GitHub Page: www.github.com/krevanth/ZAP

The ZAP Processor

User Manual and Design Documentation

12/1/2016
LibreCores.org
Revanth Kamaraj (revanth91kamaraj@gmail.com)

WARNING

THE CACHE AND MMU SYSTEMS ARE STILL EXPERIMENTAL AND CAN BE UNRELIABLE. IT IS RECOMMENDED THAT YOU DO NOT USE THEM FOR YOUR APPLICATION AT THIS TIME. I WILL REMOVE THIS WARNING ONCE I AM CONFIDENT ABOUT THE CACHE/MMU SYSTEM.

Table of Contents

1. Introduction to ZAP

1.1. Introduction

ZAP is an open source soft processor core that is binary compatible with the ARM® version 4T instruction set that includes version 1 of the Thumb® ISA. The core is released under the MIT license. The processor features a deep 8-stage pipeline , a branch prediction unit and other enhancements that offer a significant performance boost. ZAP is specifically designed for FPGA and is coded in such a way that synthesis tools are able to map the RTL efficiently onto FPGA resources like block RAMs, DSP blocks etc.

1.2. Features

- **ARM Compatible.** ZAP is binary compatible with the ARM® v4T instruction set. This includes support for v1 Thumb® as well.
- **Cache and MMU support.** ZAP features split I and D caches and split memory management units. Note that the caches are not coherent. The size of the caches and TLBs may be configured before synthesis. All caches and TLBs map to FPGA block RAM.
- **Commonality with v4T Commercial Processors.** To retain commonality with commercial cores, cache and MMU controls are laid in exactly the same way and can be accessed using CP15 instructions.
- **Designed for FPGA.** The design maps efficiently onto FPGA resources like block RAMs and DSP multipliers.
- **High Performance.** ZAP is pipelined and thus, most instructions execute in a single clock cycle. The pipeline is 8 stages long and features branch prediction to reduce pipeline flushes. Enhancements have been made to reduce pipeline stalls as well.
- **Written in Verilog-2001.** ZAP is completely described in Verilog-2001 RTL. The older version of Verilog allows for greater flexibility among vendors.

1.3. Clocks and Resets

The processor uses rising edge triggered memory elements to hold state. These elements require a clock to function. The processor requires two clocks…

- A master clock, supplied to port **I_CLK** that drives most of the core logic.
- A register clock, supplied to port **I_CLK_2X** that drives the internal register file block RAM and is twice as fast as the master clock. The design requires the block RAM to have two write ports and thus, the write is divided into two phases within a master clock cycle with each phase accomplishing part of the register write operation.

The timing relation between the clocks must be very precise (clocks must be rising edge aligned) when in steady state as shown in *Figure 1. Clock Relations*.

1.3.1. Reset Signal

Shown below are the steps to reset the device reliably, note that **I_RESET** is the reset pin which is active high.

- When the clocks are not-steady, the value of **I_RESET** does not matter but it is recommended that you hold the reset high.
- On the first steady aligned rising edge, make **I_RESET** high.
- On any rising edge of the master clock, make **I_RESET** low. Make sure the deassertion is synchronous.

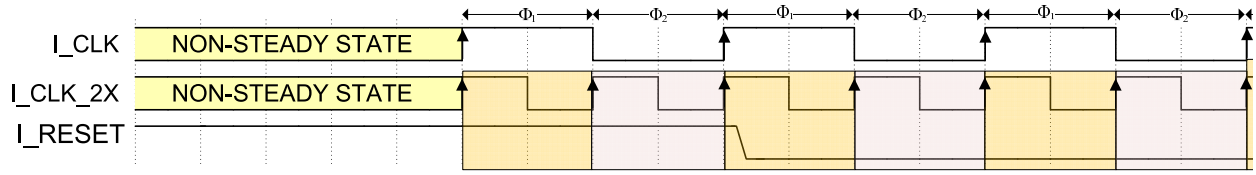

Figure 1. Clock Relations

2. Interfacing ZAP

2.1. Port List

Table 1. Port List shows the port list of the processor.

