Inkel^{TM} Pentwice Multicore Processor

Multiprocessor Architecture Project Report

Adrià Aguilà
Marc Marí
Antoni Navarro
Kevin Sala

June 23, 2017

Contents

1 Introduction

In this project for the Multicore Architecture course, we decided to get our hands onto the practical side of building a multicore. Our objective was building a multicore architecture based on an already existing single core architecture.

This single core architecture that we have used is called Inkel^{TM} Pentiun. It was written in a previous course for a project in which some of the group members also participated. The whole core is written in VHDL, and after fixing some minor issues, it became our starting point in our goal to achieve a multicore processor.

In this report, we provide a brief on how the Inkel^{TM} Pentiun works, which are its features and limitations. Next comes a second part where we explain how we made a multicore architecture by modifying it in some aspects and adding new components to our new architecture to finally build our multicore version, the Inkel^{TM} Pentwice[1][2].

2 Overall description

The Inkel^{TM} Pentwice processor is a dual-core in-order processor constituted by the following components and features:

- Two Inkel^{TM} Pentiun as the two cores of processor.
- Shared atomic bus connecting both cores' L1 and the last level cache. This bus has an arbiter to ensure that only one component is writing to the bus.
- Last level cache (LLC) to better exploit data locality and to avoid the large latency of the main memory.
- Write-invalidate bus-snooping VI cache coherence protocol between the L1s and the LLC through the atomic bus.
- Atomic instructions to allow synchronization between both cores.

Furthermore, both Inkel^{TM} Pentiun cores have the following features:

- Fetch stage with a directly-mapped instruction cache with four 16-byte entries.
- Decode stage supporting several types of instructions and a register bank with 32 total user-level registers.
- Three different pipelines: arithmetic, memory and multiplication.
- Execution stage with an ALU supporting additions, subtractions, moves and loads of immediate operands.
- 5-stage pipeline (M1-M5) for multiplication operations.
- Cache stage with a write-back data cache with four 16-byte fully associative entries and a store buffer with 10 entries.
- Reorder buffer (ROB) with 10 entries.
- Full set of bypasses.

[1] `https://github.com/kevinsala/multicore-architecture`
[2] "Blocking is for cowards, but it always works" - Development Team, 2017

3 Baseline processor: Inkel^{TM} Pentiun

In the following sections, the baseline processor and its pipeline stages are described. This baseline processor was developed by Carlos Escuín, Marc Marí and Kevin Sala in Processor Architecture, which is another MIRI subject.

3.1 ISA

In this section, the ISA of the baseline processor is described. For each instruction, the operating and its operands are detailed.

3.1.1 Arithmetic instructions

`opcode` (7b)	`rd` (5b)	`r1` (5b)	`r2` (5b)	`0` (10b)

- `add rd, r1, r2`: `opcode` 0x0
  `rd` = `r1` + `r2`
- `sub rd, r1, r2`: `opcode` 0x1
  `rd` = `r1` - `r2`

`opcode` (7b)	`rd` (5b)	`imm` (20b)

- `li rd, imm`: `opcode` 0xF
  `rd` = sign_ext(`imm`)

`opcode` (7b)	`rd` (5b)	`r1` (5b)	`0` (15b)

- `mov rd, r1`: `opcode` 0x14
  `rd` = `r1`

`opcode` (7b)	`offsethi` (5b)	`r1` (5b)	`r2` (5b)	`offsetlo` (10b)

- `beq r1, r2, label`: `opcode` 0x30
  If `r1` is equal to `r2`, jump to `PC` + shift_left(sign_ext(shift_left(`offsethi`, 10) + `offsetlo`), 2)[3]
- `bne r1, r2, label`: `opcode` 0x32
  If `r1` is not equal to `r2`, jump to `PC` + shift_left(sign_ext(shift_left(`offsethi`, 10) + `offsetlo`), 2)

`opcode` (7b)	`offset` (25b)

- `jmp label`: `opcode` 0x31
  Jump to `PC` + shift_left(sign_ext(`offset`), 2)

`opcode` (7b)	`0` (25b)

- `nop`: `opcode` 0x7F
  No operation

[3]The shift left in the branch and jump instructions is because the offset is counted in number of instructions, not in memory addresses

3.1.2 Memory instructions

`opcode` (7b)	`rd` (5b)	`r1` (5b)	`imm` (15b)

- `ldw rd, imm(r1)`: `opcode` 0x11
  Load value from address sign_ext(`imm`) + `r1` to `rd`
- `stw rd, imm(r1)`: `opcode` 0x13
  Store value in `rd` to sign_ext(`imm`) + `r1`

3.1.3 Multiplication instructions

`opcode` (7b)	`rd` (5b)	`r1` (5b)	`r2` (5b)	`0` (10b)

- `mul rd, r1, r2`: `opcode` 0x2
  `rd` = `r1`×`r2`

3.2 Functional description

The general diagram of the baseline processor is shown in Figure 1. It shows the three different pipelines and its stages separated by the segmentation registers. In the following sections, each of the stages of the pipeline is detailed.

