Project: Cache Controller for ARM Cortex M0 on a Xilinx Spartan 6

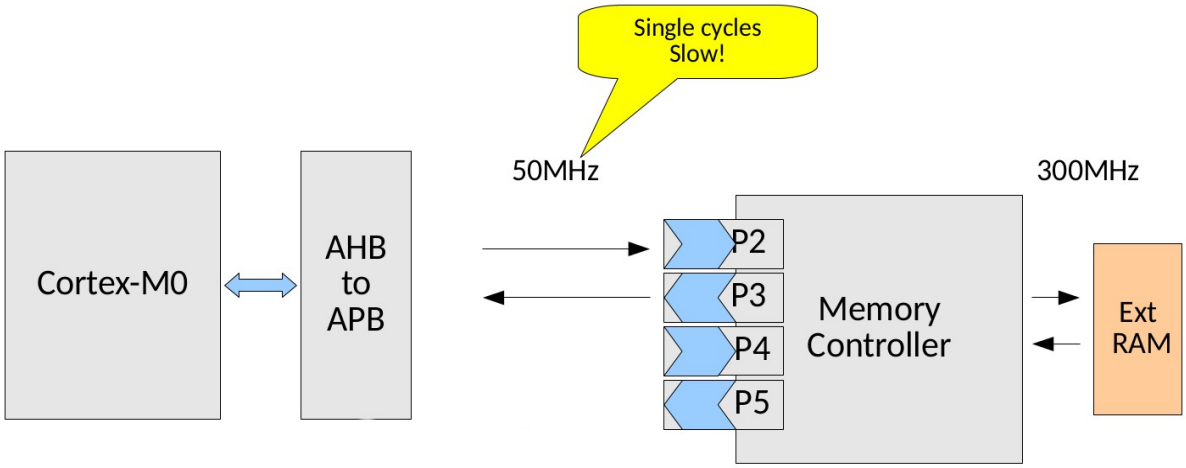
University of Heidelberg
ZITI

Tim Schneider,
Motitz Nöltner-Augustin,
Dennis Sebastian Rieber

Supervisor: Dr. A. Kugel

Table of Contents

Motivation



Drawing 1: System architecture without cache

The performance of processors and microcontrollers is heavily depending on the memory performance, especially how fast they can load instructions and data from the memory. The figure shows the current architecture of the system we are working on. It has no cache and the memory is connected to the AHB Interface of the Cortex M0, where it bridges to APB interface. The memory controller is then connected to APB interconnect. This results in multiple cycles of delay for each read and write operation that is performed, during which the processor can stall completely. The first cycle we lose is the AHB to APB bridge, then each DRAM access internally needs ~9 cycles at 300MHz to fetch or write the data, which translates to ~2 additional CPU cycles at 50MHz and finally another APB to AHB bridge cycle. So in total we have at least 4 cycles of delay for each read or write operation. This also accounts for instructions, which inflicts a huge performance penalty, even if we only work on registers in the microcontroller.

Goals

The goal of this project is to implement a cache that is connected to the AHB interface and the memory controller in order to reduce the memory access latency and increase the overall performance. Therefor, we will create an IP that will replace the AHB to APB bridge as the processors' memory connection and that contains the cache controller. This method makes the cache controller 100% transparent to the processor. The controller's design will be a direct mapped cache that is adjusted to the system's memory and processor peculiarities. The individual tasks and their results are listed in chapter 4, after the design has been explained in more detail

Structure

In the next chapter, some basics concerning the used technology will be explained. Chapter 3 is giving a brief introduction to caching strategies and technologies. Chapter 4 presents the created design and results in detail and chapter 5 gives some final remarks.