Table 1. Port List

Port	IO	Description
		CLOCKS AND RESETS
I_CLK	I	Core clock. Times most of the core logic. The register clock is basically this clock multiplied by 2.
I_CLK_2X	I	Register clock. Must be twice as fast as the core clock. Source synchronous with the core clock.
I_RESET	I	Reset. Active high reset. May be asserted asynchronously but must be deasserted synchronously.
		DATA CHANNEL SIGNALS
O_DRAM_DATA[31:0]	O	Data memory write data.
I_DRAM_DATA[31:0]	I	Data read from data memory.
O_DRAM_ADDR[31:0]	O	Data memory address. The lower 2 bits are always 0.
O_DRAM_BEN[3:0]	O	Data memory byte enables for write operations.
I_DRAM_STALL	I	Data memory stall signal. This is asserted within the same clock cycle to extend the access cycle over to the next clock cycle.
O_DRAM_WR_EN	O	Data memory write enable. **O_DRAM_DATA** is valid only when this is 1. The processor expects the data memory to clock in **O_DRAM_DATA** to address **O_DRAM_ADDR** on the upcoming core clock rising edge. If the memory is not in a position to do so, it must assert **I_DRAM_STALL** within the current cycle.
O_DRAM_RD_EN	O	Data memory read enable. When the processor asserts this, it expect the data memory to clock out data on **I_DRAM_DATA** port on the upcoming core clock rising edge. If the memory is not in a position to do so, it must assert **I_DRAM_STALL** within the current cycle.
		CODE CHANNEL SIGNALS
I_IRAM_DATA[31:0]	I	Instruction from code memory.
O_IRAM_ADDR[31:0]	O	Instruction fetch address. The lower bit of this is always 0.
I_IRAM_STALL	I	Instruction memory stall signal. This is asserted within the same clock cycle to extend the access cycle over to the next clock cycle.
O_IRAM_RD_EN	O	Instruction memory read enable. When the processor asserts this, it expect the instruction memory to clock out data on **I_IRAM_DATA** port on the upcoming core clock rising edge. If the memory is not in a position to do so, it must assert **I_IRAM_STALL** within the current cycle.

INTERRUPTS		
I_IRQ	I	Active high level sensitive IRQ request line. Hold high to request an IRQ service.
I_FIQ	I	Active high level sensitive FIQ request line. Hold high to request an FIQ service.

2.2. External Memory Timing

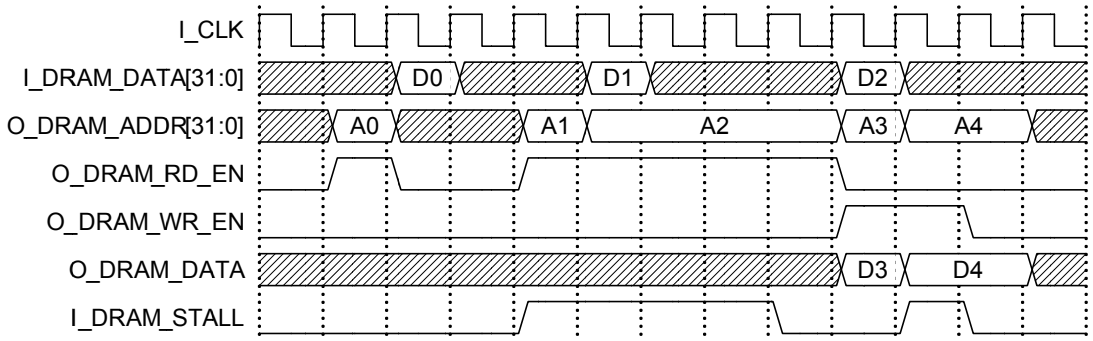

Figure 2. Data Memory Timing Sample

Figure 2. Data Memory Timing Sample shows an example timing waveform for data memory. The same can be applied to code memory except for the fact that code memory cannot be written to by the processor.

WARNING

ENSURE THAT THE EXTERNAL MEMORY CONTROLLER IS DESIGNED TO MAINTAIN COHERENCE BETWEEN CODE AND DATA CHANNELS. THIS CAN EASILY BE DONE BY ENSURING THAT CODE AND DATA CHANNELS ACCESS THE SAME PHYSICAL MEMORY SYSTEM.

3. Pre-Simulation/Synthesis Checklist

Throughout, it is assumed that **$ZAP_HOME** points to the root directory of the project.

