University of Applied Sciences
Braunschweig/Wolfenbüttel

Design and Implementation of a FPGA-based Pipelined Microcontroller

Rainer Bermbach, Martin Kupfer
University of Applied Sciences Braunschweig/Wolfenbüttel
Salzdahlumer Str. 46-48, 38302 Wolfenbüttel, Germany
r.bermbach@fh-wolfenbuettel.de, martin_kupfer@web.de

Abstract

A multi-stage pipeline version of a PIC®-compatible microcontroller core has been developed to study the actual problems in pipeline design, possible approaches and the concrete solutions. This paper describes the design, the structure and the problems as well as solutions and the achieved performance of the pipelined microcontroller. The target for the pipelined core was a Xilinx Spartan-3 FPGA. For the PIC®-compatible pipelined controller a five stage pipeline was chosen with the stages FETCH, DECODE, operand READ, EXECUTE and WRITE back. So-called data hazards generate problems with the order of instruction execution. Data hazards are resolved by data forwarding or halting the pipeline until the memory position is written. With the developed effective data hazard detection unit most of the time data forwarding can be used thus eliminating nearly all cycle losses due to data hazards. Jump and call instructions etc. interrupt the normal sequential program flow resulting in so-called control hazards. Due to the efficient handling of the control hazard detection in the developed controller unconditional jump, call and return instructions are carried out without any cycle losses. The loss of cycles due to data and control hazards is reduced to a minimum. In the current version the pipelined controller runs at 70 MHz with approximately 1 CPI delivering a threefold performance increase over the conventional FPGA design and a factor of 14 compared to standard products.

1 Introduction

In modern electronic designs discrete microcontrollers and other devices are often replaced by cores and IPs integrated in FPGAs or gate arrays. These so-called System-on-Chip (SoC) designs allow for flexible development and system implementation. A conventional PIC®-compatible VHDL-based microcontroller core, some peripherals and a hardware based debug unit with trace capability (VHDL PIC) had been developed in a feasibility study at the University of Applied Sciences Braunschweig/Wolfenbüttel [2-7], which can be used for typical System-on-Chip designs. Running at 100 MHz in a Xilinx Spartan-3 FPGA (speed grade 4) the controller performs nominally 25 million assembler instructions per second. Practically, 20-23 assembler MIPS are reached in typical programs.

To further increase the performance of the controller the development of a multi-stage pipeline version of the VHDL PIC controller core was started. Besides rising processing speed the goal was to study the actual problems in a pipeline design, possible approaches and the concrete solutions, which can not be learned from text books [8] and papers, but only through own experience. Small microcontrollers hardly ever use pipelining or only in a very simple way, e.g. overlapping code fetch and execution as does the original PIC® from Microchip [1]. Thus it was interesting what the cost would be in terms of gate count etc. of a real pipelined architecture.

This paper describes the design, the structure and the problems as well as solutions and the achieved performance of the pipelined microcontroller. The target for the VHDL PIC in the pipeline version is again a Spartan-3. The Block RAMs are used to implement program memory, register file and stack. Basic design tasks were the implementation of pipeline registers between the different pipeline stages to pass on an instruction from one stage to the following as well as mechanisms for halting the pipeline or flushing a single instruction out of the pipeline. For the PIC®-compatible pipelined controller a five stage pipeline was chosen with the stages FETCH, DECODE, operand READ, EXECUTE and WRITE back. Chapter 2 gives some basic information about pipelining in processor architectures whereas the following chapter 3 describes the structure of the pipeline of the pipelined VHDL PIC.

Several difficulties arise from the use of pipelining in microcontroller architectures. So-called data hazards generate problems with the order of instruction execution, for example when a memory position needs to be read but a former instruction which is yet in the pipeline still needs to write its result to the same memory position. Data hazards are resolved by data forwarding or halting the pipeline until the memory position is written. Jump and call instructions etc. interrupt the normal sequential program flow resulting in so-called control hazards. The pipeline has to be flushed and refilled with new instructions from the place where the program continues. Typically, these control hazards have great impact on the performance of a pipelined architecture. Chapter 4 depicts the possible hazards in general and the actual problems occurring in the pipelined PIC architecture. Chapter 5 introduces into the solutions to the hazard problem. In the following chapter 6 the overall structure and the components of the realized pipeline are presented. Chapter 7 gives some results and performance figures of the implemented controller before chapter 8 concludes this paper.

