Inkel Pentwice a.k.a. ARM Killer MP

Multiprocessor Architecture

Adrià Aguilà[1] Marc Marí[1] Antoni Navarro[1] Kevin Sala[1]

[1]Universitat Politècnica de Catalunya

June 2017

Contents

Introduction

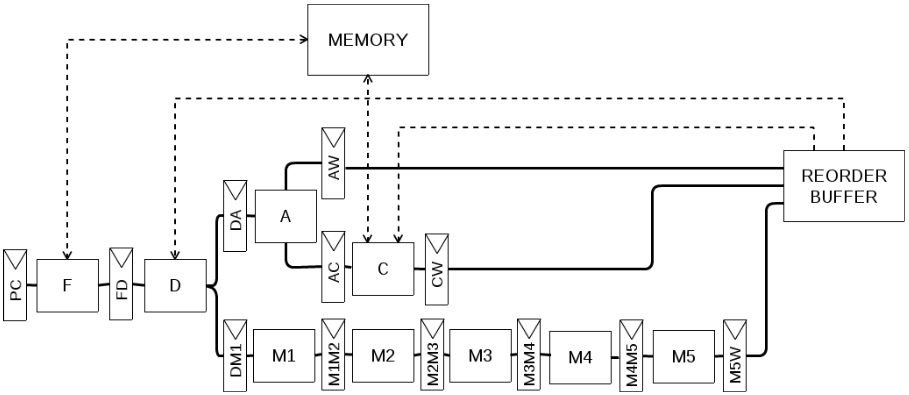
- Conversion from a single core processor to a dual core
- Based on Inkel Pentiun's design
- Written in VHDL

Blocking is for cowards, but it always works

State of the Art: Inkel Pentiun

- Pipelines and supported instructions:
  - Arithmetic pipeline:
    - `ADD r1, r2, r3`
    - `SUB r1, r2, r3`
    - `MOV r1, r2`
    - `LI r1, imm`
    - `BEQ r1, r2, label`
    - `BNE r1, r2, label`
    - `JMP label`
  - Multiplication pipeline:
    - `MUL r1, r2, r3`
  - Memory pipeline:
    - `LDW r1, imm(r2)`
    - `STW r1, imm(r2)`
- 32 user-level registers
- Important features:
  - Instruction Cache
  - Data Cache
  - Store Buffer
  - Reorder Buffer