3.1. File Hierarchy

You must find the following directory structure starting from **$ZAP_HOME**.

```
├── debug
│   ├── process_log.pl
│   └── run_sim.pl
├── docs
│   └── zap_doc.pdf
├── includes
│   ├── basic_checks.vh
│   ├── cc.vh
│   ├── config.vh
│   ├── cpsr.vh
│   ├── fields.vh
│   ├── global_functions.vh
│   ├── index_immed.vh
│   ├── instruction_patterns.vh
│   ├── mmu_config.vh
│   ├── mmu_functions2.vh
│   ├── mmu_functions.vh
│   ├── mmu.vh
│   ├── modes.vh
│   ├── opcodes.vh
│   ├── regs.vh
│   ├── sh_params.vh
│   └── shtype.vh
├── lib
│   ├── mem_ben_block.v
│   ├── mem_inv_block.v
│   ├── ram_simple.v
│   ├── reset_sync.v
│   └── sync_fifo.v
├── LICENSE.md
├── models
│   └── ram
│       └── model_ram.v
├── push_repo.pl
├── README.txt
├── rtl
│   ├── zap_alu
│   │   ├── alu.v
│   │   └── zap_alu_main.v
│   ├── zap_core.v
│   ├── zap_cp15_cb
│   │   └── zap_cp15_cb.v
│   ├── zap_decode
│   │   ├── zap_decode_main.v
│   │   └── zap_decode.v
│   ├── zap_fetch
│   │   └── zap_fetch_main.v
│   ├── zap_issue
│   │   └── zap_issue_main.v
│   ├── zap_memory
│   │   └── zap_memory_main.v
│   ├── zap_mmu
│   │   ├── zap_d_mmu_cache.v
│   │   └── zap_i_mmu_cache.v
│   ├── zap_predecode
│   │   ├── ones_counter.v
│   │   ├── zap_predecode_coproc.v
│   │   ├── zap_predecode_main.v
│   │   ├── zap_predecode_mem_fsm.v
│   │   └── zap_predecode_thumb.v
│   ├── zap_regf
│   │   ├── bram.v
│   │   ├── bram_wrapper.v
│   │   └── zap_register_file.v
│   ├── zap_shift
│   │   ├── mult16x16.v
│   │   ├── zap_multiply.v
│   │   ├── zap_shifter_main.v
│   │   └── zap_shift_shifter.v
│   └── zap_top.v
├── scripts
│   ├── bin2mem.pl
│   ├── do_it.pl
│   └── linker.ld
├── sw
│   ├── asm
│   │   └── prog.s
│   ├── c
│   │   └── fact.c
└── testbench
    └── zap_test.v
```

3.2. Core Configuration Parameters

Basic configuration of the core should be done using parameters as shown in *Table 2. Core Configuration Parameters*. These parameters may be edited in the **$ZAP_HOME/includes/mmu_config.vh**

Table 2. Core Configuration Parameters

Parameter	Purpose
CACHE_SIZE	Set the size of instruction and data caches. Both caches must be of the same size. Set this in bytes.
SECTION_TLB_ENTRIES	Set the number of section TLB entries required.
SPAGE_TLB_ENTRIES	Set the number of small page TLB entries required.
LPAGE_TLB_ENTRIES	Set the number of large page TLB entries required.

3.3. Core Configuration Defines

Primary core configuration defines are present in **$ZAP_HOME/includes/config.vh** header file (See *Table 3. Partial Defines in config.vh*). Since the header file is included in every module, you must define using the following syntax…

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

Table 3. Partial Defines in config.vh

Define	Purpose
THUMB_EN	Enables Thumb Decoder

3.4. Testbench Configuration Defines

Core/testbench configuration may be done using defines. The defines file is located in **$ZAP_HOME/includes/config.vh.** The table below (*Table 4. Bench and Core Configuration Defines*) shows the configuration defines.

Table 4. Bench and Core Configuration Defines

Define	Purpose	Comments
IRQ_EN	Testbench generates periodic interrupts.	
SIM	Generates extra messages and adds	*Do not define this when performing synthesis.*

	debug signals.
VCD_FILE_PATH	Set path to the VCD data dump.
MEMORY_IMAGE	Set path of the memory image Verilog file.
MAX_CLOCK_CYCLES	Set this to the number of cycles the simulation should run assuming no abnormal termination.
SEED	Influences randomization. Provide a 32-bit number.
STALL	The external memory stalls at arbitrary intervals.

Since the include file **config.vh** is included in every Verilog file, ensure that defines are defined using the syntax to avoid redefinition errors…

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

WARNING

ENSURE THAT SIM IS NOT DEFINED WHEN YOU ARE PERFORMING SYNTHESIS. IF YOU FIND THE SYNTHESIZER FALLIN G BACK TO FLIP-FLOP IMPLEMENTATION OF THE REGISTER FILE, YOU MAY HAVE LEFT SIM DEFINED. IF THAT HAPPENS, PLEASE UNDEFINE SIM AND TRY SYNTHESIS AGAIN.

4. Running Simulations

4.1. 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 the Perl variable, **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.