2 Pipeline Basics

Pipelining is a mechanism to accelerate instruction execution [8]. Normally, all instructions are executed sequentially, running through the different units of a microcontroller: reading the instruction code from program memory (F), decoding it (D), reading operands (R), executing the instruction (E) and writing results to memory (W), see Fig. 1.

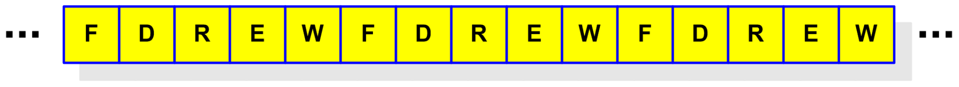

Figure 1: Sequential workflow of instruction steps in conventional architectures

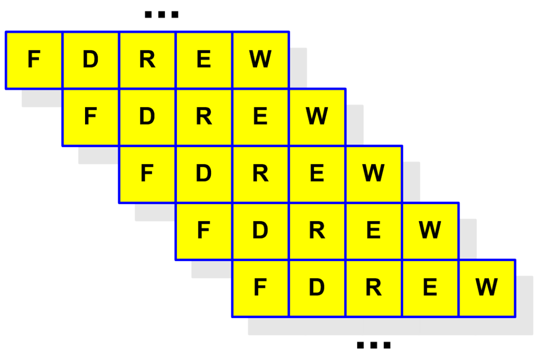

Figure 2: Parallel instruction handling in pipelined architectures

Similar to an assembly line, a pipeline architecture executes multiple instructions in parallel in the various units, so every stage handles another instruction every clock cycle (see Fig. 2). An instruction travels through all units entering the following stage with the next edge of a clock; thus, every clock cycle one instruction is finished. As one can see from the example, the pipeline accelerates instruction processing by a factor of five, i.e. the number of stages.

Unfortunately, there are limitations like hardware delays, dependencies between instructions and non-sequential code which limit the practical number of stages in a pipeline. The designer's task is to manage these limitations and develop a pipeline length that fits best.

Some problems in pipelined program execution result in reduced processing speed to prevent invalid results: Basic mechanisms for solving the problems are halting the pipeline until the execution is valid or erasing invalidated instructions from the pipeline. The main task when implementing a pipeline is to design special hardware for prevention of halting or even worse flushing the pipeline. To pass on the instruction and the progress made in one stage to the following stage a register has to be placed between these stages. Theses registers are called pipeline registers and catch all necessary data and signals for the successive stages.

3 Principal structure of the Pipeline

Following information processing in the original PIC® architecture [1] the instruction execution for the VHDL PIC is subdivided into five steps [9], so the final pipeline comprises five stages:

- Fetch: creates the program counter and fetches the instruction code from the program memory.
- Decode: decodes the instruction code and sets signals for the following stages.
- Read: reads an operand from the respective memory position.
- Execute: executes the instruction.
- Write: writes the result of the instruction to the memory position if required.

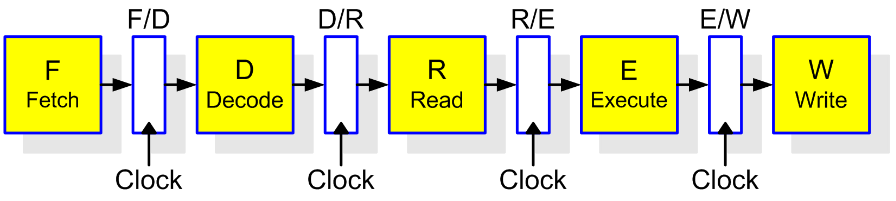

Figure 3: Pipeline stages coupled by registers

As shown in Fig. 3 adjacent stages are connected through pipeline registers. These registers store all information of an instruction at the (falling) edge of the clock to forward it to the following stage. They are named after the stages they couple, e.g. the F/D register connects the Fetch and the Decode stage. Signals created in the stage are stored in the respective register and signals from earlier stages are written to it to pass them on to their destination stage. The pipeline registers own functions for resetting, stalling and flushing:

- Reset: In case of a reset of the pipeline, all stage registers are filled with values as they would be created by a NOP (no operation).
- Stall: A stall indicates to all stages that a requested memory position is not ready to be read. So the pipeline is halted until the detected problem is resolved.
- Flush: The flush signal indicates to a stage, that it handles an invalidated instruction which has to be deleted from the pipeline. To cancel that instruction the involved registers are filled synchronously with the status of a NOP (no operation).

