ZAP Processor User Guide

Revanth Kamaraj

1 Introduction

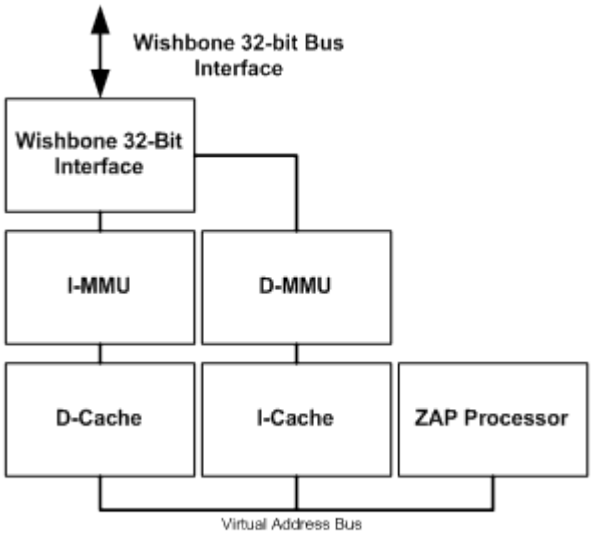
ZAP is a synthesizable open source 32-bit RISC processor core capable of executing ARMv4T binaries at both the user and supervisor level. The processor features a 10-stage pipeline that allows it to reach reasonable operating frequencies. The processor supports standard I/D cache and memory management that may be controlled using coprocessor #15. Both the cache and TLB are direct mapped. Caches, TLBs and branch memory are implemented as generic fully synchronous RAMs that can efficiently map to native FPGA block RAM to save FPGA resources. To simplify device integration, the memory bus is fully compliant with Wishbone B3. A store buffer is implemented to improve performance.

NOTE: Please use pipeline retiming during synthesis for maximum timing performance.

1.1 CPU Clock and Reset

ZAP uses a single clock called the core clock to drive the entire design. The clock must be supplied to the port I_CLK. ZAP expects a rising edge synchronous I_RESET (active high) to be applied i.e., the reset signal must change only on the rising edges of clock. The reset must be externally synchronized to the core clock before being applied to the processor.

1.2 Block Digram



1.3 Pipeline Overview

Stage	Description
Fetch	Clocks data from I-cache into instruction register. Also branch predictor memory is read out in this stage.
FIFO	Instructions and corresponding PC+8 values are clocked into a shallow buffer.
Thumb Decoder	Converts 16-bit instructions to 32-bit ARM instructions. Instructions predicted as taken cause the pipeline to change to the new predicted target.
Predecode	Handles coprocessor instructions, SWAP and LDM/STM.
Decode	Decodes ARM instructions.
Issue	Operand values are extracted here from the bypass network. In case data from the bypass network is not available, the register file is read.
Shift	Performs shifts and multiplies. Contains a single level bypass network to optimize away certain dependencies. Multiplication takes multiple clock cycles.
Execute	Contains the ALU. The ALU is single cycle and handles arithmetic and logical operations.
Memory	Clocks data from the data cache into the pipeline. Aligns read data as necessary.
Writeback	Writes to register file. Can sustain 2 writes per clock cycle although the only use for the feature is accelerate LDR performance in the current implementation.

ZAP features a 10 stage pipeline. The pipeline has an extensive bypass network to minimize pipeline stalls. A load accelerator allows data to be forwarded from memory a cycle early. Most non-multiply instructions can be executed within a single clock tick with no stalls. Exceptions to this rule are when multiplies or non-trivial shifts are used.

The following code takes 3 cycles to execute because R1 needs to be shifted and is not available until the first instruction enters the ALU:

```
ADD R1, R2, R3
ADD R4, R5, R1 LSL R2
```

If the second register is not source shifted by a register that depends on the previous instruction, a data dependency check may be relaxed (Register R9 for the second instruction can be obtained in issue itself so nothing is blocking the second instruction from using the shifter) and thus the following code takes 2 cycles to execute:

```
ADD R1, R2, R3 LSL R5
ADD R4, R1, R9 LSL R2
```

Another feature of the pipeline is that it can issue memory operations with write-back in a single cycle. The following instructions takes 2 cycles to execute assuming a perfect cache.

```
LDR R0, [R1, #2]!
ADD R1, R3, R4 LSL R1
```

