Massachusetts Institute of Technology

Department of Electrical Engineering and Computer Science

6.375 Complex Digital Systems Design

A Parametrizable Processor

Authors: Olivier Bichler, Roberto Carli, Alessandro Yamhure

Date: May 16, 2007

System-on-a-chip solutions often require simple processors designed to perform a narrow variety of tasks meeting certain performance specifications. It is important to use processors that perform the required tasks with minimal waste of chip area and power consumption, however there are engineering expenses to design an optimal processor for each SOC solution. Starting from a 3-stage pipelined SMIPS processor in Bluespec, we designed and implemented a parametrizable processor, which can be configured at compile-time to be synthesized as a one, two or three-stage processor. According to the parameter choice, the processor is synthesized only with the hardware required to run the specified pipeline configuration. Our results indicate that low-stage solutions have lower IPS performance and lower area, and the performance / area tradeoff is balanced throughout all possible processor configurations.

Contents

1 Project Description 3
1.1 Motivation 3
1.2 Design Steps 3
2 Design and Implementation 5
2.1 One-rule synchronous processor 5
2.1.1 Explicit guards.............................. 5
2.1.1.1 FIFOF Methods 5
2.1.1.1 *dataReq/RespQ_guard*... 6
2.1.1.1 wbQ_guard 6
2.1.2 Customized SFIFO 6
2.1.3 Discard and Stall 7
2.1.4 PC Register.................................. 8
2.2 Packaging stages into functions 8
2.2.1 Motivation 8
2.2.2 Implementation............................ 9
2.3 Eliminating Pipeline Stages...................... 10
2.3.1 Merging pcGen and execute......... 10
2.3.2 Merging execute and writeback ... 11
2.3.3-stage version 13
2.4 Parameter Implementation 14
3 Test Strategy 16
4 Results 18
4.1 Area 21
4.2 Instructions per Cycle 22
4.3 Performance... 23
4.4 Summary 25
5 Conclusions 27
6 Acknowledgments... 28

Figures

Figure 1: One-rule, packaged processor 10
Figure 2: 2-stage, pcG / execute merged........... 11
Figure 3: 2-stage, execute / writeback merged . 13
Figure 4: 1-stage configuration.......................... 14
Figure 5: High-level schematics 15
Figure 6: Area results... 21
Figure 7: Instructions per Cycle (IPC) results 22
Figure 8: Effective clock period results.............. 23
Figure 9: Instructions per Second (IPS) results .. 24
Figure 10: Figure of Merit (FOM) results 25

Tables

Table 1: Test variables and predicted results 17

1. Project Description

As our final project, we designed and implemented a parametrizable SMIPS ISA processor in Bluespec HDL. The parameter controlled variable in our processor is the number of pipelined stages, thus the degree of parallelism and concurrency achieved by the processor. The number of pipeline stages ranges from 1 to 3, with two versions of the 2-stage configuration.

1.1 Motivation

The motivation behind making a parametrizable processor comes from the realization that SOCs are designed to solve very specific problems within very narrow system requirements concerning area, power and performance. By having access to a parametrizable processor, an SOC designer is equipped with a menu of different processor configurations from which he can choose a version of the processor with which to tackle the task at hand most efficiently. Furthermore, having a wide choice of processor configurations allows SOC design to cut down significantly on engineering costs.

1.2 Design Steps

Implementing a parametrizable processor involved five major steps. Our starting point was a 3-stage, fully parallel SMIPS ISA processor design in Bluespec where each “pipeline stage” is a guarded atomic action, known as a *rule* in Bluespec. Our final result was a SMIPS ISA processor whose number of pipeline stages and level of stage concurrency was controlled by a 2-bit parameter, allowing for 4 different configurations of the pipeline stages. The five major steps were:

- **Identifying and merging all the *rules*** in a baseline Bluespec processor description that contained pipeline stages or actions. This was done in order to achieve very tight control of the scheduling of pipeline stages and to allow for combinational paths between pipeline stages (useful when we are dealing with one and two stage

processors). The outcome of this work was a fully functional one-rule synchronous processor, where all the processing was achieved within one guarded atomic action.