4 Possible Hazards in the VHDL PIC Architecture

As mentioned above there are some problems that limit the performance of a pipeline design. These limitations are called hazards. So-called *structural hazards* are caused by non-exclusive access to microcontroller resources like memories, registers or arithmetic units. Normally, all registers are implemented with one single port for reading and writing, thus it is impossible to perform a simultaneous read and write of a register which is necessary in a pipeline design. This also applies to the VHDL PIC pipeline architecture. Most of the registers and all of the data memory (register file) are single-ported and cause structural hazards.

Data hazards are generated by data dependencies e.g. when two or more instructions in the pipeline need access to the same memory position. Additionally, writing to one register may effect the execution of other instructions, e.g. updating the status register can influence the address building process. The most frequently data hazard occurring is a read after write hazard (RAW): One memory position is still in the pipeline and needs to be written back in WRITE while a subsequent instruction needs to read this memory position with the new value, but memory still holds the old numbers. Normally, the reading instruction and all subsequent ones must be halted until the write is performed to avoid invalid execution. The data hazards which can occur in the VHDL PIC are [10]:

Address building data hazards. The address is built by using bank bits in the status register or from the complete FSR register when using indirect addressing. Changes to one of these registers have to be completed before another valid address can be built.

RAW hazards. Read after write hazards occur when a memory position has to be read while a write to the same memory position is still to be performed. Thus the read will deliver an invalid result.

Flag hazards. Many instructions depend on the internal state of the microcontroller. When the flags in the status register are not updated before another instruction uses them during its execution the result may be invalid.

Data hazards when writing PCLATH. Building the new program counter (PC) when performing a branch (call, goto, write to PCL) is always depending on the PCLATH register holding the upper bits for the destination address. When the PCLATH register is written jumps have to be delayed to prevent an invalid program memory read.

Peripheral hazards. Peripheral registers are updated in WRITE and can have dependencies to other (peripheral) registers. So ports can be programmed as input or output by setting the respective direction register. If at least one bit of a port is an input, the direction register has to be written in WRITE before a new input value can be read.

All branches cause so-called *control hazards*. Normally, the pipeline is starting a new instruction every clock cycle. When detecting a branching instruction then the subsequent instructions which are already in the pipeline become invalid. Theses instructions have to be deleted from the pipeline to prevent incorrect updates in the microcontroller units. The program counter is loaded with the new execution position and the pipeline needs to be filled again. The possible control hazards in the VHDL PIC are:

Unconditional jumps: The PC of the next instruction is built from two bits of the PCLATH register and eleven bits of the jump opcode or fetched from the stack. GOTO/CALL jumps to a new position in program memory resp. to a subroutine (using stack). RETURN/RETLW/RETFIE returns from a subroutine normally resp. with loading a literal value into the W register or returns from interrupt service routine (all returns using stack).

Conditional jumps: The branch is performed only if the checked condition is true. In fact these instructions only skip the following instruction with a true condition. Otherwise normal execution of the subsequent instruction occurs. DECFSZ/INCFSZ decrements/increments an operand and skips the subsequent instruction if result is zero. BTFSC/BTFSS tests a single bit of an operand and skips the following instruction if the bit is zero/set.

Computed Jumps: A write to PCL (PC low byte) causes a branch (computed jump). All byte orientated file register operations with PCL as write target produce a write to PCL.

Interrupts: An interrupt suspends the normal program execution due to a predefined event and continues at PC = 0x004, where the interrupt service routine starts.

5 Handling the Hazards

The hazards in the VHDL PIC architecture described in the previous chapter have to be handled to obtain a valid and fast program execution.

The VHDL PIC owns a Harvard architecture, so there are different memories and busses for program code and data already in the standard architecture. This limits problems with *structural hazards*. An incrementer for the PC is available in the non-pipeline version, too. To realize simultaneous read and write access the file register is implemented as a dual-port RAM using the Xilinx Block RAM feature and dedicated registers work dual-ported, too. Different busses for reading and writing allow exclusive access to respective units.