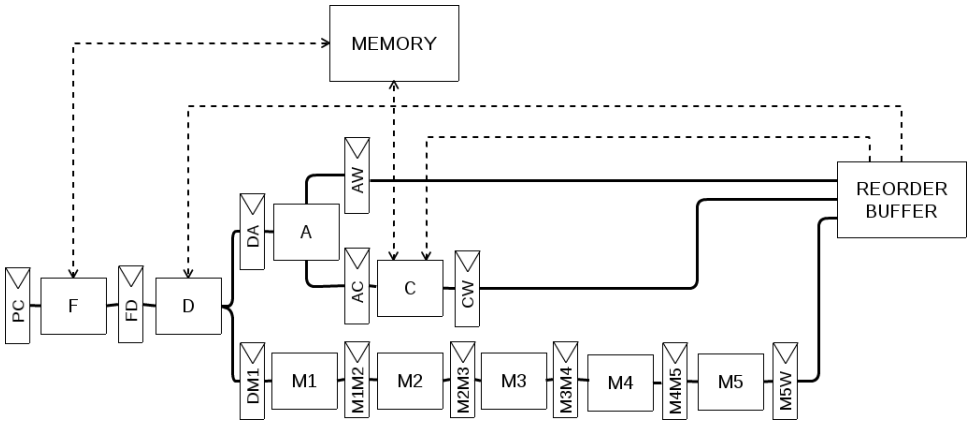

Figure 1: General stage diagram

3.2.1 Fetch

The fetch stage, shown in Figure 2, contains both program counter (PC) and instruction cache. This cache is directly-mapped and has four 16-byte entries (128 bits), so each cache line can contain at most 4 different instructions. First, when the processor boots, the initial PC is 0x1000.

In each cycle, the instruction pointed by the PC is loaded from the instruction cache. If the requested address is not present in the instruction cache, a memory request is sent to get the whole cache line and the fetch stage is blocked until it has finished.

When the requested address finally hits in the instruction cache, the instruction is returned and written on the segmentation register (`regFD`).

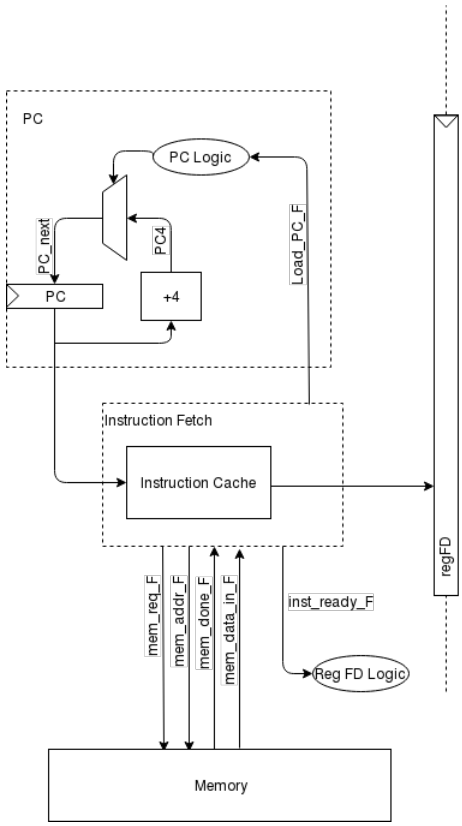

Figure 2: Fetch stage diagram

3.2.2 Decode

In the decode stage, shown in Figure 3, the instruction is analyzed, and the different control signals are set to the proper values for that instruction. If the instruction is not a valid one, an exception is raised. This stage also contains the register file, the detention unit and the bypass unit.

The register file is composed of 32 4-byte registers, and has 1 input port from the reorder buffer and 2 output ports for the source data. The detention unit is the component that detects conflicts between two in-fly instructions and stalls the corresponding stages to avoid errors during the execution. Finally, the bypass unit manages all the bypasses of the whole processor enabling the control signals of the bypasses' multiplexers.

Given the register source identifiers obtained from the instruction, the register file is accessed to get the necessary data to execute the instructions. Sometimes, this data is not up-to-date so before sending it to the next stage, there is a multiplexer to get the most recent value for each of the source registers.

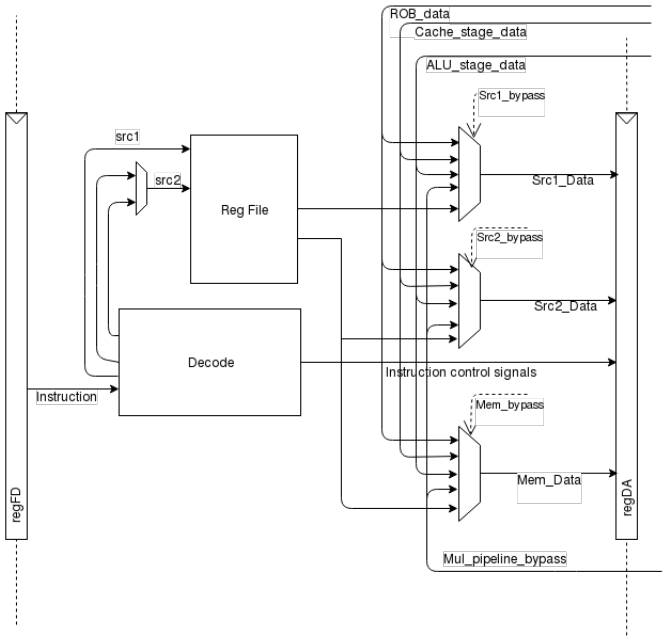

Figure 3: Decode stage diagram

Specifically, data may com from 5 different sources:

- Register file: the default case.
- ALU stage: an arithmetic instruction's result is the source of the decoded one.
- Cache stage: a load instruction's result is the source of the decoded one.
- Last multiplication stage (M5): a multiplication instruction's result is the source of the decoded one.
- Reorder buffer: the instruction that generates the source for the decoded one has finished its execution but has not been committed to the register file yet.

In some cases, bypassing the data is not possible (i.e. the data has not been computed yet) so the detention unit has to stall the fetch and decode stages to wait until the data is ready to be bypassed.

Finally, when the instruction is ready to move forward, the control signals computed by the decode unit and the operands are written to the `regDA` segmentation register.

