ION -- free, open source MIPS32r2 compatible CPU core

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.

FEATURES

- (To Be Done).

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 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.

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 (for two clock cycles).

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 (for two clock cycles).

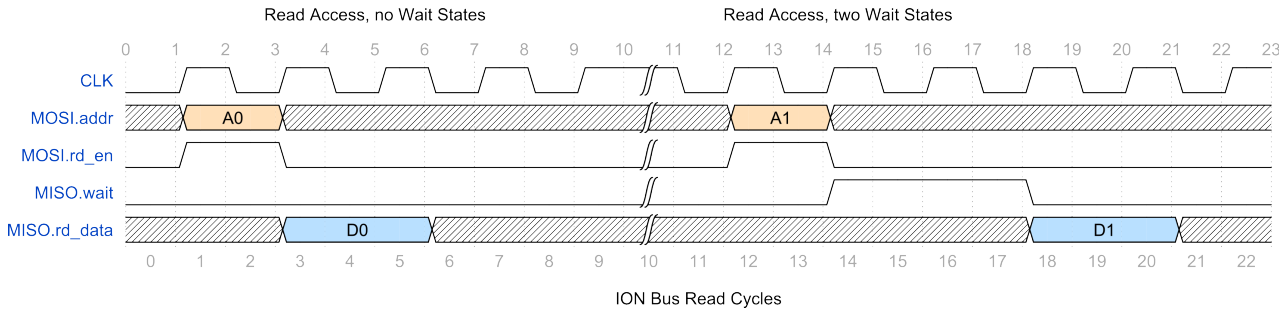


Figure 2: ***ION Bus Read Cycles***

Note that the bus slave must hold **MISO.rd_data** valid for two clock cycles, otherwise the CPU might miss it.

When the CPU detects a load hazard (a load to register R_x in instruction N followed by a reference to register R_x in instruction N+1) the pipeline is stalled for one cycle so that instruction

N+1 uses the correct, pre-load value for register R_x– this is a *load interlock*. When that happens, the bus slave must hold **MISO.rd_data** valid for an extra cycle.

Since there is no way to know the pipeline is stalled for a load hazard, the slave must do this for all read transactions. In effect, the pipeline has been simplified at the expense of the bus slave.

Both CPU buses use the same cycles, but the code bus does not have interlocks; the code bus does not need its rd_data input valid for two cycles but only one though for generality the above specs should be used for both buses until a refactor is done that fixes the undue complications.

In the present implementation of the CPU, the hazard detection logic (signal **load_interlock**) has been commented out so the pipeline is interlocked for all load instructions. The logic is there but it has not been tested, that´s why it is commented out.

Write Cycles

Write cycles are more straightforward:

- 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 slalled until this edge.

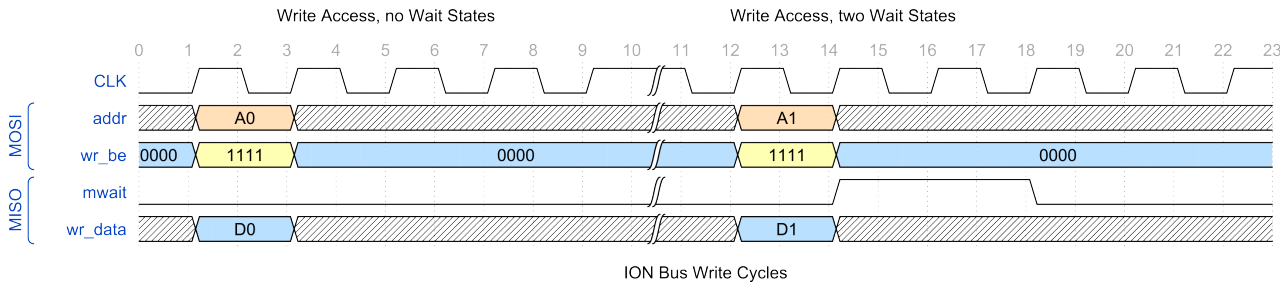


Figure 3: ***ION Bus Write Cycles***

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.

Note that the CPU in its present state does not support unaligned word writes or reads – it relies on traps for those operations. These operations are to be implemented in the new version of the core.

Pending refactors for the new version of the CPU

This read cycle interlock stuff is something that unduly complicates the bus and should be fixed before going on with the new version of the CPU.

Specifically, the following refactors are necessary before starting the development of a new cache:

1. Add a control signal indicating an interlock so the bus slave knows is has to drive **MISO.rd_data** for an extra cycle.
2. Alternatively, implement a fix for the interlock problem transparent to the bus slave.
3. Implement a properly decoded internal interlock signal (uncommenting the interlock logic), with a minimal test in the *opcodes* program.

Other refactors that would be desirable and need no radical changes to the core:

1. Unaligned loads and stores shoud be implemented.
2. The CPU might not stall in write cycles with wait states: it might go on with its business while the write operation is completed, and only stall if it has to access data memory while MISO.mwait is asserted.

2.- Structural Description

There should be here a block diagram of the core with some emphasis on the tricky parts (the PC logic). **TBD**.

PC logic

Figure 5 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.

