Operating systems for many-core

the path from time sharing to space sharing

Raphael 'kena' Poss

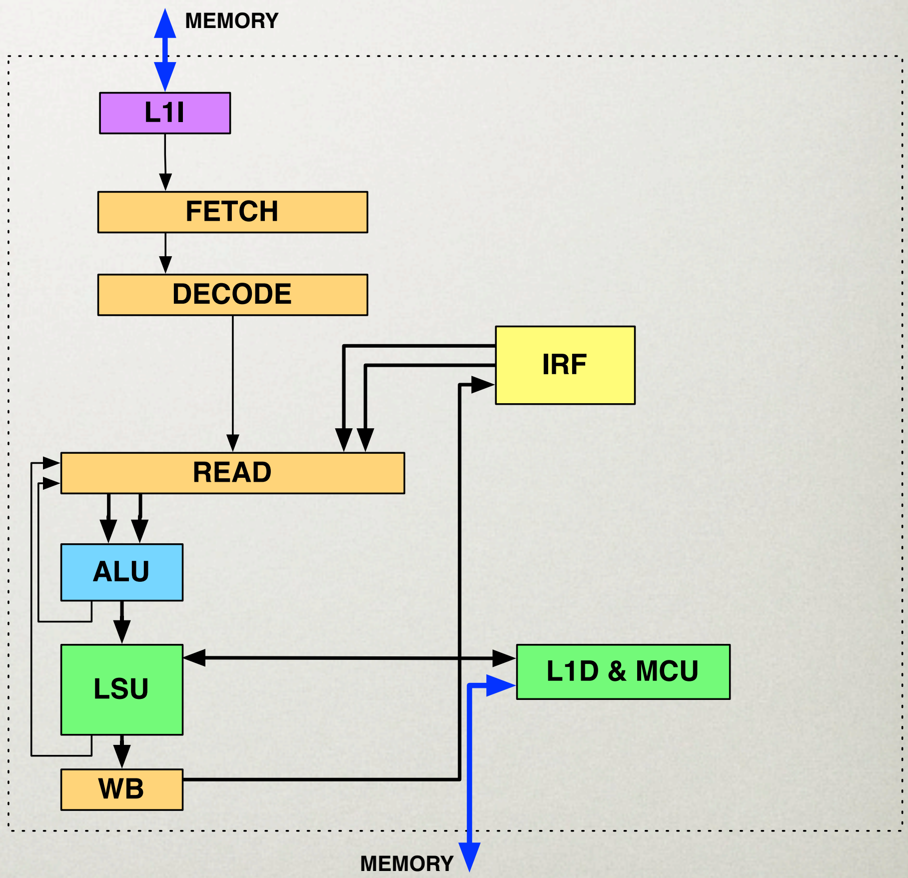
r.c.poss@uva.nl

www.apple-core.info

University of Amsterdam

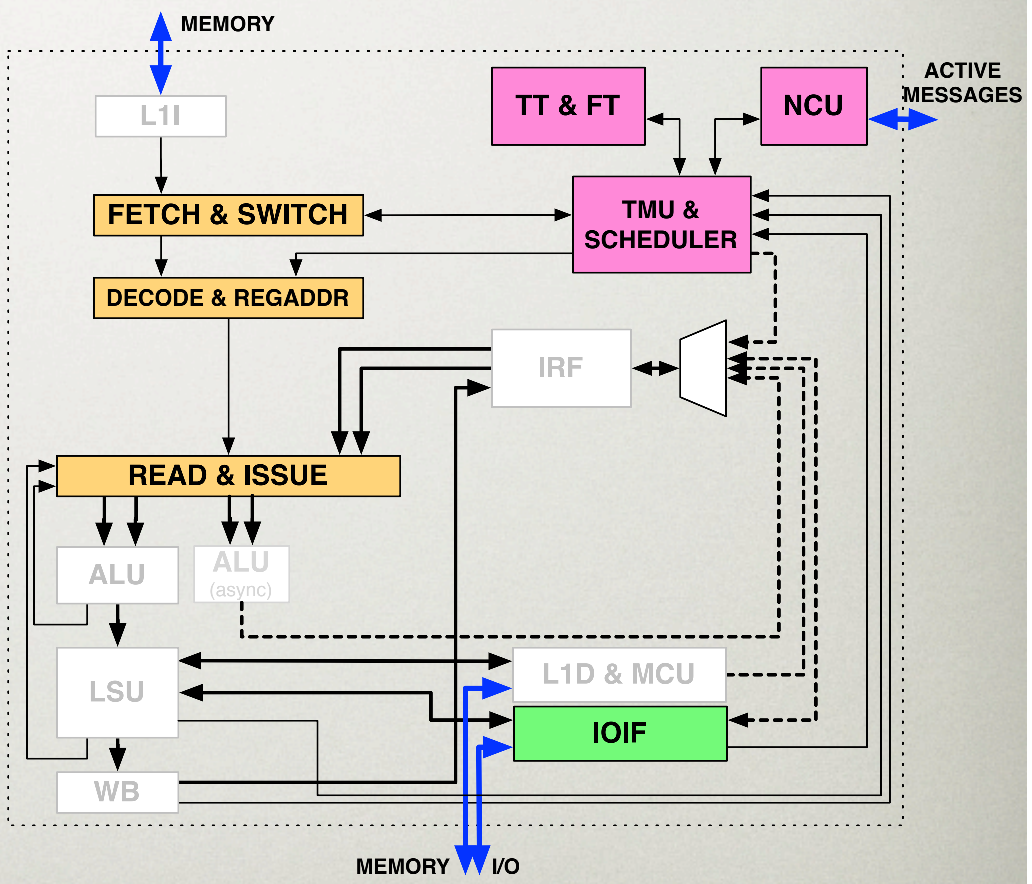
The D-RISC core
Operating system in hardware

Based on a "simple"
RISC pipeline...



