Hardware Design Lab

Report Group 6
Lukas Boland, Sven Ströher, Ana Carolina Ferreira, Julian Käuser
Supervisor: Sarath Kundumattathil Mohanan, M.Sc.
Begin: 07/31/2017 | Submission: 08/16/2017
Institute of Computer Engineering | Integrated Electronic Systems Lab | Prof. Dr.-Ing. Klaus Hofmann

TECHNISCHE
UNIVERSITÄT
DARMSTADT

1 Introduction

This semester's task in the HDL Lab is to design a general purpose processor implementing the ARM-Thumb architecture, and synthesize it with the Synopsis Design Compiler for the TSMC 45nm Standard Cell library. Further requirements are the RTL design with Verilog, the introduction of a pipelined design, the use of a given 1-Port memory and the implementation of the full instruction set. All these requirements have been met with our design, as pointed out in this report.
The paper is structured as following. At first, the different tasks are reviewed, and their distribution among the team members is outlined. An overview on our apporach is given. Then, the processor design and its submodules are reviewed. Furthermore, the RTL verification steps are shown. After a section on the synthesis process, the gate level verification results are shown. Finally, an evaluation of our design with regards to the task is done.

1.1 Tasks and Work Distribution

The task of the HDL Lab can be split into two major parts: micro-architecture design and rtl coding, and standard cell synthesis. Since the second part requires results of the first one, we started with the rtl design.

First, the necessary submodules in the processor were agreed upon, and each team member was assigned one or more modules to work on. Creating and running a testbench for each designed module must always be performed in time with the module itself, so we assigned that task to everyone designing a module. Along with the module design, requirements regarding the Thumb-architecture had to be checked and built in.

In the next step on the rtl level, finished and tested modules were integrated to verify their function. If necessary, a step back to the rtl design was taken in order to fix malfunctions in single modules or to adapt them to changing requirements (e.g. changing communication between modules, new necessary signals).

The second major part, standard cell synthesis, was performed from the point where some modules were finished with the rtl design. Of course, for every change in the rtl design, the synthesis had to be re-done. With regard to the synthesis results for every module, some minor adaptions in the Verilog code had to be done to guarantee the synthesizeability.

While the synthesis processes could only be started three days into the lab, they lasted until the final day. The RTL design also lasted until the end, since we tried to improve the design as far as possible. The approximate work distribution among the team members is shown below.

- **Lukas Boland:** RTL design (decoder) and rtl top level verification/intergration
- **Sven Ströher:** RTL design (ALU, stack, fetch), synthesis, gate level verification
- **Ana Carolina Ferreira:** RTL design (ALU, fetch, controller), synthesis, gate level verification
- **Julian Käuser:** RTL design (memory interface, register file, controller), short report

2 Implementation

In this chapter, the designed micro-architecture is presented. Achievements from verification (both RTL and gate level) are shown. Additionally, a view on the synthesis process and results is taken.

Our design implements the Thumb architecture almost completely. It is capable of executing all instructions except `CBZ` and `CBNZ`, and all instruction related to `HINT` and `STATE CHANGE` (the last two types are explicitly not required and at least `STATE CHANGE` cannot be implemented without an ARM-capable micro-architecture). We focused on implementing all instructions, since we interpreted this as a primary objective from the lab manual. With these instructions, our processor is capable of executing both the `count32` and the `memcp46` benchmark applications.

2.1 Design

Our design is built of relatively large submodules, so that not many additional elements are necessary in the top module. The structure can be seen in Fig. 2.1. The main modules are the instruction decoder, the register file, the ALU, the memory interface and the instruction fetch module. These modules are described in detail in the following sections.

In the block structure, the modules are connected with several connections, but only the address and load inputs for the memory are busses with more than one input. All other connections can only be driven by one module, which simplifies the overall structure.

2.1.1 Processor Control

The control of the execution and memory access is performed by a state machine in the decoder module. Initially, a dedicated controller for the memory access and pipeline stalls was planned; while integrating and testing the decoder, the function of the controller was integrated into the decoder. It yet had many control functions in order to perform multi-cycle instruction like `MULTI-PUSH/-POP`.

