ZAP : An ARM v4T Compatible Soft Processor

Revanth Kamaraj (revanth91kamaraj@gmail.com)

October 23, 2016

MIT License

Contents

1 Introduction **5**
1.1 Features 5
1.2 About This Manual 6

2 Configuring the core and testbench **7**

3 Compiling Code and Running Simulation **8**
3.0.1 Generating a binary using GNU tools 8
3.0.2 Generating a Verilog memory map 8
3.0.3 Invoking the simulator 9
3.1 Run Sample Code quickly... 9

4 IO Ports of the Core **11**

5 Basic Description **14**
5.1 Register Mapping 15

List of Figures

1 High level view of the ZAP pipeline 14

List of Tables

1 Configuring the core and testbench 7
2 Variables in Perl script 10
3 IO Ports 13
4 Pipeline Stage Description 14
5 Register Mapping 16

1 Introduction

ZAP is an ARM® v4T compatible soft processor core. The code is fully open source and is released under the MIT license.

1.1 Features

- **Fully ARM v4T compatible.**
  - Executes the 32-bit wide ARM v4 instruction set.
  - Executes the 16-bit wide compressed instruction set.
- **Deeper pipeline for better clock speed.**
  - The processor is built around a 9 stage pipeline to achieve a high operating frequency. The pipeline has a data forwarding capability to allow back to back instructions to execute without stalls. Non trivial shifts require their operands a cycle early. Loads have a 3 cycle latency and the pipeline will stall if an attempt is made to access the register within the latency period.
- **Supports interrupt and abort signaling**
  - Features dedicated high level sensitive IRQ, FIQ and memory abort pins.
- **Can be interfaced with caches/MMU.**
  - The CPSR of the processor is exposed as a port allowing for implementation of a virtual memory system.
  - Memory stall may be indicates to the core via dedicated ports to allow caches to be connected.
- **Coprocessor interface provided.**
  - The coprocessor interface simply exposes internal signals of the core. It is up to the coprocessor to interpret and process instructions correctly.
- **Supports M-variant multiplication instructions**
  - These instructions are supported:

    `MUL, MLA, SMULL, UMULL, SMLAL, UMLAL`
- **The core is configurable**
  - The processor may be synthesized without compressed instruction support and/or coprocessor interface support to save area and improve speed.
- **Designed for FPGA synthesis**
  - Most memory structures of the processor map efficiently onto FPGA block RAMs. The register file is overclocked by a 2x clock to allow for 2 write ports.
  - The branch predictor memory also efficiently maps to FPGA block RAM.
  - No device specific instantiations are made to allow for portability across FPGA vendors.
- **Faster performance of memory instructions**

  - Memory instructions with writeback can be issued as a single instruction since the register file is built to have 2 write ports. This may improve performance.

- **The processor core is written entirely in synthesizable Verilog-2001.**
- **A branch predictor is installed to compensate for the longer pipeline.**
  - Branches within a 2KB block of memory can be mapped into the predictor without conflict. The predictor basically uses a bimodal prediction algorithm (2-bit saturating counter per branch entry).
- **Uses a base restored abort model**
  - Uses a base restored abort model making it easier to write exception handlers. Basically, on a fault in between a multiple memory transfer, the processor rolls back the base pointer register as it were before the operation took place.

1.2 About This Manual

The purpose of this manual is to document the processor core's design. This document is very incomplete. I will try my best to update it.

2 Configuring the core and testbench

Throughout, it is assumed that `$ZAP_HOME` points to the working directory of the project.
Core/testbench configuration may be done using defines. The defines file is located in `$ZAP_HOME/includes/config.vh`.
See table 1