Fighting *data hazards* is one of the most important parts in designing a pipeline. Mechanisms detecting, preventing or resolving them have a great impact on the whole design. All data hazards need to be detected on the basis of (operand) write addresses in READ in the data hazard detection unit. All instruction combinations which can cause a data hazard are classified into so-called condition instructions and so-called trigger instructions [10]. Condition instructions initiate a potential hazard by writing a result too late for directly following reads. If the following code does not access the results of the condition instruction no hazard is created. If an (trigger) instruction needs the results of the condition instruction before they are written it activates the data hazard. Generally, the data hazard detection unit monitors all instructions and detects possible condition instructions. Their operand addresses are stored and compared against addresses of the two subsequent instructions. In addition, the results of condition instructions are kept in a two-stage shift register for so-called data forwarding.

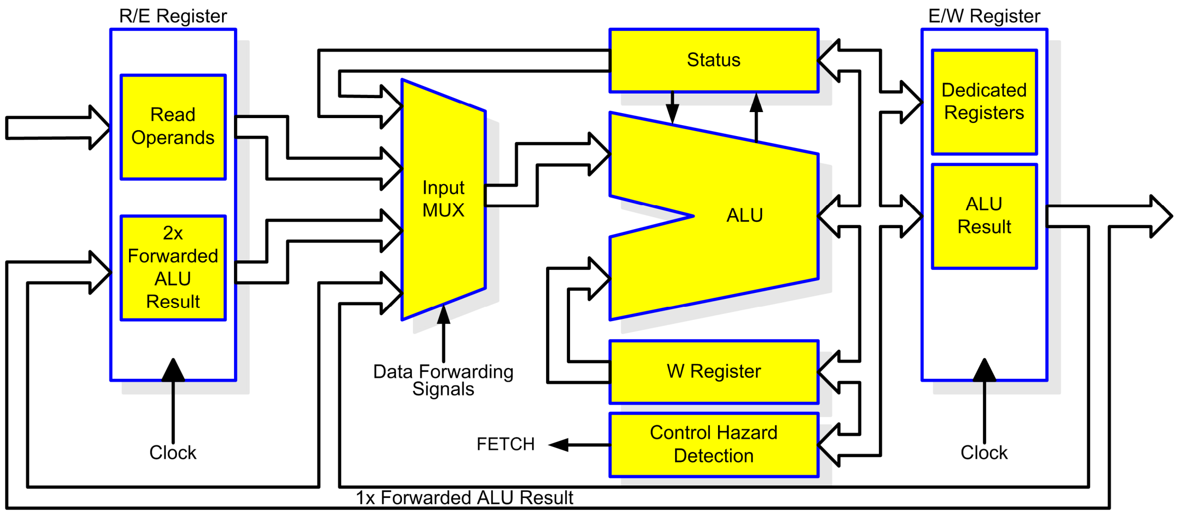

Figure 4: The EXCUTE stage with data forwarding of results

In case of a data hazard the pipeline normally has to be stalled for up to two cycles to deliver a correct result. A well known mechanism to prevent loss of cycles due to waiting for valid data is data forwarding. The data input of the EXECUTE stage can choose between different input sources which can be the data from READ, the result of the previous instruction or the result of the instruction two cycles before (see Fig. 4). Thus, the EXECUTE stage can work with the results of previous instructions directly instead waiting until these results are written to memory and then read from there. The resolution of the possible data hazards of the VHDL PIC architecture (compare to chapter 4) is implemented as follows:

> **Address building data hazards**: Building the address as late as possible in the pipeline reduces the loss of cycles; therefore it is located in READ stage. The status register is updated in EXECUTE to reduce losses during address building. If a write to status is detected in READ the subsequent instruction is halted for one cycle to prevent invalid program execution. Writing to FSR register is implemented in EXECUTE, too. If the subsequent instruction uses indirect addressing the pipeline is halted for one cycle.

RAW hazards: These hazards can be eliminated by using data forwarding as described above. The W register is written in EXECUTE, so it is always valid.

