DELFT UNIVERSITY OF TECHNOLOGY, COMPUTER ENGINEERING

ET4171 Processor Design Project
Final Report Group 8

Tijmen van Eijk (1314491)
Mark Schrauwen (4419170)
Michiel van der Vlag (4369327)

September 9, 2017

Contents

1 Introduction

This report describes performance optimizations on the baseline Plasma processor. The main goal is to create a faster processor, in absolute time of execution of the provided benchmarks, while keeping other parameters such as area and power within bounds. This can be achieved by reducing the number of cycles needed to execute the benchmarks and increasing the frequency, hence speeding up the processor. The area utilized and energy usage of the processor are taken into consideration and are reported. However they are not the main focus of the improvement strategies.

We perform a three-way strategy to achieve this goal which consists of:

- Profiling the benchmarks which are used to evaluate the performance of the processor.
- Using the timing report after synthesis to determine the critical path.
- Determine possible optimization techniques as covered in previous courses.

Our approach results in changes to the cache implementation, retiming of the critical path and the multiplier unit. Changes investigated concerning the cache are enlarging cache size, changing the set associativity and altering the write to main memory policy. Better arithmetics for multiplication are considered to reduce the number of cycles for this operation.

1.1 Obtained results

A radix-16 multiplier is implemented with a larger cache and mapping of main memory (8kB_4MB). This brought an increase of Area by about 34% while the absolute execution time was reduced by about 38%. Additionally the energy usage while executing the benchmarks has been reduced by 35%. A large improvement in terms of energy usage and absolute execution time has been achieved while trading off some additional area.

1.2 Report organisation

The remainder of the report is organized as follows. In section 2 the baseline performance is stated and discussed. The topics are: the performance expressed in cycles needed to execute the benchmarks, critical path obtained, resources needed and power consumption. Results from this evaluation are used to propose optimization avenues. Next to that the benchmarks are profiled based on the baseline Plasma emulator. This gives an indication which instructions are used most often, which in turn might suggest additional optimization avenues. In section 3 the optimization techniques implemented are explained on a detailed level. Section 4 reports the results obtained by the individual improvements and the results when the improvements are combined. A comparison between the baseline and improvements on the plasma core is made in section 4.3, which will lead to the general conclusion in section 5.

2 Improvement motivation

This section elaborates on the baseline Plasma processor what its relevant parameters are and how they might be improved. Profiling of the benchmarks can be done using the baseline Plasma emulator to evaluate the performance of the core. After this evaluation, optimization avenues are proposed and expected optimizations results are predicted.

2.1 Plasma baseline evaluation

Baseline implementation results such as critical path, resource utilization from ISE report and power consumption estimates are reported in this section.

2.1.1 Benchmarks profiling

A set of benchmarks is provided to test the performance of the Plasma processor. A more thorough investigation of the used processor instructions gives insight into the possible optimization directions. The first version of the Plasma processor was written in C-code by Steve Rhoads [1]. When compiled, this version of the processor is able to run the provided benchmarks.

This C-code is adjusted to count and output the number of times a specific instruction is executed. More than 100 different instructions are used in the processor. Similar instructions were added together to give a better overview of executed instruction. For example, the ADD, ADDIU, ADDU, SUB, SUBU instructions were combined to give an overview of the general number of additions. The results are shown in fig. 2.1.

To estimate the total duration the instructions in fig. 2.1 are multiplied with their corresponding estimated number of clock cycles. This gives an overview of the type of instruction with the longest duration fig. 2.2.

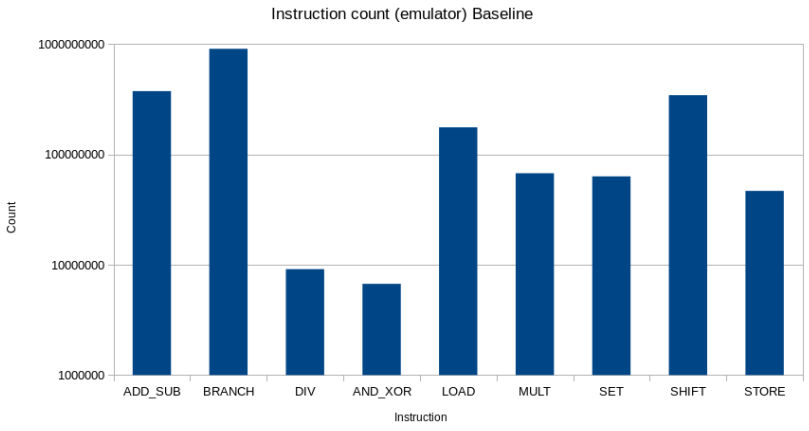

Figure 2.1: Overview of executed instructions based on all the benchmarks in the emulator. Please note that the instructions are grouped.

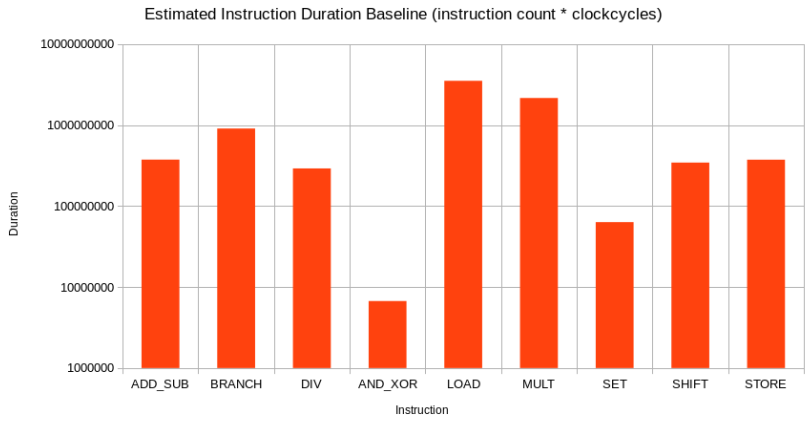

Figure 2.2: The estimated duration of an instruction based on the instruction count and an estimation of the total number of clockcycles per instruction based on fig. 2.1. The LOAD/STORE instruction duration were determined via Questasim simulation.

2.1.2 Critical path

The critical path of a design determines the maximum frequency at which the design can run reliable. The ISE tool can give insight in how the critical path is formed. Based on this information retiming (see section 2.5) insights can be gained to further improve the performance.

From the ISE timing report the top 8 maximum data paths are found of which the first one is shown in listing 1. The other 7 data paths are all directly related to the first one (other bits of the same signal vector). The timing tool lists all individual signals of data lines. We assume that the complete critical path is based on the entire 32 bit data path.

Listing 1: Critical Path

```
Source:          u1_plasma_top/u1_plasma/u1_cpu/u2_mem_ctrl...
                 .../opcode_reg_4 (FF)
Destination:     u1_plasma_top/u2_ddr/u2_ddr/data_write2_28 (FF)
Requirement:     12.631ns
Data Path Delay: 7.113ns (Levels of Logic = 8)
```

For the baseline design the critical path flows from the opcode register which resides in the *mem_ctrl* block to the data write channel for the *DDR* block. The opcode register contains the instruction received in the previous clock cycle. The corresponding instruction is either a store byte (SB), store halfword (SH) or a store word (SW). The block *mem_ctrl* transports the opcode to the *control* block for decoding, which sends the address for the data on the *rt_index* channel to the register containing block. The datum is read from the register and send back to *mem_ctrl*. *mem_ctrl* checks for the

endianness of the data and then forwards the datum to the *DDR* block. In the next clock cycle the datum will be written in memory.