The D-RISC core
Operating system in hardware

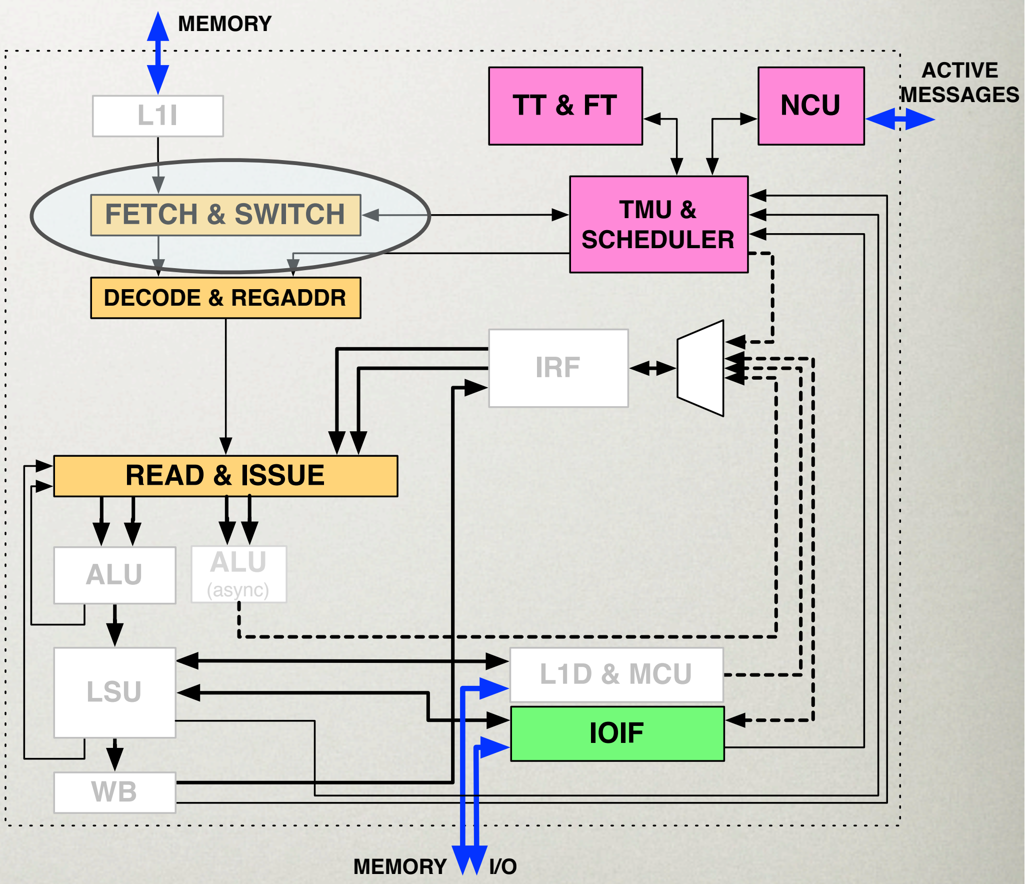
A couple of extensions...