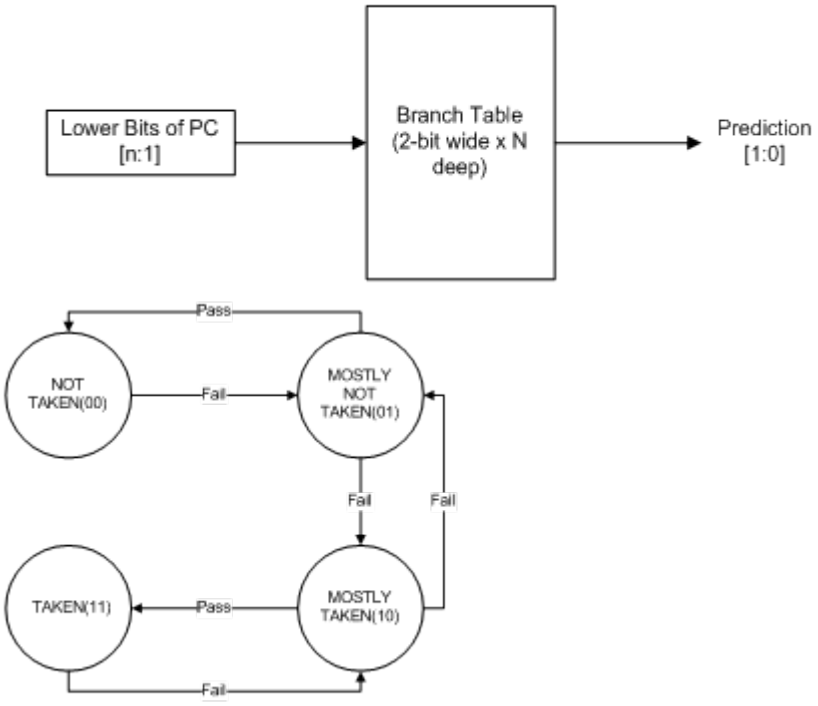
The pipeline feedback unit is designed to reasonably minimize pipeline stalls. Note however that pipeline bubble squashing is available only across the instruction FIFO.

1.4 Features

- Fully synthesizable Verilog-2001 core.
- Store buffer for improved performance.
- Can execute ARMv4T code at both the user and supervisor level.
- Wishbone B3 compatible interface. Cache unit supports burst access.
- 10-stage pipeline design. Pipeline has bypass network to resolve dependencies.
- 2 write ports for the register file to allow LDR/STR with writeback to execute as a single instruction. Note that the register file is implemented using flip-flops.
- Branch prediction supported. Uses a 2-bit state for each branch.
- Split I and D writeback cache (Size can be configured using parameters).
- Split I and D MMUs (TLB size can be configured using parameters).
- Base restored abort model to simplify data abort handling.

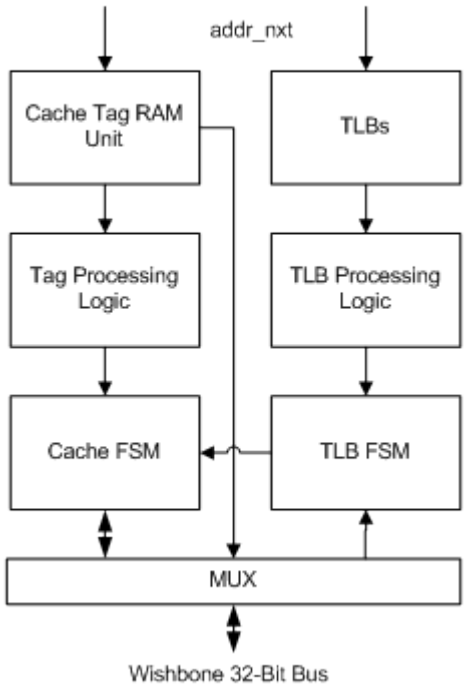
1.5 Branch prediction mechanism

ZAP uses a relatively simple branch prediction mechanism (Note that the branch table block RAM is read in Fetch). Note that when using ARM code, the amount of branch table entries is cut by half since the lower 2 bits of PC are 0. A simple state machine is used to reinforce or modify the status of a branch. The pass and fail signals are generated from the ALU.



1.6 Cache/TLB Overview

ZAP uses direct mapped cache (separate I/D) that is virtual. For decent performance, the cache is writeback. To enable writeback, each cache line has a dirty bit. The size of each cache line is fixed at 16 bytes. Since ARMv4 specifies three kinds of paging schemes: section, large and small pages, 3 TLB block RAMs are employed which are also direct mapped. ZAP has independent instruction and data caches/TLBs. The cache/MMU setup is v4 compatible. The cache should be enabled as soon as possible for good performance because the memory subsystem is efficient for burst transactions.



1.7 Running Simulations

1.7.1 Creating and Simulating Test Cases

To create a test case, create a some assembly(.s extension) and C files(.c extension) and a linker script(.ld extension) in the directory `src/ts/⟨TestCase⟩/`

Copy the `makefile` and `Config.cfg` from one of the existing test case directories to the directory `src/ts/⟨TestCase⟩/`. Edit the `Config.cfg` to suit the testcase.