2.1.3 Baseline performance results

Parameters found in table 2.1 are based on the ISE synthesis reports on timing (Frequency is the reciprocal of cycle period (Delay)), resources used (based on: $A_{RAMB16} = A_{CLB}$ and $A_{Slice} = \frac{1}{4}A_{CLB}$), power (based on the ISE XPower Analyzer with a 2 ms switching activity profile generated by QuestaSim) and the count of execution cycles of all benchmarks combined.

The derived metric D·B (absolute execution time in seconds of all benchmarks combined) is scaled to seconds. Multiplying D·B (s) and P (in W or J/s) gives the metric Energy in J. This is a good metric in a later stage to determine which optimization is best in terms of energy usage.

Table 2.1: Baseline performance results

(A)REA (CLB)	(P)OWER (W)	(F)requency (Mhz)	(D)ELAY (ns)	(B)ENCH. (M-cycles)	(E)nergy (J)
515	0.698	42.219	23.686	4323	71.47
A·P	A·D	A·B	P·D	P·B	D·B (s)
359	12186	2224233	16.53	3018	102

2.2 Cache size

To improve the benchmark score (reduce the total amount of cycles needed to execute all benchmarks), changing the cache size and cache mapping of the memory might help. Increasing the size means less replacements due to cache misses, replacements of cache words takes multiple cycles to fetch from memory, therefore this will most likely improve the benchmark score. Increasing the cache mapping of the memory might help with the latency (amount of clock-cycles needed) to replace a word in cache.

On the baseline Plasma platform the cached memory addresses are 19 bits wide (19 bit address space). Each address contains a 4 byte word (32 bits). The cache is only 4 kB, using a direct mapping policy, since only half of the cache can be reached with this addressing (a 10 bit index is implemented meaning $2^{10} = 1024$ 4-byte words). 2 MB ($2^{19} \cdot 2^2$ B) of the main memory is mapped. The cache data is confirmed to be valid by using the tag-index principle. The cached line is invalid when its tag is all 1's. The index (cache address) is 10 bits, the tag is the remaining 9 bits of the total address space.

It is expected that increasing the cache size will improve the benchmark score (reduce the total amount of clockcycles needed to execute the benchmarks).

2.3 Cache associativity

The baseline has a 1-way (direct mapped) associative cache. Multiple words in main memory map to the same place in the cache. When increasing the cache associativity to for instance 2-way, the word on a certain address can be placed in 2 spots in the cache. This will lead to less cache misses because of increased utilization of temporal locality.

Starting with a 2-way set associativity will give a clear view of possible the increase in performance. Going from 2- to 4-way might be done in a later stage when time permits.

2.4 Cache write policy

The critical path is invoked when the datum in the register needs to be stored in main memory. In the baseline a write-through policy is maintained. This means that when a datum is written in cache it is simultaneously updated in main memory. Since the system has to wait the writing to finish, fewer writes would mean faster processing.

Altering the memory write policy to for instance a write-back version reduces the number of writes to main memory. This policy states that main memory is not directly updated whenever cache is updated. Instead, main memory is only updated when data in cache is about to be overwritten while this data is not present in main memory yet. Implementing this strategy would also mean that the critical path will change. The critical path will change since the writing to main memory will start from cache instead of *mem_ctrl*.

A disadvantage would be that the cache miss time would increase. When a miss occurs not only the cache needs to be updated, but also the data from the missed place has to be written in main memory [2]. Thus this avenue could yield a decrease in execution time and an increase in processor operating frequency that could potentially outweigh the increase in cache miss time.

2.5 Retiming

Another way of speeding up the processor is to reduce the critical path by retiming the system. Retiming a component means moving registers from its outputs to its inputs or vice versa, while maintaining identical processor behavior [3]. As a result the critical path - which is the longest path between two registers - will be shortened.

As stated in section 2.1.2 the critical path starts in *mem_ctrl* and ends in the data being written into the DDR memory the next clock-cycle. The DDR memory is the end of the critical path. The DDR memory cannot be moved backwards along the critical path because this would mean moving the complete component, which is nonsense. The only option to retime the system is to move the register implemented in *mem_ctrl* forward.

2.6 MULTIPLIER

From fig. 2.2 it is gathered which instructions take the most time. Because the aim is to reduce the total duration of all the benchmarks fig. 2.2 gives insight in which operations need to be improved. The LOAD instruction has the greatest duration and is covered in sections 2.2 and 2.3. The instruction with the next longest duration is the multiply operation. This presents a good reason to try to improve the multiplier. The default multiplier takes 32 clock cycles to perform one multiplication. In the best case a multiplication can be done in 1 clock cycle (ignoring all obvious disadvantages of such a solution). In the next section the proposed improvements are explained in detail.

3 Optimization implementation

This section elaborates on how the proposed optimizations are implemented. The cache size will be increased, cache set-associativity altered, cache write policy investigated, retiming of the critical path created and a faster multiplier implemented.

3.1 Cache size

In order to increase the cache size and mapping the following alterations to the core are made. To create an 8 kB cache the index bits should be increased to 11 bits, leaving 8 bits for the tag. The tag memory should have 2048 places to have an unique tag for each cache address.

To go from 8 kB to 16 kB an additional 8 kB cache block has to be instantiated in the *cache_ram.vhd* file. Next to that the index should be increased to 12 bits, leaving 7 bits for the tag. The tag part now has to have 4096 places to have an unique tag for each cache address. Since 1 tag block (also implemented with a *RAMB16_S9* block) can store 2048 tags of maximum 9 bits, another tag block has to be instantiated to store the total of 4096 tags.

To increase the memory mapping from 2 MB to 4 MB the address range has to be increased from 19 to 20 bits $2^{20} \cdot 2^2$ B $= 4$ MB. See table 3.1 for the settings of the tag and index for each cache size and mapping.

Table 3.1: Cache address settings

Cache size	Memory	TAG range	Index range	Byte address
4 kB	2 MB	20:12 (9 b)	11:2 (10 b)	1:0
8 kB	2 MB	20:13 (8 b)	12:2 (11 b)	1:0
16 kB	2 MB	20:14 (7 b)	13:2 (12 b)	1:0
4 kB	4 MB	21:12 (10 b)	11:2 (10 b)	1:0
8 kB	4 MB	21:13 (9 b)	12:2 (11 b)	1:0
16 kB	4 MB	21:14 (8 b)	13:2 (12 b)	1:0

In table 3.2 performance results of the different cache sizes and mappings are stated because other optimization choices depend on them. Numbers in green indicate that for the specific implementation this parameter is best relative to the other implementations. Numbers in red indicate that for the specific implementation this parameter is worst relative to the other implementations.

As shown the 16 kB cache size and 2 MB mapping is fastest in absolute execution time and lowest in energy usage. This comes at a cost of highest area usage. When mapping 4 MB with the same cache size increases the energy usage relatively by a lot while the execution time is increased. Apparently adding some logic to map the 4 MB of main memory the ISE chose different placement of components which increased the critical path delay.