Almost all control signals are outputs of the decoder. These include the ALU opcode, register selects, the flag updates and memory access signals. As the memory access must be arbitrated, the instruction fetch controls the multiplexors assigning the read request and address signals from itself and the decoder module. When the current instruction accesses the memory, a stall signal to the instruction fetch can be set by the decoder. On the other hand, the instruction fetch may stall the decoder (which then stalls the rest of the processor) if the next instruction is not loaded yet. With this mechanisms, we tried to keep the control of the processor as simple as possible.

2.1.2 Pipeline Stages

Our processor is comprised of two pipeline stages. One stage includes the instruction fetch and decoder, the other one does the execution and writeback tasks. These two stages have been chosen because of the structure of the benchmark applications and the 1-port memory. The applications are both relatively memory-access-loaded. The memory allows only one access per cycle; additionally, it is halfword-aligned, while the architecture uses byte-aligned addresses. The benchmarks both frequently access whole words

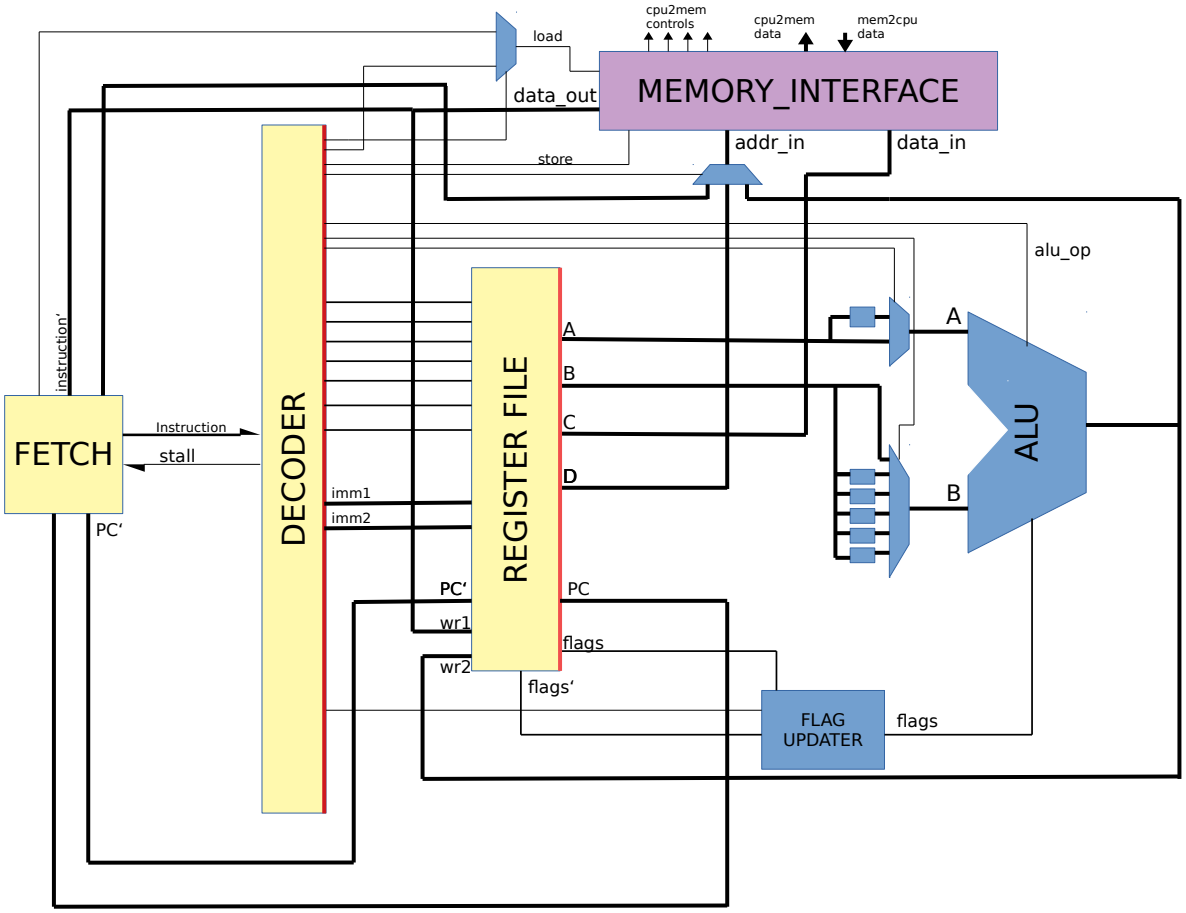

Figure 2.1: The block structure of the processor design. Register stages are marked red.

in the memory, so that in many cycles, the memory would be the bottleneck in the execution. With only two pipeline stages, few stalls would occur.

2.1.3 Modules

In the following, an insight view into the single modules' implementation is given.

ALU

The arithmetic logic unit (ALU) performs the calculations stated in the instructions.
Its 32-bit structure is purely combinatorial. All logical and arithmetical operations defined in chapter 5.4.1 in the THUMB instruction set (**TODO** are supported. Each computation can be executed within a single clock cycle.

Which operation to perform by the ALU is decided by the 4-bit opcode from the instruction fetch. This determines also if along the two signed input signals the input carry flag is needed.
The result consists of a 32-bit signed output value, as well as the four condition code flags negative (n),

zero (z), carry (c) and overflow (v). Depending on the computed instruction, the application program status register (APSR) in the register file gets updated by the Flag Updater with those flags.

Instruction Fetch

The instruction fetch (IF) is used for loading new instructions from the memory and passing them to the instruction decoder.
The module itself is build as a finite state machine and consists of four separate states. One of them is for reset, two are working states (wait and fetch) and the last one is entered when the executed program is finished.