3.2.3 ALU

The arithmetic logic unit (ALU) is the component that performs arithmetic operations given one or two source operands from the decode stage and produces a result value. Figure 4 shows a simplified diagram of this stage.

In the case of an arithmetic instruction (except multiplications, which have already branched into another pipeline), the operands are two registers, and the result has to be stored on the register file (when the instruction is committed).

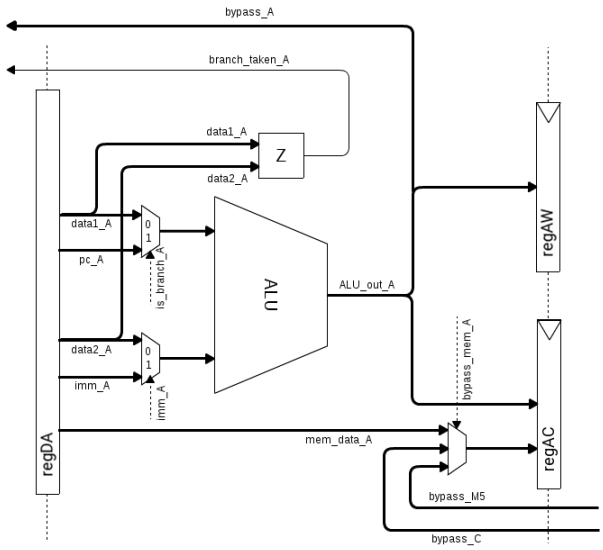

Figure 4: Execution stage (ALU).

For memory instructions, whose memory address is computed in this stage, the first operand is a source register and the second is the sign-extended immediate. In the case of the load immediate, the instruction doesn't take any source register and only outputs the sign-extended immediate. The move instructions takes only one source operand and outputs its value to a destination register.

Finally, in the case of a jump or a branch instruction, the first operand is the instruction's PC and the second operand is the offset (sign-extended immediate) to jump to, so the result of adding both operands is the jump address. For the specific case of the branch, there is also a zero (Z) module that compares the data from the source registers obtained at the previous stage and checks whether they are equal or not. When the instruction needs to jump (i.e. when the instruction is a jump or the `branch_taken_A` signal is set), these changes are performed:

- Sets the PC to the jump address.
- Squashes the instructions in fetch and decode stages by clearing those stages and decrementing the reorder buffer tail counter.

The result of the ALU may be bypassed to other stages (i.e. the decode stage). In the case of store instructions, the data to be written is not modified in this stage so it can also be bypassed from these source stages:

- ALU stage: the default case.
- C stage: a load instruction's result is the source of a store instruction.
- M5 stage: a multiplication instruction's result is the source of a store instruction.

Note that in this stage, data cannot be bypassed from the reorder buffer (ROB). The analysis of dependencies showed that if it was necessary to bypass data, it had already been bypassed in the previous stage (decode).

In case the result is ready to move forward, it is written to the corresponding segmentation register. If it is a memory instruction it is moved to the cache stage by writing to the `regAC` segmentation register. Otherwise, the instruction has finished and it has to wait until it is committed, so the instruction has to move to the reorder buffer stage by writing to the `regAW` segmentation register.

3.2.4 Multiplication Stages

Multiplication instructions are not processed by the ALU stage but by the multiplication pipeline. The MUL pipeline consists on five multiplication stages (M1 to M5). The first stage (M1) is connected after the decode stage through the `regDA` segmentation register, which is the same as in the ALU stage.

All the multiplication operands must be ready before the instruction leaves decode. Besides, the result of the multiplication will not be ready until the last stage (M5). Consequently, instructions that have a dependency with a multiplication instruction should stall at decode until the multiplication finishes and the result is bypassed from the M5 stage. The result can also be bypassed to the ALU stage, as explained previously.

Finally, when a multiplication instruction finishes its execution, it is moved forward to the reorder buffer's stage where it is added to the buffer waiting to be committed.

3.2.5 Data Cache

Memory instructions (loads and stores) continue their journey through the pipeline on the cache stage. Figure 5 shows the basic structure of the cache stage.

This stage has a write-back fully associative data cache with four 16-bytes entries and a 10-entry store buffer to store pending store accesses waiting to be committed by the reorder buffer. The cache is connected directly to memory through a private bus and it is also connected to the store buffer to send and receive information about the pending stores. At the same time, the store buffer is connected to the reorder buffer to receive commit commands of pending stores.

The address of an access is taken from the result of the ALU stage which is computed as the sum of the first source register and the immediate. In the following paragraphs, the operation of this stage is explained.

In the case that the access is a store, its address and the value to be stored is saved into the first free entry of the store buffer. This store will be committed onto the cache when the instruction is committed by the reorder buffer.

In parallel with the store buffer, the cache checks if the memory line of the address is present in the cache. In the case that the memory line is not present, there is a cache miss and the cache has to send a line request to memory. This cache miss can cause an eviction of a valid line which, in turn, can create a conflict between this evicted line and a pending store that targets the same line. In this case, the cache stage stalls until the conflict disappears, this is when the reorder buffer commits the pending store and definitively saves it in cache. Then, the line can be finally evicted and sent to memory. Finally, the original memory request can proceed. With this technique, each pending store assures that its target line is present and valid in the cache.