Basics

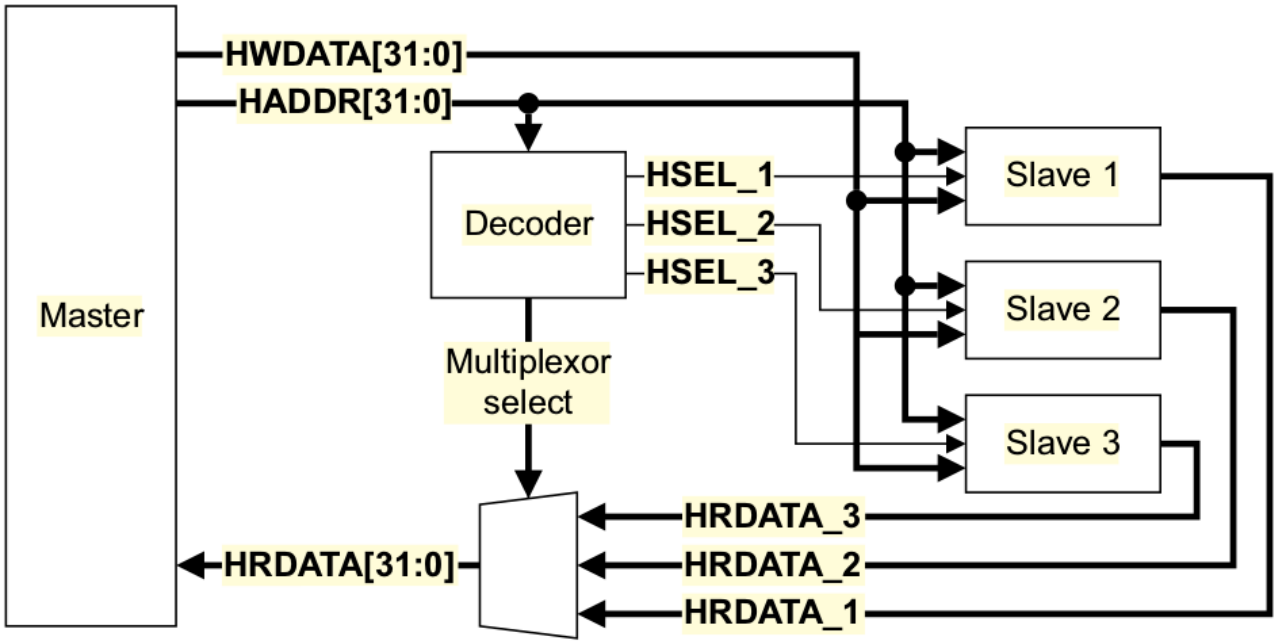
ARM Cortex M0

The ARM Cortex M0 is the smallest available ARM Cortex M model. It is classified as a 32bit RISC microcontroller with a 3-stage pipeline. To keep this pipeline filled, the Cortex M0 has an instruction buffer for two instruction, plus two instructions that can be fetched at once. This means the next four instructions are always available. (http://infocenter.arm.com/help/topic/com.arm.doc.dai0321a/DAI0321A_programming_guide_memory_barriers_for_m_profile.pdf, page 16)

Since we have an estimate delay of four cycles for each load, the buffer and instructions on the fly can barely suffice the processor's needs for an instruction at each cycle. Additionally, the instruction loads are competing with the data loads and stores for memory access. And since data and instructions are usually in separate parts of the memory, additional latency might occur due to possible bank switching in the DRAM memory.

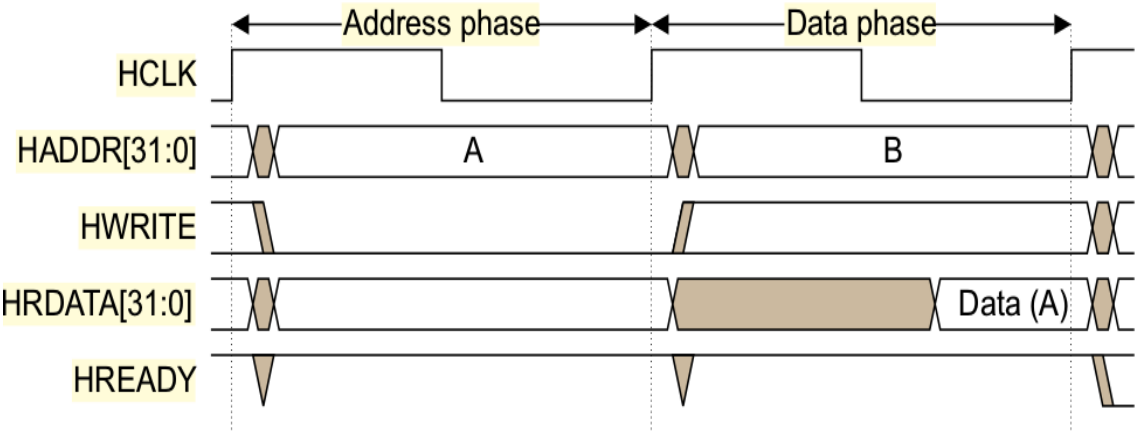
AHB-Lite

AHB-Lite stands for “Advanced High performance Bus” and names a protocol and interface that is used on the ARM AMBA bus systems to provide high-bandwidth operations for the participators. It is a Master/Slave split phase transaction bus system where the master request reads/writes from/to slaves, which is implemented as a memory mapped interface. The two phases are address phase and data phase, which overlap each other. This means that during the data phase, the next address is already written to the address signal HADDR, so that the next cycle can begin as soon as the previous cycle is completed.