Flag hazards: The flags may be changed by an instruction with the status register as write target. This produces a loss of one cycle, see address building data hazards. In addition, flags are changed by many instructions to signal internal results writing the flags in EXECUTE. When a subsequent instruction has to read the flags from the status register, status is forwarded to the ALU input in EXECUTE instead of the read data from READ. Therefore no loss occurs when reading flags.

Data hazards when writing PCLATH: As mentioned above PCLATH is involved in building a new PC. Writing to PCLATH forces subsequent CALL/GOTO instructions to wait for storing of the new PCLATH value before they can execute. To speed up branch instructions PCLATH is written in EXECUTE and forwarded to FETCH to build a new PC while the write to PCLATH is executing.

Peripheral hazards: A write to a PORT is normally done at the end of WRITE; the new value is ready to be read one cycle later due to sampling mechanisms. A PORT can not be forwarded, because some of its bits can be declared as input (by writing to the respective TRIS register) and an input can not be forwarded. So it takes three cycles after detecting a write to a PORT or its respective TRIS register in READ, before the PORT can be read. This is the only data hazard where acceleration by forwarding is impossible.

Detecting *control hazards* in the pipeline as early as possible reduces loss of cycles. So the best way is to detect all jumps in FETCH and build the jump destination there. But this is not easy to implement due to instructions in the pipeline that impact on the jump or information for detection that is not yet available. The control hazards differ in the stage where they can be detected and the location where the jump destination is built: Some of the control hazards are making use of the ALU either to decide to skip or not, or to calculate the new PC. For a computed jump the address has to be computed in EXECUTE, too. Without optimization the pipeline needs to halt until the result is calculated. The handling of the possible control hazards of the VHDL PIC architecture (compare to chapter 4) is implemented as follows [11]:

Unconditional jumps: An unconditional jump is detected by analyzing its opcode in DECODE and signaled to FETCH. The jump destination address is built from eleven bits from the opcode and two from the PCLATH register and used to read the instruction code from the program memory in FETCH in the same cycle as it is detected. Therefore no loss of cycles is produced. (For stack handling see chapter 6.)

Conditional jumps: All conditional jumps are detected in DECODE, but the instruction has to pass EXECUTE where the test is made before the address of the next valid instruction is known. In modern processor architectures branch prediction is used to handle conditional jumps. Because the VHDL PIC's conditional jumps only skip or execute the subsequent instruction complex branch prediction is not necessary. Instead, all conditional jumps are assumed to not skip the next instruction, some kind of static branch prediction. When the forecast is true no cycles are lost. In case of a wrong prediction the subsequent instruction has to be flushed from the pipeline. Often a conditional jump is followed by an unconditional jump (loop), which means that the instruction in DECODE is the target of the unconditional jump and has to be flushed from the pipeline, too. In case the unconditional jump is falsely skipped (loop end) the pipeline has to be restarted with the PC of the conditional jump plus two. This generates a two cycle loss where one of these cycles belongs to the skipped instruction.

Another aspect is to make sure a wrong prediction does not invalidate the stack. Therefore the respective stack pointers of every instruction (see chapter 6) are stored and reloaded with their values when a prediction shows to be wrong. Only the stack pointers need to be restored, the stack entries never become invalid.

Computed Jumps: A computed jump is detected in READ. Then the instruction in DECODE is flushed from the pipeline because it is invalid. Additionally, building of a new PC and stack access is inhibited in FETCH. One cycle later, when the instruction is in EXECUTE, the result is forwarded to FETCH where the new PC is built and the instruction now in DECODE is flushed from the pipeline. This ensures that no internals (e.g. stack) are manipulated in an illegitimate way.

Interrupts: An interrupt is used for fast reaction on events. Therefore the interrupt functionality is implemented in a way that the microcontroller enters the ISR as fast as possible after an interrupt occurrence. When an interrupt occurs the PC is loaded with 0x004 where the ISR is located and the return address of a valid instruction is stored on the stack. Due to detected or still undetected control hazards it is not easy to decide which instruction is valid. Generally, the address of the instruction in READ is used as return address and the instruction in EXECUTE is the last executed before the ISR. Therefore only a computed jump or a conditional jump in EXECUTE can invalidate the instruction in READ. In that case the destination address of this branch is stored on stack. If the instruction in READ is an unconditional jump then this instruction is already executed and may have changed the stack. Therefore the destination address of this instruction is saved on the stack and the stack pointers are adjusted.

