The ZAP Processor

User Manual and Design Document

NOTE : The cache and MMU are both currently experimental. I am working on testing them.

12/1/2016
Document Version 0.0.1 DRAFT
Revanth Kamaraj

CACHE/MMU EXPERIMENTAL STATUS:
IF YOU INTEND TO USE THE CORE FOR SERIOUS PROJECTS, I RECOMMEND THAT YOU DO NOT USE THE CACHE AND MMU FEATURES AT THIS TIME SINCE THEY ARE EXPERIMENTAL. I WILL REMOVE THIS WARNING ONCE I AM CONFIDENT ABOUT THE CACHE/MMU SYSTEM.

MIT License

Trademarks

ARM is a registered trademark of Acorn RISC Machines.

1 INTRODUCTION TO ZAP

1.1 Overview

ZAP is an open source ARM v4T processor core with support for cache and MMU intended to be used on FPGA. **The cache and MMU section of design as it stands is in a *very experimental* state.**

1.2 Features

- Synthesizeable ARM core designed for FPGA and described entirely in Verilog – 2001.
- Note that the processor always operates in **Little Endian** mode i.e., lower bytes are at a lower address in relation to more significant bytes within a word.
- Binary compatible with version 4 of the ARM instruction set and version 1 of the Thumb instruction set.
- Built around a high performance 8 stage pipeline. The deeper pipeline allows for the core to meet higher frequencies.
- The pipeline has operand feedback to allow for high instruction throughput and reduce pipeline stalls. Operands are fed-back into the issue and shift stage to allow back to back execution without stalling. A memory accelerator allows memory results to be forwarded a cycle early.
- (**Experimental**) Configurable split I and D caches that map to FPGA block RAM. Both caches must be configured to the same size. Cache coherence is not implemented. Cache may be configured using CP15 instructions **MCR** and **MRC**.
- **(Experimental)** The MMU is v4 compatible except for the partial TLB purge function which reverts to an entire TLB flush (although this does not functionally affect software, it hurts performance). Features split I and D TLBs whose depth may be configured. The TLBs map to block RAM. MMU may be configured using CP15 instructions **MCR** and **MRC**.

1.3 Clocks

Table 1. Clocks and Resets

Clock Name	Reset Pin	Apply On Port	Frequency	Description
Register Clock.	**I_RESET**	**I_CLK_2X**	$2f$	Drives the register block in the core.
Core Clock.	**I_RESET**	**I_CLK**	f	The base clock

				that drives most of the logic in the core.

Table 1 shows the clocks and resets used by the processor. The reset signal **I_RESET** is passed through an internal reset synchronizer to generate a **RESET_SYNC** signal that is used as a synchronous reset throughout the core. You should design your system in such a way that resets continues to be asserted until at least 1 cycle after both clocks have stabilized and should deassert synchronously with respect to the rising edge. Note that the clocks you generate on the FPGA should look exactly as shown in the figure below. The reset signal must be high for at least as long as that shown in the figure below. **Ensure that the two clocks are edge aligned.**

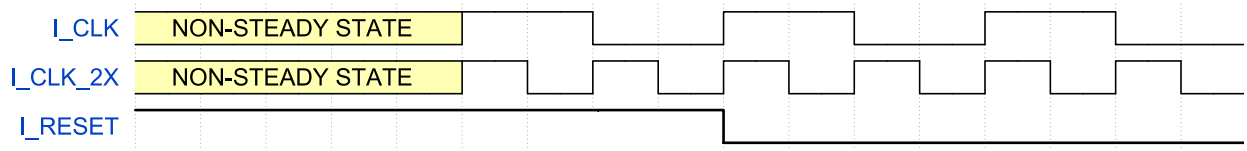

Figure 1. Clock Relation and Reset Deassertion

1.4 Ports

Table 2. Port List (Top Level)

Port Name	IO	Description	Comments
I_CLK	I	Core Clock.	
I_CLK_2X	I	Register Clock.	
I_RESET	I	Reset.	Active High Reset. Internally synchronized.
I_IRQ	I	Level Sensitive Interrupt.	Active High.
I_FIQ	I	Level Sensitive Fast Interrupt.	Active High.
O_DRAM_WR_EN	O	Data memory Write Enable.	
O_DRAM_RD_EN	O	Data memory Read Enable.	
O_DRAM_DATA[31:0]	O	Data memory Write Data.	
O_DRAM_BEN[3:0]	O	Data memory Byte Enable	Used only for writes.

O_DRAM_ADDR[31:0]	O	Data memory access address.	
I_DRAM_DATA[31:0]	I	Data from Data memory.	
I_DRAM_STALL	I	Data memory Stall.	Indicates data RAM has stalled. Extends the current access cycle.
I_IRAM_STALL	I	Instruction memory Wait Qualifier.	Indicates that instruction RAM has stalled. Extends the current read cycle.
I_IRAM_DATA[31:0]	I	Instruction Input from Instruction memory.	
O_IRAM_ADDR[31:0]	O	Program Counter to address instruction memory.	