Define	**Purpose**	**Required** for	**Comments**
CORE CONFIGURATION			
`THUMB_EN`	Enabling compressed instruction support	Core Setup	Enabling this increases core area and reduces performance.
`COPROC_IF_EN`	Enabling coprocessor suport. Extra ports get added.	Core Setup	Enabling this increases core area and reduces performance.
TESTBENCH CONFIGURATION			
`IRQ_EN`	Generates periodic IRQ pulses.	Testbench	–
`SIM`	Generates extra messages.	Testbench	*Must be UNDEFINED for correct synthesis of the core in Xilinx since some debugging structures in RTL are removed if this is not defined.* **==¿ MUST BE UNDEFINED FOR SYNTHESIS ¡==**
`VCD_FILE_PATH`	Set the path to the VCD data dump.	Testbench	–
`MEMORY_IMAGE`	Path to the memory image Verilog file.	Testbench	–
`MAX_CLOCK_CYCLES`	Set the number of cycles the simulation should run before terminating.	Testbench	–
`SEED`	Set the testbench seed. The seed influences randomness.	Testbench	–

Table 1: Configuring the core and testbench

3 Compiling Code and Running Simulation

NOTE: If you want to quickly test the processor with sw/asm/prog.s and sw/asm/prog.c sample programs, see Section 3.1

3.0.1 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.

3.0.2 Generating a Verilog memory map

`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;
```

3.0.3 Invoking the simulator

Ensure `config.vh` is set up correctly.

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

```
iverilog $ZAP_HOME/rtl/*.v $ZAP_HOME/rtl/*/*.v $ZAP_HOME/testbench/*.v
$ZAP_HOME/models/ram/ram.v -I$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 situations, 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.

3.1 Run Sample Code quickly...

A sample .s and .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 table 2...

Variable	Purpose
ZAP_HOME	Set this to the project working directory.
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 your startup assembly file.
C_PATH	Set this to the location of your C file.
LINKER_PATH	Set this to the location of the linker script.
TARGET_BIN_PATH	Set this to the target bin file location.

VCD_PATH	Set this to location where the VCD is to be created. **This must match what is in config.vh**
MEMORY_IMAGE	Set this to the location where the memory image must be created. **This must match what is in config.vh**

Table 2: Variables in Perl script

4 IO Ports of the Core

Table 3 lists the ports of the processor core.

Port	Direction	Description	Comments
i_clk	I	Core clock	–
i_clk_2x	I	Register file clock	Must run synchronously with the core clock at twice the frequency.
i_reset	I	Active high reset	This runs through a reset filter (using 2 series D flip-flops) that then drives a global reset throughout the design.
i_instruction[31:0]	I	32-bit instruction from the instruction cache.	–
i_valid	I	Indicates that the instruction on the 32-bit bus is valid and ready to be clocked into the instruction register (Fetch stage). If this is 0, the PC does not change.	–
i_instr_abort	I	Indicates that the instruction cache experienced an instruction abort.	When abort is asserted, the i_valid port is treated as a DON'T CARE.
o_read_en	O	Current memory access is a read access.	–
o_write_en	O	Current memory access is a write access.	–
o_address[31:0]	O	Address of the current memory access.	The address is always 32-bit aligned. Specific bytes are addressed using byte enables.
o_mem_translate	O	Current memory access must adopt a user view of memory	–.
i_data_stall	I	Indicates that the data memory has stalled. The entire pipeline freezes when this happens (PC does not change).	–

i_data_abort	I	Indicates that the data memory access aborted. When this is asserted, the i_data_stall port must be deasserted.	
i_rd_data[31:0]	I	Data received from data memory is clocked in from this port if there is no data memory stall.	–
o_wr_data[31:0]	O	Data to write out to memory	–
o_ben[3:0]	O	Byte enables. [0] deals with [7:0] of the data and [3] with [31:24].	–
i_fiq	I	FIQ level sensitive signal (Active high)	–
i_irq	I	IRQ level sensitive signal (Active high)	–
o_pc[31:0]	O	Program Counter	Lower bit is ALWAYS 0.
o_cpsr[31:0]	O	Current PSR	–.
i_copro_done	I	Coprocessor done indication.	–
o_copro_dav	O	Data on the coprocessor output ports of the core are valid	–
o_copro_word [31:0]	O	The entire 32-bit coprocessor instruction is presented on the port.	–
i_copro_reg_en	I	Coprocessor wishes to take charge of the register file.	–
i_copro_reg_wr_index [5:0]	I	Coprocessor write register index.	–
i_copro_reg_rd_index [5:0]	I	Coprocessor read register index	–
i_copro_reg_wr_data [31:0]	I	Coprocessor data to write to the register mentioned in i_copro_reg_wr_index.	–

o_copro_reg_rd_data [31:0]	O	Data read from register file to be read into coprocessor. This corresponds to register specified on i_copro_reg_rd_index although it is delayed by 1 clock cycle.	–