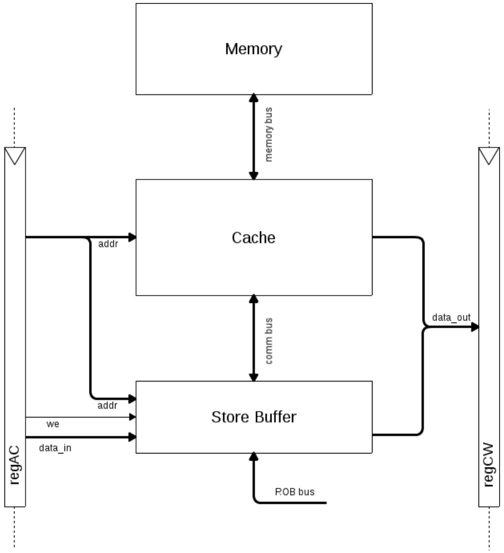
State of the Art: Inkel Pentiun



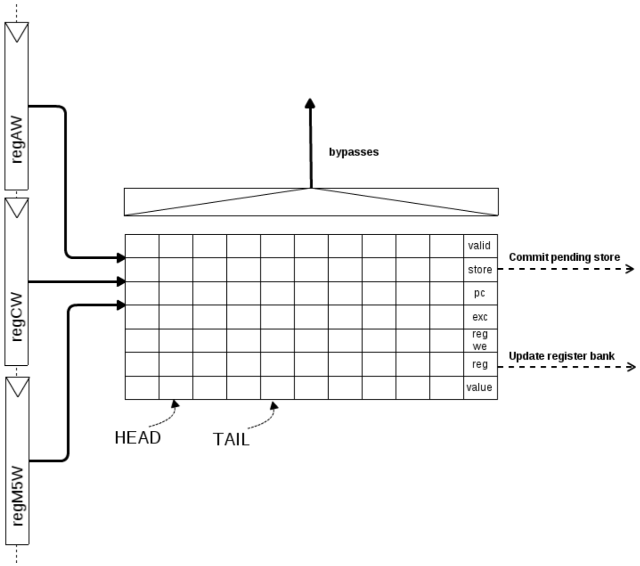
Memory Hierarchy

- L1 Instruction Cache
  - Direct mapping
  - 4 lines of 16B
  - 64B total size
- L1 Data Cache
  - Full associative mapping
  - 4 lines of 16B
  - 64B total size
- Memory
  - 256KB of storage
  - 16B per line (4 words)
  - Each operation takes 5 cycles
  - User code in address 0x1000
- Caches are directly connected to the memory (without L2 cache)

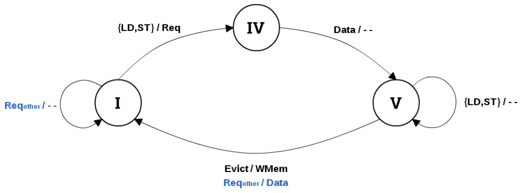
Cache Stage

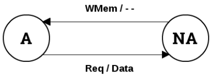

Memory
memory bus
Cache
addr
comm bus
data_out
addr
Store Buffer
we
data_in
ROB bus
regAC
regCW

Reorder Buffer



VI Coherence Protocol for L1 Caches





- VI protocol
- Delayed writes
- No modified state, always invalidate
- Memory keeps track of the status of each block

Store Buffer conflict with the Coherence Protocol

- While observing an invalidation request of a cache line, the store buffer could have a pending store to that line
- Solution: Do not reply the request with the data until the pending store is committed by the Reorder Buffer

Blocking is for cowards, but it always works

Memory Controller

Since we were unsure on how the cache-memory system would look like at the end, we had to decouple the idea of Memory from the managing of block states.

- Decided to add a memory controller.
- If coherence protocols must be changed, the memory won't suffer any change, only its controller.
- Requests from L1s (and later on the LLC) sent to the Controller.
- Two states: Either a block has available data or not (modified in any Cache)

Atomic Bus & Arbiter

Atomic Bus

- Adding an atomic bus was a requirement prior to going Multicore
- Being plugged into the bus allows observing transactions
- However... someone has to manage accesses into the bus

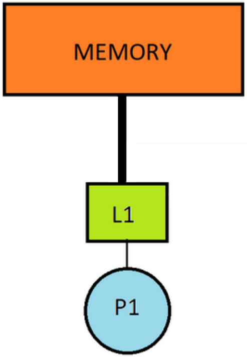
Arbiter

- Fully controls the bus
- Accepts requests and gives access under a certain policy
- We tried two policies, we finally sticked to the first:
  - **Fairness:** Based in a kind of "Least Recently Worked"
  - **Random:** Introduces variability by giving access to a random requester

New instructions

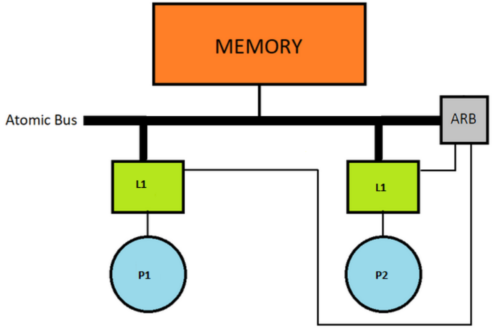
- Test and set (memory operation): `tsl rd, imm(rs)`
- Get processor ID (ALU operation): `pid rd`

Going Multicore (I)



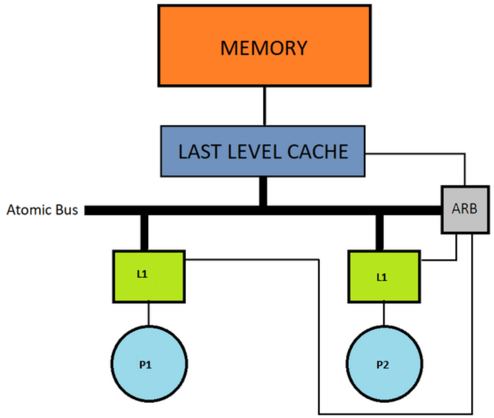
- Single core
- Direct bus between L1 and memory

Going Multicore (II)



- Add arbiter to the bus
- Make bus atomic
- Add observer process
- Double it!