Table 2 shows the top level ports of the processor (note separate I and D memory interfaces).

1.5. External Memory Timing Interface

Memory timing diagrams are shown below to indicate how external memory should react. See the figure below for details. Note that in the figure, the port directions are w.r.t the processor.

- Memory writes occur as usual, provide address and data on the memory bus and indicate a write. A lack of a **STALL** indicates that the data will be clocked in on the upcoming rising edge. If a **STALL** is seen on the upcoming rising edge, keep the access signals constant to permit the memory to continue working.
- Memory reads are a bit different. Assuming you provide address, data and a read command on edge N, the memory should indicate the readiness to supply data on the next edge using the **STALL** signal within the current cycle itself. If you see a **STALL** on the rising edge of clock, keep the access signals constant to permit the memory to continue working.

Even if you have a **STALL**, you can cancel the operation and change the access signals, the memory system must simply abandon the access it was doing and focus on the current request and re-compute STALL once more.

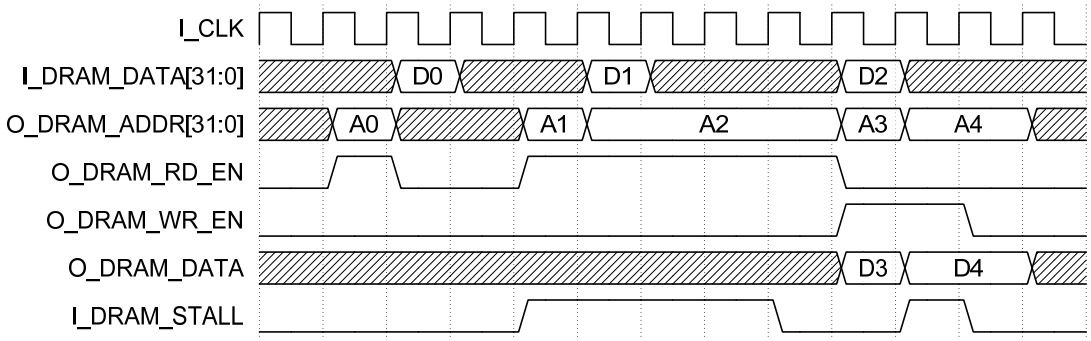

Figure 2. Memory Timing Diagram

WARNING: Ensure that external memory uses a single memory dual read port setup (i.e., both I and D channels operate on a common memory area – or at least appear to) to maintain coherence else expect a loss of coherence. Caches aren't coherent so JIT code may need to perform cache invalidation to guarantee coherence.

1.6. Basic Pipeline Structure

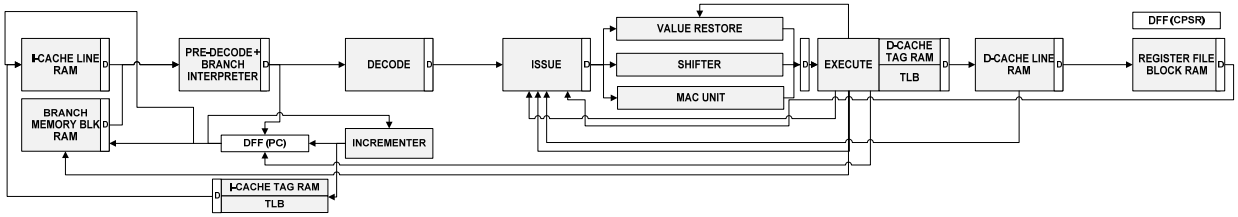

Figure 3. Basic Pipeline Structure

The figure above shows the basic pipeline structure of the processor. The longer pipeline allows for a faster processor. The branch predictor and the value restore unit compensate for the increased pipeline length by reducing the frequency of pipeline flushes and stalls respectively.

2 CONFIGURING THE CORE AND TESTBENCH

Throughout, it is assumed that **$ZAP HOME** points to the working (base) directory of the project.

2.1 Basic Setup

Core/testbench configuration may be done using defines. The defines file is located in **$ZAP HOME/includes/config.vh**. The table below shows the configuration defines.

