ION -- free, open source MIPS32r2 compatible CPU core

Revision 1 - March 17, 2014 Core Design Notes

OVERVIEW

This document contains design notes meant for the CPU maintainer or for anyone interested in the design of the ION core.

None of the information in this document is necessary to use the core itself.

In time, this document will explain the design of the CPU, its implementation and the rationale behind them.

This document has been hastily put together and is very far from complete. It must be considered a work in progress.

CAVEATS

This document is part description and part wishlist, since the core is in the middle of a major refactor.

Some of the specifications laid out in this document may change before the core is completed – some haven´t even been aggreed on.

Table of Contents

1.- ION Internal Memory Bus

This is the memory bus used internally in the core; it´s not used outside the **ion_core** entity or in its external interfaces so its details will only interest you if you want to debug or develop the core itself. For lack of a better name, this document will call it the “ION Bus”.

It is a plain, point-to-point, pipelined 32-bit bus meant to be connected to synchronous memories of the usual FPGA BRAM kind. Its operation reflects some of the quirks of the pipeline, which makes it unsuitable for general purpose use.

The bus is used to interface entity **ion_cpu** with the caches. The CPU is always the bus master.

It is formally encoded in two record data types, one for the master outputs (**MOSI**) and other for the master inputs (**MISO**). The signals are described in the following table.

Table 1: ***ION Internal Memory Bus Signals***

Signal	Width	Description
t_cpumem_mosi		**Master output, slave input**
addr	32	Active high synchronous reset.
rd_en	1	Read Enable.
wr_be	4	Write Byte Enable. Bit 0 is for wr_data[7..0].
wr_data	32	Data bus, write.
t_cpumem_miso		**Master input, slave output**
rd_data	32	Data bus, read.
mwait	1	Asserted to stall a read or write cycle.

The bus only supports simple read and write cycles as described in the following sections. Since the code bus is slightly different, fetch cycles (read cycles over the code bus) are described separately.

Data Read Cycles

For a read cycle without any wait states, this is what happens on the bus on active (positive) clock edges:

- Edge 1: Master drives **MOSI.addr** and asserts **MOSI.rd_en**.
- Edge 2: Master deasserts all **MOSI** signals.
- Edge 2: Slave drives **MISO.rd_data** and deasserts **MISO.mwait**.
- Edge 3: Slave must drive **MISO.rd_data** until this edge (i.e. for one clock cycle).

For a read cycle with wait states, the edge-by-edge sequence is this:

- Edge 1: Master drives **MOSI.addr** and asserts **MOSI.rd_en**.
- Edge 2: Master deasserts all **MOSI** signals.
- Edge 2: Slave asserts **MISO.mwait** for as long as needed (say K cycles).
- Edge 2+K: Slave drives **MISO.rd_data** and deasserts **MISO.mwait**.
- Edge 2+K+1: Slave must drive **MISO.rd_data** until this edge (i.e. for one clock cycle).

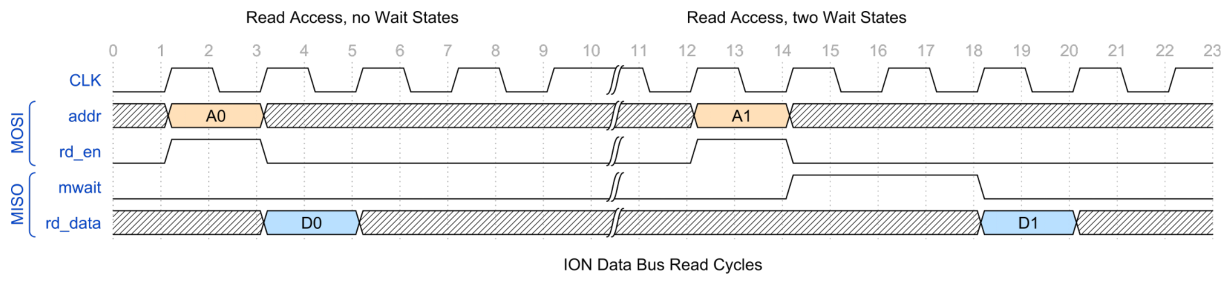


*Figure 2: **ION Data Bus Read Cycles***

Note that in the present implementation of the CPU, all instructions following a load instruction are stalled for one clock cycle – see last section of this chapter.