Table 3.2: Performance results for different cache size and mapping

CACHE (kB)	RAM (MB)	(A)REA (CLB)	(P)OWER (W)	(F)requency (Mhz)	(D)ELAY (ns)	(B)ENCH. (M-cycles)	(E)nergy (J)
4	2	515	0.698	42.219	23.686	4323	71.47
8	2	513	0.698	40.245	24.848	4246	73.63
16	2	532	0.698	42.860	23.332	4136	67.36
4	4	517	0.698	42.233	23.678	4323	71.45
8	4	511	0.698	40.473	24.708	4001	69.01
16	4	532	0.698	40.011	24.993	4136	72.15
		A·P	A·D	A·B	P·D	P·B	D·B (s)
4	2	359	12186	2224233	16.53	3018	102
8	2	358	12735	2175824	17.34	2963	105
16	2	371	12401	2198263	16.29	2887	97
4	4	361	12230	2232879	16.53	3018	102
8	4	371	13296	2200331	17.45	2887	103
16	4	357	12632	2045689	17.25	2793	99

From this table it can be concluded that enlarging cache size is effective in the sense that the amount of cycles needed goes down (B). Enlarging the mapping to main memory however proves not to have the desired effect. While the execution time for 4 MB mapped onto 8 kB still is faster then 2 MB onto 8 kB, 4 MB onto 16 kB proves to be slower than its 2 MB variant. The sweet spot here is thus in the region of 2 MB onto 16 kB or 4 MB onto 8 kB.

One reason for the 4 MB variant not to be indefinitely faster (as far as this table goes), probably has to do with the fact that the program in total does not need more then 2 MB. The total memory is not utilized and hence misses its desired effect. On the other end the index is growing larger and will take more time to decode. The reading and writing from a larger address space does not outweigh the fact that less cache misses occur.

3.2 Cache associativity

For the implementation of the 2-way cache associativity the 16 kB 2MB cache variant was chosen since in the baseline version it has the quickest execution time and lowest energy usage. As can be seen in table 3.1 in row 3 the tag range is 7 bits and the index range is 12 bits. Making it 2-way set associative means that the index can be reduced by 1 bit since the data can be stored in 2 sets (each 8 kB) in the cache and increasing the tag by 1 bit to keep the same cache mapping (2MB).

To implement a 2-way cache associativity all types of cache actions have to be considered:

1. A read action from CPU with a cache hit.
2. A read action from CPU with a cache miss following in an update action from main memory to cache.
3. A write action from CPU.

The cache has to be instantiated having 2 places for a tag (8-bit) and 2 places for the data (32-bit) at one index address (11-bit). It also needs 1 bit to use for the replacement policy (FIFO (First In First Out), LRU (Least-Recently Used) or random). LRU is chosen since it has a more straightforward implementation than random and is optimized for temporal locality.
T_1 stands for clockcycle 1, T_2 for clockcycle 2 etcetera.

Read-hit action
T_1: The CPU asks for a word in cache.
T_2: The cache compares the tag to the two tags stored at the index of the address it wants to read from. One of the tags stored is equal to the tag part of the read-address. The data stored at the index is propagated to the output.
T_3: The LRU bit of that index is updated.

Read-miss action
T_1: The CPU asks for a word in cache.
T_2: The cache compares the tag to the two tags stored at the index of the address it wants to read from. Both tags are not equal to the tag part of the read-address. A miss is stated. The LRU bit of the cache index is stored to make sure that the word that is fetched from main memory is stored at the correct place in the cache.
T_3: The LRU bit needs to be set to indicate that the "other" cached word on the same index needs to be replaced first and the tag has to be stored.
T_n: After n cycles the requested data was fetched from main memory and is written to the cache in the correct index and set.

Write action
T_1: The CPU requests a write to the cache, LRU-bit of the write-address index is requested.
T_2: The data is made ready to be written to set 0 or 1 of the address-index based on the LRU bit.
T_3: The data is written to the cache, the tag and LRU-bit are updated.

The design drawn in fig. 3.1 is implemented in the Plasma processor while keeping the original state machine intact. A lot of alterations had to be made to maintain the same functionality as in the baseline processor. For instance each clockcycle a read or write request can be done to the cache, at this moment the cache might still be busy with handling a request that was done in the previous clockcycle, for instance a write

operation. Therefore the memoryblocks of the tag, data and LRU are instantiated as 2 port blocks (RAMB16_S9_S9).

One port (port A) is used only for read operations, the other port (port B) is used only for write operations. This way a read and write operation can occur in the same clockcycle. However, some logic is added that determines for each type of memory (tag, data and LRU) if the write and read operation is allowed to occur at the same time, since reading and writing to the same address at the same time is not allowed. This is implemented by enabling/disabling the read port when the addresses of the read and write port are the same. Some logic is also added that determines which output (from the read or write port (port A or B)) of the memory must be used in the state machine and propagated out to be used in plasma.vhd.

A design overview of the 2-way set associative cache is shown in fig. 3.2. Only relevant signals are drawn.

