HWSW Synthesis

authors

October 2014

Contents

1 Goal of the project **2**

1.1 Specification . . . 2

2 Hardware **4**

2.1 Control Unit . . . 4

2.2 Scheduler . . . 4

2.3 Pipeline . . . 4

2.4 Instructions . . . 7

2.5 Simulation . . . 7

3 Software **9**

3.1 Device Interface . . . 9

3.2 Host Software . . . 10

4 Conclusion **12**

1 Goal of the project

The main subject of this project is to develop a hardware/software system for accelerating numerical computations. One Task is to design a coprocessor, which is then synthesized on an FPGA. This project has a software part as well. Embedded software is written and consists of a driver running on a host PC and a driver running on a CPU on the FPGA.

1.1 Specification

Host API The host computer is responsible for managing the coprocessor and is also in charge of the device memory. Allocating/Freeing and moving data are operations the host API has to be capable of. Furthermore, the host controls the execution of coprocessor kernels, to be more specific, the host has to be able to start and stop a kernel, query its current state and wait for it to finish.

Device API The PCP has to process kernels written in an own assembly language. It can execute 256 concurrent threads, which are able to diverge. Function calls or a stack are not supported. Different threads can communicate by accessing a scratchpad memory and synchronization.
Memories and Registers:

Device Memory	512MiB of DDR3-SDRAM on the FPGA-board
Scratchpad Memory	16kiB, low latency, can load or store a 4-byte word per thread in each cycle
Instruction Memory	8kiB, for up to 2000 instructions
Registers	full set of registers per thread (32 general purpose 32-bit registers, 8 boolean condition registers)

Instructions The PCP's architecture is similar to the architecture of a RISC processor. The instruction set consists of arithmetical and logical, comparison, load/store and control flow operations.

Execution Model Threads are executed in groups, called *warps*. Those warps run on the processor in a SIMD[1] fashion similar to CUDA, so all threads within a warp are implicitly synchronized. If some threads diverge within one warp, the diverging threads are set inactive, which means they won't execute anything until the remaining active threads reach the same program counter as the inactive threads. The decision which part of the warp is set inactive has to be made by the programmer.
Warps can be executed independently of each other so there is no need for explicit synchronization. Nevertheless, the possibility of synchronizing warps

[1]Single Instruction Multiple Data

exists as a *SYNC* instruction. If there is a data-dependency within one program execution, it is not handled explicitly because warps are only scheduled again after they have passed through the pipeline and are completed. This means that instructions within one thread are executed in program order.

2 Hardware

In this section the pipeline structure of the PCP and it's pipeline stage compositions are explained and described in detail.
When the processor is supposed to execute a program, it is necessary to fill it's instruction memory first. This initially happens via the AXI-bus. The last instruction sent to the instruction memory is a *START* instruction which defines how many threads should be executed and starts the processor.

2.1 Control Unit

After the instruction memory has been filled and the processor has been started, the control unit initially fills the scheduler with required data for the first run.and then unstalls the pipeline.
In case of a *TERM* instruction, the control unit stalls the pipeline and a bit is written to the AXI bus signaling that the processor has terminated.

2.2 Scheduler

The scheduler consists of three separate FIFO[2] queues of which two get warps as inputs from the pipeline. The third FIFO receives inactive warps as inputs if a *SYNC* instruction is executed.
After a warp has passed through the pipeline it is put back into a FIFO. The first two FIFOs are filled by the *write back* stage or the memory controller respectively, since the memory works asynchronous and a warp can only be rescheduled if a *LOAD* from memory is completed,
The third FIFO is also filled by the *write back* stage, but only with warps that execute the *SYNC* instruction. If this FIFO contains all active warps, its content becomes available for scheduling again.

2.2.1 FIFO

For the FIFOs, the Xilinx "FIFO_SYNC_MACRO" is used. It is synthesized on the FPGAs block RAM and can be configured as 18kB or 36kB memory. For this project, a size of 36kB is used. The macro allows up to 72 bit data width, out of which 54 are used here. For initializations, this macro need 5 clock cycles. Because filling the instruction memory happens before the FIFOs are initially filled, no counter is needed, since 5 cycles will have passed when the instruction memory is filled.

2.3 Pipeline

Since the coprocessor is synthesized on an FPGA, a pipeline is used to exploit concurrency. Since enough space is available on the FPGA, it makes sense to

[2]FIFO: first in first out

use this logic for a pipeline. This pipeline is quite similar to a standard RISC-pipeline and is composed of five pipeline stages.

