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)

NOTE
THE DESIGN IS IN AN EXPERIMENTAL STATE. THUMB IS NOT SUPPORTED.

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 4 instruction set (Thumb is not supported). The processor is equipped cache and virtual memory. 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® v4 instruction set.
- **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 v4 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.

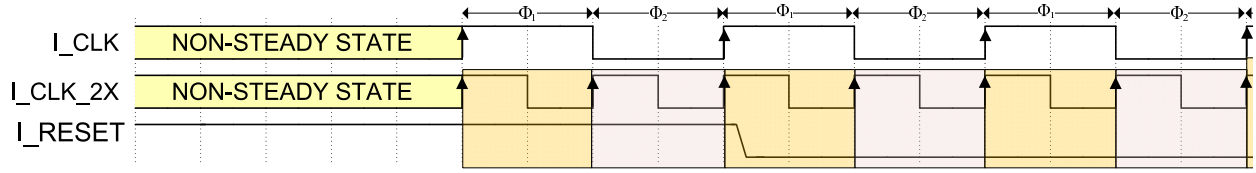

Figure 1. Clock Relations

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. When a memory is stalled, read enable has no effect on the output.
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. When a memory is stalled, read enable has no effect on the output.
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

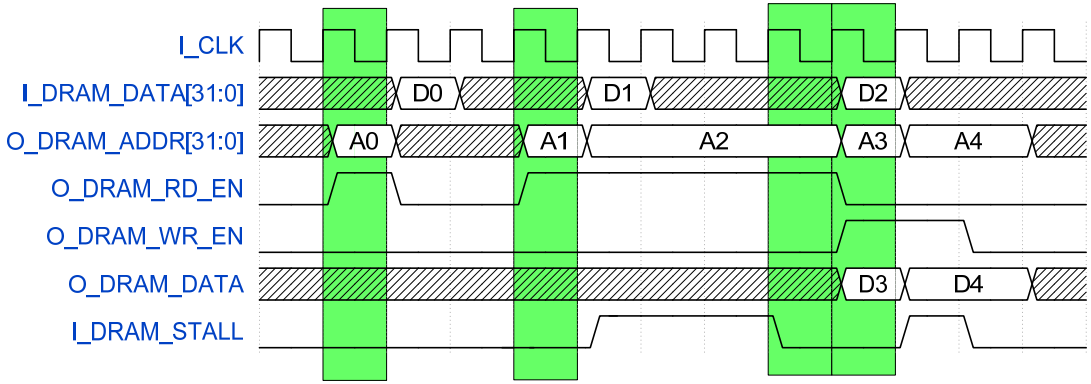

Figure 2. Data Memory Timing Sample

Figure 2. Data Memory Timing Sample shows an example timing waveform for data memory (the port directions are w.r.t the processor). The same can be applied to code memory except for the fact that code memory cannot be written to by the processor. The green areas indicate a valid address phase. A valid address phase in the context of a memory write indicates that the data present in the cycle will be clocked into memory on the upcoming rising edge. In the context of a read, a valid address phase indicates that the memory will produce output data (read data) on the upcoming rising edge.

Active address phases occur in the above diagram when the following condition is satisfied...

ACTIVE ADDRESS PHASE = (O_DRAM_RD_EN|O_DRAM_WR_EN) & !I_DRAM_STALL

NOTE
• 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. • MEMORY CHANNEL SIGNALS INCURR SOME ADDITIONAL TIME DELAY AFTER THE RISING CLOCK EDGE BEFORE STABILIZING. THIS IS BECAUSE THE SIGNALS PROPAGATE THROUGH SOME COMBINATIONAL LOGIC IN THE CACHE. I AM CURRENTLY WORKING ON MAKING THESE OUTPUTS REGISTERED FOR HIGHER PERFORMANCE.

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**.

```
.
├── 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_block128.v
│   ├── mem_inv_block.v
│   ├── ram_simple.v
│   ├── reset_sync.v
│   └── sync_fifo.v
├── LICENSE.md
├── push_repo.pl
├── README.md
├── 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_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
├── run
│   ├── bench_files.list
│   └── rtl_files.list
├── scripts
│   ├── bin2mem.pl
│   ├── do_it.pl
│   ├── linker.ld
│   ├── post_process.pl
│   └── run_sim.pl
├── sw
│   ├── asm
│   │   └── prog.s
│   └── c
│       └── fact.c
└── testbench
    ├── model_ram.v
    └── 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. Core Configuration Defines in config*.vh). Since the header file is included in every module, you must define using the following syntax...

Table 3. Core Configuration Defines in config.vh

Define	Purpose
CMMU_EN	Enable cache/MMU support.

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 Configuration Defines*) shows the configuration defines. If you do not intend to define a macro, it is better to specifically undefine it instead of assuming it undefined. Note that some defines need to just ***exist*** (Shown in ***bold-italics***) while some must have a specific definition.

Table 4. Bench Configuration Defines

Define	Purpose	Comments
IRQ_EN	Testbench generates periodic interrupts.	
SIM	Generates extra messages and adds debug signals.	*Do not define this when performing synthesis.*
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.	
STALL	The external memory stalls at arbitrary intervals.	
FORCE_I_RAND_CACHEABLE	Set random cacheability parameter for I-Cache.	*Do not define when performing synthesis.*
FORCE_D_RAND_CACHEABLE	Set random cacheability parameter for D-Cache.	*Do not define when performing synthesis.*
FORCE_I_CACHEABLE	Force every I-cache request to go through the cache subsystem and access main memory only if entry is not found. Useful to allow code cacheability when MMU is disabled.	*Do not define when performing synthesis.*
FORCE_D_CACHEABLE	Force every D-cache request to go through the cache subsystem and access memory only if entry is not found. Useful to allow data cacheability when MMU is disabled.	*Do not define when performing synthesis.*

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

NOTE

ENSURE THAT *SIM* IS NOT DEFINED WHEN YOU ARE PERFORMING SYNTHESIS. IF YOU FIND THE SYNTHESIZER FALLING 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 Provided Sample Code

Set an environment variable called **ZAP_HOME** to point to the base directory of the project. You must set this before proceeding.

To perform a quick test using the provided sample code, you may follow the steps in this section. To run your own code, please refer to the next section.

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/scripts/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 script invokes Icarus Verilog. Ensure that you have it installed.

NOTE: The sample program essentially calculates the factorial of 5 (that is stored at byte address 2000 (decimal)) and writes the result to byte address 2001. The 32-bit resulting value starting at location 2000 must be 0x00007805 where address 2000 contains 5, 2001 contains 0x78 and both 2002 and 2003 are 0x0. After factorial calculation, some multiplications are also performed. The code switches on cache and MMU and uses identity mapping using a section descriptor placed at 16KB.

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).
POST_PROCESS	Command that should be executed after simulation is complete and log files is generated.
RTL_FILE_LIST	Set this to point this to the RTL file list.
BENCH_FILE_LIST	Set this to point this to the testbench file list.

3.2. Running Your Own Code

Step 1: Generating a binary using GNU tools

You can use the existing GNU ARM 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 to generate an ELF file...

```
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 contains a set of lines describing the bytes at each memory address like this...

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

Step 3: Running a Simulation

NOTE: Ensure **config.vh** is set up correctly (especially check that the memory image and VCD dump paths are good). Assume that **$BENCH_LIST** points to the testbench *files list*.

NOTE: The testbench structure must contain an instance of the processor (zap_top) and some simulated memories connected to the processor to simulate a real system.

Your command must look like this...

```
iverilog -f $ZAP_HOME/run/rtl_files.list -f $BENCH_LIST -DSEED=22
```

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.