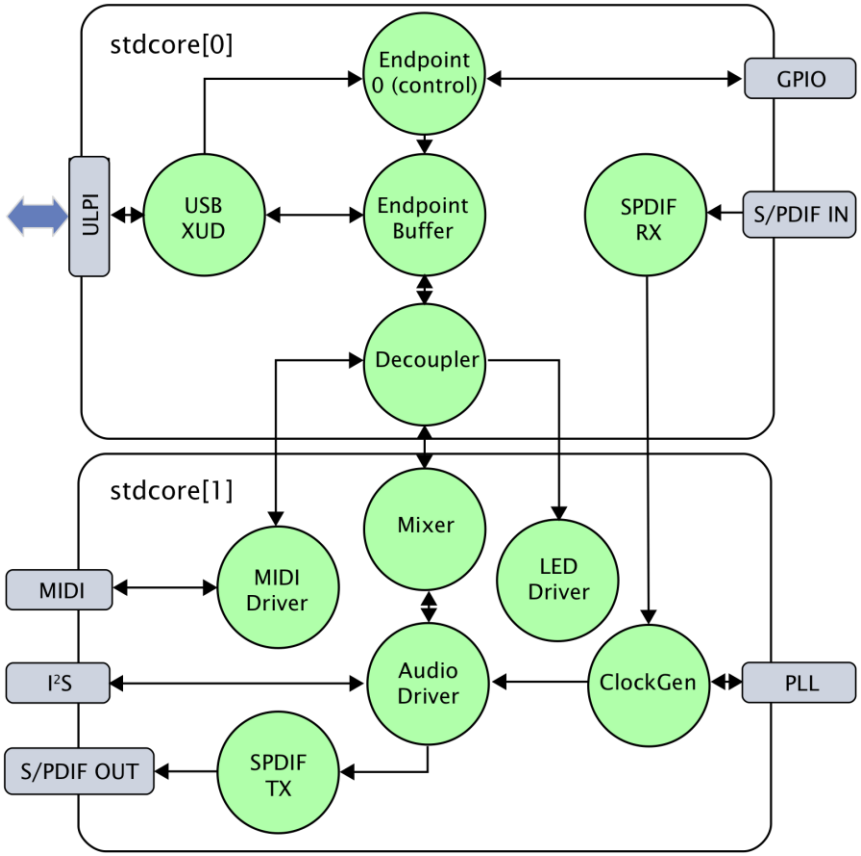


Software tasks



- Thread 0
- Thread 1
- Thread 2
- Thread 3
- Thread 4
- Thread 5
- Thread 6
- Thread 7

Example: USB Audio Application



Partitioning vs. Placement

- Partitioning
  - How do I split up my problem into different tasks?
- Placement
  - Where to I put my tasks?

Partitioning != Parallelization

- Splitting a system up into tasks is not necessarily the same as parallelizing a sequential task
- Often in embedded systems there are plenty of naturally concurrent tasks

Task Partitioning/Placement: Key Factors

- Requirements
  - Resource Requirements
  - Communication Latency
  - Communication Bandwidth
  - Environment Response Behaviour
  - Communication Response Behaviour
- Implementation concerns
  - Blocking/synchronization behaviour
  - Buffering
  - Communication Mechanisms
  - Task sharing

Requirements

Resource Requirements

- Does a particular task need to run on a processor with specific resources:
  - Memory
  - I/O
  - Processing capability

Communication Latency and Bandwidth

- Bandwidth is the easier one to handle
  - Still need to be careful about protocol overhead
- Latency is doubly hard:
  - Working out what latency you task requires
  - Working out what the latency is between two processors

Response requirements

- Environment response
  - How quickly does a task need to respond to a particular I/O event
- Communication Response
  - How quickly does a task need to respond to an event from another task

Implementation

Blocking/Synchronization Behaviour

- Who waits for whom?
- Who initiates transactions between tasks?
  - Who is the master? who is the slave?
- Sometimes roles can change dynamically
- Notification protocol is useful:
  - Current slave task posts a notification in a buffer
  - Master tasks notices notification and initiates transaction

Buffering

- Buffering serves three purposes:
  - Provides a range of data for a specific purpose (e.g. ffts)
  - Copes with short term changes in data rates (i.e. copes with bursty traffic)
  - Decouples response time to processing time
- Downsides:
  - Buffering adds latency
  - Buffering needs more memory

Communication Mechanisms

- Once blocking behaviour, synchronization needs, response requirements and buffering requirements are known….
- We can choose the right tool for the job:
  - Message passing/shared memory
  - Which protocol to use

Problems

- How do you express task placement?
- How do you express communication?
- How do you ensure response requirements are met?
  - Proper protocols help but still need worst case timing analysis
- How do you manage/debug multicore programs?

XMOS solutions

- Naturally concurrent multicore architecture
- XC – C language extensions for parallelism and communication
- XTA – static timing analysis tool
- Multicore debug and tracing

Conclusion

- Multicore programming starts with the application
  - Task placement/partitioning is key
- Implementation decisions follow from the requirements (as always)
- Particularly easy on XMOS devices ;)