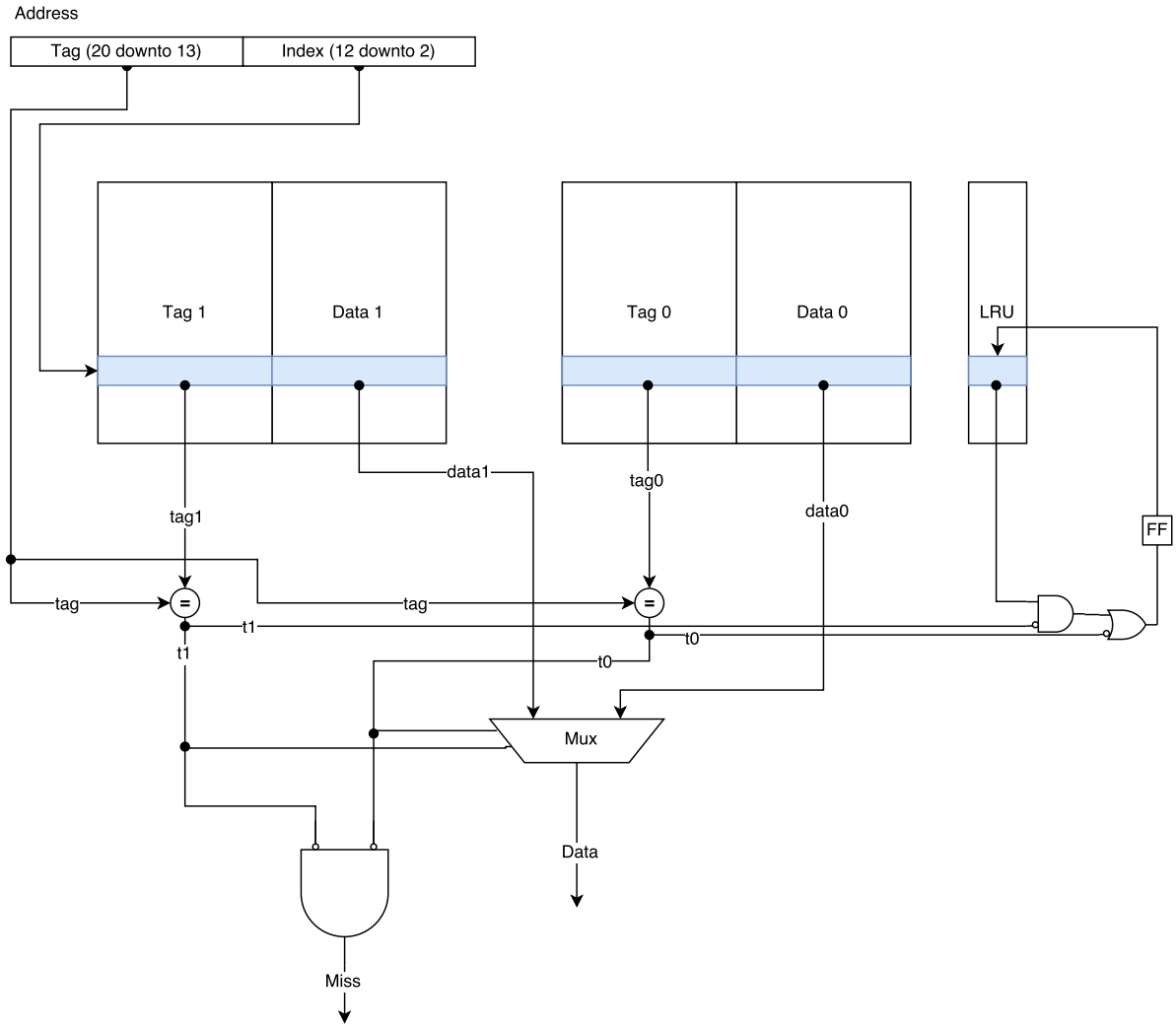

Figure 3.1: 2-way set associative cache concept

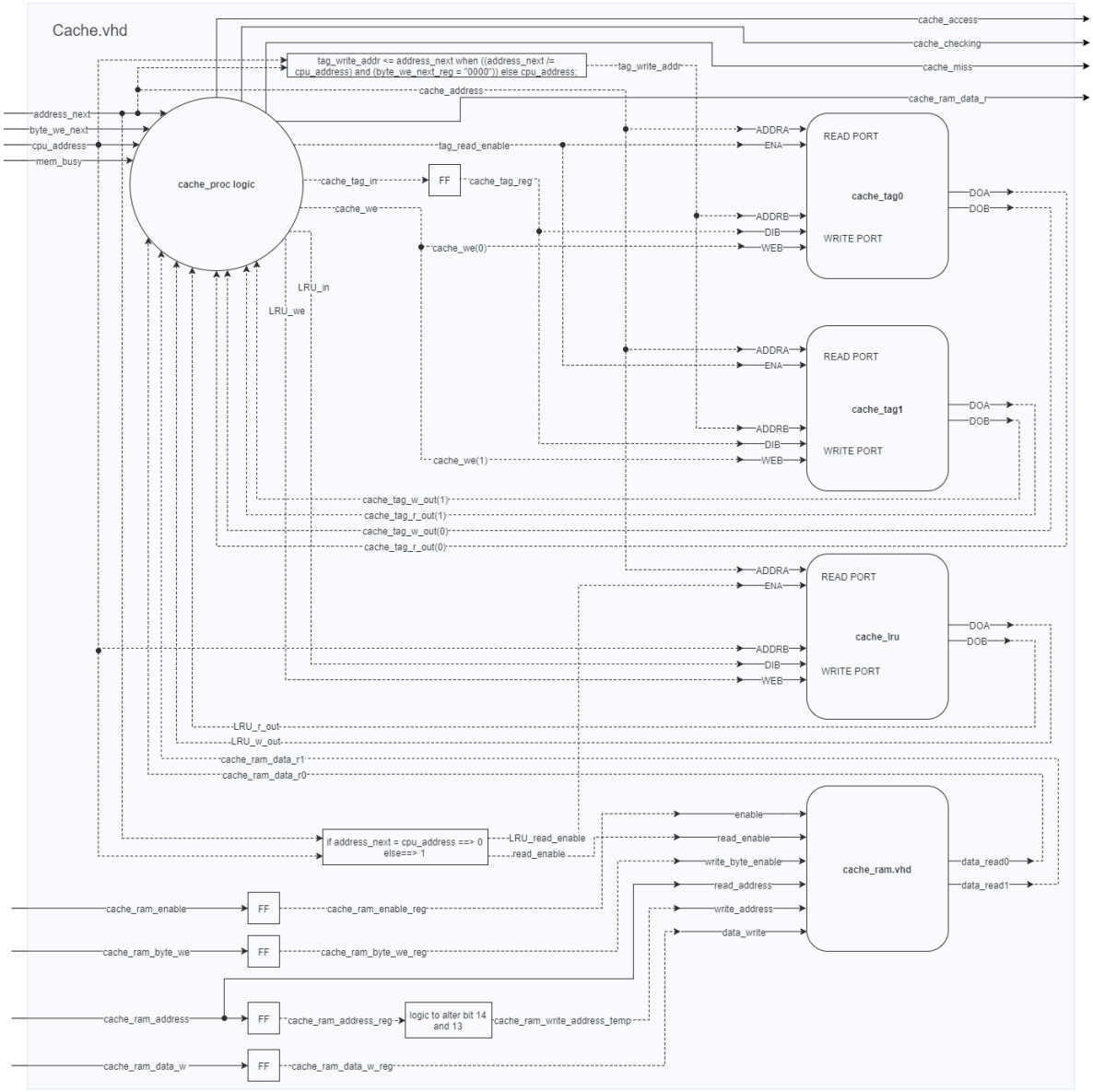

Figure 3.2: 2-way set associative cache design

3.3 Cache write Policy

The write-back policy could be implemented in the following way. If the datum needs to be written in cache, main memory would not be updated directly. A dirty bit will be set in the tag, indicating that this cache block differs from the content in main memory. If a cache miss occurs from a block with a dirty bit set, the cache will be triggered to write the cache content with the dirty bit set to main memory and update the cache content with the new datum. The cache request that causes a miss at a location with a dirty bit set, triggers a two way writing and reading from main memory and cache. The old data with the dirty bit set in the tag needs to be written to main memory. The

address in main memory is represented by the address of the old tag and the index (the index is the same as the new request). The new tag which differs from the old, hence causing the miss, should be stored in cache tag memory. The data of the index should be stored in cache itself. The dirty bit should of course then be reset to 0 for this block [2].

Since the mapping of the 2MB main memory to 16kB cache has proven to be one of the faster implementations for cache, this configuration will be the target to implement the write-back policy. Moreover, the tag is 7 bits wide, which leaves 2 bits (tag memory is 8 bits wide plus a parity bit for this configuration) free for setting the dirty bit. Eventually the changes made for the set associativity could also be implemented for this configuration without any memory hardware having to be added.

Due to complexity issues it was decided not to implement this avenue.

3.4 Retiming

The *mem_ctrl* component contains the register at which the critical path starts. The first component encountered when moving along the path is *control*. The register in *mem_ctrl* contains the opcode which is send to *control*. In figure fig. 3.3 the original situation for the signal opcode_out is illustrated. The triangle means this signal is buffered in a register. Only a rising edge of the clock makes this signal become available for *control*. The *opcode_out* output is connected not only to the *control* but also to the *pc_next* component.

