

Implementation of a fully pipelined ARM compatible microprocessor core

Abstract

This article describes the implementation of a microprocessor core, instruction compatible with the ARM8/9 processor. The implementation uses an internal structure, similar to that of the ARMv4 and is completely written in VHDL. The heart of the design is a 5 stage pipe-line, which notably improves the performance. The design has been extensively verified using VHDL-Simili[1] and other applications written for the task, these allow a flexible and appropriate testing strategy for the core. Once high level simulations showed the correct functionality of the core, a prototype has been implemented using a Virtex XC800 FPGA from Xilinx. This prototype had the addition of a simple peripheral designed for the demonstration. Finally the design has been synthesized using the AMS 0.35u standard cell library and a mega-cell layout created.

1. Introduction

In recent years the number of custom designs using FPGA or ASIC technologies have multiplied. This has led to the development of designs in high level description languages, like VHDL or Verilog which allow the designer to conceive the design at the level of RTL, without reference to the final technology of vendor used for the final implementation. This high level design process has created a market of IPs or complete pre-designed and tested macro-cells which can be incorporated within the designer's project hierarchy [2].

On the other hand, the family of ARM processors [3] has experienced a massive growth, not only in the technology but also in their commercial success.

These factors, coupled with the past experiences [4] in the implementation of such cells has led to the development of this project .

2. The ARMv4 processor architecture

The name ARM is an acronym for "Advanced RISC Machine". The ARM processors based on the ARMv4 architecture [6] have the following features: A bank of 37, 32bit registers; ALU with inputs and

outputs of 32 bits, and an input barrel shifter; 32x32bit multiplier with 64bit result; user and 6 other operational privilege modes of operation; exceptions with priority handling, Reset, Abort (memory failure), FIQ (fast interrupt) IRQ, SWI and *undefined*; The possibility to choose little endian or big endian memory; the standard ARM ISA instruction set.

3. Pipeline Architecture

The basic principal of this architecture is that it doesn't wait for an instruction to write its results in the destination before it begins processing the next instruction. With this structure, in any given moment there are as many instructions being processed by the CORE as stages in the *pipeline*. In the steady state, when the processor is working, one instruction is completed and one instruction loaded on every clock cycle. That is, in a pipeline of d stages one instruction takes d cycles to be processed, but there exist d instructions in the different parallel stages of execution, only after the final stage does an instruction write its results into the corresponding processor register.

The functionality of a pipeline processor is directly related to a parameter called *speedup*, whose value gives an idea of the increase in processor speed obtained using a pipeline structure with respect to the time required to execute the same instruction in a single stage processor. If the both types of processor use the same technology for their implementation, the value of *speedup* is define as:

$$speedup = \frac{CPI_{not\ pipelined} \times t_{cyc}^{not\ pipelined}}{CPI_{pipelined} \times t_{cyc}^{pipelined}} = \frac{CPI_{not\ pipelined}}{CPI_{pipelined}}$$

In the ideal situation, a pipeline processor with each stage optimised to take an equal fraction of the processing would take $n+d$ cycles to execute n instructions, whilst a non-pipelined processor would take n cycles. Given the increased complication of the single stage implementation, this design would have to reduce the clock frequency by a factor of d , giving an equivalent time of $n.d$ pipelined cycles.

Normally $n \gg d$, so that we can express the speedup factor as:

$$speedup = \frac{n \cdot d}{n + d} \approx d$$

In practice, the situation is not so ideal, other phenomena such as *interlocking* exists between processor registers. Figure 1 shows the five stages of the pipeline, IF: Instruction fetch, DE: decode, EX: execute, ME: memory and Wr: write, in the case of an instruction i_3 which writes to a register used by the instruction i_4 an interlocking system is activated. The instruction i_4 can not be executed until the result of the instruction i_3 has been saved and no operation instructions called *bubbles* are inserted in the execute stage until i_3 has left the pipeline.

Time	IF	DE	EX	ME	Wr
1	i_1				
2	i_2	i_1			
3	i_3	i_2	i_1		
4	i_4	i_3	i_2	i_1	
5	i_5	i_4	i_3	i_2	i_1
6	i_5	i_4	●	i_3	i_2
7	i_5	i_4	●	●	i_2
8	i_6	i_5	i_4	●	●

Fig. 1. Interlocking between registers

Another situation that reduces the *speedup* of a pipeline processor occurs with the execution of jump instructions. All jump instructions are conditional, and can therefore only be detected in the execute stage, if the jump is taken, the instructions already loaded in fetch and decode must be discarded. Figure 2 shows such a situation, where i_2 is a jump instruction, similar situation exist with exceptions and interrupts.

All of these situations reduce the effective value of the *speedup* factor. Memory wait states and 64 bit multiplies also effect this factor.

4. Implementation

The scope of this work is the design of a processor CORE, compatible with the ARM instruction set, using an ARMv4 type architecture. It was decided to not implement the Block Data Transfer instruction (which takes a variable number of cycles, and can be implemented as a software subroutine) and the co-

processor instructions (as the co-processor has not been implemented). The BDT instruction also breaks the RISC nature of the ARM instruction set and is not easily implemented in our pipeline structure.