The D-RISC core
Operating system in hardware

Fetch from FIFO of active threads

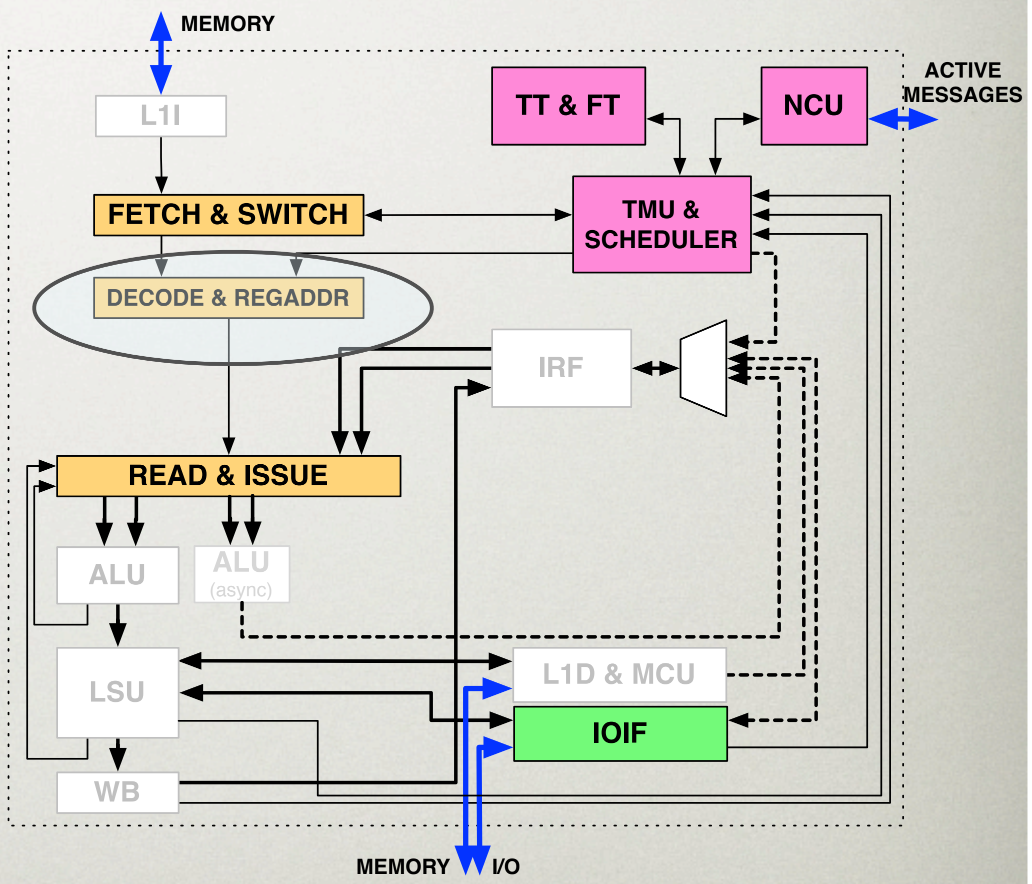
Switch based on instruction annotations



The D-RISC core
Operating system in hardware

Each thread has its own register window of configurable size

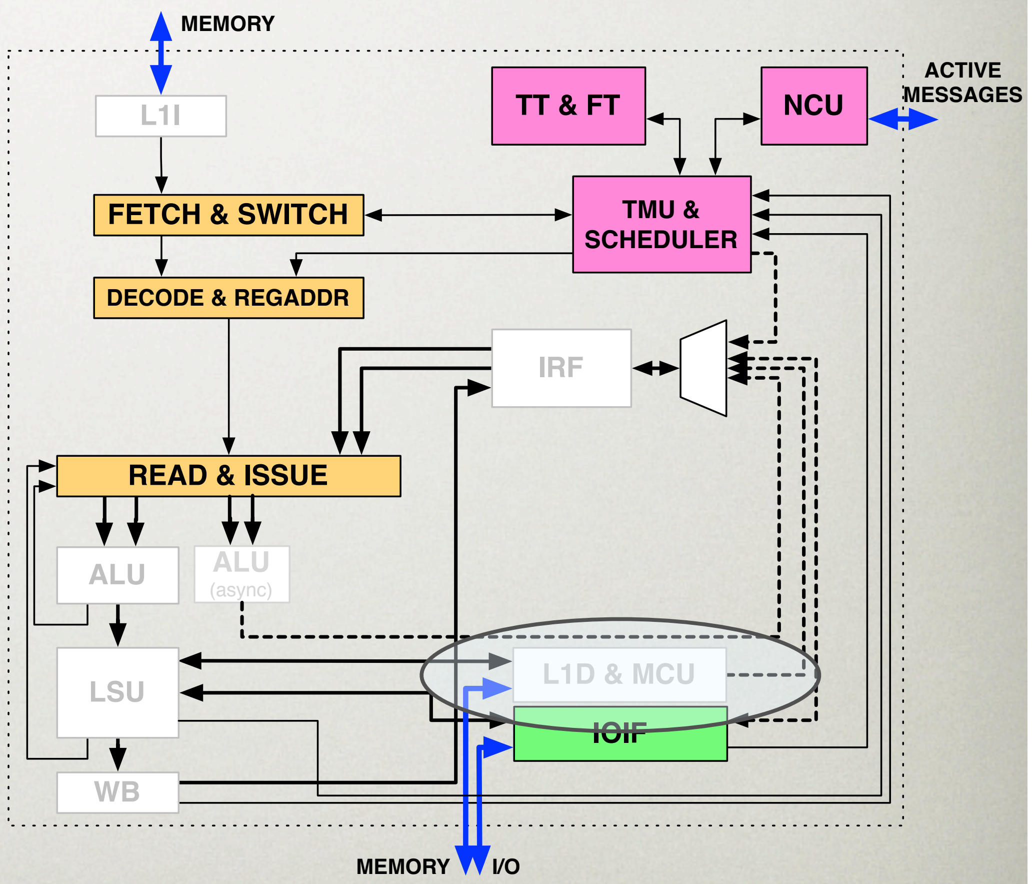
Base address in IRF computed during decode



The D-RISC core
Operating system in hardware

Upon L1 miss the thread is allowed to go through

a "waiting" state is written to the output register, only further instructions dependent on it will suspend