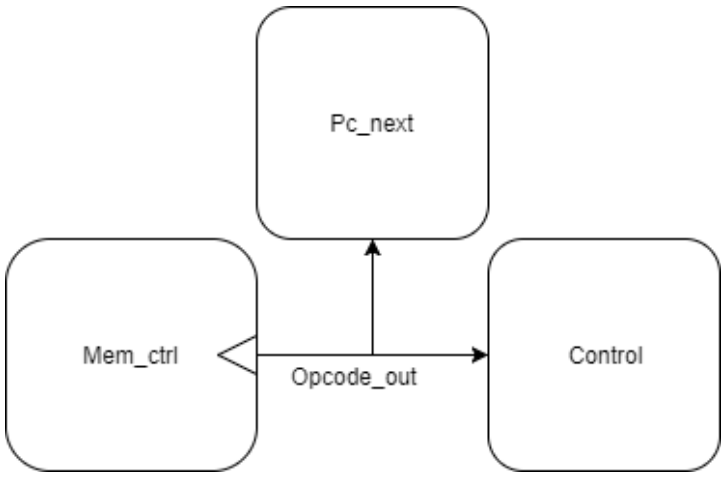

Figure 3.3: Original situation for the opcode_out signal with connected components

The processor is retimed by placing the register that holds the opcode signal from the *mem_ctrl* into the *control* component. The opcode signal connected to the *pc_next* component remains intact (registered). The *mem_ctrl* component is therefore equipped with two *opcode* outputs. One will remain the registered version which is connected to the *pc_next* component and the other is the version which is connected directly to

the *control* component. All other *mem_ctrl* registers will remain unchanged. This first retimed design is illustrated in figure fig. 3.4.

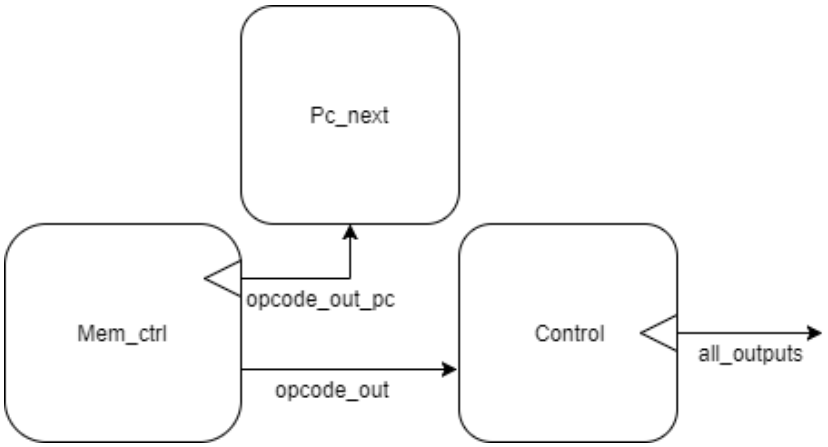

Figure 3.4: First step to retime: the opcode_out signal is split in two

The retimed situation will entail that the opcode will now pass through *mem_ctrl* and reach *control* a clockcycle earlier. As a consequence all output signals from *control* are put into registers to ensure that the overall timing of the system does not change.

An optional next step in the retiming process could be by placing the registers even further down the critical path. The next component down the critical path is the *reg_bank* component. By placing the registers of *control* into *reg_bank* decreases the critical path even further. This second design is illustrated in figure Figure 3.5.

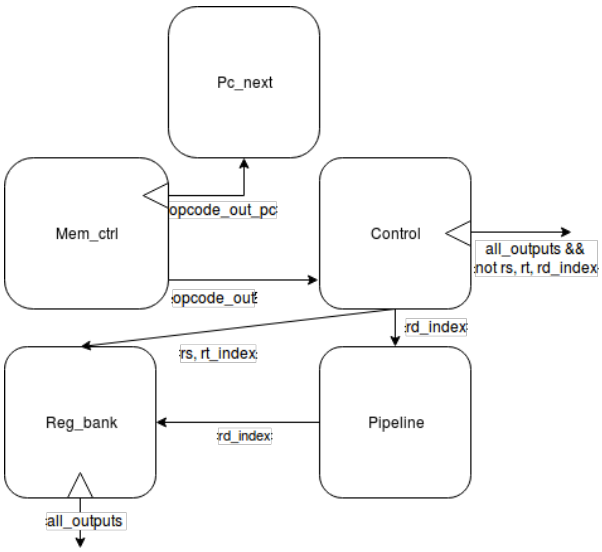

Figure 3.5: Second step to retime: all outputs from control are still buffered except rs, rt and rd_index

The outputs concerned with this operation are the *rs, rt, rd_index* signals. These signals must pass unregistered through *control* to *reg_bank*. If these signals were connected directly to *reg_bank*, this next retiming step would be simpler. However the *rd_index* signal passes first through the pipeline component, in which it is also delayed one clock cycle. Luckily, inside *pipeline*, besides the clock cycle delay there is no logic performed on this signal. Thus *pipeline* does not need to be altered and the *rd_index* can be treated as straight through connection. Since the *rs, rt_index* signals are connected both to *pipeline* and *regbank*, these signals are split into a pair that is connected directly to *regbank* and the original delayed signals which are connected to *pipeline*. As a consequence the outputs of *reg_bank* need to be put in registers in order to ensure that the timing of the system remains identical.

The first step in ensuring the functionality of the new system is equal to the old, is by looking at the opcodes benchmark. If all the opcodes can be executed by the retimed version the next step in the verification process can be taken.

This next step is to ensure that the retimed system has the same timing as the old setup. This is researched by comparing the timing diagrams of the baseline simulated in Questasim versus the retimed version. The outputs of the modified *control* should have the same values on the same moments in time as the old setup. The only difference in timing is that the *opcode_out* signal is reached by *control* a clock-cycle earlier then in the previous situation. The timing of all the other signals should not have been changed.

3.5 Multiplier

The performance of a multiplier can be improved in two ways: by reducing the number of cycles and by adding the operands faster. The baseline multiplier takes 32 cycles to compute a result. The most efficient multiplier could give a result in 1 cycle. This kind of improvement will lead to a significant FPGA area increase. The full CSA-multiplication tree will lead to a 1 cycle implementation. The partial CSA-multiplication tree is the best option as the area metrics are taken in to consideration (see: fig. 3.6).

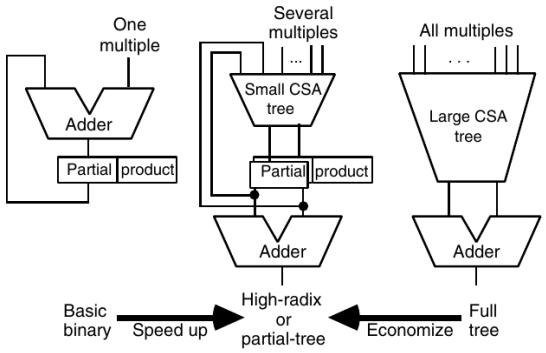

Figure 3.6: Overview of different CSA-multipliers. From a Radix-2 to a full CSA multiplier. Source: [4].