Data Write Cycles

This is the event sequence for a write cycle:

- Edge 1: Master drives **MOSI.addr** and **MOSI.wr_data** and asserts **MOSI.wr_be**.
- Edge 2: Master deasserts all **MOSI** signals.
- Edge 2: Slave asserts **MISO.mwait** if necessary for K cycles.
- Edge 2+K: Master remains stalled until this edge.

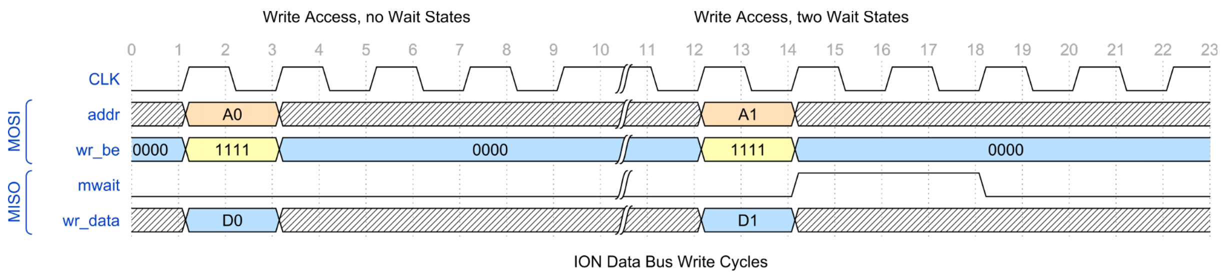


*Figure 3: **ION Data Bus Write Cycles***

The master will remain stalled for as long as **MISO.mwait** is asserted.

The bus supports four 'byte lanes' as described in table 4.

*Table 4: **ION BUS Byte Lanes***

MOSI.wr_be	**Byte Lane**	**Description**
1111	31 .. 0	SW on address ending in 0100.
1100	31 .. 16	SH on adress ending in 0010.
0011	15 .. 0	SH on adress ending in 0000.
1000	31 .. 24	SB on adress ending in 0011.
0100	23 .. 16	SB on adress ending in 0010.
0010	15 .. 8	SB on adress ending in 0001.
0001	7 .. 0	SB on adress ending in 0000.

Code Read (Fetch) Cycles

A code fetch cycle is identical to a data read cycle with one exception:

- **MISO.rd_data** must remain valid from the edge **MISO.mwait** is deasserted to the edge after **MOSI.addr** changes again.