At the startup, the decoder does not posses any instruction to perform, so the IF begins in fetch state. To get the first instruction, an address, which is two less than the current value of the program counter (PC), and a load request is sent to the memory interface.
When the wanted data arrives back, it gets passed to the decoder for one clock cycle. Furthermore the PC gets incremented by two, which matches the length of one instruction, 16 bit. The IF switches to wait state.
As long as the decoder is executing the new instruction, the IF gets stalled and remains in its current state. When finished, the instruction fetch switches back to fetch state again and starts to load the next instruction.
When the last instruction got performed, the instruction fetch switches to finish state and stops gathering new instructions.

If the execution of an instruction is already finished after a single clock cycle, the decoder does not have to stall the instruction fetch. Thus the IF is able to gather a new instruction every second cycle. While there is no new instruction to pass to the decoder, the no operation signal is sent.

Instruction Decoder

The instruction decoder generates all necessary control signals for the execution stage (register file, alu, memory controller, multiplexers, flag updater). The instructions have different formats that are characterized by specific bit patterns in the instruction bitstring. The instruction decoder tests the instruction againts these bit patterns to recognize the instrution format and extract the instruction parameters (alu operation, operation sources and target).

It can decode all instructions on the„Thumb 16-bit Instruction Set Quick Reference Card“except the CBNZ and CBZ instructions, for which there was no time left to implement them, and the instructions in the „Processor state change“and „Hints“sections which should not be implemented. Also, the change to ARM state is not implemented because the processor can only work in Thumb mode.

For operations that access the memory or the program counter, it stalls the instruction fetch to prevent interfering with the memory access of the execution stage or using a wrong program counter value for fetching a new instruction.

Instructions that can not be executed in one step are split into multiple smaller steps. This is the case for the instructions PUSH, POP, LDMIA and STMIA because they can store or load up to eight values in the memory, and for the BL instruction because the execution of its second part needs three consecutive alu operations on registers. During the execution of the steps of a split instruction, the instruction fetch ist stalled by the decoder.