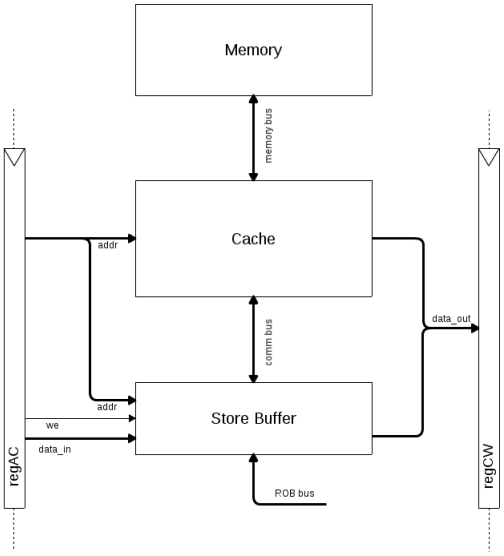

Figure 5: Cache stage

To do the communication between the cache and the store buffer discussed above, a bus between both components is used. This bus contains signals to know which address is being replaced by the cache, the store being committed by the store buffer (and therefor, the reorder buffer) and its value.

The processing of a load instruction is much easier. In parallel, the store buffer and the cache are checked. In the case of a miss in tags, the line has to be requested to memory. If it was necessary to evict a line, the cache would proceed as already explained with stores. In the case that both components report a hit, the data from the store buffer has priority to be served because it's the most recent value.

The value retrieved by a load can be bypassed to the decode and the ALU stages. When the memory instruction finishes, it can be moved forward to the reorder buffer's stage. Finally, it should be noted that an unaligned access can rise an exception.

3.2.6 Reorder Buffer

The reorder buffer (ROB) is a circular buffer of 10 positions with 3 inputs (one input port for each pipeline) and 1 output. For each in-fly instruction, it stores the destination, the data generated and whether the instruction is an store or not. At the same time, it commits this information in the correct sequential order. Figure 6 shows a general diagram of the reorder buffer.

For this reason, the reorder buffer issues "tickets" to each instruction at the fetch stage. These "tickets" indicate the position where the instruction will be written on the reorder buffer. Every cycle, if the head of the buffer has valid data and can be committed, this data is written to the proper place on the register bank. In case the instruction is a store, it forces the store buffer to commit it to the cache.

The information stored in the reorder buffer may be bypassed to any stage that requests that data. In order to do the bypasses, the order of the data in the ROB must be taken into account. Therefor, entries in the ROB should be checked from tail to head.

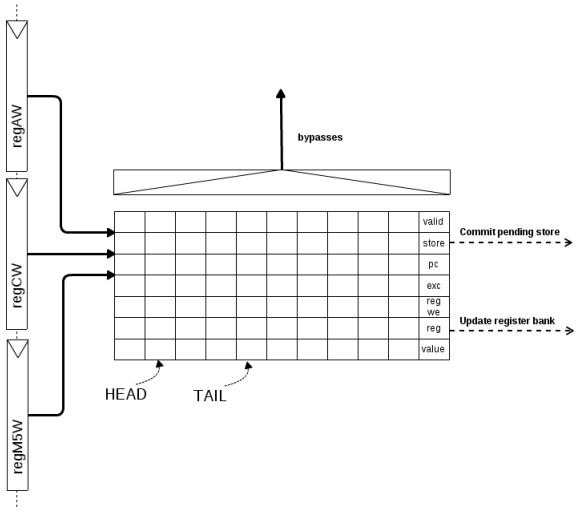

Figure 6: Reorder buffer diagram

3.2.7 Exceptions

In order to handle exceptions when they happen at some point in the pipeline in the correct sequential order, the exception code and data are propagated to the reorder buffer and stored there. When the ROB commits the instruction, if it has raised an exception the privilege status bit is set to one, the following instructions are killed (invalidate all entries in the ROB and kill any instruction currently on the pipeline), and the next instructions fetched are the system instructions. At the moment, all exceptions are classified as *fatal error*, so the system code is only an infinite loop that makes the processor to stall.

The exceptions the processor can raise are:

- Invalid memory access (i.e. unaligned memory access).
- Invalid instruction (i.e. instruction code not supported by the processor).

3.2.8 Memory

The memory is a buffer of 256 KB, arranged in 16K words of 16-byte width. When the processor boots, it loads the file `memory_boot` (user code) on address 0x1000, and the file `memory_system` (system code) on address 0x2000. This memory only has one read port and one write port, and, therefor, it is governed by an arbiter.

The arbiter attends the requests in FIFO order. In case that two requests arrive at the same time, one from the instruction cache and the other one from the data cache, the request from the data cache is served first because the instruction from the cache stage is older than the instruction from the fetch stage.

Requests to memory take a minimum of 6 cycles. 5 of these cycles are the latency of the memory operation itself, and another cycle is the latency of the arbiter. But memory will not attend any new request until it finishes with the current one, which may cause a longer access latency.

3.3 Preparing to move to multicore

Originally, the baseline processor wasn't as complete as we have described in the previous sections. There were two main versions of the processor, the working one and an advanced prototype. The working version of the processor had all the features discussed before but without the store buffer and the reorder buffer.

The advanced version was a prototype with the store and reorder buffers, but without any connection between them. This way, the store buffer committed pending stores only when the cache stage was free, without waiting for the instruction to be committed in the ROB. It should be noted that this last version wasn't working correctly due to some bugs in the design and the implementation.

Before starting with the multicore project, we had to correct the advanced version by solving all its problems and finally, refactoring the store buffer and the ROB to allow communication between them. After these changes, the ROB sent commit commands to the store buffer, and then, the store buffer committed the pending store to the cache, as desired.

4 Multicore processor: Inkel^{TM} Pentwice

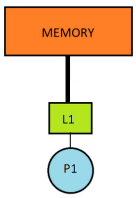

Figure 7: Single core

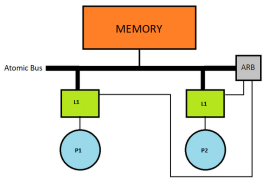

Figure 8: Multicore

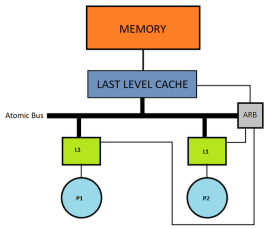

Figure 9: Multicore with last level cache

With the single core working (Figure 7), we started making the necessary changes that needed to be done to plug a second Inkel Pentiun to the system to have our multicore architecture, Inkel Pentwice (Figure 8). The following sections explain in detail the different elements necessary to do this transition.