Drawing 2: AHB-Lite Architecture (Image: Xilinx User Guide 388)

- **Address Phase:** The master writes the desired address to HADDR. The decoder snoops it and notifies the responsible slave with the HSEL signal. The data phase begins as soon as the Master signals HREADY. With the HWRITE signal, the Master defines if this is a read or a write transaction.
- **Data Phase:** The Slave is then reading/writing the data. If the action that needs to be performed takes longer than one bus cycle, the slave signals this with the HREADYOUT signal. It can extend the data phase until it can deliver the data or secured the read. The slaves write to HRDATA, which is multiplexed back into the master.



Drawing 3: AHB-Lite read transfer. Notice that the address for transfer B is already put on the bus, while transfer A is still in the data phase. (Image: Xiling User Guide 388)

Actually, there is a number of additional signals and interactions that can occur during a bus transaction, but for the understanding of the project, the given description is sufficient. For more information see Xilinx UG388.

Memory Controller

To access the DRAM the standard Xilinx memory controller IP is used. It has six channels to which devices can connect. Each channel is divided into a command path, a read path and a write path. Two channels are bidirectional, four channels are unidirectional. To access the DRAM, one bidirectional channel is used. This channel uses the same clock as the cache controller itself. More details on this IP have been provided during the lecture.

The project uses 16 Mbyte of DDR2 memory operating at ~300MHz.

Performance Indicators

Since the project aims to improve the memory throughput, the most significant performance indicators are the memory access latency and the available read and read/write bandwidth. These metrics can be used to benchmark applications that are predominantly memory bound. However, on algorithms that are calculation bound, we still can achieve a speedup by reducing the instruction fetch latency.

Bandwidth and latency however, can only be used to indicate the overall application performance development and tell us nothing about the cache controller. To check how well our cache is actually performing, the most important metric would be the hit rate. It is the percentage of memory reads that actually result in a successful cache lookup. The higher this number is, the better.

Since the microcontroller has no data cache, and the instrcution buffer is to small to hold a significant number of instructions, the performance is expected to be very low. We

measured the read bandwidth that we can achieve with the current setup and the results are displayed in the table below.

Benchmark	Read Bandwidth (Kbyte/s)
Global Sum (Read only Access)	2
Prefix Sum (Read and Write Access)	1

We used two benchmarks to determine the read performance, the single most significant area of performance improvements by caches. A global sum, which only requires a subsequent number of reads and a prefix sum that consists of two reads and one write. Since one of the reads is the value of the previous iteration, we can assume that it remains in a register for use in the next calculation. So we have one read and one write. This is also displayed in the benchmark results, where the global sum has two times the read bandwidth, compared to the prefix sum where reads and writes compete for memory access.

Both benchmarks have been performed with 32bit integer data to eliminate delays due to floating point arithmetic. Since we have no caches or other resources that can influence the performance over different problem sizes, the benchmark results are consistent over different (reasonable) problem sizes.

Cache

There is a huge demand on large amounts of high speed memory. High speed memory is cost-intensive and of small size, commonly less than 1 MB, high density memory with large capacity is cheap and slow. An economical way to satisfy this demand is to use memory hierarchies, which take advantage of locality and trade-offs in the cost-performance of memory-technologies.

The principle of locality, says that most programs do not access all code or data uniformly. Locality occurs in time (temporal locality) or in space (spatial locality). This principle led to hierarchies based on memories of different speeds and size.

Since fast memory is expensive, a memory is organized into several levels, each smaller, faster and more expensive per byte than the next lower level, which is closer to the main memory. The hierarchy is in most cases transparent to the CPU and moreover to the programmer, which means that there is a virtual memory address space to the CPU and the data is physically stored somewhere in the memory hierarchy (comp. Computer-Architecture-A-Quantitative-Approach-5th edition, Page 72 – 73).

The memory is usually organised in bytes and a group of bytes are called a block or a line with e.g. 2, 4, 8, 16 or more bytes.

When a word is not found in the cache, the word must be fetched from a lower level memory and placed in the cache before continuing. For efficiency reasons more than one word, the so called cache line is moved in the cache, because it is likely that they are needed soon due to the spatial locality. Each cache line includes a tag to indicate which memory address it represents.