For the split of the instructions, a step register is used which masks the already executed steps with zeros. A one-hot-encoding was chosen instead of binary encoding to allow using a part of the instruction bitstring as a selector for the steps to execute (e. g. for bits [7:0] = 1000 0001 of an STMIA instruction, only the steps „store register R0“and „store register R7“are executed and after the first step, the step register ist set to 1111 1110 to mask bit 0 which corresponds to the first step).

For the conditional execution of up to four instructions in an IT block, a register "itstate" is used that stores the execution condition for up to four instructions. The following instructions which are subject to the execution condition are executed only if the correspondig condition is true. Otherwise a NOP ist executed. This register is set when an IT instruction is executed and shifts left with every following instruction until all execution coditions loaded by the IT instruction are shifted out. Then it is reset to zero.

In retrospect, it would have been better to place the logic for conditional execution in the instruction fetch module because it would have been possible to fetch only the instructions that are really executed. That logic would have needed the capability to identify CMP instructions because they update the condition flags which are used to determine if an instruction is executed or not despite being in an IT block and could have altered the instructions to fetch. Other instructions that normally update the condition flags do not do so when in an IT block.

The decoder also has two outputs that control two multiplexers in the two operand paths to the alu. they can select modified versions of the operands as inputs to the alu which are used for some instructions, e. g. REV16 or REVSH.

Register File

Memory Interface

Other Modules

The remainder of the necessarry hardware for the processor is divided over only few parts, which are not all part of a distinct module. These include:

- **Update of flags:** The required flags are either updated or left in their old state, depending on control flags from the decoder.
- **Input modification of ALU:** The inputs of the ALU always come from outputs A and B of the register file. Nevertheless, some instructions require manipulations on one or more bits. These are selected via multiplexors.
- **Address select for memory interface:** The address which is fed into the memory interface can be chosen from the instruction fetch, a register (output D of register file) or the ALU output with a multiplexor.
- **Memory load select:** As for the address, the load signal for the memory can be selected between decoder load and instruction fetch load.

2.2 RTL Verification

The testbench for the register transfer level verification of count32 and memcpy46 sample test programs consists of the memory module and the cpu top module. The clock and reset signals are generated in

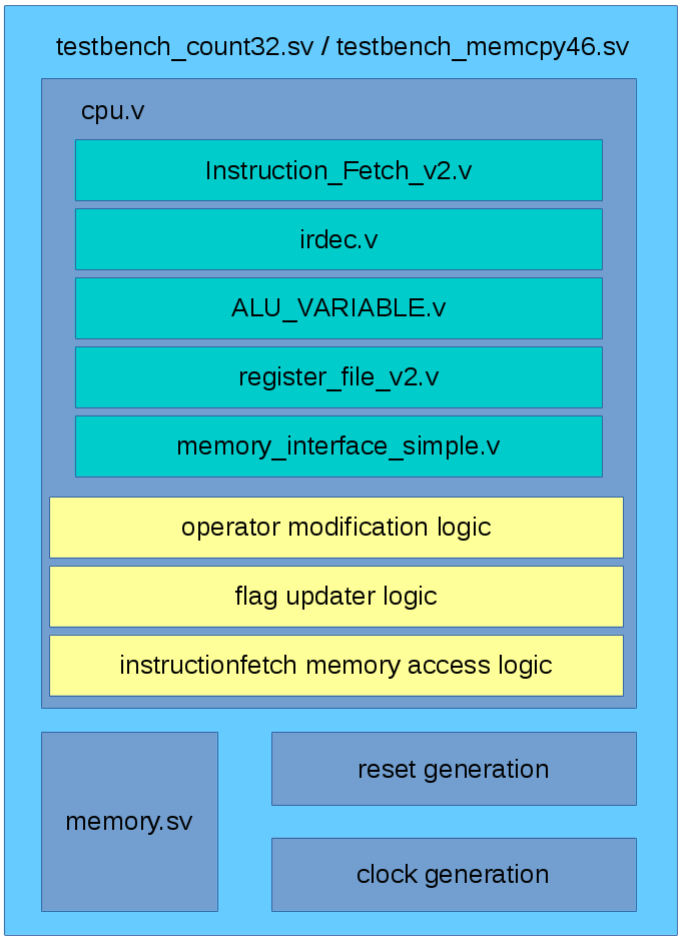

Figure 2.2: Structure of the rtl top level testbench

the testbench module. The cpu module reads the instructions from the memory module, which is loaded with the initial date for the sample programs, and also saves the results of the program executions in the memory.