Based on the instruction count it can be estimated how multiplier speedup (of 1, 2, 4, 8, 16 or 32) would improve the average duration of the benchmark suite (see, table 3.3). Based on table 3.3 a speedup of 4 will give a relatively good reduction of the total execution time. This would mean a radix-16 multiplier (see: fig. 3.7). Higher speedups will give diminishing returns and are not further investigated.

Table 3.3: Estimated speedup of multiplier in relation to the average reduction of the benchmark score

Speedup	Reduction
1	0%
2	-13%
4	-20%
8	-23%
16	-25%
32	-26%

In [5] it is argued that combining the partial tree CSA multiplier with Booths-recoding multiplication gives best balanced version of a multiplier in soft-core FPGA implementations. It was decided to implement the radix-16 multiplier and later add Booths-recoding. The basic design is based on [4] figure 10.12.

A subgoal was to implement a radix-16 multiplier in a non-invasive way. The improved multiplier was implemented alongside the baseline multiplier. The multiplier state machine was adjusted in such a way that the values of the multiplier/multiplicand are only send in strict multiplication phases. Ignoring the *MULT_WRITE_LO* , *MULT_WRITE_HI* and the division phases. The vhd-files are structured in an hierarchical way see fig. 3.7.

The implementation of Booths-recoding did not happen as it can be seen as an extension of the current multiplier and with time being a limited resource it can always be added later on.

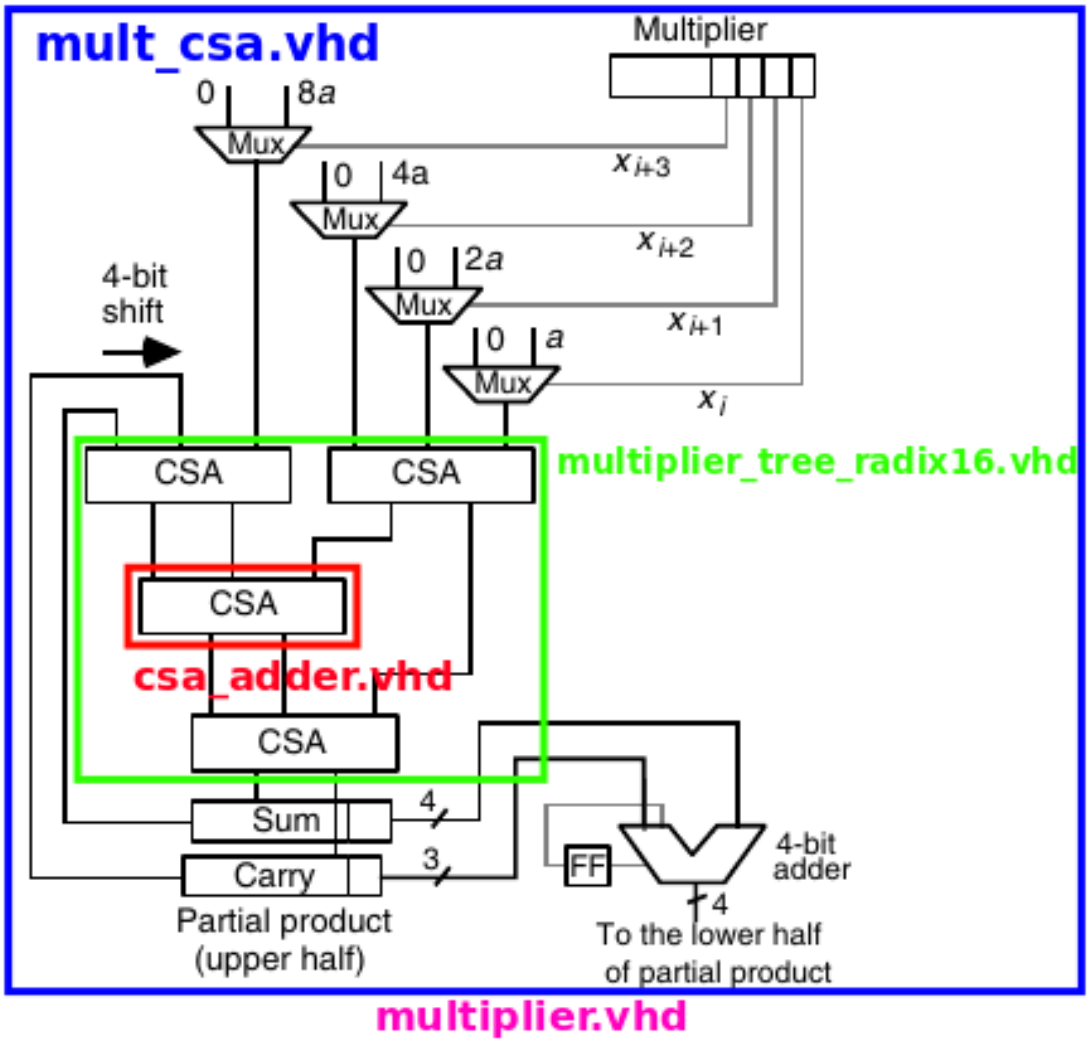

Figure 3.7: Optimized multiplier implementation with file hierarchy. A radix-16 CSA multiplier. Original image source: [4].

4 Performance results

Performance of each individual improvement will be stated first, then the performance of a combination of improvements will be stated, narrowing down the possibilities. Finally the performance of the most optimal combination of improvements will be stated and compared to the baseline processor.

4.1 Individual Improvements

This section reports on the performance results of each individual improvement.

4.1.1 Cache Size

Increasing the cache size has been done in the first phase of the design process. The results are printed in table 3.2. Since the aim of the project is to reduce the absolute time needed to execute the benchmarks, the 16 kB - 2 MB cache implementation is used in the design process to create the 2-way set associative cache.

4.1.2 Cache associativity

A 2-way set associative cache has been implemented in the Plasma processor. The processor has been simulated thoroughly. The testbench *opcodes* gives proper results in output.txt. The testbenches running the benchmarks *multiply* and *susan* have been simulated and all signals from *cache.vhd* and its higher level file *plasma.vhd* have been compared to the baseline processor. In either case no differences have been found that could influence the behavior of the processor. Sadly the synthesized design does not work on the actual FPGA.

4.1.3 Retiming

From the timing report it can be concluded that in theory the reduction of the critical path is a success. In listing 2 the new critical path is reported.

Listing 2: Critical Path after retiming, the 8 kB 2 MB cache version

```
Source:           u1_plasma_top/u1_plasma/u1_cpu/u3_control/...
                  rt_index_reg_2 (FF)
Destination:      u1_plasma_top/u2_ddr/u2_ddr/...
                  data_write2_26 (FF)
Requirement:      12.631ns
Data Path Delay:  6.448ns (Levels of Logic = 4)
```

The critical path is still the critical path as before but now it does not start in *mem_ctrl* but it starts in *control*. The data path delay is decreased by 0.665ns and 4 logic levels when compared to the critical path in the baseline processor.

The opcodes benchmark runs successfully in Questasim; meaning that all the opcode instructions can be executed by the new design.

The next step is to look at the timing diagram and compare it to the baseline version. For this the *multiply* and *ssearch* benchmarks are used. In figure 4.1 is illustrated that the timing for the outputs is identical to the old setup. The opcode however - seen in the top of the figure - lags forward to the old setup, as is expected since in the old setup the opcode_out was received a clock-cycle later. So the behavior of the retimed design is - as far as the eye can see - identical to the original design.