Time	IF	DE	EX	ME	Wr
1	i_1				
2	i_2	i_1			
3	i_3	i_2	i_1		
4	i_4	i_3	i_2	i_1	
5	i_a	●	●	i_2	i_1
6	i_b	i_a	●	●	i_2
7	i_c	i_b	i_a	●	●
8	i_d	i_c	i_b	i_a	●

Fig. 2. Jump instruction execution

From the start it was decided to use a modern and efficient structure, for this reason the design is implemented using the Harvard structure. As for internal structure, the design is styled on a RISC machine, and taking into account the characteristics of the ARM ISA, the pipeline architecture was optimised to provide the best results. When it came to selecting the number of stages in the pipeline, it was decided to use a pipeline of 5 stages rather than the traditional 3 or 4 stage pipelines. This decision was motivated by two important factors: Experimental results indicate that the maximum clock frequency for an FPGA implemented pipeline is given by $\log_2(\text{bus width})$ [7] in our case the busses are of 32 bits and hence indicate a 5 stage pipeline, and also the commercial ARM8/9 is known to use a 5 stage pipeline.

In the trade off between speed and area, we have designed for speed. The target architecture was a Virtex FPGA which while more than capable of implementing the design, the characteristics of the design (32bit busses) is not very speed friendly for FPGA implementation.

Before beginning the implementation it was first necessary to develop an architecture that could support the ARM instruction set, by this we mean the internal bus structure, register access control logic, program counter (PC) etc. In second place it was necessary to specify the architecture of an ALU suitable for use with the ARM instruction set. The ALU is rather complex, it requires 3 input busses A, B, and C and includes a barrel shifter that permits

any rotation of the 32 bit B input, in this way it can implement any of 16 distinct operations on any rotation of the B input. The ALU also contains a multiplier which is capable of implementing multiply and accumulate instructions on 32 bit inputs with a 64 bit result. The reading of the 64 bit multiplier result requires two clock cycles to read the result, this complicates the pipeline structure and was solved by inserting a dummy instruction in the execute stage following a multiply instruction. The multiplier of the design is implemented in a separate file, allowing the use of precompiled multipliers, megacell multipliers or the disabling of this feature.

With these two steps completed, it is now possible to implement the five stages the make up the pipeline architecture. Figure 3 shows a simplified view of the structure used. Each stage has been designed so as to simplify the following stages, once the instruction is decoded in the decode stage, a configuration word is produced, this decoded description simply enables or disables processing units, but no further instruction decoding is necessary. Once all the functional blocks of the pipeline where implemented, the decode stage simple produces configurations that are passed from stage to stage enabling the correct logic as they go.

To finalise the design it was necessary to implement the control structure to handle exceptions and a memory interface for the complete core. This memory interface provides just the minimal support to allow the core to be included within an unknown system structure.

A more detailed description of the complete architecture of the design can be found in [5].

5. Simulations

The simulation of the design is perhaps the most important design step, and at the same time the most complex and critical. The entire instruction set must be verified in all conditions, in such a way that all errors can be detected, this is not easy and clearly an automated testing setup is required.

The solution adopted is based on the freely available tool VHDL-Simili. This tool is able to “compile” a VHDL description and then execute the compiled design at a very high speed. The results are not as reliable as post synthesis simulation, but have the advantage of speed and flexibility. A testbench was developed in VHDL to stimulate the processor and record the results in 3 text files.

The simulation process is the following. A text file is generated containing the hexadecimal representation of the list of instructions to be simulated. Using a simple tool, written for the task, this instruction file is converted into a VHDL ROM model, that can be used as executable code for the processor core. Using a batch process, the ROM description together with the core and testbench are compiled with VHDL-Simili and executed. The testbench provides the system clock, and records the values of all the registers, internal busses and control signals in three files called traza.dat, pipeline.dat and memory.dat.

Figure 4 shows the complete simulation flow. The output files record, cycle by cycle all the information necessary to validate the correct execution of the ROM code, together with detailed information on the state of every stage of the pipeline. One advantage of the our design strategy is that the

