mor1kx IP core specification

REVISION HISTORY

NUMBER	DATE	DESCRIPTION	NAME

Contents

Chapter 1

Introduction

This document describes the mor1kx processor block and its various configurable components. The core is an implementation of an OpenRISC 1000 compliant processor which is highly configurable and contains multiple pipeline implementations along with a configurable set of peripherals such as caches, timers, debug and bus interfaces. This documentation will contain a section detailing each optional block such as pipeline, caches, bus interface etc. It can outline as much implementation detail as it wants but should ultimately provide information to users on how to program and use the peripheral, or what to expect during execution of software on a particular configuration.

Document Status

This documentation is far from complete. It is expected that the documentation will become more extensive over time.

1.1 OpenRISC Family

OpenRISC 1000 is architecture for a family of free, open source RISC processor cores. As architecture, OpenRISC 1000 allows for a spectrum of chip and system implementations at a variety of price/performance points for a range of applications. It is a 32/64-bit load and store RISC architecture designed with emphasis on performance, simplicity, low power requirements, scalability and versatility. OpenRISC 1000 architecture targets medium and high performance networking, embedded, automotive and portable computer environments.

1.2 mor1kx CPU

The mor1kx implementation was developed in order to provide a better platform for processor component development than previous implementations. The goal of the implementation is to provide a greater level of flexibility in terms of implementation trade-offs such as area and performance.

The blocks within the core have been designed for maximum re-use within different configurations. Based on this, different pipeline implementations are a major focus of the core. With this, the core should be very useful for developers and users alike. For developers as a base for either high-performance or low-overhead pipeline implementations based on re-usable components eg. decode and ALU block. For users as a wider variety of capabilities should be available to suit the processor's use case.

Chapter 2

Implementation Overview

The implementation is heavily modular, with each particular functional block of the design being contained within its own Verilog module or modules.

The implementation configuration makes use of Verilog parameters. There should be no configuration performed with the use of Verilog defines.

2.1 Hierarchy

The top few levels of hierarchy are as follows

mor1kx
Top-level, instantiatng bus interfaces and CPU top-level

- mor1kx_bus_if_xx - Bus interface, depending on desired bus standard
- mor1kx_cpu - Pipeline implementation wrapper
  - mor1kx_cpu_xx - Pipeline implementation, depending on configuration
    - mor1kx_fetch_xx - Pipeline-implementation-dependent fetch stage
      - mor1kx_icache - Instruction cache implementation
      - mor1kx_immu - Instruction memory management unit implementation
    - mor1kx_decode - Generic decode stage
    - mor1kx_execute_alu - Generic ALU for execute stage
    - mor1kx_lsu_xx - Pipeline-implementation-dependent load/store unit
      - mor1kx_dcache - Data cache implementation
      - mor1kx_dmmu - Data memory management unit implementation
    - mor1kx_wb_mux_xx - Pipeline-implementation-dependent writeback stage mux
    - mor1kx_rf_xx - Pipeline-implementation-dependent register file
    - mor1kx_ctrl_xx - Pipeline-implementation-dependent control stage, usually containing features such as tick timer, interrupts etc.

The above hierarchy is not always the same for each pipeline implementation (from *mor1kx_cpu_xx* level down) but as a good indicator as to how the existing pipeline implementations have been structured.

2.2 Coding Style

The coding style is a relatively simple Verilog style and should be adhered to for any future development. All of the following detail how the existing code base has been implemented, and it should be realtively easy to continue in that style, but if in doubt, the following details some of the rules:

- module names should be prefixed with *mor1kx* and be in lowercase and stored in a source file of the same name plus the standard verilog suffix, *.v*.
- port names should have their direction abbreviation appended to the name
  - E.g the instruction bus address output should be ibus_adr_o
  - The only exceptions are the clock and reset ports, commonly referred to as *clk* and *rst* in the Verilog, and don't really require direction suffixes. The only exception is if a clock or reset generation module is outputting one of these, in which case it should have _o appended, otherwise for clock and reset inputs it can be inferred that they are inputs from the absence of _o.
- Parameter names should be in UPPERCASE, with parameter values, if strings, also in uppercase.
- Verilog reg and wire names should be named usefully, in lowercase, and make use of underscores in the name.
- There should be no reason to alter the mor1kx-defines.v or mor1kx-sprs.v files unless it is expanding upon or fixing what is already there. *No new class of defines controlling implementation should be added* (use parameters instead!) Those files are only for storing architectural information in Verilog format, which can then be used in the design.

There has been heavy use of the Emacs Verilog mode auto-implement and wire features throughout the code. This is a useful tool and it would be useful to stick to using this for future pipeline implementations, to speed up the tedious task of wiring up new blocks.

2.3 Architecture compliance

The mor1kx aims to be fully OpenRISC 1000 compliant. It implements the VR2, AVR and ISRs (still largely unused). The current OR1K architecture version supported is 1.0.