The cache storage scheme defines where a cache line can be placed in the cache. The set associative scheme separates the cache into multiple sets. A set is a group of cache lines. A cache line is first mapped onto a set and then the cache line can be placed anywhere within that set. Finding a cache line consists of first mapping the cache line address to the set and then search the set – usually in parallel – to find the cache line. The set is normally chosen by the address of the data, by way of modulo operation.

If there are n cache lines in a set, the cache placement is called n-way associative. The end points of set associativity have their own names. A direct-mapped cache has just one cache line per set i.e. a cache line is always placed in the same location and a fully associative cache has just one set i.e. a cache line can be placed anywhere (comp. Computer-Architecture-A-Quantitative-Approach-5th edition, Page 74).

If data in a cache line is modified the main memory needs to be updated somehow. There are two major strategies to handle the consistence of the main memory. The write-through cache updates the item in the cache and writes through to update the main memory. A write-back cache only updates the cache and when the cache line is about to be replaced, the cache line is copied back to the main memory. Both write strategies can use a write buffer to allow the cache to proceed as soon the data are placed in the buffer rather than wait until the write to main memory completes.

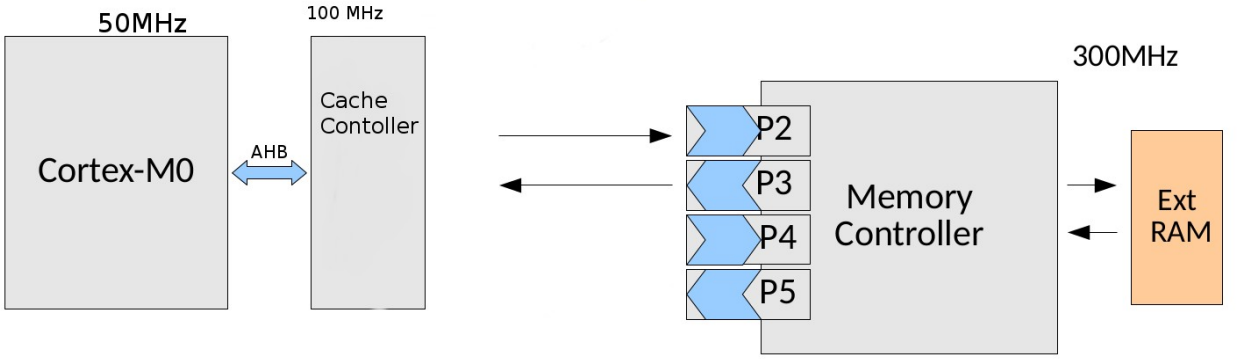
If a cache stores both, instructions and data, it is called a unified cache.

Implementation

Cache Design

For this project we decided to use a direct mapped cache. This type of cache is straightforward to implement and can deliver data with the least latency, because there is no need to a hit-compare between the sets.

To cache the 16Mbyte of memory we will create a unified cache with the size of 4kByte, divided into cache lines of 32 bytes each. This means 128 cache lines that can hold, for example, 8 32bit values or 16 16bit instructions. The cache is byte addressable. The cache uses the write-through strategy. This means that a write is always relayed to the DRAM, while the cache line is updated.



Drawing 4: System architecture with cache. Now the memory is directly connected to the AHB bus, without a bridge to the APB interconnect.

Besides the 8 words of data, a cache line consists of the TAG which stores the base address of the data in the cache line and a bit that indicates whether the data in the cache line is valid. The TAG consists of the upper bits of the memory address, since the lowest 12 bits are used to address the data in the cache. It is used to determine if the cache line that is in the cache right now belongs to correct memory region, because after 4kByte of memory, the lowest 12 bit repeat themselves.

The cache is empty after reset, meaning all cache lines are invalid, but by design a cache line cannot become invalid during normal operation. However, there may be other devices accessing the memory through another channels of the DRAM controller, for example the DMA module or the framebuffer. To allow for sustained consistency of the cache, the processor can request address ranges to be flagged as invalid, if present in the cache.

A downside of direct mapped caches is possible thrashing of data, when instructions and data reside in different areas of the memory, that map to the same cache line. However, since we are on an embedded system, we can avoid that by manually placing the data in a memory area that is not affected by this (granted, the data is small enough).

We aim to run the cache controller at 100MHz clock-synchronous with the 50MHz AHB-Lite bus. In case of a read access, this gives us two bus cycles to determine a cache-hit and deliver the data to the bus. This would result in a zero cycle hit response. A cache miss results is an extended data phase, since the data needs to be fetched from the DRAM. A write request can almost always be serviced in a single cycle. This logic will be embedded into two state machines that run independent form each other, one for reading and one for writing. The state machines are directly serving the AHB and memory controller interfaces. A more detailed behavioral analysis is given in the section describing the state machines.