- **Identifying and packaging the purely combinational pipeline stages** of the SMIPS processor Bluespec description into separate and independent Bluespec *functions.* The functions were constant and were “blind” to the value of the parameter. This was done to allow us to *call* the different functions as needed by the desired configuration and to specify the way the functions interact with each other depending on the level of parallelism chosen.
- **Introducing a single 2-bit parameter** that controls the interaction and arrangement of the functions written above to achieve a certain degree of parallelism in the processor. The outcome of this step is a processor whose number of pipeline stages can be specified by setting the parameter value.
- **Optimizing each processor configuration** with respect to performance and area. The objective was to have each configuration be as efficient as possible. Moreover, our goal was to have each configuration contain the minimum amount of hardware needed to achieve the desired level of parallelism. This was done by abusing the aggressive optimization approach of the Bluespec compiler in addition to the use of compiler macros that specified optimizations.
- **Comparing the different processor configurations.** In order for our parametrizable processor to be useful, each configuration must have its own benefits and advantages over the other configurations.

2. Design & Implementation

2.1 Making a one-rule synchronous processor

2.1.1 Explicit guards

Including the whole processor in one Bluespec rule required us to lose all the advantages of guarded atomic actions and implicit guards. In a multi-rule processor, each rule can attempt to call methods such as FIFO enqueue and dequeue without checking FIFO states first. If the methods are not available, the rule stalls and retries next cycle. In a one-rule processor, however, a failed attempt would cause the entire processor to stall indefinitely; therefore the processor must verify the state of every element before attempting a method call. In other words, in order to maintain processor functionality and correctness, we needed to explicitly define the *WILL_FIRE* signal of each pipeline stage which we expressed with *IF* conditions that "guarded" each pipeline stage. This way, we achieved complete control over when each stage fires and could thus specify the precise schedule of the processor stages.

The new explicit conditions guarding the various processor stages were expanded versions of the guards used in the original multi-rule Bluespec description. All guards conditioned on the type and nature of the instruction being processed were implemented using purely combinational logic that takes the instruction as its input (returned by the instruction memory) and returns a Boolean result. These Boolean results, logically combined with the state of the relevant FIFOs, determine the *WILL_FIRE* signal of the individual pipeline stages.

2.1.1.1 FIFOF Methods

Explicit guards require that the processor checks the state of the connecting FIFOs. In the absence of implicit guards, we had to explicitly ensure that the FIFOs were capable of handling a pipeline stage's requests before executing the stage itself. This required all FIFOs in the processor to be changed into FIFOFs, which are identical to FIFOs but have the additional

capacity of being asked whether they are empty or full, which is crucial for ensuring the feasibility of *enq* and *deq* of the FIFO. Additionally, new interfaces were written in Bluespec to allow for the interaction of the FIFOFs with the instruction and data memory modules.

2.1.1.2 *dataReqQ_guard/dataRespQ_guard*

Guards also require that the processor checks the nature of the instruction being processed. Certain instructions (all but *LOADS* and *STORES)* do not require any interaction with data memory. Thus there is no need to wait for the data memory queues to accept/return a request/response to or from data memory. These conditions were made explicit by writing two Boolean values, *dataReqQ_guard* and *dataRespQ_guard*, and inserting them in the explicit guards of execute and writeback stages.

2.1.1.3 *wbQ_guard*

Certain instructions (certain *branches* and *jumps*) do not write to the register file. Thus, for both correctness and performance reasons, we made it explicit that there is no need to wait on the readiness of the *writeback* stage and its associated FIFOs when dealing with such instructions. This logic is implemented with a *wbQ_guard* Boolean, also integrated appropriately in the explicit guards.