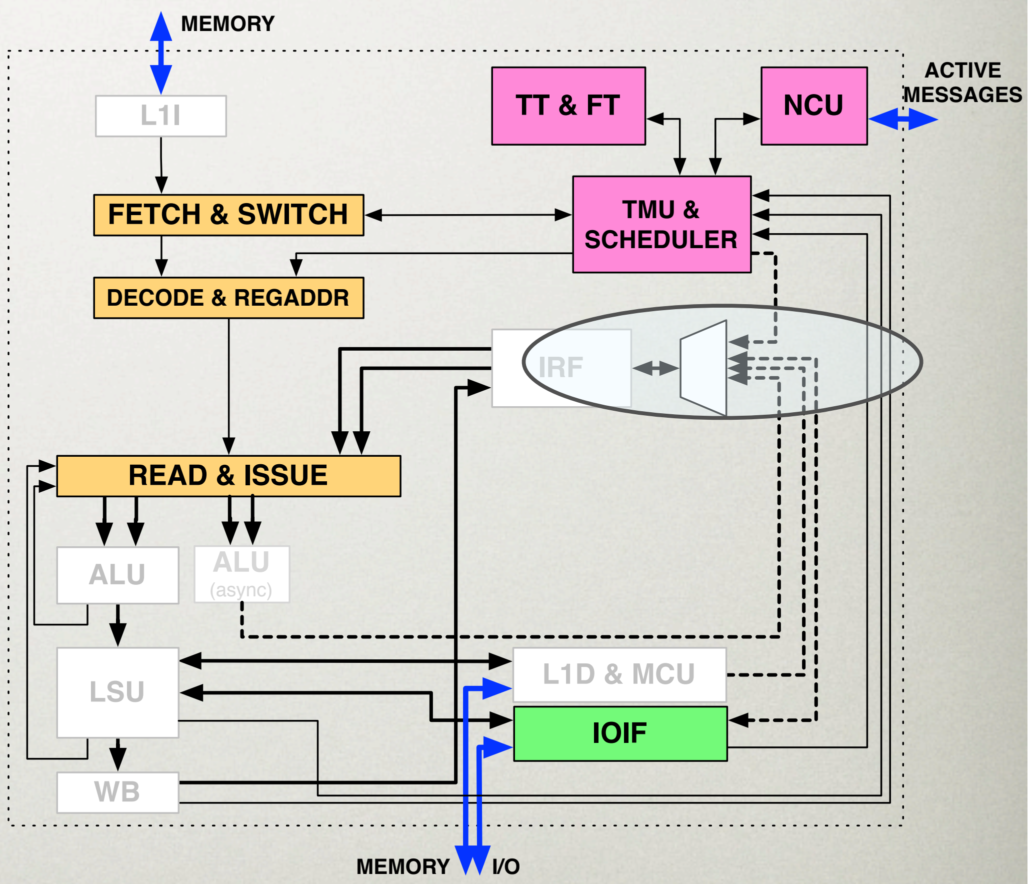
The D-RISC core
Operating system in hardware

Upon completion of a long-latency operation

(multi-cycle ALU, memory load)

the value is written back asynchronously to the RF

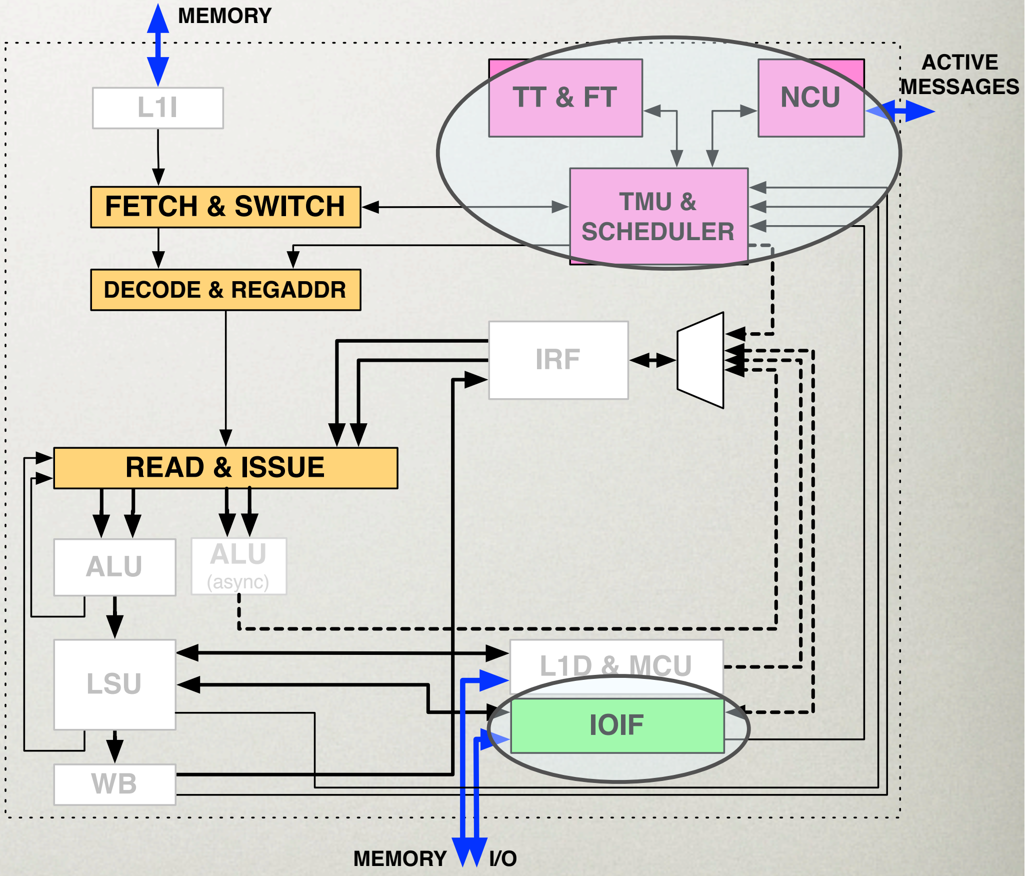
this wakes up any waiting thread(s) by bringing them back on the active FIFO