Additionally the cache controller will have a set of registers. One pair of registers is used to indicate a range that needs to be invalidated. The second set of registers contain a hit and a miss counter for benchmarking purposes. For the processor, these registers are read only.

Tasks

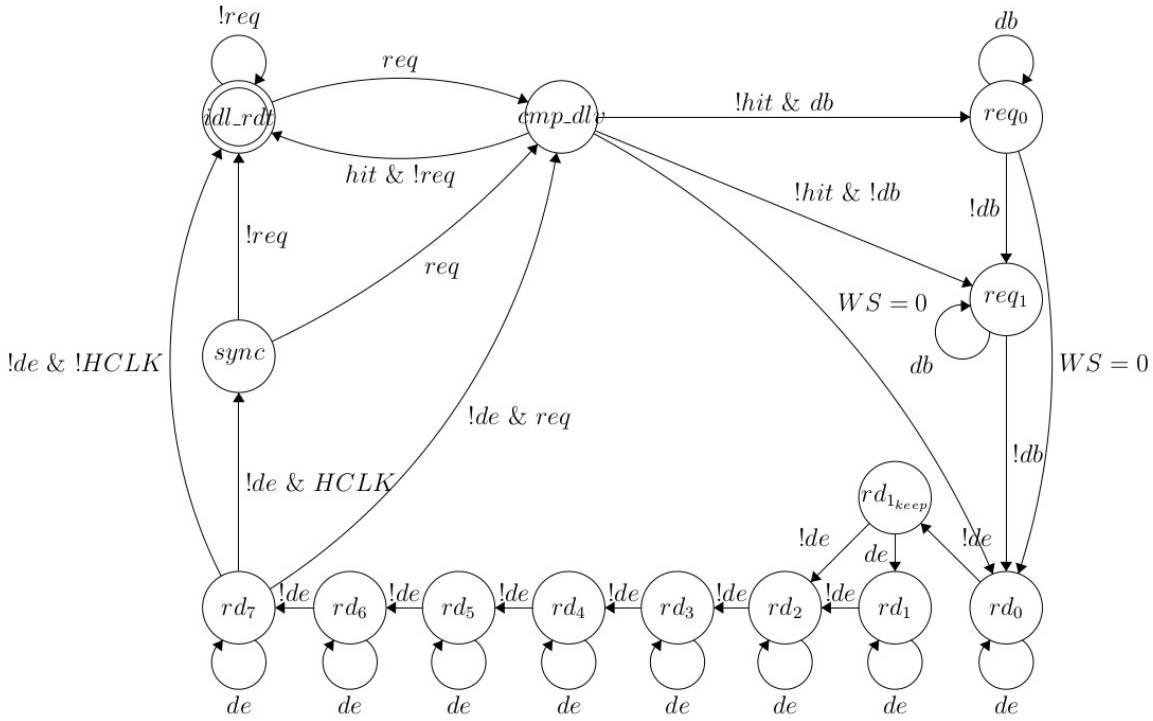
From the created design, these individual tasks can be derived:

- Implement two dual-port memories, one for the tag and one for the data
- Implement two state machines, one for reading and one for writing data
- Implement a registerfile for invalidation and hit/miss statistics
- Implement a testbench for design verification
- Integration into the existing project
- Testing and benchmarking of the Implementation

The following sections describe the results of these tasks in detail. The testbench is not described in detail because it offers no interesting features that would be worth elaborating. The dual-port memories are implemented according to the Xilinx User Guides and offer no novel features, either.

Read state machine

Designing the read state machine was one of the biggest challenges during the implementation, since a lot of corner cases, especially after a cache miss, need to be considered. Unlike consumer hardware, which can transfer a full cache line in one clock cycle using a wide bus, the memory controller in this design is restrained by the 32bit-wide data path. Thus, reading a full cache line requires at least the eight clock cycles for reading the data from the DRAM sequentially, granted that the data is already in the DRAM read FIFO, so we had to implement a more sophisticated logic consisting of multiple reading cycles to efficiently read a cache line from memory.



Drawing 5: state machine for read access to DRAM

Since the used memory controller does not support out-of-order bursts, two read requests are used to fetch all data: The first requests the remainder of the cache line, beginning with the word requested by the processor, the second reads the beginning of the line up to (excluding) the word requested by the processor. Splitting the read access allows to finish the AHB transaction as soon as the DRAM delivers the first word of data since the requested word is the first to appear in the DRAM read FIFO. This way, the cache-miss latency is minimized, allowing the processor to resume its work and freeing the bus within a minimal delay. Obviously, no second read request to the DRAM is needed or issued when the processor has requested the first word of a cache line.