2.1.2 Customized SFIFO

The transition to a single-rule processor also required us to customize the searchable FIFO (referred to as *SFIFO*) that connects the *execute* and *writeback* stages of the processor (referred to as *wbQ)*. It is searchable in order to protect against data hazards and had to remain searchable for the same reason in the single-rule merged version of the Bluespec description.

The customization of our *wbQ* involved introducing two additional methods and reordering the method schedule. The new SFIFO's schedule, including new methods, is:

first, find, find2, notFull, notEmpty, deq < enq < clear

We added the FIFOF methods *notFull* and *notEmpty*, with proper interface, so that the explicit guard can verify the state of the SFIFO before firing stages.

To implement the new schedule, we used EHR registers to hold both the data and the valid bits. Having execute and writeback in the same rule implies that one rule will *enq* while the other will *deq* in the same cycle. The valid-bit structure of the SFIFO implies that both enq and deq will read and write registers in the SFIFO. Using EHR registers allows us to schedule the reads and writes as needed to ensure correctness and avoid conflicts and structural hazards.

Pulse wires were avoided in order to allow enqueue and dequeue to occur in the same cycle. With full parallelism, when all stages fire concurrently, wbQ needs to *deq* and *enq* simultaneously. Thus we eliminated the pulse wire found in the base-line SFIFO.

Another issue is that of bypassing. Our register file is write-before-read, EHR-based and bypassable in order to avoid RAW hazards during full stage parallelism - once dequeued by writeback, an instruction writes the new value before the execute stage fires. Our initial SFIFO was bypassable too, which caused a combinational loop between execute and writeback. We solved this by scheduling the notFull and notEmpty as read_0, so that if deq and enq are requested in the same cycle on an empty SFIFO, the dequeue will see wbQ as empty and not fire.

Finally, we scheduled the SFIFO methods in according to the schedule of the processor stages, expressed by the explicit guards. To ensure correct stage scheduling, we allow the *find, notFull* and *notEmpty* methods used by the explicit guards to happen before the SFIFO methods that are required when the stages fire. Therefore, *find, notFull* and *notEmpty* make use of read_0. Furthermore, in order to avoid scheduling conflicts between execute and writeback, we altered the scheduling properties of the SFIFO so that deq < enq which translates into writeback < execute. This schedule is consistent with the register file and prevents a conflict.

2.1.3 Discard and stall

Discard, implemented as part of the one-rule processor, is used to eliminate stale instructions between the pc generation and execution. Its guard is mutually exclusive with the execute

condition, so that it fires only when the pcQ is locked with old data. Correctness is guaranteed by the 1-bit value *hasToken* and the *wait4token* state of execute. When the execute stage takes a branch, it sets a *wait4token* state, and it fires only when the new good instruction, carrying *hasToken*, reaches the first position in pcQ.

The stall logic is implemented as a function, concerned with avoiding RAW hazards in the execute-writeback pipeline. It works by searching the wbQ. It only necessary when the execute and writeback stages are pipelined.

2.1.4 PC register

The pc register was modified to an EHR register in order to ensure correctness during a branch instruction. Since pcGen and execute are in the same rule, and pcGen always updates the pc value, during a branch a pc write by execute would cause a structural hazard. In our implementation, execute performs a write_1 (overwrite) on the PC register while to pcGen performs a write_0 in the case of a branch/jump instruction.

2.2 Packaging stages into functions

After obtaining a working one-rule version of the processor, we proceeded to packaging the main combinational functioning of the three stages into parameter-independent functions.

2.2.1 Motivation

There are several reasons why we decided to use the various stages as functions.

- Unchanged combinational workhorse: While different degrees of pipelining influence the processor's internal datapath and scheduling, the combinational working of instruction retrieval, ALU operations and writeback remain unchanged. Therefore it makes sense to isolate this combinational work into abstracted blocks.
- Function-call mechanisms: In Bluespec, a function is converted into appropriate hardware at compile-time, in instances when it is called. Therefore, specifying combinational stages as functions gives an additional degree of flexibility to the system,

wherein, for example, working ALU hardware can be instantiated in more than one place, and the corresponding hardware only synthesized if and where the function is called. The designer can easily build combinational paths between hardware by using recursive calls, without worrying about timing details in communication. These choices provide higher flexibility for future system expansions.