The D-RISC core
Operating system in hardware

The Thread Management Unit handles thread creation, synchronization, and provides thread metadata to the pipeline

Can be controlled from the pipeline (ISA extensions), control NoC messages or other I/O events



Apple-CORE
a perspective shift

Core i7 Linux	**Thread creation** (pre-allocated stack) **>10000 cycles in pipeline**	**Context switch** syscalls, thread switch, trap, interrupt **>10000 cycles in pipeline**	**Thread cleanup** **>10000 cycles in pipeline**
D-RISC with TMU in hardware	**Family creation** (metadata allocation for N threads) **~15 cycles** **(7c sync, ~8c async)** **Thread creation 1 cycle, async**	**Context switch** at every waiting instruction, also I/O events **<1 cycles**	**Thread cleanup 1 cycle, async** **Family cleanup 2 cycles, async**

Apple-CORE
a perspective shift

Core i7	**Function call** with 4 registers spilled **30-100 cycles**	**Predictable loop** requires branch predictor + cache prefetching to maximize pipeline utilization 1+ cycles / iteration overhead
D-RISC with TMU in hardware	**Family creation** of 1 thread, 31 "fresh" registers **~15 cycles**	**Thread family** 1 thread / "iteration" reuses common TMU and pipeline no BP nor prefetch needed 0+ iteration overhead

Apple-CORE
a perspective shift

	Unit of programming
In a mostly sequential CPU	The **sub-sequence,** defined by structured **blocks** **Repetition** defined by loop **control expression** and **statement body**
In a multithreaded, many-core chip	The **microthread** defined by **thread programs** **Repetition** defined by Family **logical index space** and **thread program**

Apple-CORE
a perspective shift

	Synchronization style
In a mostly sequential CPU	**Implicit** One mechanism: **sequential ordering**
In a multithreaded, many-core chip	**Semi-explicit** **sequential** within threads **dataflow** within families **bulk synchronization** across families

Apple-CORE
a perspective shift

	Basic mechanism for Program composition
In a mostly sequential CPU	**Control flow transfer** to/from **named entry point** Defines **singular** functions and composition with **call/return** protocol **Traps** and **interrupts** for async. events = implicit call/return
In a multithreaded, many-core chip	**Family creation** and **family synchronization** of multiple threads running from the same **named entry point** Defines **multi-way** functions and composition with **create/sync** protocol Optional: execution **placement** "at" a **named cluster of cores** Asynchronous families for async. events = implicit create/sync

Evolution or revolution?

- Looks like a *revolution*:
  - **Can’t reuse existing OS** code as-is
  - **Can’t reuse existing low-level techniques** based on preemption and software schedulers
    eg. signals, interrupt handlers, “callbacks”
  - Must use **ISA concurrency in code generation** to exploit; requires **language extensions** and shakes **compiler assumptions**
- Can we really afford this?

An evolution, really (1)

- Low-level machine code generation:
  - **Lift loop bodies as separate functions**
    - reuses techniques from GPU/accelerator world
  - **A thread is really a virtual processor**
    – threading is well-know in compilers already
  - **Higher-level compilers** can generate threaded low-level code from "productivity" languages

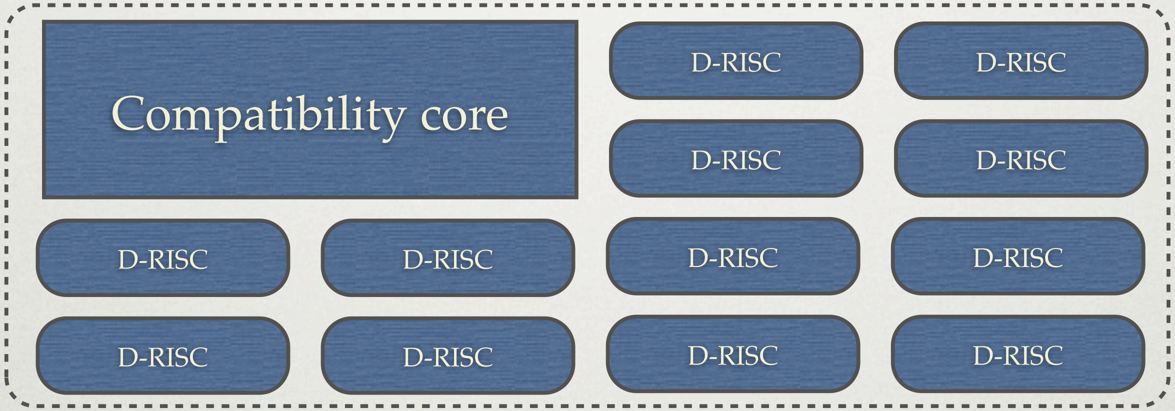
- Really a **convergence** of mature technology