To run simulations using the scripts provided, you will need Icarus Verilog 10.0 or higher and the ARM baremetal GCC toolchain. Object files and waveform dumps can be found in the `obj/ts/⟨TestCase⟩/` directory.

Enter `src/ts/⟨TestCase⟩` and type `make`.

NOTE: Normally C files are directly converted to object files. To view the assembly code for a C file (say file.c), do `make c2asm X=file.c`. This will create a file called `file.c.asm` in the corresponding object directory.
NOTE: To clean object files, enter `src/ts/⟨TestCase⟩` and type `make clean`.
NOTE: To clean object files and the extracted GCC files, enter `src/ts/⟨TestCase⟩` and type `make cleanall`.

NOTE: You can set the testbench and processor configuration in the `Config.cfg` file that contains a Perl hash. This file is present in the same directory as the test case.

Here's a sample of what the `Config.cfg` file should contain.

```
%Config = (
# CPU configuration.
DATA_CACHE_SIZE            => 4096,# Data cache size in bytes
CODE_CACHE_SIZE            => 4096,# Instruction cache size in bytes
CODE_SECTION_TLB_ENTRIES => 8,   # Instruction section TLB entries.
CODE_SPAGE_TLB_ENTRIES   => 32,  # Instruction small page TLB entries.
CODE_LPAGE_TLB_ENTRIES   => 16,  # Instruction large page TLB entries.
DATA_SECTION_TLB_ENTRIES => 8,   # Data section TLB entries.
DATA_SPAGE_TLB_ENTRIES   => 32,  # Data small page TLB entries.
DATA_LPAGE_TLB_ENTRIES   => 16,  # Data large page TLB entries.
BP_DEPTH                   => 1024,# Branch predictor depth.
INSTR_FIFO_DEPTH           => 4,   # Instruction buffer depth.
STORE_BUFFER_DEPTH         => 8,   # Store buffer depth.
SYNTHESIS                  => 1,   # Make this to 1 to simulate compile
                                   # from a synthesis perspective.

# Testbench configuration.
WAVES               => 1,     # Logs VCD when 1.
UART0_TX_TERMINAL => 0,     # 1 Enables UART TX terminal 0. 0 disables it.
UART1_TX_TERMINAL => 0,     # 1 Enables UART TX terminal 1. 0 disables it.
UART0_RX_TERMINAL => 0,     # 1 Enables RX terminal 0. Characters entered go to
UART1_RX_TERMINAL => 1,     # 1 Enables RX terminal 1. Characters entered to to
EXT_RAM_SIZE        => 32768, # External RAM size in bytes.
SEED                => -1,    # Seed. Use -1 to use random seed.
DUMP_START          => 2000,  # Starting memory address from which to dump.
DUMP_SIZE           => 200,   # Length of dump in bytes.
MAX_CLOCK_CYCLES  => 100000,# Clock cycles to run the simulation for. 0 for infi

# Make this an anonymous has with entries like
# "r10" => "32'h0" etc. These represent register
# values (expected) at the end of simulation.
REG_CHECK                      => {"r1" => "32'h4",
                                   "r2" => "32'd3"},

# Make this an anonymous hash with entries like
# verilog_address => verilog_value etc. These
# represent expected  memory values at the end of
# simulation.
```

```
FINAL_CHECK                     => {"32'h100" => "32'd4",
                                    "32'h66" => "32'h4"}
);
```

2 IO Ports and Configuration

Signal Name	IO	Description
I_CLK	I	Core clock.
I_RESET	I	Core reset.
O_WB_CYC	O	Wishbone CYC signal.
O_WB_STB	O	Wishbone STB signal.
O_WB_ADR[31:0]	O	Wishbone 32-bit address.
O_WB_SEL[3:0]	O	Wishbone byte lane enables.
O_WB_WE	O	Wishbone write enable.
O_WB_DAT[31:0]	O	Wishbone write data.
O_WB_CTI[2:0]	O	Wishbone cycle type indicator. 0b010 = Incrementing burst. 0b111 = End of burst. The interface shall only generate one of the above 2 codes. A single transfer is effectively a burst of length 1.
O_WB_BTE[1:0]	O	Burst type extension tag. This always reads 0x0 to indicate that the processor can only perform incrementing linear bursts (32-bit width).
I_WB_DAT[31:0]	I	Wishbone read data.
I_WB_ACK	I	Wishbone acknowledge.
I_IRQ	I	IRQ Interrupt.
I_FIQ	I	FIQ Interrupt.