- Higher-level abstraction: Isolating combinational stages also implies theoretical advantages. It sets a higher level of abstraction into the system, wherein data routing can be separated from combinational workings. The internal logic of the functions is completely parameter-independent, which guarantees the description of an elegant parametrization, where the only difference between pipeline and non-pipeline resides in the routing.

2.2.2 Implementation

Each state was separated into a *shell* and a function. Shells correspond to the parameter-dependent routing logic, which also interacts with state elements, and calls stage-functions with the appropriate data.

While theoretically they should be fully combinational, we implemented stage functions as ActionValue functions, which partially interact with the processor's state. We chose this solution since some interactions, for example those with instruction memory, are completely parameter-independent. Integrating these actions into stage functions provided for a much cleaner shell code, and highly simplified the complexity of the functions' input/output types. If new parameters were to be added which conflict with function actions, it would be easy to modify state functions to be purely combinational for higher expandibility.

Figure 1 shows the state of our processor after one-rule merging and stage packaging.

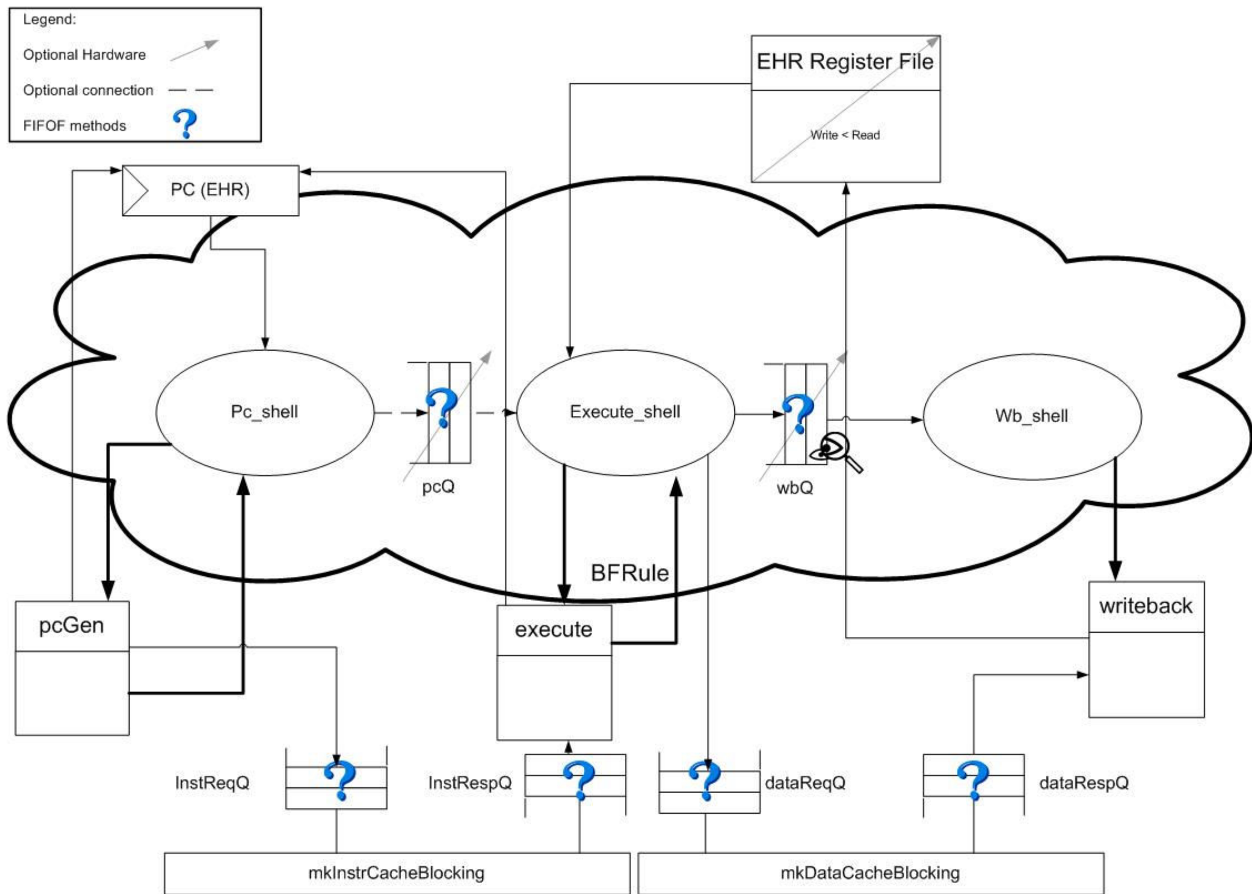