2.3.1 Version Register 2 definition

Bits 23:0 of the VR2 are implementation-specific. They are defined, for mor1kx, to be 3 bytes, interpreted as 3 unsigned 8-bit integers, as the following:

- 23:16 - Major version number
- 15:8 - Minor version number
- 7:0 - Pipeline identifier
  - Where this pipeline identifier means
    - 1 - Cappuccino
    - 2 - Espresso
    - 3 - Pronto Espresso

However, the mor1kx does not implement the following at present:

- arithmetic exception control/status registers (AECR, AESR)
- l.muld[u] and thus access to full 64-bit result as there is no MAC unit

Part I

CPU Components

This section will outline each of the CPU component modules.

Chapter 3

CPU Pipeline Implementations

3.1 Available Implementations

At present the following pipeline implementations are available. These combine various of the components, outlined in a following section, to implement the bulk of the processor.

- cappuccino - 6 stage, single issue, delay slot, debug unit, timers, PIC, tightly coupled cache, MMU, ORBIS32 support
- espresso - 2 stage, single issue, delay slot, debug unit, timers, PIC, ORBIS32 support
- pronto espresso - 2 stage, single issue, no delay slot, debug unit, timers, PIC, ORBIS32 support

3.2 Cappuccino pipeline

A 6 stage pipeline. (address, fetch, decode, execute, control/memory and writeback)

Caches supported (optional).

MMUs supported (optional).

It has a delay slot on jump and branch instructions.

It features the EVBAR.

Pipeline consists of the following modules:

- Fetch:
  - mor1kx_icache
  - mor1kx_immu
  - mor1kx_fetch_cappuccino
- Decode
  - mor1kx_decode
- Execute stage
  - mor1kx_execute_alu
  - mor1kx_wb_mux_cappuccino
  - mor1kx_rf_cappuccino

  - mor1kx_execute_ctrl_cappuccino
- Control/memory stage
  - mor1kx_dcache
  - mor1kx_dmmu
  - mor1kx_lsu_cappuccino
  - mor1kx_ctrl_cappuccino
  - mor1kx_ctrl_branch_cappuccino

The following sections outline the pipeline-specific modules

3.2.1 mor1kx_ctrl_branch_cappuccino

This is the pipelines branch control unit, selecting the jump/branch address and opcode input from execute stage, with flag input from control stage. Indication of whether a branch needs to be evaluated (based on flag) comes from the control stage.

The block then outputs the appropriate indication of whether a branch is going to occur and the target address to the fetch stage.

It is wholly combinatorial.

3.2.2 mor1kx_ctrl_cappuccino

This module contains a lot of the core functionality of the pipeline, such as:

- SPRs (NPC, PPC, etc.) and accesses to them
- PIC
- Debug unit
- Tick timer
- Pipeline control signals
  - Advance/stall signaling to each pipeline stage
  - Exception handling

It's in a big monolithic file but perhaps things like the PIC and tick timer could be split out and made genering among pipeline implementations.

3.2.3 mor1kx_execute_ctrl_cappuccino

Determine the status of execute-stage units in play (ALU or LSU) and:

- Determine when they're done
- Control the write-enable to the register file for any result
- Correctly propagate exception signals from either execute or earlier stages

3.2.4 mor1kx_fetch_cappuccino

Fetch stage, tightly coupled with instruction cache.

3.2.5 mor1kx_lsu_cappuccino

Load/store unit.

Performs accesses of the generic bus which may or may not be then going to a data cache, and finally out onto the bus via the selected bus interface.

Is 32-bit specific.

Combinatorial outputs to pipeline control logic to reduce latency, may introduce long paths effecting timing, though.

Handles sign extension if load/store requires it. Generates alignment exception, and handles bus error exception back to the mor1kx_execute_ctrl_cappuccino module.

3.2.6 mor1kx_rf_cappuccino

Register file for the pipeline. 2 lots of 32 general purpose registers (GPRs.)

Handles forwarding from control/memory and writeback to execute stage.

Instantiates a RAM for each of the two register files (*mor1kx_rf_ram* module.)

3.2.7 mor1kx_wb_mux_cappuccino

Writeback stage mux. Inputs are ALU result, LSU result, SPR value for l.mfspr instruction. Generates link address for jump-and-link instructions.

3.3 Espresso pipeline

The espresso pipeline essentially contains two stages: a fetch and "the-rest" stages. There is no registering in the decode stage, so the register outputting the fetched instruction from the fetch stage is what is used for the remainder of that instruction's processing.

No support for caches.

It has a delay slot.

Supports DSX bit in SR.

Pipeline consists of the following modules:

- Fetch:
  - mor1kx_fetch_espresso
- Decode
  - mor1kx_decode
- Execute/memory stage
  - mor1kx_execute_alu
  - mor1kx_lsu_espresso
  - mor1kx_wb_mux_espresso
  - mor1kx_rf_espresso
- Control stage
  - mor1kx_ctrl_espresso

The following sections outline the pipeline-specific modules

3.3.1 mor1kx_fetch_espresso

This is the fetch stage for the espresso pipeline.

It is tightly coupled with the control stage.