Going Multicore (III)



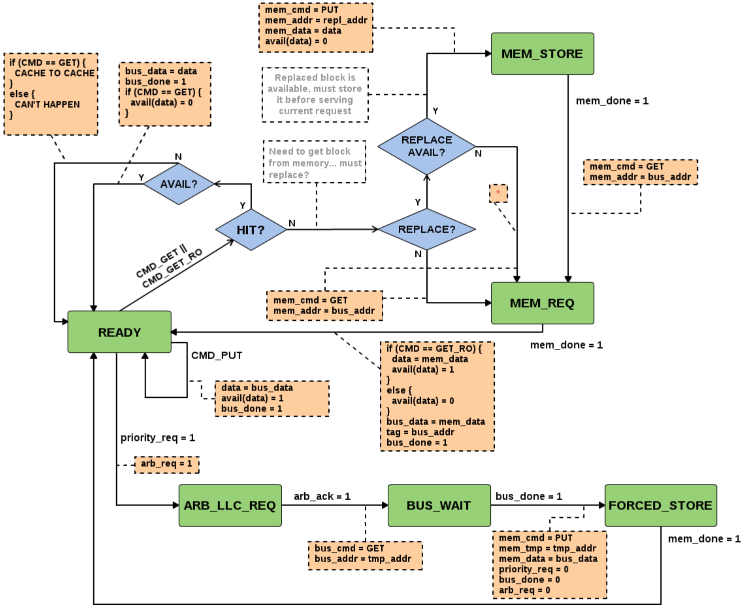
- Add LLC between L1s and memory
- Decouple direct access from L1 to memory

Last Level Cache (I)

Listed below are some of the reasons that pushed us to add a Last Level Cache to the Multicore:

- Acts as a memory that's closer to L1s in a real system
- Gives the Multiprocessor a bigger data bank that has lower response times than memory
- Due to lower response times from L1 requests, the overall performance is increased
- Stores blocks regardless of them containing data or instructions

Last Level Cache (II)



Assembler

```
li r0, 0
li r1, 1
li r2, 4
li r3, 0x10000
li r4, 0
li r5, 32

# This is a loop
loop:
ldw r6, 0(r3)
add r3, r3, r2
add r4, r4, r1
bne r4, r5, loop
```

```
1e000000
1e100001
1e200004
1e310000
1e400000
1e500020
22618000
00318800
00420400
65f217fd
```

Simulator and Validator

Simulator:

- Run code on a processor model to verify its behavior
- Dump caches, register banks and memory to check the result

Validator:

- Run the software simulation and the VHDL simulation in parallel
- Compare and check outputs (PC, register bank and memory)

Voyeur

- Snoop all bus transactions
- Print them in a "human-readable" format

Compiler

- Based on SISA (used on undergraduate FIB subjects) compiler
- Adapted it to 32 bits and 32 registers (previously, 16 bits and 8 registers)
- Too little time to continue developing it

Performance

(Results for vector sum code)

	I. Pentwice	**I. Pentiun w/ bus**	**I. Pentiun**
Cycles	12087	22730	20695
Instructions (p. core)	5130	10247	10247
CPI (p. core)	2.356	2.218	2.019

From Inkel Pentiun to Inkel Pentwice: 1.88x

Future Work

- Addition of the Compare & Swap atomic instruction.
- Adding an L2 level of cache between L1s and the LLC.
- Adapt the compiler (as previously explained).
- Implementation of an MLI based coherence protocol (MOSI, MOESI...)
- Adapting to FPGA! (They are the future! ... or so it is said...)
- Boot Linux

Conclusions

- It has helped us understand how a multiprocessor works
- Even the simplest protocol in theory becomes challenging in practice
- There are multiple ways to implement it, we just chose one of many

Hands On By Presenters!

Thanks for your attention!