Verilog parameters can be used to statically configure the processor instance as shown in the table below.

Parameter	Description	Notes
DATA_CACHE_SIZE[31:0]	Data cache size in bytes.	1
CODE_CACHE_SIZE[31:0]	Instruction cache size in bytes.	1
CODE_SECTION_TLB_ENTRIES[31:0]	Section TLB entries (CODE).	2
CODE_SPAGE_TLB_ENTRIES[31:0]	Small page TLB entries (CODE).	2
CODE_LPAGE_TLB_ENTRIES[31:0]	Large page TLB entries (CODE).	2
DATA_SECTION_TLB_ENTRIES[31:0]	Section TLB entries (DATA).	2
DATA_SPAGE_TLB_ENTRIES[31:0]	Small page TLB entries (DATA).	2
DATA_LPAGE_TLB_ENTRIES[31:0]	Large page TLB entries (DATA).	2
FIFO_DEPTH[31:0]	Depth of the fetch buffer in the pipeline.	3
BP_ENTRIES[31:0]	Depth of the branch predictor memory.	3
STORE_BUFFER_DEPTH[31:0]	Set the depth of the store buffer. Do not set it to a value less than 16.	4

NOTE.

1. Should be a power of 2 and greater than 128 bytes.

2. Should be a power of 2 and must be at least 2 entries.

3. Should be a power of 2 and must be greater than 2.

4. Depth must be 16 or more and a power of 2.

3 CP15 commands

ZAP features an ARMv4 compatible cache subsystem (cache and MMU). This subsystem may be configured by issuing commands to specific CP #15 registers using coprocessor instructions. A list of supported CP #15 commands/registers are listed

in the table below:

WARNING: In particular, cleaning and flushing of specific locations is not supported. The OS should avoid issuing such commands.

<table>
<tr><th>Reg.</th><th>Name</th><th>Description</th><th>Note</th></tr>
<tr><td>0</td><td>ID</td><td>[23:16] Always reads 0x01 to indicate a v4 implementation.
Other bits are UNDEFINED.</td><td>1</td></tr>
<tr><td>1</td><td>CON</td><td>[0] MMU enable.
[2] Data cache enable.
[8] S bit.
[9] R bit.
[12] Instruction cache enable.
READ ONLY bits are described in note 2.
Bits other than the ones specified here and in note 2 are UNDEFINED.
2</td><td></td></tr>
<tr><td>2</td><td>TRBASE</td><td>Holds 16KB aligned base address of L1 table.</td><td></td></tr>
<tr><td>3</td><td>DAC</td><td>Domain Access Control Register.</td><td></td></tr>
<tr><td>5</td><td>FSR</td><td>Fault address register.</td><td>4</td></tr>
<tr><td>6</td><td>FAR</td><td>Fault status register.</td><td>4</td></tr>
<tr><td>7</td><td>CACHECON</td><td>Data written to this register should be zero else UNDEFINED operations can occur.
CACHECON control table.
<table>
<tr><th>Opcode2</th><th>CRm</th><th>Description</th></tr>
<tr><td>000</td><td>0111</td><td>Flush all caches.</td></tr>
<tr><td>000</td><td>0101</td><td>Flush I cache.</td></tr>
<tr><td>000</td><td>0110</td><td>Flush D cache.</td></tr>
<tr><td>000</td><td>1011</td><td>Clean all caches.</td></tr>
<tr><td>000</td><td>1010</td><td>Clean D cache.</td></tr>
<tr><td>000</td><td>1111</td><td>Clean and flush all caches.</td></tr>
<tr><td>000</td><td>1110</td><td>Clean and flush D cache.</td></tr>
</table></td><td>3</td></tr>
</table>

<table>
<tr>
<td>8</td>
<td>TLBCON</td>
<td>Data written to this register should be zero else UNDEFINED operations can occur.
TLBCON Control table.
<table>
<tr><th>Opcode2</th><th>CRm</th><th>Description</th></tr>
<tr><td>000</td><td>0111</td><td>Flush all TLBs</td></tr>
<tr><td>000</td><td>0101</td><td>Flush I TLB.</td></tr>
<tr><td>000</td><td>0110</td><td>Flush D TLB.</td></tr>
</table>
</td>
<td>3</td>
</tr>
</table>

NOTE:
1. Read only. Writes have NO effect.

2. Processor does not check for address alignment ([1] reads 0), only supports Little Endian access, full 32-bit, write buffer always enabled ([7:4] reads 0b0011), does not support high vectors ([13] reads 0) and always has a predictable cache strategy ([11] reads 1) i.e., direct mapped.

3. Reads are UNPREDICTABLE.

4. Only data MMU can update this. For debug purposes, these are RW registers.