An evolution, really (2)

- Managing **asynchrony of “external” events**
  – I/O, traps, remote “syscalls”
  - **An event handler is really a thread**
    – reuse as is, just entry/exit is different
  - Requires **mutual exclusion of shared state**
    – already accepted in OS/library design
- The **benefit of extra bandwidth and lower latency** will justify the req. adaptation, if any

An evolution, really (3)

- Sometimes **legacy OS and library code** cannot be adapted – typically device drivers, proprietary interfaces



- Solution: integrate a "compatibility" core on the same chip using **same NoC protocol for concurrency management**, then **delegate syscalls** behind library entry points
- With same ISA and shared memory **APIs can be kept as-is**
- **The "accelerator pattern", transposed!** – similar to Cray XMT, on chip

The “real” issues uncovered in Apple-CORE

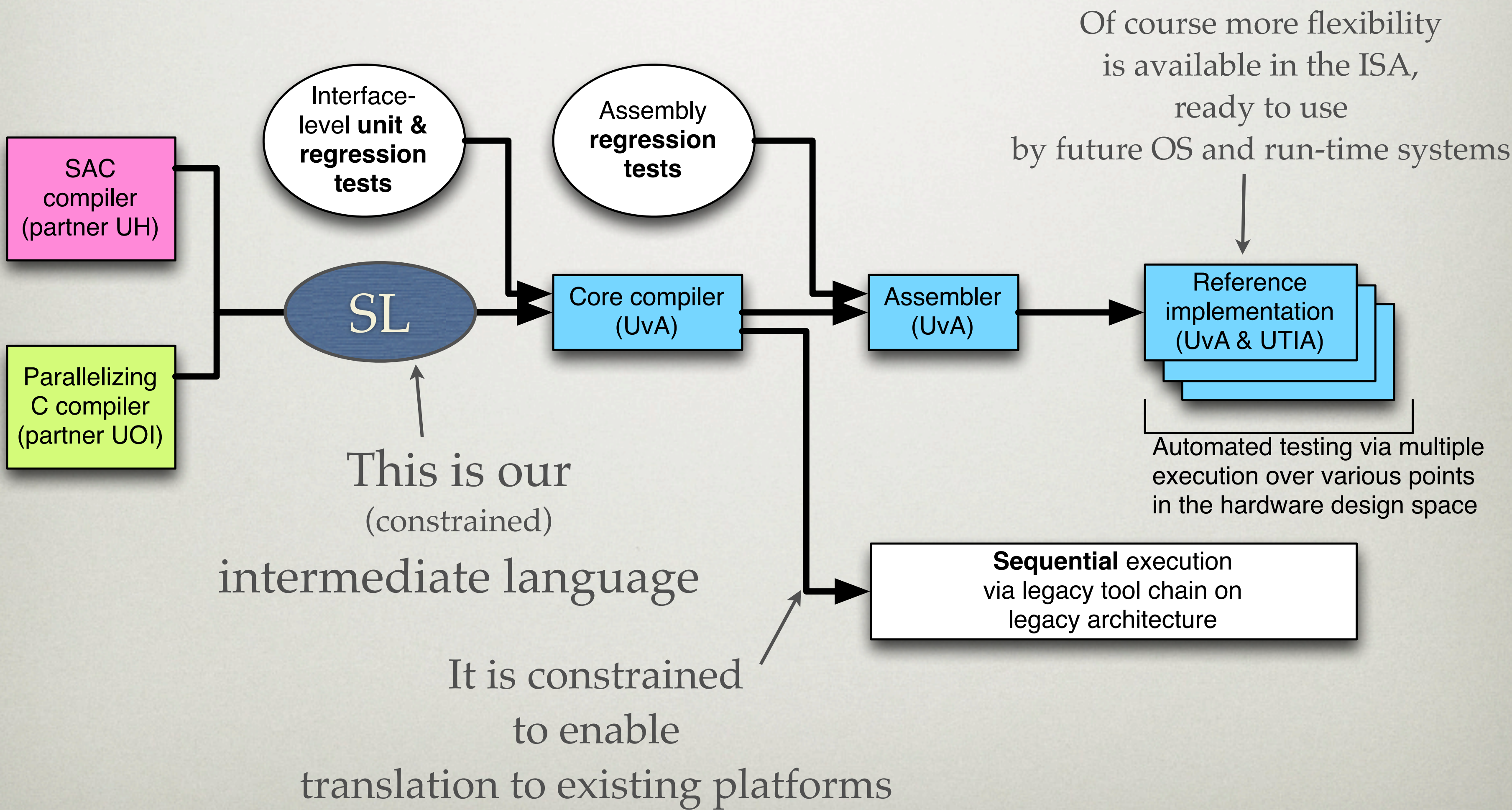
- **Validation**: how to detect detect errors, then compare with existing systems
  – need reference / base lines
- **Security**: isolation and observation
  – and how to reduce interference with performance
- **Resource management**:
  cores, but also **memory** and NoC **channels**
  – how to reduce management overheads

- NB: these issues are general to many-core, but they are exarcerbated in Apple-CORE