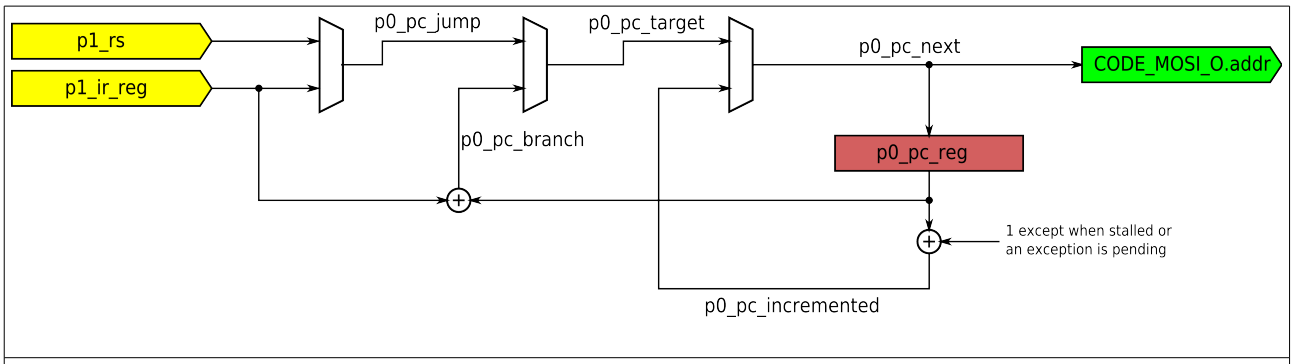


Figure 5: ***PC Update Logic***

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.

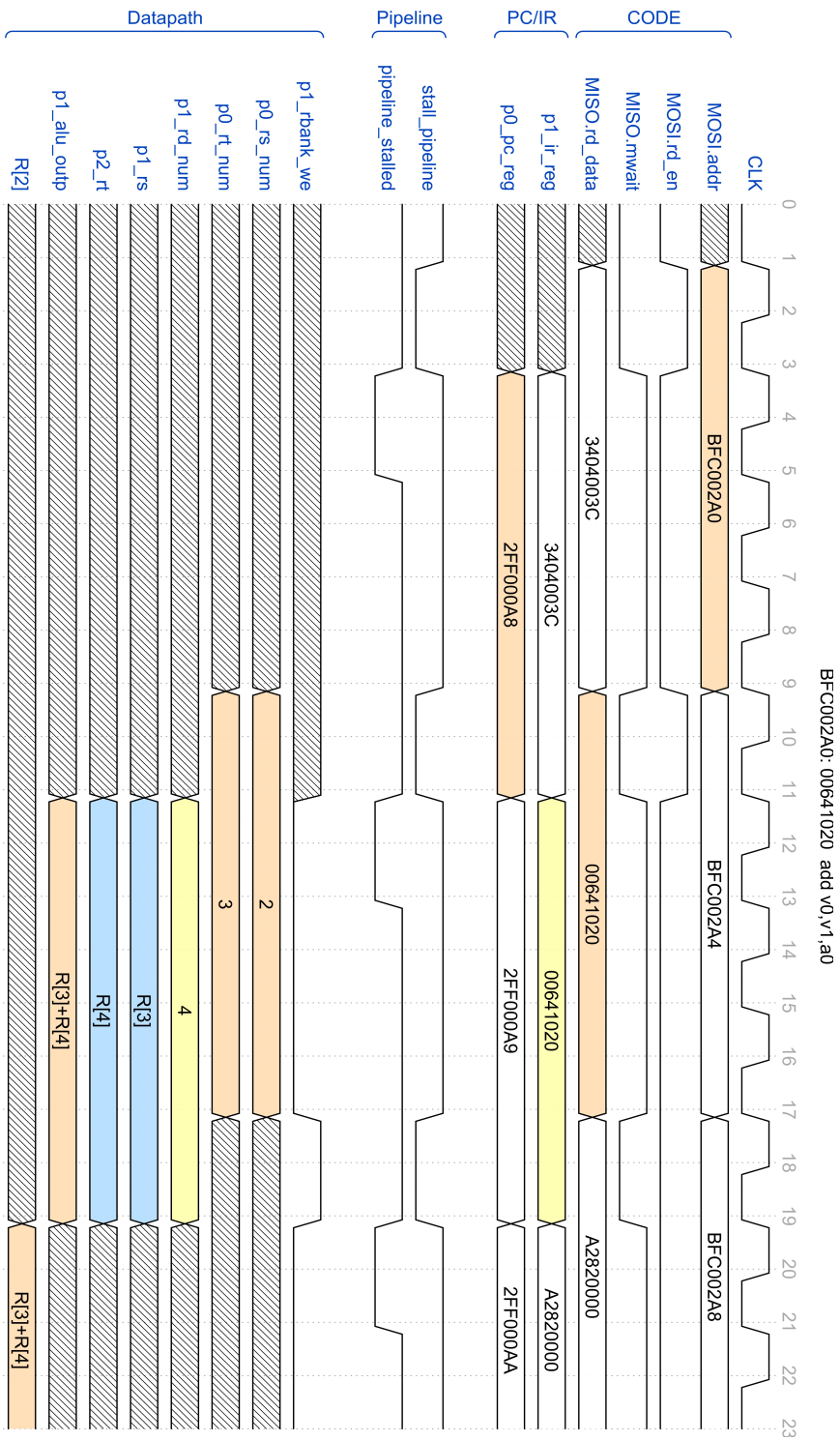
So from edge (1) to edge (11) the instruction is in its first pipeline stage.

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 6: ***Execution of "ADD" instruction***