The first step was to introduce a bus between the processor caches and memory. Every cache can see what happens in the bus and react accordingly, but only one has access to the bus at any given point in time. This access is granted by a new component, the arbiter, in charge of controlling who and when gets the bus.

Later, a second core was introduced. In order to coordinate the memory accesses of both processors, a VI coherence protocol was implemented. When this version of the multicore was working, we decided to go one step further and make it slightly more similar to a real core. For this reason, we decided to add a Last Level Cache (Figure 9).

5 Memory System

We were unsure on how the system between processors, caches and memory was going to be like at the end of the project. We also were unsure on the amount of cache levels that there were going to be. Hence why we decoupled the management of data blocks from the memory itself. This was mainly done so that any change in the coherence protocol wouldn't affect the memory's structure or code.

In the next subsections, the memory controller, the atomic bus and the arbiter are introduced and a brief discussion is made as to why they are needed.

5.1 Memory Controller

As aforementioned, a memory controller is needed to decouple the management of blocks from the memory. Any request from a cache is then redirected to the memory controller instead of the memory itself. The memory controller answers these requests and it is the only agent in the system to have real access to memory.

5.2 Atomic Bus

Introducing a variable amount of processors and, therefor, L1 caches in the system, raised the need for a way to serialize accesses to memory. This need was satisfied by introducing an atomic bus between memory and caches, which later on became an atomic bus between L1 caches and the last level cache.

The bus allows processors not actively using it to passively observe it in order to act accordingly if a conflict happens or if they must evict one of their blocks and serve it to another processor. However, there must be an agent carrying the task of giving access to the bus, task that is performed by an arbiter.

5.3 Arbiter

The arbiter has full control of the bus in order to serialize memory accesses. It accepts requests from any other agent plugged into the atomic bus, and gives access following a certain policy. Listed next are the policies implemented:

- **Random:** In order to introduce variability in the system, a random access policy was implemented and tried. With this policy, the requester that gets access to the bus is randomly chosen.
- **Fairness:** To balance the capabilities of all the requesting agents, a fairness policy was also tried. This policy is based on a Least Recently Used algorithm. For each possible requesting agent, it keeps track of when they last worked. When requests are made, the Arbiter gives access to the requester that has not worked for a long time (more than all the others). This policy was the chosen one as it ended up balancing the work between processors and gave a greater performance.

6 Coherence Protocol

In order to keep coherent access to memory between the processors' caches, it was necessary to decide on a protocol. For the sake of simplicity, we decided to start with the simplest coherence protocol, a VI (Valid-Invalid) with only two states.

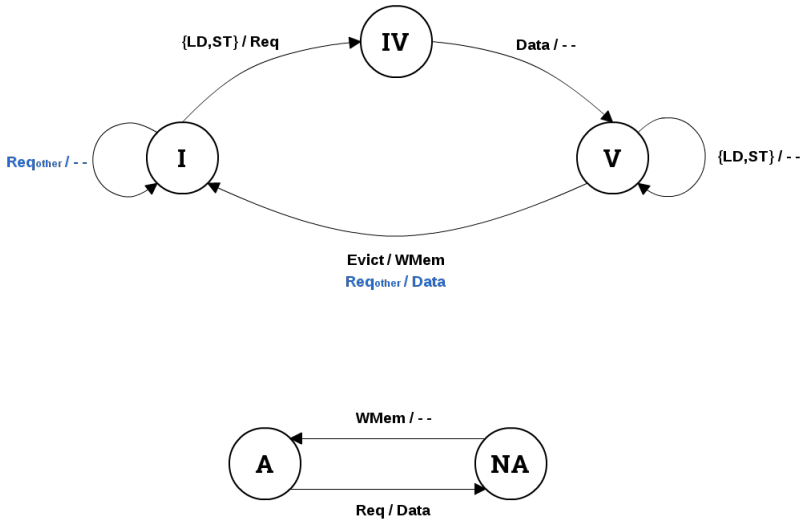

Figure 10: Coherence protocol

In figure 10 is the diagram for both the actuator and the observer on the data caches. The actuator does not imply changes, as it is the same that the single core used: when an access to cache misses (it doesn't matter if it is a load or store, because there is a single valid state), it makes a request for that cache line (it sends a `CMD_GET` command through the bus) and goes into a temporary state where it waits for the block. When the block comes on the bus, either from memory or another cache, it becomes valid on the requesting cache and any subsequent accesses will hit, until the replacement policy decides to evict that block. In this case, the block is sent to memory (command `CMD_PUT`) and the state becomes invalid again.

The memory controller on the other side, keeps track of whether a block is available or unavailable: when it gives a block to a cache, it marks it as unavailable, and marks it as available again when this block is written to memory after an evict.

In blue in the figure are the actions that correspond to the observer process. This process is constantly snooping the bus, and ignores every request on the bus unless it is a requesting a block that the observer has as valid. In this case, the cache observing the request must invalidate the block and put the data in the bus. Therefor, memory is not aware of the owner of the data, it just knows whether it has the latest version.

To simplify implementation, we decided that instruction caches would not have a bus observer. This decision is based on the assumption that memory blocks corresponding to instructions are not modified. It is then necessary to let memory know that it should not mark the requested block as invalid. For this reason, instruction caches send the command `CMD_GET_RO` to the bus when they request a memory block, and they don't send a `CMD_PUT` when they evict the block (they evict silently).