A summary of the loss of cycles generated by the different hazards is given below:

Address building hazards: one cycle delay, due to waiting for the new bank bits before another address can be built. Writes to the status register are seldom and the loss can even be reduced by the programmer if a write to status is not followed by a file register instruction. The same applies for using indirect addressing after a write to FSR register.

RAW hazards: no loss, all hazards are eliminated by data forwarding.

Flag hazards: no loss, for writing status see address building hazards.

Data hazards when writing PCLATH: one cycle delay – the loss can be reduced by the programmer: do not use a write to PCLATH followed by a CALL or GOTO instruction.

Peripheral hazards: up to two/three, can partly be reduced by sorting the instructions.

Unconditional jumps: no loss.

Conditional jumps: dependent on the behavior of the conditional jump – no loss, if no skip; one cycle delay if no unconditional jump is skipped and two cycles if an unconditional jump is skipped.

Computed jumps: two cycles delay – cannot be accelerated without spending a lot of hardware (reduction of maximum clock frequency)

Interrupts: normally two cycles – there are positive side effects with other control hazards, their losses are reduced then.

The original VHDL PIC needs four cycles to execute one instruction (fetch is performed in parallel) without any loss from data hazards, but all branches need additional four cycles. Thus all control hazards (except not skipping conditional jumps) need eight cycles to be performed. Compared to that, the loss of cycles in the pipelined VHDL PIC is always very low.

6 The Implemented Pipeline

This chapter describes the structure of the complete pipeline, for the principle configuration see Fig. 3. The purpose of the FETCH stage (see Fig. 5) is to build the PC for the next instruction and to read the instruction code from the program memory. Additionally, the stack handling is located in this stage. The PC is built asynchronously and accesses the program memory where the opcode is read with the falling edge; the PC is concurrently stored in a register. The building of the PC depends on control hazards and is controlled by signals from

DECODE, READ and EXECUTE. When no control hazard is detected the PC is incremented to build the new PC. The PC is saved (PC Branch) for another cycle for restarting the pipeline in case of a falsely skipped unconditional jump. All control hazard information goes to this stage; thus, signals for flushing instructions from the pipeline are set here, too.

As the stack is only for return addresses and not for data, this makes handling of the stack not too complex. The stack is implemented as a dual-port Block RAM with one port for write and one for read access. Thus, it is possible to read a return address or to write a return address simultaneously. The reading and writing ports have their own stack pointers. They are stored for two cycles to allow for restoring them in case of a falsely stack access due to a conditional jump. In case of a subroutine call the PC is saved on the stack and the stack pointers are incremented; in case of a return the already accessed return address is used to build the PC and the stack pointers are both decremented.

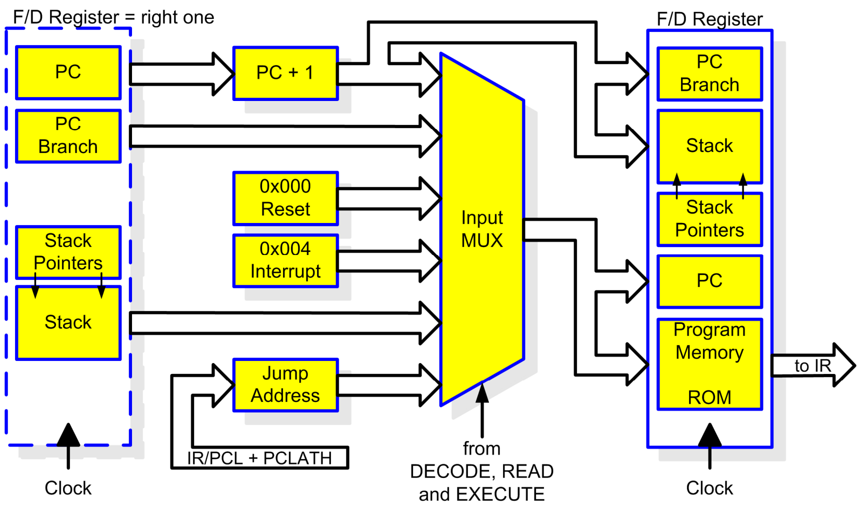

Figure 5: Structure of the FETCH stage

In the DECODE stage the opcode of the instruction read from the program memory is analyzed and signals for later stages are set for execution of this instruction. These signals are also used to detect control hazards. A detected control hazard is signaled to FETCH.