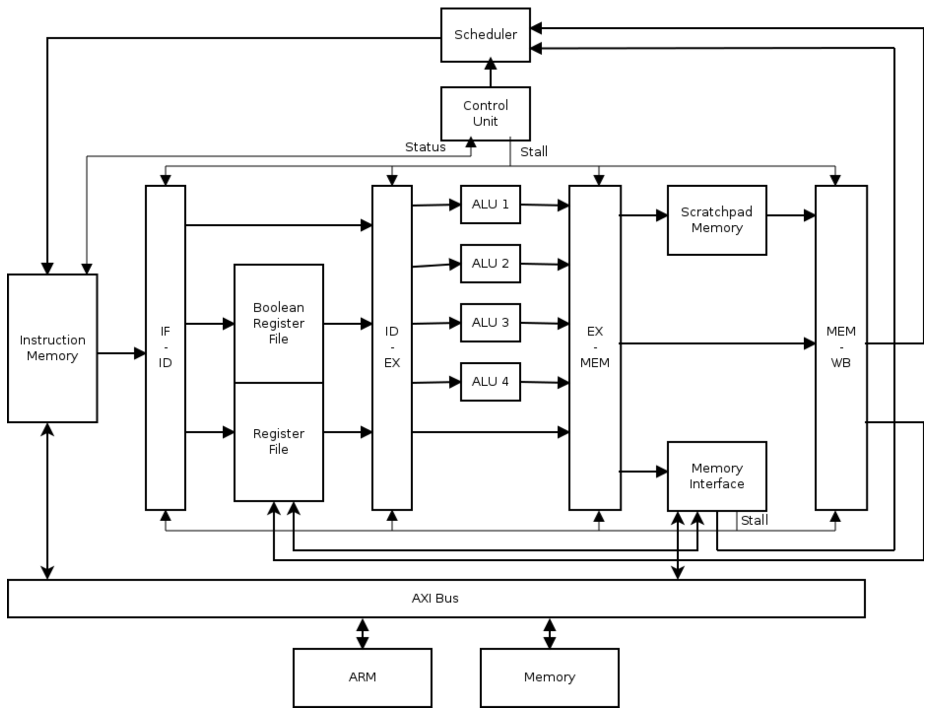

Figure 1: Pipeline of the processor

2.3.1 Instruction Fetch

During this pipeline step an Instruction is fetched from the instruction memory which requires one clock cycle, like all other pipeline stages. The instruction memory has a size of 8kiB and is implemented as single port ram, so it can be synthesized on the block RAM. The instruction width is 32Bit. It is directly connected to the AXI bus and can be filled through it.

2.3.2 Instruction Decode

The Instruction Decode pipeline stage contains one normal and one boolean **register file**. Values needed for the current instruction are loaded from those and are passed to the next stage. It also has inputs for results from the memory controller and the *write back* stage.
The normal register files hold 32 registers of 32 bit width for each thread, the boolean register file contains 8 one-bit registers for each thread. The first register of each thread discards all reads and always return zero for normal and true for boolean registers..

Each register file consists of four register entities. An entity holds the registers of one fourth of the threads, lies in the FPGA's block RAM and allow one read and one write in each clock cycle. The four entities in the register file are double clocked to allow a total of two reads and writes per cycle for four threads. The order of reads and write is irrelevant since the executions model excludes all cases where the same register is written and read in the same cycle.
This is necessary to, one one hand, read two operands for one instruction in one cycle and, on the other hand, handling results from asynchronous *LOAD* instructions while still being able to write results from the write back stage..

2.3.3 Execute

Since four identical instructions are executed at the same time with possibly different data, the execute stage has four **ALUs**[3]. Although the opcode is the same for all four ALUs, they execute four possibly different calculations since their data might differ. Only the ALU results of active threads are passed through to the next stage. For instructions like memory access instructions that do not use the ALUs, data is passed through again.

2.3.4 Memory Access

In this pipeline stage, memory access instructions read from or write to the memory and scratchpad memory access instructions access the scratchpad memory. The data memory is connected asynchronously via the AXI bus.
If a memory access instruction occurs, the pipeline has to be stalled until the bus is ready. If the bus is ready, the instruction is executed, but the current warp will not be written back into one of the FIFOs of the scheduler yet. This happens when the reply arrives.Therefore all information that is needed to do the write back and reschedule the warp, like target register and warp id, are stored by the memory controller.
The scratch pad memory is used for communication between different threads, has a size of 16kiB and is able to store 4 bytes per thread per cycle.