The block attempts to stream in bursts from the bus interface.

The block outputs register addresses for the next read from the RF.

3.3.2 mor1kx_lsu_espresso

A LSU specific to the espresso pipeline. Its features are similar to the mor1kx_lsu_cappuccino block.

3.3.3 mor1kx_wb_mux_espresso

Writeback stage mux for the espresso pipeline. Similar to the mor1kx_wb_mux_cappuccino block.

3.3.4 mor1kx_rf_espresso

Register file for the espresso pipeline. Simlar to the mor1kx_rf_cappuccino block.

3.3.5 mor1kx_ctrl_espresso

This module contains a lot of the core functionality of the pipeline, such as:

- SPRs (NPC, PPC, etc.) and accesses to them
- PIC
- Debug unit
- Tick timer
- Pipeline control signals
  - Advance/stall signaling to each pipeline stage
  - Exception handling
  - Branch indication to fetch stage
  - Register file write enable

3.4 Pronto Espresso pipeline

The pronto espresso pipeline essentially contains two stages: a fetch and "the-rest" stages. It is based on the Espresso pipeline, but does not have a delay slot on jumps and branches. As such, it reuses a lot of Espresso's pipeline.

No support for caches.

It has *no* delay slot.

Pipeline consists of the following modules:

- Fetch:
  - mor1kx_fetch_prontoespresso

- Decode
  - mor1kx_decode
- Execute/memory stage
  - mor1kx_execute_alu
  - mor1kx_lsu_espresso
  - mor1kx_wb_mux_espresso
  - mor1kx_rf_espresso
- Control stage
  - mor1kx_ctrl_prontoespresso

The following sections outline the pipeline-specific modules

3.4.1 mor1kx_fetch_prontoespresso

This is the fetch stage for the pronto espresso pipeline

It is tightly coupled with the control stage.

The block attempts to stream in bursts from the bus interface.

The block outputs register addresses for the next read from the RF.

It takes into account that the pipeline does not support a delay slot on jumps/branches.

3.4.2 mor1kx_ctrl_prontoespresso

This module contains a lot of the core functionality of the pipeline, such as:

- SPRs (NPC, PPC, etc.) and accesses to them
- PIC
- Debug unit
- Tick timer
- Pipeline control signals
  - Advance/stall signaling to each pipeline stage
  - Exception handling
  - Branch indication to fetch stage
  - Register file write enable

It is based on the espresso pipeline, however is modified in such a way as to make sure it doesn't have a delay slot on branches.

Chapter 4

Components

4.1 mor1kx_bus_if_wb32

This module is a Wishbone bus interface block and sits between the pipeline's fetch and load/store units and the Wishbone bus.

It is Wishbone version B3 compliant and can perform burst reads.

At present there are two configurations of the block, one is "classic" configuration and the other is "B3 read bursting".

The "classic" configuration performs all access as single cycle reads or writes to the bus.

The "B3 read bursting" mode will perform burst reads over the bus, but writes are still single cycle accesses.

4.2 mor1kx_cpu

The CPU pipeline wrapper layer. This selects the appropriate pipeline CPU implementation toplevel.

Additionally, some signals intended to be used as hooks for monitor modules are provided.

4.3 mor1kx_decode

This is a generic OR1K decode stage module, which can ideally be reused by each CPU implementation.

The module can either register the its decode output or be wholly combinatorial.

It generates ALU, LSU and control operation signals for the remainder of the pipeline units. Exceptions caused in this stage (ie, illegal instruction, system call etc.) or earlier (fetch stage exceptions like bus error) are also generated or passed through.

4.4 mor1kx_execute_alu

This is a generic ALU implementation. It contains all of the integer arithmetic and logical operations which are supported in the ORBIS32 instruction set.

The following features are optional. All can be disabled, or enabled with the implementation options listed, if any:

- Multiplier
  - Three stage, three cycle, full 32-bit parallel multiplier
  - Serial, 32-cycle serial multiplication implementation

  - Simulation, single cycle multiplication, not advisable for synthesis
- Divider
  - Serial, 32-cycle serial division implementation
  - Simulation, single cycle division, not synthesisable
- Shift-right-arithmetic
- Rotate right
- Shift instructions, logical shift left and right, and shift right arithmetic and rotate right can be chosen to be implemented in a single-cycle barrel shifter implementation or done serially to save implementation area.
- Conditional move
- Find first and last *1*

The following is not yet supported:

- Add with carry
- Sign extension instructions

The module also implements comparison logic for the set flag instructions.

4.5 mor1kx_icache

Instruction cache module.

4.6 mor1kx_dcache

Data cache module.

4.7 mor1kx_immu

Instruction memory management module.

4.8 mor1kx_dmmu

Data memory management module.

4.9 mor1kx_rf_ram

4.10 mor1kx_spram

Generic single port ram with seperate read and write addresses.
Has explicit bypass logic to correctly present write-first behaviour on different platforms.

4.11 mor1kx_dpram_sclk

Generic single clocked dual port ram.

Chapter 5

Index