In the READ stage (see Fig. 6) the address for file register access is built and used for reading the data memory on the rising edge of the clock. The data read is stored in the pipeline register altogether with the information from former stages. The data hazard detection unit is located in READ, too. It uses the addresses for read or write access to detect any sort of data hazards and eventually sets appropriate signals for eliminating data hazards for halting the pipeline or activating data forwarding.

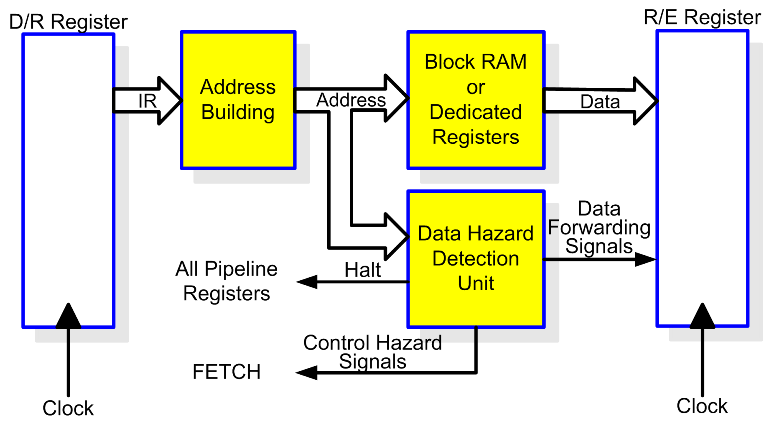

Figure 6: Structure of the READ stage

The EXECUTE stage (see Fig. 4) computes the result of the instruction depending on the inputs from the earlier stages. Therefore it includes the arithmetical logical unit to execute the instruction and calculate its result. The ALU is controlled by several signals passed on from earlier stages, mainly from DECODE. Some define the inputs of the ALU and others select between the different modes of the ALU. This stage decides whether a conditional jump skips or does not skip the subsequent instruction. Therefore a signal goes to FETCH to signal a skip. Additionally, the PCLATH and the result are forwarded to FETCH for fast reaction on control hazards. Some dedicated registers are written in this stage e.g. status, to reduce execution time. In the WRITE stage only data is written back to the file register.

7 Results

The implemented pipelined VHDL PIC was compared to the standard VHDL PIC. At present the pipeline runs with 66 – 70 MHz without any optimization versus 100 MHz in the standard version due to increase in hardware and complexity. In spite of that, processing is rising significantly. Two benchmark programs shall give an impression of the acceleration of the processing speed (see Tab. 1). The first one is a program which demonstrates usage of some peripherals on the Spartan3 starter kit board – a typical embedded program including a delay loop with four instructions. Here the positive effects of the control hazard handling show up.

Table 1: Acceleration of processing speed

Program	Standard (cycles)	Pipelined (cycles)	Ratio	Speedup (x 70/100)
Embedded I/O	1,249,700	250,452	4.99	3.49
Floating Point	3,348	1,007	3.32	2.33

The second program calculates a 32-bit floating point division. Here 106 of the 707 executed instructions test bits with 75 jumps taken, the static prediction being false. The speedup is lower due to the price paid for the control hazards. Though this code is atypical for microcontrollers it marks the lower limit of acceleration. Typical programs speedup in the range of 3.

Pipeline implementation and hazard handling increase the hardware requirements. As Tab. 2 shows from the map report, typically only 28% additional resources are needed (results after synthesis are comparable). Most of the extra hardware is used to implement the data hazard detection unit followed by the input multiplexers etc. of the PC and of the ALU.