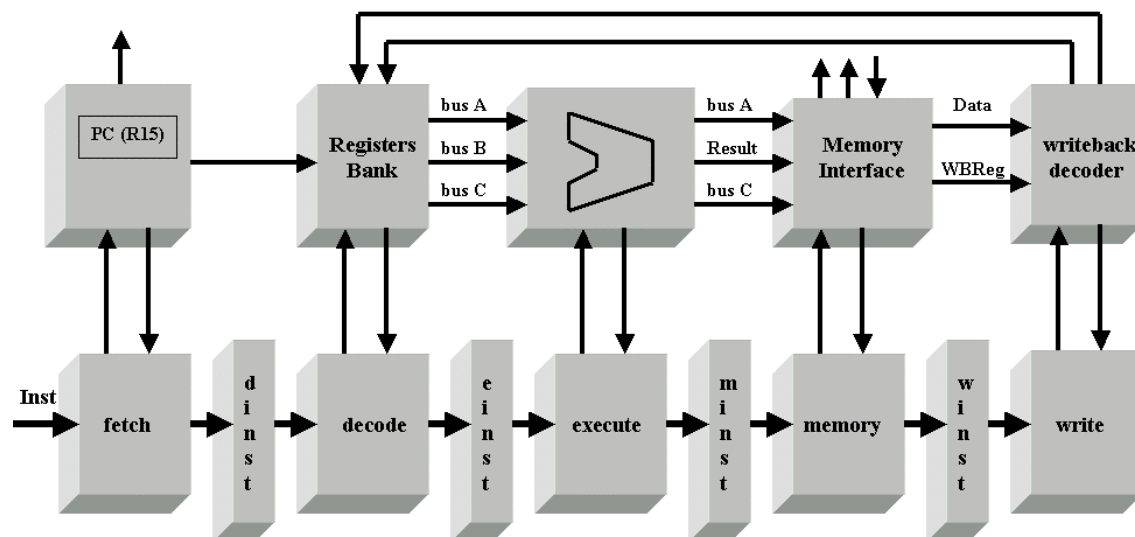


Fig. 3. The architecture implemented

pipeline could be completely simulated as soon as one instruction was implemented, and conceptual problems and critical conditions (eg. Interrupts arriving during the interlocking process) corrected whilst the design was still in the early stages. The design of the configurations generated by the decode stage, means that the addition of more instructions has little effect on the pipeline architecture and the instructions previously implemented. This notable helped the debugging of the complete design.

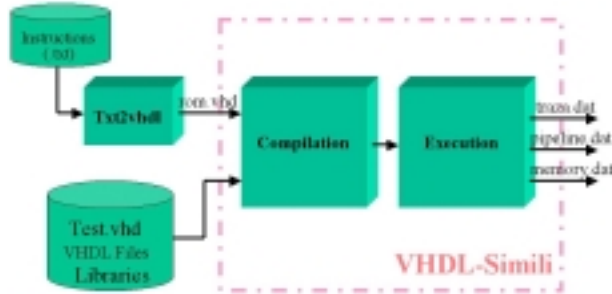


Fig. 4. Simulation flow

6. Layout generation and optimisation

The project was originally orientated at the design of a FPGA core, but due to the process independent nature of VHDL, it is a simple task to produce a standard cell ASIC description of the design. To this end, a standard-cell netlist was synthesised using SYNOPSIS, and a layout produced with Cadence Silicon Ensemble.

The design was implemented in the AMS 0.35u 3 metal process. The size of the mega-cell, without the multiplier was 2.4x2.4mm with a working frequency of 30 Mhz.

It was observed that the maximum frequency of the design was limited by the ALU, in particular instructions that use the shifter followed by the

adder. It was observed that the bus used as input to the shifter is available in the decode stage, and it was therefore possible to modify the design so that the shifting was applied during decode, and thereby reducing the length of the critical path in the ALU. These changes resulted in a 15% improvement in clock speed.

7. FPGA Prototype

In order to complete the design process, and taking in account the size of implementation, an FPGA prototype was necessary to completely validate the simulations results.

As the core does not include any input/output devices a simple memory mapped output device was designed. This device consists of a display made up of four seven segment LED displays, controlled by a twelve bit bus. Four bits were used to select the display to be powered on, and the rest select the individual LEDs segments to light up. A program was written in ARM machine code to refresh each element of the display at high speed, and write a message that slowly moves to the left, giving the effect of a rotating text message.

Figure 5 shows the working system. The design has been implemented using 50% of a Xilinx Virtex XC800 FPGA on an Xess evaluation board. The program being executed by the processor was written to use the maximum number of different instructions and the design worked as expected.

8. Conclusions and future work

A fully pipelined, ARM compatible processor core has been successfully implemented. The design is has been verified and a working prototype demonstrated. This design [5] will be placed under a



Fig. 5. The prototype design working

public license and downloadable from following URL: <http://www.gte.us.es/~jon/PFCS>

Future work on the design will include a more detailed study of the critical delays within the execute stage, and the possible improvements that can be made. A co-processor together with the missing instructions may be implemented and an efficient way to implement the BDT instruction designed.

References

- [1] SymphonyEDA, VHDL Simili V1.0 Documentation <http://symphonyeda.com/doc/simulimanual.pdf>
- [2] W. Hobbs "Model Availability, Portability and Accuracy, An IC Vendor's Perspective, Efforts and Vision for the Future" Workshop on Libraries, Component Modeling and Quality Assurance , Nantes, April, 1995
- [3] Advanced RISC Machines Ltd. <http://www.arm.com>
- [4] Hidden for blind review
- [5] Hidden for blind review
- [6] D. Jagger. "ARM Architecture Reference Manual", Morgan Kaufmann Publishers, 2nd edition, ISBN: 0201737191, 2000
- [7] Xilinx, "X-Files – News from the Leading Provider of FPGA and Desktop ASIC" 2000