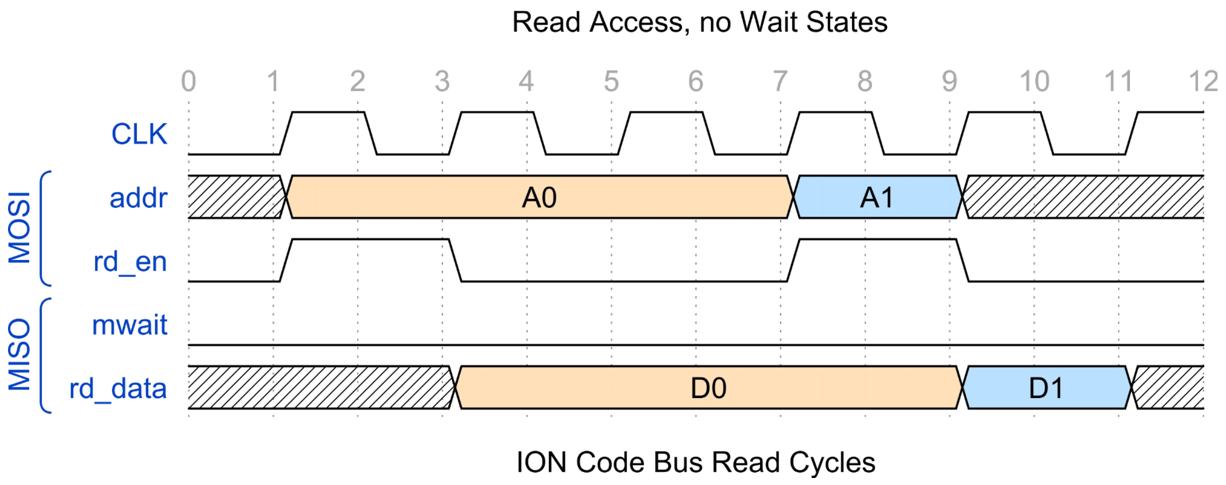


Figure 5: ***ION Code Bus Read Cycles***

Other than that extra requirement, and the fact that it will never execute a write cycle, the code bus is identical to the data bus.

The reason for this extra requirement is explained in the next section.

Limitations of the ION bus implementation

The CPU data interface has several important limitations:

1. Unaligned loads or writes are not implemented. The core relies on unimplemented opcode traps for those.
2. All instructions following a load instruction are stalled for one clock cycle.
3. Code fetches require the code word to be valid for more than a single cycle.

A brief rationale for these implementation quirks follows.

Unconditional stalls in data load cycles

The reason for the unconditional load stall is that the register bank has a single write port. The register writeback of a load instruction happens one cycle later than a normal instruction writeback; so a load and a post-load instruction would both clash trying to write to the register bank at the same time.

The simplest solution to this problem is an unconditional stall; the best solution would be adding a second write port to the register bank. A compromise solution (stalling only those post-load instructions that write to the register bank) might improve performance slightly.

For the time being, the simplest, cheapest solution has been implemented: unconditional stall.

Code fetch data valid requirement

The Ir register (**p1_ir_reg**) will not be loaded if the CPU is stalled for any of four reasons: code wait, data wait, multiplier wait or load interlock – see logic for **stall_pipeline** in line ~966 of ion_cpu.vhdl.

This means that the moment the opcode is actually fetched does not depend ony on the code bus master logic; the cpu may be stalled for some reason other than a code wait and thus the code word may be latched at any time from the end of the code wait to the beginning of the next fetch cycle. So the slave has to make sure the word is there when latched.

Since the ion code bus is only meant to be used internally, this is not a terribly costly requirement in terms of complexity, actually, it helps reduce the complexity of both CPU and cache, since the behavior matches that of a common FPGA RAM block.

2.- Structural Description

What follows is a description of the implementation of some selected parts of the core. This section will eventually cover the entire circuit in a later revision of this document.

Core Structure

The **ion_core** entity wraps the CPU itself (ion_cpu) together with its caches and a TCM (Tightly Coupled Memory) block for each data and code:

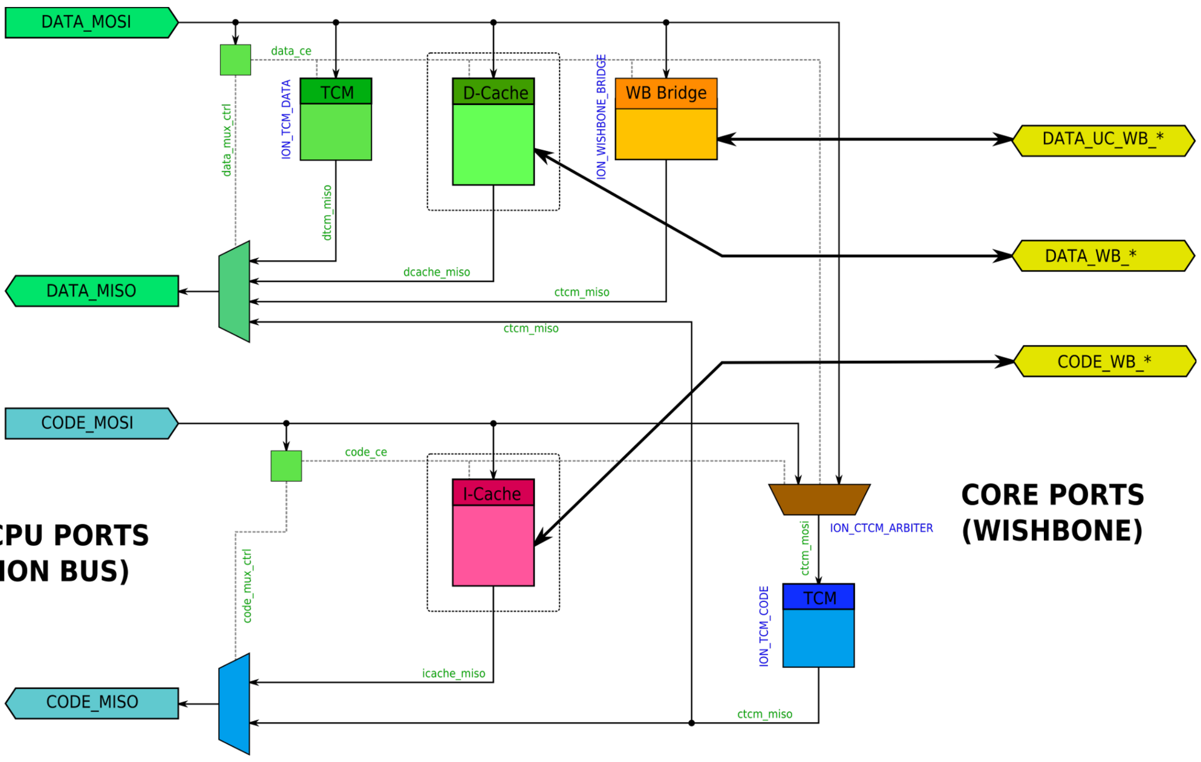


Figure 6: ***Core Interconnect***

Figure 6 depicts in detail the structure of the interconnect within the **ion_core** entity.

All slaves within the code or data spaces share the same master MOSI bus; their MISO nuses are multiplexed back to the respective CPU MISO port.

An address decoding combinational block generates a select signal for each slave (**data_ce** and **code_ce**) and the control signals for the MISO multiplexors (**data_mux_ctrl** and **code_mux_ctrl**).

The caches are not implemented in the current state of the source. They are meant to be optional, an dof omitted the respective MISO bus will just be zeroed and the address space will be vacant.

Analogously, the TCM blocks can be omitted by setting their size generic to zero. If omitted the address space is left vacant and the MISO bus is zeroed.

Note that the code TCM is reachable from both the code and data spaces. This is meant to enable the SW to read constants without resorting to special load scripts and code loading tricks. It will also enable the SW to initialize the code TCM from an external source.

In case of access conflict, a data read or write cycle will be performed with no delay penalty whereas a code fetch will be delayed for one cycle.

The location of the TCM within the addressing spaces is configurable by way of generics but can't be changed on runtime.

Cache Wishbone Ports

Each of the caches is the master of a Wishbone bus that it uses for refills. The precise behavior of this WB port is still to be defined – we might implement burst reads or some other features beyond the basic pipelined cycles. What´s pretty sure is that the WB buses need only implement pipelined accesses since they are meant for interfacing to internal peripherals – such as memory controllers – and those are expected to be synchronous.

At any rate, the WB master functionality is supposed to be implemeted within the cache itself so no need for a WB bridge is anticipated here.

The feature set supported by this port will be fully described in the core datasheet, since it is part of its external interface. No need to do so here.

Direct Wishbone Port

There is a third WB port through which the CPU can access non-cached devices bypassing the caches. The core uses a ION-to-Wishbone bridge to provide this port.

The CPU does not support bursts in the ION bus so the WB bridge does not need to support them either. On the other hand, it will have to support cycle retrials, if the standard mandates them. The features of this port are left to be specced in a later revision of this document.

The datasheet will describe the functionality of this port, since it is part of the core external interface.

The above description is sketchy and incomplete, to be fleshed up in a later revision.

PC logic

Figure 6 shows a simplified diagram of the PC update logic. This logic handles the PC increments and the regular (non-exception) jumps and branches.

In the diagram, rectangles represent registers whereas green and yellow pointed shapes represent connections to other parts of the core.