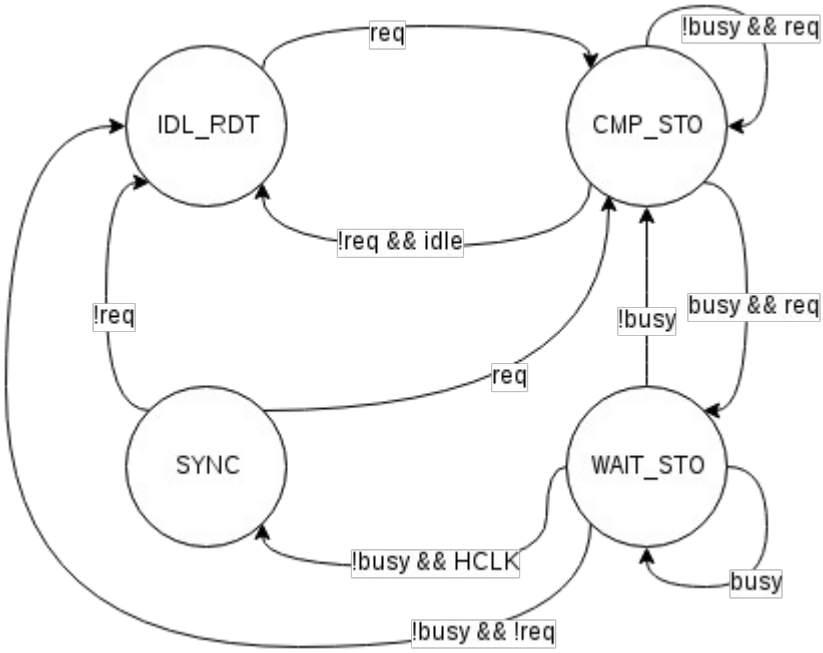
Because the DRAM can be accessed by multiple devices concurrently, we have no guarantee on how long it will take for the data to arrive in the read FIFO. This means that the state machine has to loop waiting for data to arrive at DRAM output FIFO to read each 32bit word.

So, if a read request appears we sample the address and begin to check our data and TAG memories if we can serve the requested data out of the cache. If we have a hit, we forward the data to the AHB interface and there it is written to the bus. This happens in a single bus transaction cycle. Then we can serve the next read request. If we cannot serve the read request from our caches, we send all our required read requests to the command FIFO and begin the reading sequence. In between there might be a number of wait cycles

because the DRAM is already busy. As soon as the data requested by the processor arrives, we store it for forwarding it to the bus, the others are directed towards the data cache. If we are done reading, the state machine synchronizes with the bus clock and waits for the next request.

Write state machine

The write state machine consists of four states, IDL_RDT, CMP_STO, WAIT_STO and SYNC. Usually the write state-machine simply forwards the data to the memory controller and updates the corresponding cache-line, if it is residing in the cache and is valid. In this



Drawing 6: state machine for write access to cache and DRAM

case, the state machine transitions form IDL_RDT to CMP_STO and back as soon as the request is done. If there are subsequent requests, the state-machine resides in the CMP_STO state as long as requests keep coming in.

If the memory controllers' write-queue is full, the state-machine transitions into the WAIT_STO state and stalls the bus until it could write the data to the memory controllers' write queue. Since the AHB-Lite interconnect has overlapping data and address phases, we need two stages of registers to save the address until we get the data from the bus.

The SYNC state common to the read and write state machines is used to re-set the state machines in-phase with HCLK when returning to the IDL_RDT or CMP_* states and is only reached when RD_7 state for the read- or WAIT_STO state for the write state machine is left out-of-phase in respect to the HCLK.

Registerfile

The four registers mentioned above will be placed in the existing registerfile, to use the existing address range. Each has a size of 32 bit.

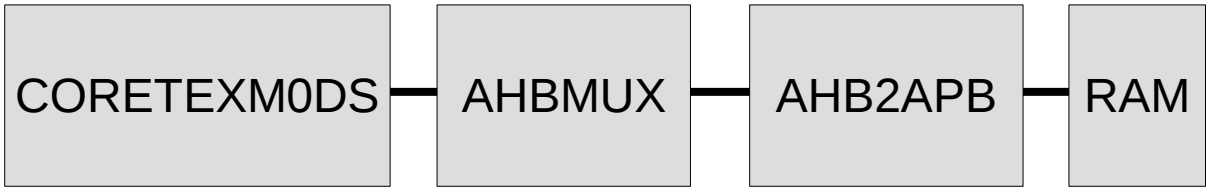
Since the ARM Cortex M0 has no coherence protocol implemented, we need to find another solution to make sure that memory regions used by multiple devices are not tainted by the cache. Therefore we introduce a range of registers that can be used to invalidate a range of cache lines. The first register contains the start address, the second one an end address. Every cache line that is in this range will be invalidated.