Table 2: Usage of resources

Map report (total)	Pipelined	Standard	Increase in hardware
Number of slices (1,920)	772 (40%)	601 (31%)	+28%
Slice Flip Flops (3,840)	515 (13%)	356 (9%)	+45%
4 input LUTs (3,840)	1,097 (28%)	864 (22%)	+27%

8 Discussions

A five-stage pipeline version of a PIC®-compatible microcontroller core has been developed for a Xilinx Spartan-3 FPGA to study the actual problems in pipeline development. The design, the structure and the problems as well as solutions and the achieved performance of the pipelined microcontroller were discussed. In its five stages FETCH, DECODE, READ, EXECUTE and WRITE occurring data hazards are handled efficiently by the data hazard detection unit. Most of the time data forwarding can be used, thus eliminating any cycle losses due to data hazards. Control hazards inferred by various kinds of branches have to be dealt with not to loose the performance of the pipeline: Unconditional jump, call and return instructions are carried out without any cycle losses. Conditional jumps executing or skipping the following instruction are assumed to execute the following instruction – a kind of static branch prediction. If the assumption is correct no cycles are lost (typically true for over 90% in loops and for 50% in standard branching). In case of a false prediction the subsequent instruction is invalid and has to be deleted from the pipeline (one or two cycles lost). Performing computed jumps and interrupts cost two cycles, too.

The loss of cycles due to data and control hazards is reduced to a minimum with an increase of only 28% in hardware resources; very few losses are left e.g. due to waiting for necessary calculations in the EXECUTE stage for seldomly used instruction combinations. In those cases waiting seemed more effective than spending additional hardware. At present, the pipelined VHDL PIC controller runs at 70 MHz with approximately 1 CPI delivering a three-fold performance increase over the conventional FPGA design and a factor of 14 compared to standard products. The source code will be available for non-commercial usage at [12].

9 References

[1] Microchip Technology Inc.: PICmicro Mid-Range MCU Family Reference Manual. Chandler, Arizona, 1997.

[2] Cramm, I.: Entwicklung eines PIC-kompatiblen Mikrocontrollerkerns in VHDL. Diploma Thesis, University of Applied Sciences Braunschweig/Wolfenbüttel, Germany, 2003.

[3] Bermbach, R.: Entwicklung eines Mikroprozessorkerns. Research Reports, University of Applied Sciences Braunschweig/Wolfenbüttel, Germany, 2003/2004.

[4] Andreas, V.: Optimierung eines PIC-kompatiblen VHDL-Mikroprozessorkerns. Diploma Thesis, University of Applied Sciences Braunschweig/Wolfenbüttel, Germany, 2004.

[5] Kupfer, M.: Entwicklung der Hard- und Software eines Debug-Moduls für einen VHDL-basierten Mikrocontrollerkern. Diploma Thesis, University of Applied Sciences Braunschweig/Wolfenbüttel, Germany, 2005.

[6] Bermbach, R., Kupfer, M.: Development of a Debug Module for a FPGA-based Microcontroller. IFAC Workshop on Programmable Devices and Embedded Systems, p. 275-280. Brno, Czech Republic, 2006.

[7] Germann, U.: Erweiterung eines Debug-Moduls für einen VHDL-basierten PIC-kompatiblen Mikrocontroller um Hardware Watches und einen Trace Buffer. Diploma Thesis, University of Applied Sciences Braunschweig/Wolfenbüttel, Germany, 2007.

[8] Hennessy, J.L.; Patterson, D.A.: Computer Architecture: A Quantitative Approach. 4th Edition, Morgan Kaufmann, San Francisco, 2006.

[9] Kupfer, M.; Bermbach, R.; Patz, R.: Implementation of a Multi-stage Pipeline for an 8-bit Microcontroller. Procceedings of the 1st Research Student Workshop 2007, p. 109-110. University of Glamorgan, Faculty of Advanced Technology, Pontypridd (Wales), Great Britain, 2007.

[10] Breustedt, P.: Analyse und Behandlung von Datenhazards in der Pipeline eines VHDL-Mikrocontrollers. Diploma Thesis, University of Applied Sciences Braunschweig/Wolfenbüttel, Germany, 2007.

[11] Krüger, M.; Meinecke, A.; Renz, A.; Weiß, N.: Behandlung von Kontrollflusshasards in einer fünfstufigen Pipeline. Project Thesis, University of Applied Sciences Braunschweig/Wolfenbüttel, Germany, 2008.

[12] http://public.fh-wolfenbuettel.de/~bermbach/vm/vhdlmicro.htm