6.1 Conflicts in the Store Buffer

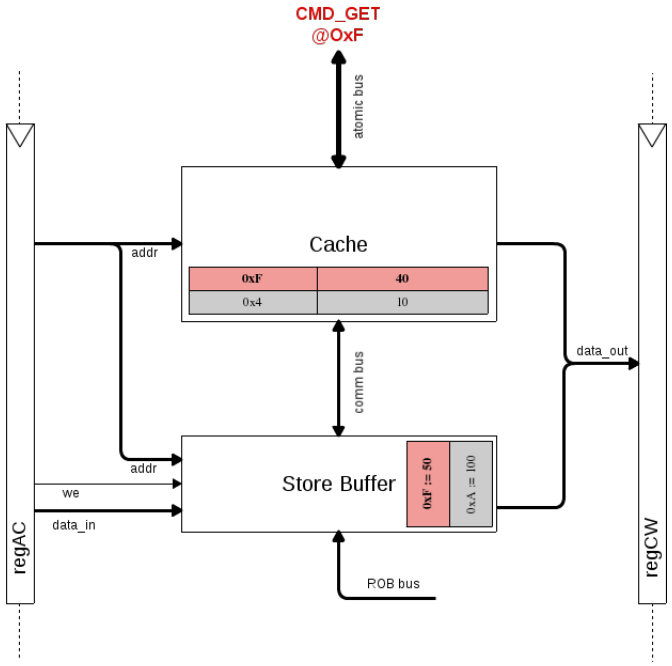

Figure 11: Conflict in the store buffer when using the coherence protocol

Adding a coherence protocol to the processor leads to an important conflict between the data cache, the store buffer and the coherence protocol. Figure 11 shows this conflict. For instance, the problem is shown when the cache has a valid line (in this case, with address 0xF), there is a pending store to that line in the store buffer and the cache's observer detects a `CMD_GET` in the bus at the same address. This means that the line has to be invalidated in the cache and its data has to be sent through the bus. We saw two main solutions:

- Block the bus request until the pending store is committed to the cache, and then, when the conflict has disappeared, send the most recent data (in this case is 50) through the bus.
- Reply immediately the bus request by sending the data in the cache (in this case is 40) and invalidating the cache line. This technique breaks the property explained before which ensures that every pending store has its target line valid in the cache. Therefor, when a pending store is committed, it's possible that the target line where it has to write has been invalidated, so the line would have to be requested again.

As can be seen, the first approach is easier than the second one. Therefor, we decided to implement the first one for simplicity reasons.

7 New Instructions

To allow both processors to synchronize when running codes, two new instructions were added: test & set (`tsl`), which is executed on the cache pipeline, and get processor ID (`pid`), which is executed on the ALU pipeline.

`opcode` (7b)	`rd` (5b)	`r1` (5b)	`imm` (15b)

- `tsl rd, imm(r1)`: `opcode` 0x15
  Atomically load value from address sign_ext(`imm`) + `r1` to `rd` and store a 1 to the same address if the previous value was 0

`opcode` (7b)	`rd` (5b)	`0` (20b)

- `pid rd`: `opcode` 0x40
  `rd` = processor ID, which can be 0 or 1

8 Last Level Cache (LLC)

The Last Level Cache is a fully associative cache with 32 16-byte entries, that is used to reduce memory latency and extend the capacity of the L1 caches. Figure 12 shows all the actions performed by the LLC as a consequence of L1 requests, its reactions as a consequence of observing certain L1 actions in the bus and all the states the LLC goes through.

The LLC behavior is explained next as a top down list, explaining every detail that leads to a certain state and the actions that take place in it.

- If the LLC is in the `READY` state and...
  - L1 requests a data block (`CMD_GET`):
    - If the request hits (block is in the LLC and it is valid) and the data in the block is available, it is put in the bus and marked in the LLC as not available. Next state is `READY`.
    - If the request hits but the data in the block is not available, nothing should be done, because the L1 cache that has the block as valid will submit it to the bus.
    - If the request misses, the LLC needs to ask memory for the requested block, but first it must check if it has to evict a block. If it doesn't have to evict, it simply makes a memory request for the block. Next state is `MEM_REQ`.
    - If the request misses, and the LLC needs to evict a block, and that particular block is available, it must store the evicted block into memory first. Next state is `MEM_STORE`.
    - If the request misses, the LLC needs to evict a block and that particular block is not available, it must force an eviction on the L1 that has the block that is going to be evicted, and store its value in memory. To do this, the LLC will first serve the requesting L1, and then mark that there is a pending request in the LLC. The next state is `MEM_REQ`.