Figure 1: One-rule, 3-stage processor with packaged functions. The comprehensive rule fires every clock cycle, and the firing of individual stages is regulated by explicit guards, which invoke FIFOF methods (question marks) to check the state/availability of FIFO elements. The combinational working of each stage, together with part of the memory interfacing, is performed by the function-blocks. The *shell* parts take care of all parameter-dependent routing and of invoking stage functions.

2.3 Eliminating Pipeline stages

2.3.1 Merging pcGen and execute

The pcGen and execute stages can be merges by eliminating the FIFOF *pcQ* and inserting a synchronizing register *allow_pcgen.* Since merged stages always work on the same instruction, execute can read the pc value directly from the pc register for branch instructions. The hardware for *pcQ*, as well as all discarding mechanisms, is eliminated, and the *instrReq/RespQ* FIFOFs are reduced to size 1.

The 1-bit Boolean register *allow_pcgen*, initially set to true, ensures synchronization between the two stages by enforcing mutual exclusion. The pcGen explicit guard requires *allow_pcgen* to be true, and sets it to false after firing. Execute requires it to be false, and sets it to true after firing. The *allow_pcgen* mechanism does not allow concurrent firing of the two stages, however, the instruction memory cycle latency never allows a combinational path between the stages. In the following section, however, *allow_execute* is compatible with single-cycle stage firing. Figure 2 shows the resulting 2-stage implementation, with the pcGen and execute stages merged.