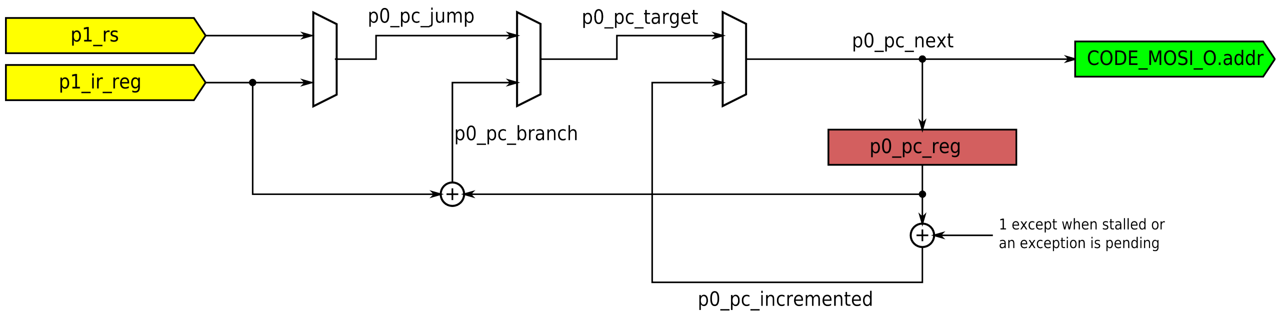


Figure 7: ***PC Update Logic***

As was to be expected, the Pc logic is a plain loadable counter. The low 2 bits of PC are not stored in the 30-bit **p0_pc_reg** signal, so the conuter increments by 1, not 4.

When no exception is going on, signal **p0_pc_reg** is unconditionally loaded with the new value; but the increment itself is conditional – the value added will be zero if the pipeline is stalled or an exception is pending. This is a slightly counterintuitive implementation detail that should be simplified.

The only omission in this diagram is that it does not include the logic that generates the EPC value – the value stored in EPC upon exceptions. This logic is slightly different from that of **CODE_MOSI_O.addr** and will be included in a final version of this document.

Also omitted is the logic that loads **p0_pc_reg** with the exception vector or the exception return value; again, this remains to be done.

3.- Instruction Execution

In order to illustrate the internal working of the CPU, we will explain nw the execution of a few representative instructions step by step as they traverse the pipeline.

All the instructions used as examples have been taken from a run of the opcodes test with the ion_cpu_tb.do simulation script. The execution context can be found in the program listing file.

Note that the listing uses the conventional register mnemonics, whereas in the examples the register index is used instead, for clarity.

BFC02A0: 00641020 ADD r2, r3, r4

This instruction is one of the simplest ones; all it does is read two registers from the register file, add them and write the result into a third register. No exceptions will be

In the waveform diagram 6, you can see that the simulation has been run with 3 wait cycles per code space memory access. Therefore, each pipeline stage takes 4 clock cycles instead of one.

Execution commences when the address for the instruction is placed in CODE_MOSI.addr (1). As you see, the address is put on the bus one cycle before the PC register p0_pc_reg is updated in edge (3). Note that the Pc register is “shifted left” by 2 in the chronogram, because it´s missing the two LSBs.

The code memory puts the opcode on CODE_MISO.rd_data after CODE_MISO.mwait is deasserted, at (9).

At this point, the register bank read address signals for ports Rs and Rt, p0_rs_num and p0_rt_num, take their values directly from CODE_MISO.rd_data – they do not take their values from the IR register p1_ir_reg.

This means that the register bank, implemented with synchronous memory, can be read at the same time the IR is loaded at (11), which saves one pipeline stage.