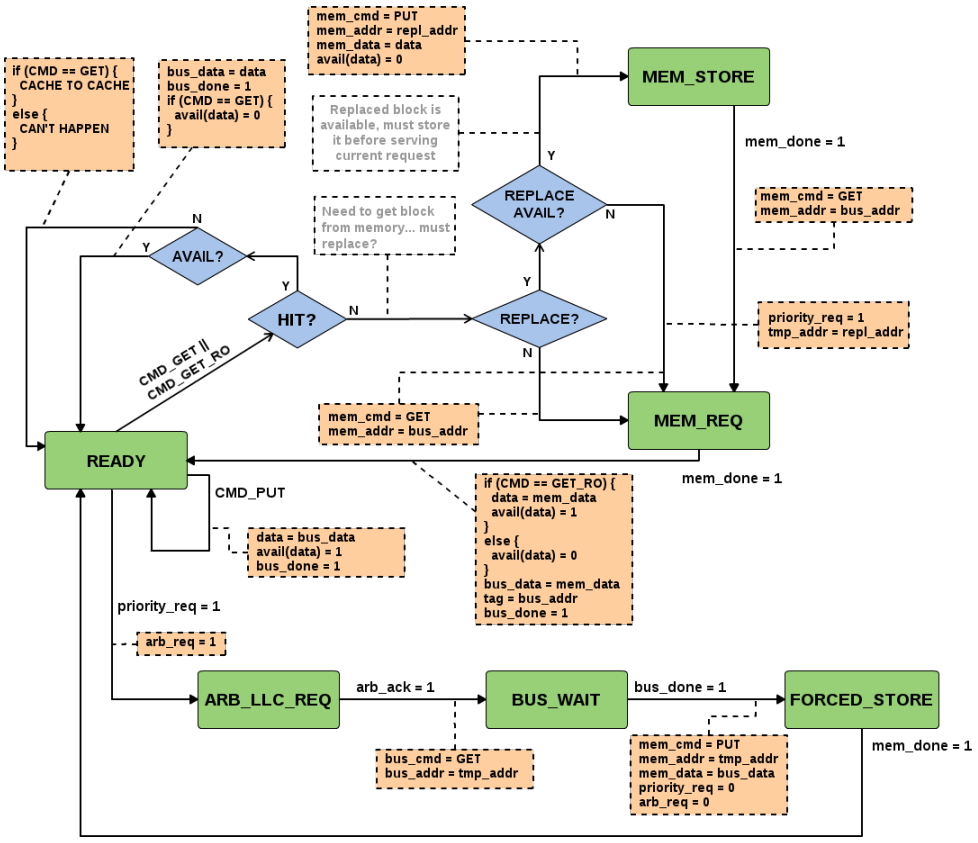

Figure 12: Last Level Cache States and Actions

- L1 requests an instruction block (`CMD_GET_RO`): The behavior is exactly the same as the previous request (`CMD_GET`), however there are two different details:
  - When the request hits and the block has available data, the block in the LLC is not set as not available once it serves the request because instruction blocks cannot be modified (as previously explained).
  - When the request hits and the block doesn't have available data. This combination of events is impossible because if a request hits and it is for an instruction block, the block must be available in the LLC.
- L1 evicts a block and so it must be stored in the LLC (`CMD_PUT`): As the block has to be in the LLC, it simply stores the block and marks it as available. The next state is `READY`.
- If there is a pending request on the LLC (this has priority over all other possible requests previously explained, to have as many as one pending conflict at a certain time): This means that the LLC must ask permission to the Arbiter in order to have access to the bus so that it can force an eviction. This request to the arbiter has priority over all L1 requests, in order to avoid new conflicts arising. The next state is `ARB_LLC_REQ`.

- If the LLC is in the `MEM_REQ` state and...
  - Memory has sent the requested data: the LLC puts the data in the bus for the requesting L1. If the requesting block is a data block (`CMD_GET`) the block in the LLC is marked as not available. However, if the requesting block is an instruction block (`CMD_GET_RO`) the block in the LLC is marked as available. The next state is `READY`.
- If the LLC is in the `MEM_STORE` state and...
  - Memory marks that the requested data has been stored: the LLC continues and makes the memory request it had to do to serve the L1. The next state is `MEM_REQ`.
- If the LLC is in the `ARB_LLC_REQ` state and...
  - The arbiter has granted access to the bus: the LLC forces an eviction of the block whose data is located in some L1 by sending a `CMD_GET` command to the bus. This request will make the L1 that has the block evict it as if it was a cache to cache transaction. The next state is `BUS_WAIT`.
- If the LLC is in the `BUS_WAIT` state and...
  - The L1 has already evicted the block: the LLC gets the data from the bus and stores it in memory. The next state is `FORCED_STORE`.
- If the LLC is in the `FORCED_STORE` state and...
  - Memory marks that the requested data has been stored: the LLC moves to state `READY`.

It should be noted that this LLC is a non-inclusive non-exclusive cache. All data blocks (blocks requested by data caches) are inclusive, if they are in one of the L1s, they are on the LLC. But instruction blocks (blocks requested by instruction caches) are not inclusive, because the LLC can evict them silently, as they are always marked as available in the LLC. This is not a problem, because, as explained previously, instruction caches also evict silently.

9 Tools

9.1 Assembler

In order to simplify the operation of the Inkel Pentwice and, previously, the Inkel Pentiun, several tools were implemented. The first of this tools is an assembler written in Python that converts mnemonics (which have already been introduced with the processor ISA) to its machine representation, in a format readable by VHDL.

The elements that this assembler recognizes are:

- Comments: lines starting with `#` are not interpreted.
- Labels: text lines ending with a colon (`:`) are interpreted as labels.
- Instructions, in the format of `inst op0, op1, op2`, where `inst` is a mnemonic as introduced previously, and the operands may be one or more of the following (depending on the instruction):
  - Register operands: `rX`, where `X` is the register number
  - Immediate operands: binary numbers preceded by `0b`, octal numbers preceded by `0o` or `0`, decimal numbers or hexadecimal numbers preceded by `0x`
  - Labels
  - Memory operands: composed of a register operand and an immediate operand in the format `imm(rX)`

To run this assembler:

```
python compiler/compiler.py <source>
```

The output is on `memory_boot` file.

9.2 Simulator

The other tool that has been implemented to allow debugging codes is a simulator. This simulator is written in Python and models the behaviour that the VHDL processor should have. This simulator contains all the components that the processor has (instruction and data cache, memory subsystem...), but runs only one instruction per cycle, instead of using a pipeline as the real processor does.

To run this simulator:

```
python simulator/run_code.py <machine code file>
```

This prints the values in memory, caches and the register bank at the end of the execution of the given program.