Table 3: IO Ports

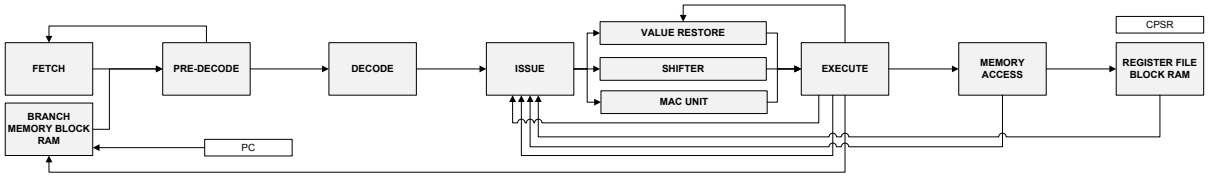
NOTE: If COPROC_IF_EN is not defined, the coprocessor ports are not available!

NOTE: The coprocessor must generate translated register numbers. For example, if it wishes to write to R13_FIQ, it must generate a write to R25.

5 Basic Description

Figure 1 should give a basic overview of the pipeline structure (8 stage pipeline). Table 4 gives an overall description of the pipeline stages. Note that the pipeline is an 8 stage pipeline.

Figure 1: High level view of the ZAP pipeline



Stage	Description
FETCH	Clocks in data from the instruction cache. Also computes PC+8 in case the instruction refers to R15 getting PC from the third stage).
BRANCH PREDICTOR RAM	This consists of block RAM dedicated to hold branch state for up to 2KB of instructions.
PREDECODE	Decompresses 16-bit instructions, handles LDM/STM instructions and SWAP/SWAPB. Also handles the coprocessor interface. Basically consists of a series of state machines.
DECODE	Decodes 32-bit instructions into a format that can be understood by the downstream logic.
ISSUE	Performs preliminary register read by sniffing out register value from the bypass network.
VALUE RESTORE SHIFTER MAC-UNIT	The value restore unit restores correct values by getting them from the ALU. This is useful for executing back-to-back instructions that typically get read incorrectly in issue. The shifter handles all shift operations and the MAC unit performs multiply- accumulate.
EXECUTE	The execute unit consists of a 32-bit ALU. This unit also generates memory addresses and other memory control signals.
MEMORY ACCESS	This stage clocks in data from the data cache.
REGISTER FILE	This stage is the register file and is basically 4 block RAMs to allow for 4 independent read ports (TODO: Can be reduced to 3 since one port is redundant).

Table 4: Pipeline Stage Description

5.1 Register Mapping

Table 5 shows how banking is implemented in ZAP. The register map to the register file as shown below...

Program Register	Physical Register
USR_R0	0
USR_R1	1
USR_R2	2
USR_R3	3
USR_R4	4
USR_R5	5
USR_R6	6
USR_R7	7
USR_R8	8
USR_R9	9
USR_R10	10
USR_R11	11
USR_R12	12
USR_R13	13
USR_R14	14
–	15
RAZ_REGISTER	16
–	17
FIQ_R8	18
FIQ_R9	19
FIQ_R10	20
FIQ_R11	21
FIQ_R12	22
FIQ_R13	23
FIQ_R14	24
IRQ_R13	25
IRQ_R14	26
SVC_R13	27
SVC_R14	28
UND_R13	29
UND_R14	30
ABT_R13	31
ABT_R14	32
DUMMY_REG0	33
DUMMY_REG1	34
FIQ_SPSR	35
IRQ_SPSR	36
SVC_SPSR	37
UND_SPSR	38
ABT_SPSR	39
SWI_SPSR	40

Table 5: Register Mapping

The file that describes this translation is in `includes/regs.vh`. The function that does the translation is in `includes/global_functions.vh`