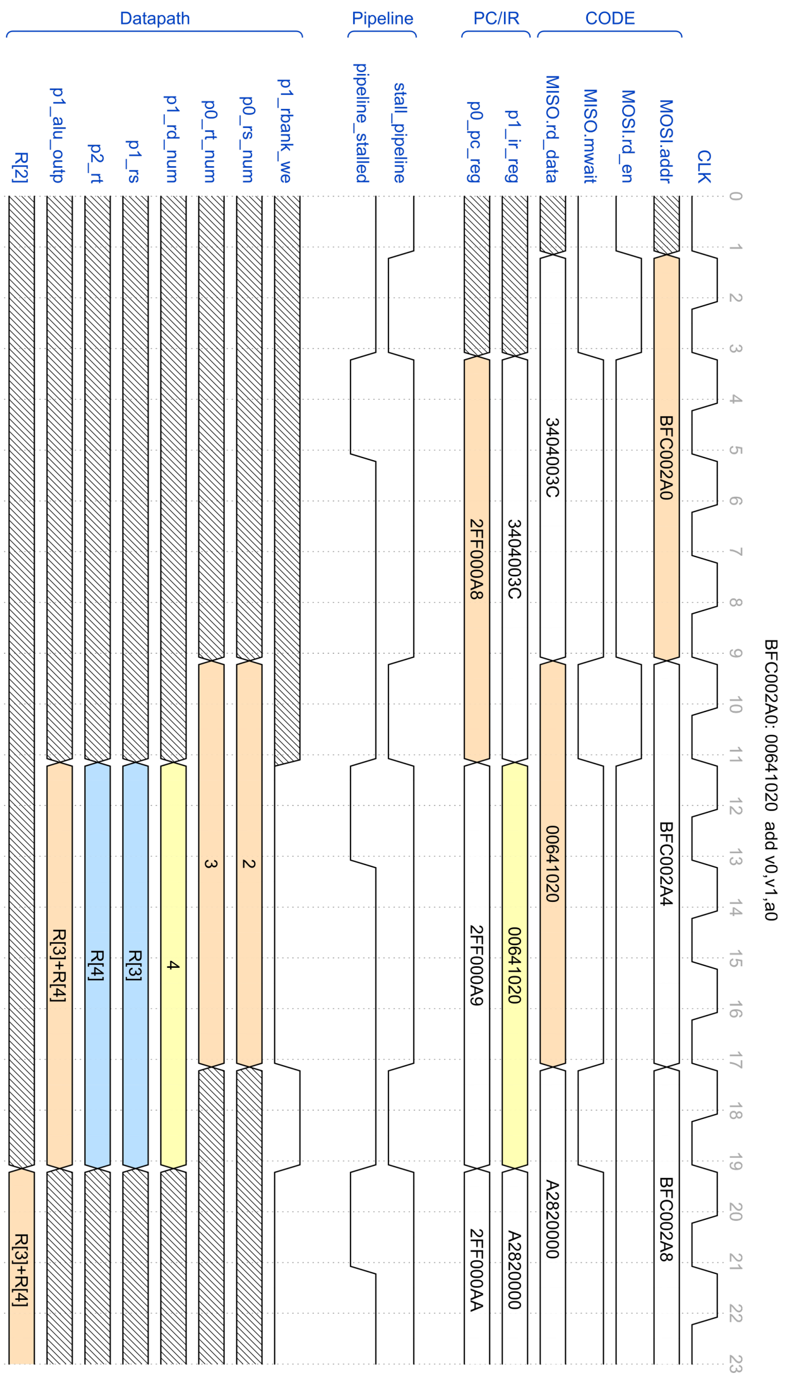
So from edge (1) to edge (11) the instruction is in its first pipeline stage.

At edge (11) the register outputs for Rs and Rt, p1_rs and p1_rt, take their correct values. At the same time, the alu control signal p1_ac, and other datapath control signals, take their values from the IR. The ALU does its work between edges (11) and (13) – the ALU must complete its work in a single cycle even if the pipeline is stalled, the core does not have any multicycle path..

The second stage of the pipeline for this instruction takes from edge (11) to edge (17); as explained, it only takes 4 clock cycles because of the 3 delay cycles of every code fetch in this simulation.

Signals stall_pipeline and its delayed version pipeline_stalled control the stalling of the pipeline, as can be imagined. Each one controls a different part of all the stages of the pipeline and their behavior is arguably the trickiest part of the system. They will be explained carefully in a later version of this document.

Finally, at edge (17), the instruction completes its third pipeline stage by updating the Rd register – as enabled by the register bank WE p1_rbank_we. The new value is available for the next instruction because the register bank implements “data forwarding” when necessary.



Waveform Diagram 8: ***Execution of “ADD” instruction***