Validation

- Solution:
  1. Choose a **subset of the ISA** that can be emulated in legacy platforms
  2. Design the intermediate language SL to use only this subset to **constrain programs**
  3. Implement **compilation to both** the new platform and legacy systems and perform **comparative testing**
- This subset resembles **fork/join with families** and **forward-only dataflow synchronization**
- It is **deadlock-free**, mostly **deterministic** and **can be serialized** (cf Cilk, Chapel)

Validation



Of course more flexibility
is available in the ISA,
ready to use
by future OS and run-time systems
SAC compiler (partner UH)
Parallelizing C compiler (partner UOI)
Interface-level **unit & regression tests**
Assembly **regression tests**
SL
Core compiler (UvA)
Assembler (UvA)
Reference implementation (UvA & UTIA)
Automated testing via multiple execution over various points in the hardware design space
Sequential execution via legacy tool chain on legacy architecture
This is our
(constrained)
intermediate language
It is constrained
to enable
translation to existing platforms

Security

- Isolation via virtual address spaces is expensive
- Instead Apple-CORE has fine-grain **capabilities** on memory and concurrency management
  Concept imported from microkernels: Opal, Mungi, etc.
  VT still available at the level of core cluster or entire chip
- **Observation** via **out-of-band hw monitors**
  Can observe performance **remotely**
  including TMU – active/suspended threads, etc.
  "Out of band" means in parallel to pipeline execution, also potentially **separate from memory** on different NoC

Resource management

- At the **finest grain**:
  provide TLS to threads created by TMU
  Solution: **pre-allocate** and **partition statically**
- **Algorithms:** distributed memory allocator, garbage collection using reference counting
- **Application components**:
  OS allocates and deallocates cores, memory and network links for top-level family entry points
  – this is called **SEP** and is **distributed**
- Either **explicit allocation** in programs (Apple-CORE)
  Or annotated static requirements, aggregated at run-time by RTS/OS (upcoming project ADVANCE)

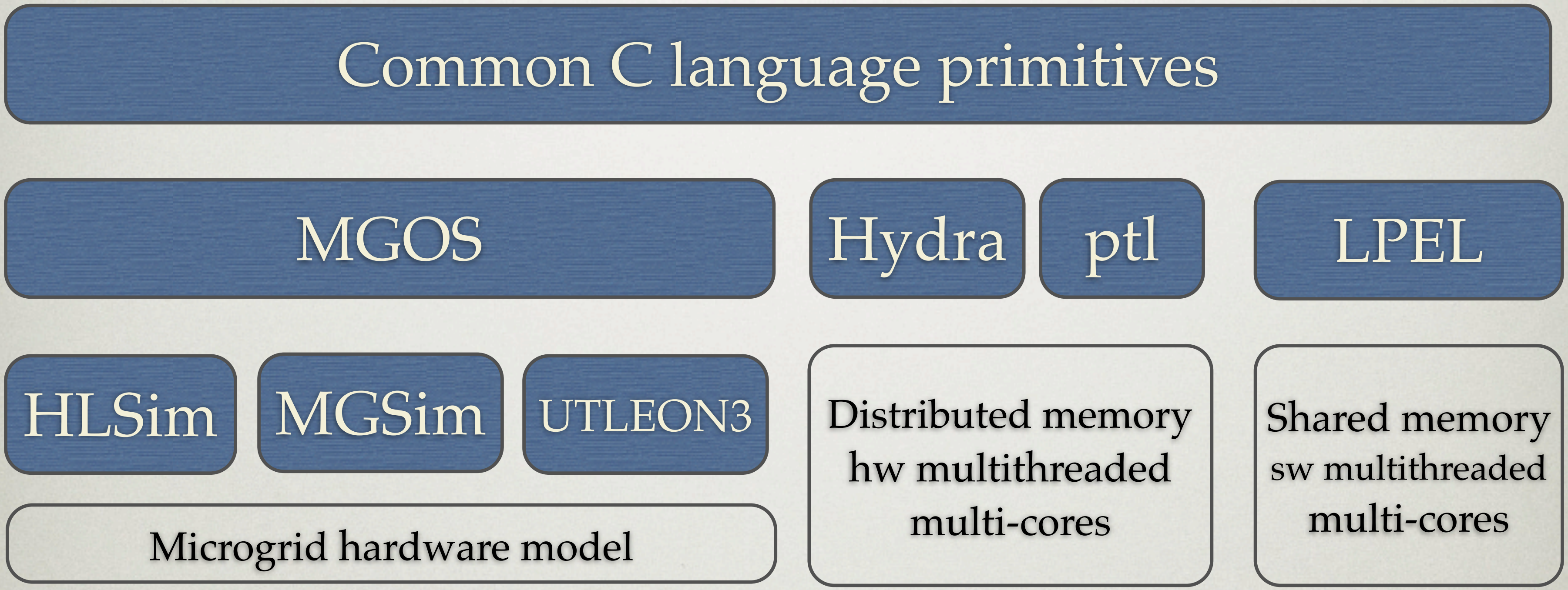
The System Virtualization Platform

- Apple-CORE: was bottom-up from DRISC
- Main outcome: common **concurrency management protocol** on chip
- Can be generalized to other many-core chips: define **placed family creation** as basic primitive instead of function calls, message passing and IPIs
- This defines **a new form of processor virtualization** where **concurrency is the norm**
- Apple-CORE makes this most efficient (in hw) but the protocol **can be used with other platforms**