2.3.5 Write Back

In the write back stage, results from arithmetic operations and loads from scratchpad or immediate values are written back into the registers. Also, except for the asynchronous memory operations, the current warp is rescheduled.
Further, in this stage, jumps are handled. In the case of jump instructions, the corresponding threads are set inactive and if active threads catch up with active threads, they are set active again.

[3]ALU: arithmetic logic unit

2.4 Instructions

Register Type (ADD, SUB, AND, OR, XOR)						
Cond	Instr	RegTarget	Unused	RegA	RegB	Unused
0..2	3..7	8..12	13	14..18	19..23	24..31

Shift (SLL, SRL)						
Cond	Instr	RegTarget	Unused	RegA	Amount	Unused
0..2	3..7	8..12	13	14..18	19..23	24..31

Immediate (LOADI,LTID)						
Cond	Instr	RegTarget	Immediate/Unused			Unused
0..2	3..7	8..12	13..28			29..31

Bool Register Type (EQ, LT)						
Cond	Instr	RegTargetA	RegTargetB	RegA	RegB	Unused
0..2	3..7	8..10	11..13	14..18	19..23	24..31

Mem Type (LW/LWS)						
Cond	Instr	RegTarget	Unused	AddrReg	Unused	
0..2	3..7	8..12	13	14..18	19..31	

Mem Type (SW/SWS)						
Cond	Instr	Unused		AddrReg	DataReg	Unused
0..2	3..7	8..13		14..18	19..23	24..31

Jump (JD, JDN)						
Cond	Instr	Unused	JumpAddr			Unused
0..2	3..7	8..12	13..28			29..31

Misc (TERM, SYNC)						
Cond	Instr	Unused				
0..2	3..7	8..31				

Processor Control (START)						
Cond	Instr	ThreadCount		Unused		
0..2	3..7	8..15		16..31		

2.5 Simulation

For testing purposes, simulation is used to test the coprocessor. After all important components of the processor are tested by distinct simulation testbenches, all inputs to the coprocessor are set by a simulation testbench for the whole coprocessor. For every type of instruction, a constructor is written, so that one function can create all inputs to the coprocessor for loading an instruction into the instruction memory.
Example code for simulation covering the most important instructions and instruction types:

```
inst(0,op_LTID,1,0);              LTDI(R1)
inst(0,op_LOADI,2,1);             LOADI(R2,1)
inst(0,op_EQ,1,2,1,2);            EQ(R1,R1,BR1,inverseBR2)
inst(1,op_JD,5);                  JD(5)
inst(0,op_LOADI,5,5);             LOADI(R5,5)
inst(0,op_LOADI,4,1024);          LOADI(R4,1024)
inst(0,op_SWS,5,4);               SWS(R5,R4)
inst(0,op_LWS,6,5);               LWS(R6,R5)
inst(0,op_SW,5,4);                SW(R5,R4)
inst(0,op_LW,6,5);                LW(R6,R5)
inst(0,op_JD,4);                  JD(4)
inst(0,op_SYNC);                  SYNC()
inst(0,op_TERM);                  TERM()
inst(0,op_C_START,7);             START(7)
```

Besides various instructions, more complicated scenarios like diverging threads and terminating the program are tested. To test mem type instructions, a dummy memory is connected to the AXI bus.

3 Software

3.1 Device Interface

The interface to the parallel co-processor is implemented in C on the ARM Cortex-A9 processor on the Zync-7000 SoC. It handles communication with the host system, memory allocation and freeing, data transfers and directly controls the PCP.
On the network side, the lwIP[4] stack is used to communicate over TCP/IP using a simple protocol. Packets always start with the string *PCP*, followed by a single character specifying the type of the packet. The rest of a packet is used for an optional payload. Commands are always issued by the host, used types are shown in table 1. The ARM processor sends back either an acknowledgment, a packet with type *A* and an optional payload, or an error packet with type *E* and an error code specifying the type of error that has occurred.
Memory operations executed directly on the specified addresses using the standard *alloc*, *free* and *memcpy* functions. Communications with the PCP are done through the AXI bus, which is memory mapped on the ARM processor, using the Xilinx provided *Xil_In32* and *Xil_Out32* functions.