The invalidation registers are written by processor and then evaluated by the cache controller. It will stall until all cache lines have been invalidated. This is an expensive operation since the complete TAG memory has to be traversed and checked if the TAG is in the range specified by the processor. The controller then fills the registes with value `0x"FFFFFFFF"` to indicate the invalidation has been completed. New invalidation requests issued by the processor will be ignored. It is the processors responsibility to check if the invalidation is finished.

The hit and miss registers are read-only for the processors and contain the number of cache hits and misses. The counters for these values reside in the controller, and only the updated values are written into the register file.

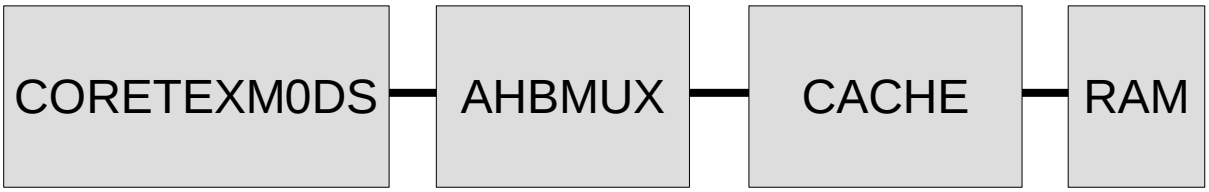
Integration

The RAM module was attached to the APB interconnect as set forth above and shown in Drawing 7. To overcome the performance issues of the APB Bus we decided to attach the cache directly to the AHB bus as a replacement for the AHB2APB bridge. Moreover, this has the side effect of a backward compatible integration of the cache to the supplied arm project.



Drawing 7: Memory access chain before cache integration

To perform the integration of the newly build cache we had to attach the cache in the memory access chain as shown in figure 8. Therefor, we made the following changes to the arm base project.



Drawing 8: Memory access chain after cache integration

First, we integrated our new AHB cache module in the armspecs.vhl as shown in code listing 1.

Now we are able to address the cache from software within the address range 0xcd000000 until 0xcdffffff. As some of you may noticed, that the MEM range is still in place, this is on purpose. It allows us to test the cache integration and measurement while the code is running on the AHB2MEM4 module on the APB Bus and vise versa. This gives us a system that is able to boot and run, even if the cache controller is not working properly.

```
-- order of ids and names must match! check when updating
type addrRangeName is array (0 to decoderRanges - 1) of string(5 downto 1);
constant rangeNames: addrRangeName := (
        "MEM  ",
        "UART ",
        "REGS ",
        "TIMER",
        "AUDIO",
        "FPU  ",
        "TRACE",
        "HIL  ",
        "ENET ",
        "APB  ",
        "DMAC ",
        "SPI  ",
        "SPI1 ",
        "PWM  ",
        "SDRAM" – “RFU1 ”
);
```

Code 1: armspecs.vhd

After assigning the address range to the AHBL2SDRAM module, we attached the module itself to the AHBMUX with the following modifications in the SignelArm.vhd as shown in code listing 2. The integration to the AHBMUX is highlighted in bold for a better understanding. Furthermore, we attached the AHBL2SDRAM module, which includes the cache itself to the bidirectional port P1 of the DRAM memory controller and left the old APB memory module untouched at the unidirectional ports p2 and p3. To do so, we routed

the P1 port from the memory controller from the singleArm_top.vhd down to the SingleArm.vhd.