Foreground produced

- Technology:
  - various **simulators** for a many-core chip with hardware concurrency management (Microgrid)
  - MGOS: **OS and library components** to drive the hardware architecture, including **resource allocators** and **API bridges**
  - **compilation tools** to/from the SVP intermediate language
  - **software run-time systems**
    for commodity multi-cores using SVP semantics
  - **Tests** and **benchmarks** to validate and evaluate
    fine-grained concurrency management
- + Accompanying documentation & know-how

Foreground produced



Summary

- A true **perspective shift** for the basic OS/compiler abstractions:
  - from *sequence* to *concurrency*
  - from *loops* to *microthreads*
  - from *function calls* to *family creation*
  - new focus on **placement** and locality
- Revolution in hardware, yet only an evolution in software
- Middle ground: a common set of primitives
  = **a concurrency management protocol** on chip
- This is generalized from D-RISC towards portable SVP

Thank you!

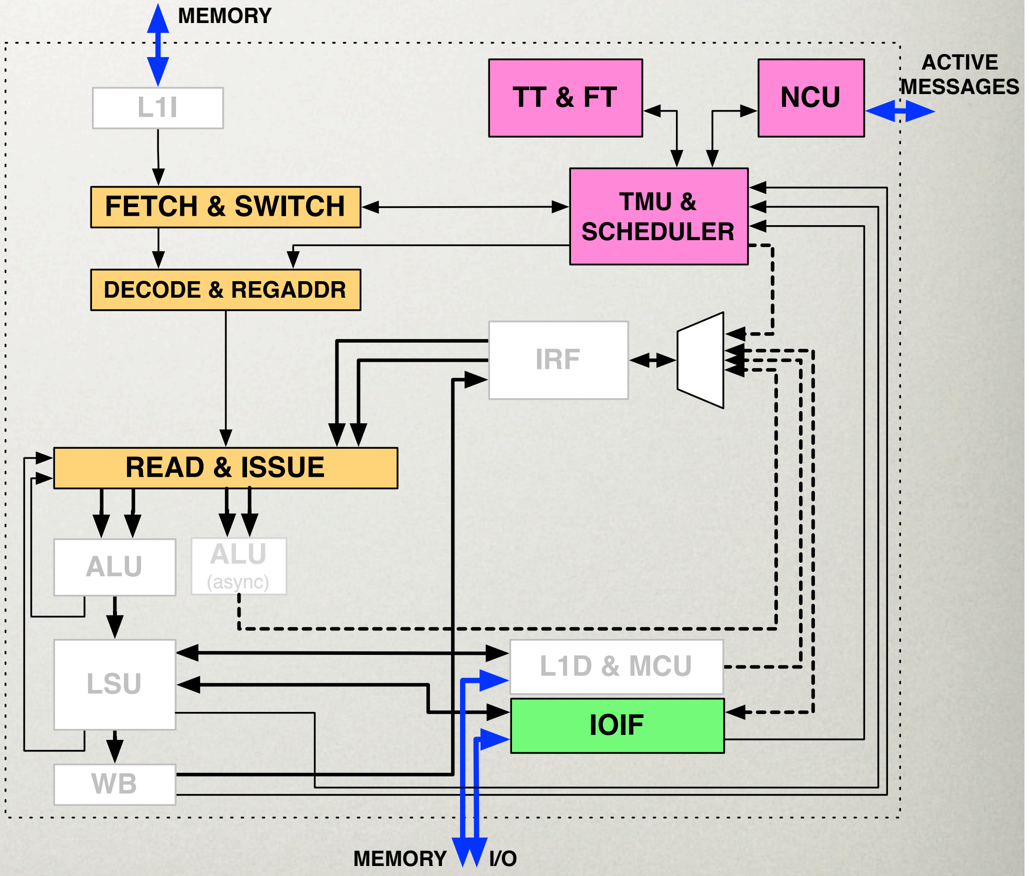
- More information:
  - http://www.apple-core.info/
  - http://www.svp-home.org/

Concurrency management protocol

allocate $\$Place \to \$F$	Allocate a family context
setstart/setlimit/ setstep/setblock $\$F, \$V \to \emptyset$	Prepare family creation
create $\$F, \$PC \to \$ack$	Start bulk creation of threads
rput $\$F, R, \$V \to \emptyset$ rget $\$F, R \to \$V$	Read/write dataflow channels remotely
sync $\$F \to \$ack$	Bulk synchronize on termination
release $\$F \to \emptyset$	De-allocate a family context

The D-RISC core
Operating system in hardware

- Reference parameters:
  - 1GHz clock
  - 1Kregs IRF
    + 1Kregs FRF
  - 256 thread contexts
    32 family contexts
  - 64-bit
    Alpha-like ISA
  - Asynchronous FPU,
    2 cores / FPU



The D-RISC core
Operating system in hardware

- On FPGA (UTLEON3)
  - 20MHz clock
  - 512-1024 registers
  - 64-128 thread contexts
    16-32 family contexts
  - 32-bit
    SPARC-like ISA