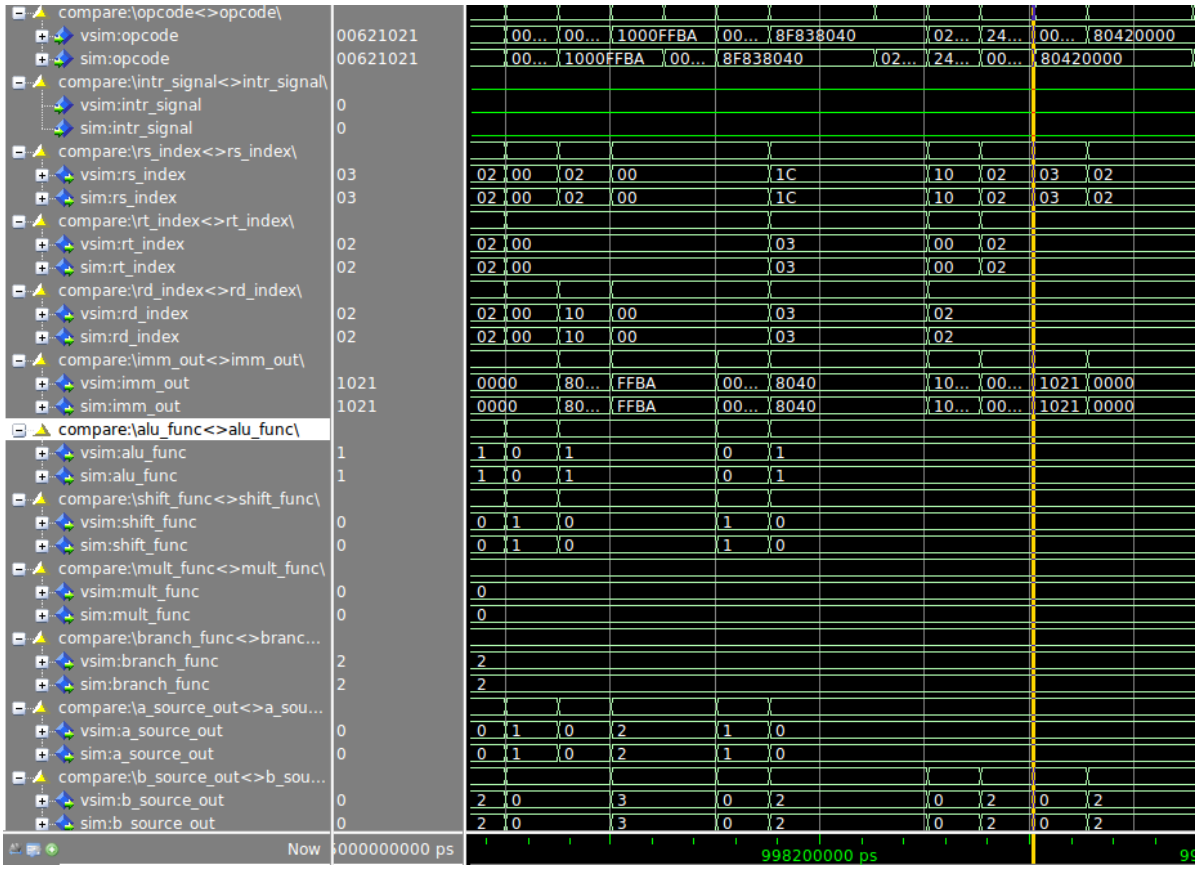

Figure 4.1: The output of the Questa simulation for the *control* component

When run on the FGPA the benchmarks are executed. In table 4.1 the performance results from the baseline, retiming the baseline and retiming with cache size/mapping 8kB_4MB and 16kB_2MB are printed. The baseline metrics are printed in the same table so they can be compared easily.

Table 4.1: Performance results of baseline and retimed design with various cache sizes/mapping

CACHE (kB)	RAM (MB)	(A)REA (CLB)	(P)OWER (W)	(F)requency (Mhz)	(D)ELAY (ns)	(B)ENCH. (M-cycles)	(E)nergy (J)
Baseline		515	1.257	42.219	23.686	4323	128.214
8	2	533	1.238	44.875	22.284	4323	119.3
8	4	533.5	1.263	44.201	22.624	4001	114.3
16	2	544	1.235	42.878	23.322	4136	119.1
		A·P	A·D	A·B	P·D	P·B	D·B (s)
Baseline		647	12186	2224233	29.77	5434	102
8	2	660	11883	2305240	27.59	5352	96
8	4	674	12070	2134534	28.57	5053	91
16	2	672	12687	2249984	28.80	5108	96

When compared to the results of the baseline version in table table 2.1 it is concluded that the retiming process for the 8kB_2MB design (baseline) was a success. The total number of clockcycles remains the same and there is speedup of $\frac{102-96}{102} = 5.9\%$. A small speedup was to be expected since the critical path is reduced by a small part.

The next step is to integrate this working version with the 8kB_4MB and 16kB_2MB version which is designated as the fastest cache implementation.

The critical path of this retimed cache version is further investigated as reported in listing 3. It is the same critical path as the previous cache version with only a slight increase in the Data Path Delay.

Listing 3: Critical Path after retiming, the 8 kB 4 MB cache version

```
Source:            u1_plasma_top/u1_plasma/u1_cpu/u3_control/...
                   rt_index_reg_0 (FF)
Destination:       u1_plasma_top/u2_ddr/u2_ddr/data_write2_1 (FF)
Requirement:       12.631ns
Data Path Delay:   6.225ns (Levels of Logic = 4)
```

The results of the retimed 8kB_4MB and 16kB_2MB cache designs are reported in table 4.1 from which can be concluded that when compared to non-retimed versions, the number of cycles remains the same and there is again a speedup of $\frac{103-91}{103} = 11.7\%$ for the 8kB_4MB retimed design.

The critical path for the retimed 16kB_2MB cache design is reported in listing 4. It is also the same critical path as before but with a different data path delay.

Listing 4: Critical Path after retiming, the 16 kB 2 MB cache version

```
Source:          u1_plasma_top/u1_plasma/u1_cpu/u3_control/...
                 rt_index_reg_0 (FF)
Destination:     u1_plasma_top/u2_ddr/u2_ddr/data_write2_1 (FF)
Requirement:     12.631ns
Data Path Delay: 6.494ns (Levels of Logic = 4)
```

The results of the retimed 16kB_2MB cache design are reported in table 4.1. There is a slight speed up of $\frac{97-96}{97} = 1\%$. See table 3.2 for the non-retimed performance results.

The candidate for combining designs clearly is the retimed version with 8KB_4MB. This implementation yields the best performance for retimed designs.

4.1.4 MULTIPLIER

Implementing the improved multiplier resulted in the parameters as shown in table 4.2. The implementation with 16 kB of cache and 2 MB mapping did not provide the best results regarding total execution time. The best implementation is the 8 kB with 4 MB mapping. This is probably due to the higher frequency of that specific design in combination with a synthesis sweet spot (e.g. parts of the design are more optimally placed on the FPGA).