The cache is running at DCLK - 100 MHz - out of the main PLL of the core from inside the

```
theSDRam: AHBL2SDRAM
    PORT MAP (
    HCLK => HCLK,
    HRESETn => HRESETn,
    HSEL => HSEL_OUT(getRange("SDRAM")),
    HADDR => HADDR,
    HWRITE => HWRITE,
    HSIZE => HSIZE,
    HTRANS => HTRANS,
    HREADY => HREADY,
    HWDATA => HWDATA,
    HREADYOUT => HREADY_OUT(getRange("SDRAM")),
    HRESP => HRESP,
    HRDATA => HRDATA_OUT(getRange("SDRAM")),
    p1_cmd_clk => p1_cmd_clk,
    p1_cmd_instr => p1_cmd_instr,
    p1_cmd_addr => p1_cmd_addr,
    p1_cmd_bl => p1_cmd_bl,
    p1_cmd_en => p1_cmd_en,
    p1_cmd_empty => p1_cmd_empty,
    p1_cmd_error => p1_cmd_error,
    p1_cmd_full => p1_cmd_full,
    p1_wr_clk => p1_wr_clk,
    p1_wr_data => p1_wr_data,
    p1_wr_mask => p1_wr_mask,
    p1_wr_en => p1_wr_en,
    p1_wr_count => p1_wr_count,
    p1_wr_empty => p1_wr_empty,
    p1_wr_error => p1_wr_error,
    p1_wr_full => p1_wr_full,
    p1_wr_underrun => p1_wr_underrun,
    p1_rd_clk => p1_rd_clk,
    p1_rd_en => p1_rd_en,
    p1_rd_data => p1_rd_data,
    p1_rd_full => p1_rd_full,
    p1_rd_empty => p1_rd_empty,
    p1_rd_count => p1_rd_count,
    p1_rd_overflow => p1_rd_overflow,
    p1_rd_error => p1_rd_error,
    DCLK => clk100,
    mem_calib_done => mem_calib_done,
    cache_hit => cache_hitReg,
    cache_miss => cache_missReg,
    cache_invalidate_start => cache_invalidate_startReg,
    cache_invalidate_end => cache_invalidate_endReg
      );
```

Code 2: SingleArm.vhd

SingleArm.vhd. This allows us to fulfill the requirement of the single CPU clock cycle access to the cache.

Additionally, we added 4 registers to the existing register module registerSet.vhd, two counting status registers and two address registers. The counting status registers count the cache hits und cache misses and the address registers are the invalidation range registers to invalidate a cache range. For simplicity the address registers are byte addressable and the cache line alignment of the start and end address is done in the cache module itself.

```
write(fline,string'("#define REGS_GPOUT "));
write(fline,4*regs_outReg);
writeline(mappingFile,fline);

write(fline,string'("#define REGS_CACHE_HIT "));
write(fline,4*regs_cache_hitReg);
writeline(mappingFile,fline);
write(fline,string'("#define REGS_CACHE_MISS "));
write(fline,4*regs_cache_missReg);
writeline(mappingFile,fline);
write(fline,string'("#define REGS_CACHE_INVALIDATE_START"));
write(fline,4*regs_cache_invalidateStartReg);
writeline(mappingFile,fline);
write(fline,string'("#define REGS_CACHE_INVALIDATE_END"));
write(fline,4*regs_cache_invalidateEndReg);

writeline(mappingFile,fline);
write(fline,string'("#define REGS_LED REGS_GPOUT"));
writeline(mappingFile,fline);
```

Code 3: SingleArm.vhd

After the integration of the AHBL2SDRAM in the vhdl project as mentioned above, we extended the hxxgen.vhd to generate a working hxx header for the software as shown in code listing 3.

Performance with Cache

By the time this report, no performance benchmarks could be conducted, because the cache controller was still in testing/development. See presentation for benchmarks with cache controller.

Concluding Remarks

The development of this cache controller showed that there are certain design specifics that are different for customized systems like the one we used, compared to commodity hardware. Especially the load from the DRAM is much more complicated since we have no cache line wide interface to the memory. Also there are no coherence protocols implemented in the system, which is why the coherence needs to performed manually by the processor.

Future work on this project might explore the possibilities of a set-associative cache or a more generic design that allows customization of cache size. This might be interesting if

larger memory areas need to cached. To reduce the risk of possible data thrashing between data and instructions, the cache module could be duplicated a separate instruction cache could be created. It would be even simpler, since it would need no write logic, and might gain performance from longer cache-lines. This can be implemented on this system since certain signals on the AHB-Lite interface allow distinguishing command and data read operations.

Tasks and Contributions

Task	**Workers**
Design	Moritz Nöltner-Augustin, Tim Schneider, Dennis Rieber
Dual Port RAM	Mortitz Nöltner-Augustin, Dennis Rieber
State Machines	Moritz Nöltner-Augustin
Integration	Tim Schneider, Dennis Rieber
Register File	Tim Schneider
Testbench	Dennis Rieber
Software Benchmarks	Dennis Rieber
Report and Presentation	Dennis Rieber, Tim Schneider

Overall Contribution

Moritz Nöltner-Augustin	33%
Tim Schneider	33%
Dennis Rieber	33%

Illustration Index

Illustration Index