Type specifier	Library method	Parameters/ Payload	Description
S	start	thread_count	Starts *thread_count* threads on the PCP.
O	stop	-	Stops execution
T	state	-	Queries the PCP state.
U	load	[instructions]	Uploads a kernel to the PCP.
M	malloc	word_count	Allocates *word_count* words in the device memory.
F	free	address	Frees the specified memory *address* on the device.
D	memcpy_to_dev	address, word_count, [data]	Copies *word_count* words of *data* to memory *address* on device.
H	memcpy_to_host	address, word_count	Copies *word_count* words from the device memory to the host.
C	memcpy_dev_to_dev	source, target, word_count	Copies *word_count* words from *source* to *target* address on the device.

Table 1: Packet types

[4]https://savannah.nongnu.org/projects/lwip/

3.2 Host Software

On the host side, we provide a small Python library to work with the PCP. It consists of two parts: A *PCP* class to abstract network communication and a simple assembler.
After the class is instantiated with IP and port of the PCP, the methods listed in table 1 can be used to access the PCP. For most methods, all parameters are passed to the PCP unchanged. The only exception is the *load* method, it takes additional parameters to the kernel, which is passed as a text string, and passes the kernel to the assembler first to generate PCP machine code.
During assembly, first, comments (lines beginning with #) and white space is stripped from the program. Then parameter placeholders in immediate commands , e.g. memory addresses, which are prefixed with a *!* are replaced with the given values. Immediate commands are, if necessary, expanded to multiple instructions. If the immediate value exceeds 16 bit, the load is split up into four instructions, an immediate load of the lower 16 bits into the target register, an immediate load of the higher 16 bits into register 31, a left shift by 16 bit of this register and an or operations of both into the target register. For this reason, register 31 is reserved and should not be used for other purposes.
Finally the lines are enumerated and jump labels are replaced by the corresponding jump addresses. Then instructions are simply translated one by one by converting the single tokens to their binary representations.
A full overview of all assembly commands is given in table 2, all instructions are prefixed by a conditional register of the form $br[0-7], all registers have the from $r[0-31].

Instruction	Function
reg_cond term	End kernel
reg_cond sync	Synchronize all threads
reg_cond jd label	Jump to label, threads which take the jump continue
reg_cond jdn label	Jump to label, threads which don't take the jump continue
reg_cond add reg_target reg_a reg_b	Add a and b into traget
reg_cond sub reg_target reg_a reg_b	Subtract a and b into traget
reg_cond and reg_target reg_a reg_b	Logical and of a and b into traget
reg_cond or reg_target reg_a reg_b	Logical or of a and b into traget
reg_cond xor reg_target reg_a reg_b	Logical xor of a and b into traget
reg_cond sll reg_target n	Logical left shift of target by n bits
reg_cond srl reg_target n	Logical right shift of target by n bits
reg_cond loadi target_reg value	load value into target
reg_cond ltid target_reg	load thread id into target
reg_cond eq reg_cond_a reg_cond_b reg_a reg_b	Check if a and b are equal, write result to a and inverse result to b
reg_cond lt reg_cond_a reg_cond_b reg_a reg_b	Check if a is less then b, write result to a and inverse result to b
reg_cond lw reg_target reg_addr	Load addr from memory into target
reg_cond sw reg_addr reg_src	Store src to addr in memory
reg_cond lws reg_target reg_addr	Load addr from scratchpad into target
reg_cond sws reg_addr reg_src	Store src to addr in scratchpad

Table 2: Assembly

4 Conclusion

Due to time constraints, the project wasn't completely finished. In simulation, the processor was thoroughly tested and working, because finding and fixing bugs was in simulation proved to be quite simple.
On real hardware, however, it took a lot of time. Completely rebuilding the project for every debug signal and fix is really slow and tedious. So in the end, the processor was able the receive and execute kernels correctly, but not able to access the main memory in any way. The cause is yet unknown. Therefor it was not possible to do any performance testing. The clock speed also had to be reduced to fulfill the timing constraints of the real hardware.
In spite of everything, the project proved to be an interesting experience, because designing hardware was a new experience and is quite different in thinking and writing code, and is, after some initial problems, not that much trouble.
Working with the provided tools was not that easy, since they are on some points a bit broken and not very intuitive. The documentation was really detailed, but it was sometimes really difficult to find the right parts.
So, in the end, we had a real hands on experience in hardware design, learned a lot, and, with some further debugging and testing, it should be possible to get the processor running how it was supposed to be.