Listing 5: Critical Path of Multiplier implementation

```
Source:            u1_plasma_top/u1_plasma/u1_cpu/...
                   ...u2_mem_ctrl/opcode_reg_2 (FF)
Destination:       u1_plasma_top/u2_ddr/u2_ddr/...
                   ...data_write2_29 (FF)
Requirement:       12.631ns
Data Path Delay:   7.004ns (Levels of Logic = 7)
```

Table 4.2: Performance results of optimized multiplier in relation to different cache/RAM implementations. Please notice other power values then previously mentioned in this document. The following power values are based on the Xpower Analyser from Xilinx ISE.

CACHE (kB)	RAM (MB)	(A)REA (CLB)	(P)OWER (W)	(F)requency (Mhz)	(D)ELAY (ns)	(B)ENCH. (M-cycles)	(E)nergy (J)
Baseline		515	1.257	42.219	23.686	4323	128.2
4	2	691	1.297	44.01	22.722	3188	94.0
8	2	685	1.297	43.25	23.123	311-	93.4
16	2	698	1.281	42.22	23.686	2995	90.9
4	4	699	1.306	41.07	24.347	3188	101.4
8	4	691	1.294	43.52	22.978	2792	83.0
16	4	699	1.295	41.38	24.166	2995	93.7
		A·P	A·D	A·B	P·D	P·B	D·B (s)
Baseline		647	12186	2224233	29.77	5434	102
4	2	896	15701	2202908	29.47	4135	72
8	2	889	15845	2130853	29.99	4033	72
16	2	894	16539	2091259	30.34	3837	71
4	4	912	17006	2227028	31.80	4164	78
8	4	894	15878	1929272	29.73	3613	64
16	4	905	16886	2092756	31.29	3879	72

4.2 COMBINED IMPROVEMENTS

Combining the retimed design with the improved multiplier, yields the following critical path listed in 6 and the results in table 4.3. The critical path is the same as the previous paths except for the Data Path Delay.

Listing 6: Critical Path of Retimed Multiplier implementation

```
Source:          u1_plasma_top/u1_plasma/u1_cpu/...
                 ...u3_control/rt_index_reg_0 (FF)
Destination:     u1_plasma_top/u2_ddr/u2_ddr/...
                 ...data_write2_17 (FF)
Requirement:     12.631ns
Data Path Delay: 7.221ns (Levels of Logic = 4)
```

Table 4.3: Performance results of a retimed multiplier with 8KB 4MB cache

CACHE (kB)	RAM (MB)	(A)REA (CLB)	(P)OWER (W)	(F)requency (Mhz)	(D)ELAY (ns)	(B)ENCH. (M-cycles)	(E)nergy (J)
8	4	698	1.304	41.339	24.190	2792	88.672
		A·P	A·D	A·B	P·D	P·B	D·B (s)
8	4	910	16885	1948816	31.54	3641	68

From the results it is gathered that the best multiplier implementation without retiming has a critical path that runs through 7 logic levels. After implementing the retimed design, the number of logic levels was reduced, however there was no significant speedup, because the data path delay is increased. The design which integrates a retimed multiplier with 8kB_4MB cache runs on 41.339 Mhz. The unretimed multiplier with 8kB_4MB cache runs on 43.52 Mhz. Regarding the results of the retimed designs from the previous chapter, in which a speed up of 11% was obtained, this is an unexpected result. It was expected to have at least some increase in operating frequency. Unfortunately, this is not the case.

This can be explained due to the fact that the multiplier adds a lot of extra hardware. Because of this the place and route of ISE is not able to place the components closer together such that the critical path is reduced. Therefore the 8kB_4MB cache with radix-16 multiplier without retiming leads to the highest improvement of absolute execution time.

4.3 COMPARISON

Table 4.4: Comparison between baseline and the improved processor

Design	(A)REA (CLB)	(P)OWER (W)	(F)requency (Mhz)	(D)ELAY (ns)	(B)ENCH. (M-cycles)	(E)nergy (J)
Baseline	515	1.257	42.219	23.686	4323	128.214
Improved	691	1.294	43.52	22.978	2792	83.0
	A·P	A·D	A·B	P·D	P·B	D·B (s)
Baseline	647	12186	2224233	29.77	5434	102
Improved	894	15878	1929272	29.73	3613	64

Obviously the radix-16 multiplier has the largest effect on the increase of area. This comes at a significant reduction in needed cycles to execute all benchmarks. The cache size increase helps to reduce this metric even further. Because the cycles are reduced, the frequency is higher and power has not increased significantly in comparison to the baseline a reduction in energy needed is achieved. The metrics Band F also account for the large reduction absolute execution time.

5 Conclusion

In order to make an improved design the baseline design is evaluated. The goal of this project was to improve the absolute execution time of the processor. The simulator was used to count the amount of instructions executed in the benchmark suite that will evaluate the improved processor design. This gives a preliminary insight into which instructions are invoked often and narrows down the possible improvement avenues. The baseline processor parameters such as critical path, power consumption and area provide a firm base on which the improvements can be compared against.

Several improvement avenues are investigated such as cache size and mapping, which shows a sweet spot for execution time between 8kB_4MB and 16kB_2MB. A 2-way set associative cache is designed which promised lower miss rate, however at this moment it did not return any performance results since the synthesized design could not be properly loaded into the FPGA.

In order to reduce the critical path two improvements are investigated and one successfully implemented. One is altering the write to memory policy from write-through to write-back. Removing the current critical path. The second is retiming the critical path and thus reducing the total propagation delay. The last improvement is successfully implemented with a reduction in critical path. However when combined with the improved multiplier the propagation delay is higher than without retiming. This can be explained because the multiplier needs a lot more area which might increase the datapaths of multiple components of the processor on the FPGA. The critical path is still the path that runs from *control* to *ddr*.

Next to the store/load instructions, multiplication instructions are invoked often and take up to 32 cycles to execute. A radix-16 multiplier is successfully implemented, reducing the execution time drastically.

Table 5.1: Improvement percentages, 100% is the baseline performance for each metric

	(A)REA	(P)OWER	(F)requency	(D)ELAY	(B)ENCH.	(E)nergy
Difference	134.3%	102.9%	103.1%	97.0%	64.6%	64.5%
	A·P	A·D	A·B	P·D	P·B	D·B
Difference	138.3%	130.3%	86.7%	99.9%	66.5%	62.7%

Table 5.1 shows that though area has increased by about 34%, however the compound metric A·B has been reduced by approximately 13% due to the fact that the total amount of clock cycles needed to execute all benchmarks have been reduced by a lot. This is also the reason why the energy usage has been lowered to 64.5% of the original usage while while the power increased by about 3$.

REFERENCES

[1] *Plasma - most MIPS I(TM) opcodes*, https://opencores.org/project,plasma

[2] David A. Patterson, John L. Hennessy. (2008) *Computer Organization and Design, Fifth Edition*. Morgan Kaufmann.

[3] De Micheli. (2008), *Synthesis and optimization of digital circuits*. USA: McGraw-Hill.

[4] Parhami, B. (2000). *Computer Arithmetic, algorithms and hardware designs*. New York: Oxford University Press.

[5] Kumm, M. (2015), et all. *An Efficient Softcore Multiplier Architecture for Xilinx FPGAs*. Conference paper, january 2015.