In earlier stages of development, testbenches for the modules were used. After the modules were written and tested, they were integrated into the top module one by one and tested together. Before the integration of the instruction fetch module, the decoder and execution stage were tested together with different complicated instructions like IT, LDMIA and STMIA. The instructions were asserted directly in the testbench. The hex codes for the test programs were generated by the GNU ARM assembler which translated the written assembler test programs to the hex codes needed in the testbenches. C programs were unsuited for the testing at this stage because they did not necessarily generate the exact instruction sequences desired.

2.3 Synthesis

During RTL coding, special attention was given to good coding practice for synthesis, in order to ensure that the synthesized hardware would behave exactly as specified, without any glitches or timing issues. Therefore, each module was separately and iteratively synthesized to make the debugging process more clear and modular. The main concern and reason for iterations was the need to avoid using latches in our design, which was solved by carefully assigning all outputs for all input conditions in case or if-else statements. The main reason to avoid the usage of latches is that they are asynchronous storage elements, which would potentially cause timing issues and racing in a fully synchronous design.

Once the final top module was completed, the synthesis iterations were started in order to obtain the fastest hardware possible based on the informations given in the timing report. Starting with a period of 1ns as a constraint, the value was continuously reduced until it was no longer possible to obtain a slack greater than or equal to zero. A negative slack in the timing report means that it is not possible to generate a hardware to the given design that do not violate any timing request with the given period, while a positive or zero slack means that the synthesized hardware could be further optimized. The value used to synthesize the final hardware was 0.7 ns.

After the synthesis of the final hardware, an analysis of the schematic and the critical path was made in order to compare with the expected from the designed architecture. According to the timing report, the critical path consists of the instruction decoder reading a value from the register file, which is then processed by the ALU and written back to the register file. This path is expected due to the huge amount of combinational logic involved in the process, and the corresponding delay of the critical path is 0.69 ns.

As the guideline was to obtain a processor with the highest possible maximum frequency, there was no concern in optimizing the area or the energy consumption of the final hardware, nor to balance the three performance indicators. As a result, a processor with an area of 22453.12 μm^2 and a power of 6,9367 mW was obtained.

2.4 Gate Level Verification

After the synthesis, a gate level simulation is required to check if the behaviour of the final hardware corresponds to the logic simulation performed in the RTL coding stage, as well as to determine the maximum clock frequency under which the processor can run without any timing violations.

The two timing violations that must be avoided are the setup and hold time violations. A setup violation occurs when a signal does not arrive at a flip-flop with sufficient time, the so called setup time, before the next rising edge of the clock cycle. This means that the circuit is too slow for the critical path. On the other hand, the hold time of a flip-flop is the time that the input signal must remain there after the rising edge of the clock for the output to be safely switched. Therefore, a hold time violation occurs in the shortest paths and means that the circuit is too fast in that region.

With these concepts in mind, a testbench was run for each of the benchmark programs provided for the validation of the final synthesized processor design. Both testbenches were repeatedly run for increasing values of clock speed until any of them presented any timing violations. With this method, the lowest reached clock period under which the processor ran without any errors was 1.26 ns, which corresponds to a maximum frequency of 793 MHz. The final value of the clock period is way different than the one used as a timing constraint during the synthesis process, which was 0.7 ns.

2.5 Other

-

3 Evaluation

here some stuff on how good this processor is

- what could be better: - IT-instruction in fetch - better memory controller -lacks time - ALU with better solution for multiply (not single cycle) - decoder frees fetch earlier -

- process: -first benchmarks, then the rest -better look at rewuirements -better overview on what should happen

- conclusion: - very much work, lack of time - trotzdem instruction set working - have a more detailed look on requirements, e.g. byte-alignment -