ZAP : An ARM v4T Compatible Soft Processor

Revanth Kamaraj

October 19, 2016

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 super-pipelined with 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`.

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.*
`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	–

3 Compiling Code and Running Simulation

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...

```
iverilog $ZAP_HOME/rtl/*.v $ZAP_HOME/*.v $ZAP_HOME/testbench/*.v $ZAP_HOME/models/ram/ram.v
-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 the table below...

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**

4 Top Level Description