9.3 Validator

The simulator introduced in the previous section can also be run in parallel with the VHDL model in order the check the correctness of the VHDL. This validator is based on the Co-routine Co-simulator Test Bench (Cocotb [4]) using the GHDL VHDL simulator [5] in the background.

This validator runs the VHDL model until an instruction finishes execution (exits the ROB and is commited), and then it runs a step on the simulator. After this, the PC, register bank and memory of the VHDL and the simulator are compared. If there is a mismatch, an error is reported.

This validator does not help in debugging errors specifically, but it allows to run long programs and check its behavior automatically, instead of running manually on ModelSim. If this validator reports an error, it is then easier to go to the reported cycle on ModelSim and debug that specific error.

It is important to note that cycles in this validator are 2ns long, while they are only 1ns long in ModelSim. This is due to Cocotb not allowing steps of 0.5ns.

Cocotb and GHDL interact using the Verilog Procedural Interface (VPI) interface which, in the case of GHDL, does not support VHDL arrays (used in, for example, memory). For this reason, it was necessary to introduce a signal in the processor (`debug_dump`) that dumps memory and register bank to text files, that can later be read from the validator interface. For this reason, the validator is slow (around 100ns simulated every second, depending on the code).

The instructions to run this validator are in the project Github repository [6].

9.4 Voyeur

The same tools used for the validator (Cocotb and GHDL) have been used to snoop on all bus transactions.

Every time the user presses the `<Enter>` key, the VHDL simulation advances to the next bus operation, and displays it in human-readable form. For example:

```
P1, dCache has permission to use the bus
Sent a PUT petition on address 0x10160
Data: 0x00000000000000000000000000000000
Transaction finished in 2 cycles
```

As this tool has no knowledge about the code that is being run on the processor, if the processor reaches the end of the program the tool will hang, because there won't be any more transactions to memory.

[4] `https://github.com/potentialventures/cocotb`
[5] `https://github.com/tgingold/ghdl`
[6] `https://github.com/kevinsala/multicore-architecture`

9.5 Compiler

We were also able to obtain the code for a very simple C compiler for the SISA architecture[7]. As this was a much simpler compiler than the compiler for a mature architecture such as x86, ARM or MIPS we thought it might be simpler to modify and adapt.

In the time that we worked on this compiler, we were able to change the processor word from 16 to 32 bits, and increase the number of registers from 8 to 32.

It is now necessary to start modifying the ISA of the compiler and add missing but important instructions on the processor.

10 Performance Results

Table 1 shows the performance results when executing an example vector sum code with the Inkel Pentwice (multicore) with and without LLC, Inkel Pentiun (unicore) with the atomic bus and the baseline Inkel Pentiun.

	I. Pentwice		**I. Pentiun**	
	LLC	**no LLC**	**bus**	**no bus**
Cycles	12087	13630	22730	20695
Instructions (p. core)	5130	5130	10247	10247
CPI (p. core)	2.356	2.657	2.218	2.019

Table 1: Performance results executing vector sum

The increase in CPI between the Inkel Pentiun with and without atomic bus is due to the extra cycles required to acquire the bus and put the necessary data on the bus. This change produces a slowdown of nearly 10%.

On the other hand, the decrease in CPI between the Inkel Pentwice with and without LLC, is due to the reduced latency to perform some memory operations. The introduction of an LLC produces a speedup of nearly 13% for this specific code.

Although the serial and parallel versions cannot be directly compared (because the code executed is not the same), it is still interesting to check some of the numbers. The speedup in executed cycles between the best unicore version (the baseline processor) and the best multicore version (with LLC) is of 1.71x. This is not too far from the theoretical 2x speedup that could be obtained when moving to dual-core. And it is even more surprising considering the number of memory conflicts (and therefor, stalls) that appear when inserting a second core on the system.

[7]Salagre, P. C., & Navarro, J. J. (2004). Generació de codi per a una arquitectura RISC pedagògica. `http://cataleg.upc.edu/record=b1264636~S1*cat`

11 Conclusions & Future Work

This project has represented an amount of work that none of us anticipated. Taking into account that we chose one of the most simple coherence protocols, we were expecting to finish it rather quickly and do bigger and better stuff. When we started working with the code, we realized that even this simple protocol is anything but trivial, and that conflicts can arise everywhere, depending on how you chose to design and implement previous steps.

We have acquired a deeper knowledge of how a multicore can work or rather, one of many ways you can get it to work, as we have encountered many forks in the road where we were forced to make a decision and go one way or another. The result, we believe, would have been the same regardless of the path we had chosen, but certainly the implementation and troubles would have been a whole different story. What is undeniable is that we have also acquired a lot of experience in VHDL along the way.

Since implementing a VI protocol with the LLC took us way more time than we expected, there are many things that we wanted to do which, unfortunately, we didn't have time to. Some of the things we could have done but there was no time to fit them in are:

- Adding more atomic instructions, like Compare&Swap, Fetch&Add and some others.
- Adding a second level of cache L2 for each Inkel Pentiun, to have more levels of cache and resemble more a real processor.
- Implementing a more complex protocol. We started with VI because we thought it would be quick to implement and a good starting point for a more advanced protocol, like MLI, MOESI, etc. We misjudged the amount of work it would take to get to VI from scratch, but we believe that it would be easier to evolve to another protocol now that we have one that works, even if it is simple.
- Fully adapting the SISA compiler to our needs, but this was left out because of time constraints.

As a much more long term future work, we would also like to see if this core can run in an FPGA, and if we could get it to boot a real OS with the appropriate (and probably abundant) changes. But for now, our work for the course ends here, and we consider the result to be satisfactory, even if we weren't able to do everything we would have liked to.