Table 3. Bench and Core Configuration Defines

Define	Purpose	Required For	Comments
`THUMB_EN`	Enable Thumb v1 support.	Core	Enabling Thumb results in a larger and more sluggish core.
`IRQ_EN`	Testbench generates periodic interrupts.	Bench	
`SIM`	Generates extra messages and adds debug signals.	Bench	**DO NOT define this for FPGA synthesis.**
`VCD_FILE_PATH`	Set path to the VCD data dump.	Bench	
`MEMORY_IMAGE`	Set path of the memory image Verilog file.	Bench	
`MAX_CLOCK_CYCLES`	Set this to the number of cycles the simulation should run assuming no abnormal termination.	Bench	
`SEED`	Influences randomization. Provide a 32-bit number.	Bench	

Ensure that defines are defined using the syntax…

```
`ifndef `MYDEFINE
`define MYDEFINE
`endif
```

2.2. Advanced Setup (Experimental)

This section deals with setting memory parameters. Edit the parameters in the **mmu_config.vh** found in the **includes** directory. The parameters are listed below.

Table 4. Advanced Configuration Parameters (Experimental)

Parameter	Purpose	Required For	Comments
SECTION_TLB_ENTRIES	Set number of section TLB entries.	Core	
LPAGE_TLB_ENTRIES	Set number of large page TLB entries.	Core	
SPAGE_TLB_ENTRIES	Set number of small page TLB entries.	Core	
CACHE_SIZE	Set this to the size of the cache in bytes.	Core	Setting to 1024 gives a 1KB I and D cache, for example. Both the I and D cache that result will have the same size.

3 RUNNING SIMULATIONS

3.1. Run Sample Code

A sample **prog.s** and a sample **fact.c** file is present in **$ZAP HOME/sw/s** and **$ZAP HOME/sw/c** respectively. To translate them to binary and to a Verilog memory map, you can run the Perl script (See NOTE below).

```
perl $ZAP_HOME/debug/run_sim.pl
```

NOTE: Ensure you set all the variables in the Perl script as per the table below. Also ensure **config.vh** is set up properly. Often, you need to set up only **ZAP_HOME**. Data logs and value change dumps are sent to **/tmp** by script default values.

NOTE: The sample program essentially calculates the factorial of 5 (that is stored at byte address 5000 (DECIMAL)) and writes the result to byte address 5001. The 32-bit resulting value starting at location 5000 must be 0x00007805 where address 5000 contains 5, 5001 contains 0x78 and both 5002 and 5003 are 0x0.

Table 5. Perl Script Variables

Variable	Purpose
ZAP_HOME	Set this to the project base directory. Most paths are computed relative to this.
LOG_FILE_PATH	Set this to the place where you want the log file to be created.
ASM_PATH	Set this to the location of the startup assembly file.
C_PATH	Set this to the location of the C file.
LINKER_PATH	Set this to the location of the linker script.
TARGET_BIN_PATH	Set this to the target bin file location where it is supposed to be created.
VCD_PATH	Set this to where the VCD must be created.
MEMORY_IMAGE	Set this to the location where the memory image must be created (Verilog file).

3.2. Running Your Own Code

STEP 1: Generating a binary using GNU tools

You can use the existing GNU toolchain to generate code for the processor. This section will briefly explain the procedure. For the purposes of this discussion, let us assume these are the source files...

main.c
fact.c
startup.s
misc.s
linker.ld → *This is the linker script.*

Generate a bunch of object files.

```
arm-none-eabi-as -mcpu=arm7tdmi -g startup.s -o startup.o
arm-none-eabi-as -mcpu=arm7tdmi -g misc.s -o misc.o
arm-none-eabi-gcc -c -mcpu=arm7tdmi -g main.c -o main.o
arm-none-eabi-gcc -c -mcpu=arm7tdmi -g fact.c -o fact.o
```

Link them up using a linker script...

```
arm-none-eabi-ld -T linker.ld startup.o misc.o main.o fact.o -o prog.elf
```

Finally generate a flat binary...

```
arm-none-eabi-objcopy -O binary prog.elf prog.bin
```

The .bin file generated is the flat binary.

STEP 2 : Generating a Verilog memory map

Run the command,

```
perl $ZAP_HOME/scripts/bin2mem.pl prog.bin prog.v
```

The prog.v file looks like this...

```
mem[0] = 8'b00;
mem[1] = 8'b01;
…
```

STEP 3: Running Simulation

NOTE: Ensure **config.vh** is set up correctly (especially check that the memory image and VCD dump paths are good).

Your command must look like this (It is a single command)...

```
iverilog $ZAP_HOME/rtl/*.v $ZAP_HOME/rtl/*/*.v $ZAP_HOME/testbench/*.v
$ZAP_HOME/models/ram/model_ram.v -I$ZAP_HOME/includes -DSEED=22
```

The rtl/*.v and rtl/*/*.v collect all of the synthesizable Verilog-2001 files, the testbench/*.v collects all of the testbench (In this situation, the ram.v file is a part of the testbench).

Provide some seed value (22 is used in the example). Ensure you edit the config.vh file before running the simulation to correctly point to the memory map, VCD target output path etc for the simulator to pick up.

NOTE: Ensure you instantiate the ZAP processor and an external memory system in the testbench and initialize the memory to the contents of the Verilog memory image**.**