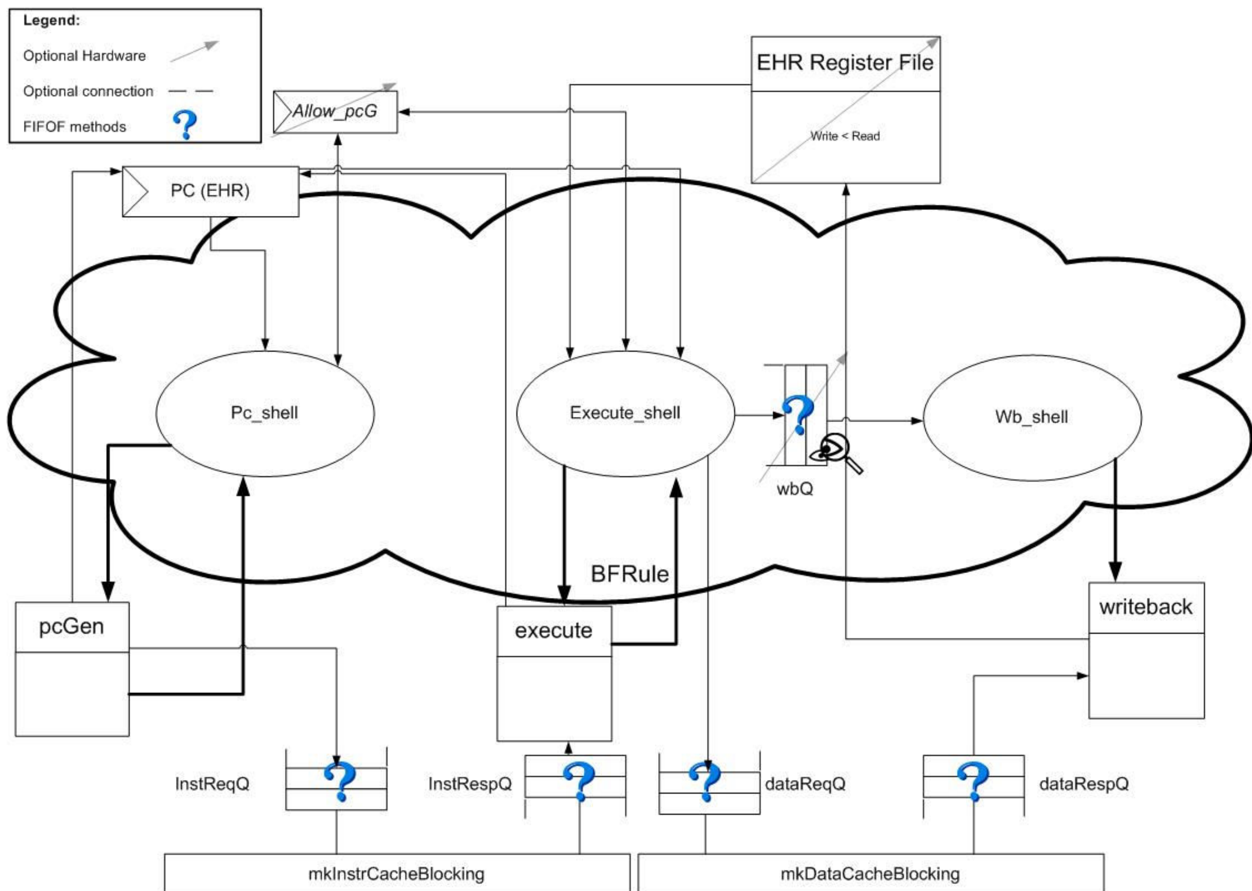


Figure 2: 2-stage implementation with pcGen and execute merged. During branches, execute reads directly from the PC register. The *allow_pcG* Boolean register coordinates the firing for the two stages by enforcing mutually-exclusive firing.

2.3.2 Merging execute and writeback

We merged execute and writeback by using a data-holding register, *rescomm*, and a synchronizing register, *allow_execute*.

The EHR register *rescomm* is used as a temporary holder for the output of execute. It is set so that execute can write before writeback reads, so as to allow concurrent firing of the two rules. This register needs to be particularly flexible in its workings, since its workings depend on the type of instruction processed. In case of ALU instructions, which do not need data memory, *rescomm* works as a wire between the two stages. For load/store instructions, *rescomm* acts as a register, holding its value until data memory responds. For branch-type instructions, finally, *rescomm* is set as invalid, since writeback does not need to fire at all after a branch.

The Boolean synchronizer *allow_execute* works similarly to *allow_pcG*, but allows single-stage firing, and has a few more complications. It allows concurrent firing by being EHR, so that execute can read it, fire, and write it before writeback does the same. Also, execute only sets the register to False after non-branching instructions, since in those cases writeback does not need to fire, and execute can fire again instead.

This implementation can use a regular read<write register file, as opposed to the standard EHR solution. In fact, with writeback and execute working on a single instruction, there are no read-after-write pipeline hazards. With the previous implementation, it was necessary to use a write<read register file for rule scheduling issued, as well as for correctness in the stall logic. This change to a regular register file implies high savings in hardware, as well as preventing a combinational cycle between execute and writeback, since, in a 2-stage implementation, the former fires before the latter.

Finally, the entire stall logic, implemented as a function, is never called, thus never synthesized. Figure 3 shows the 2-stage implementation that results from merging execute and writeback.

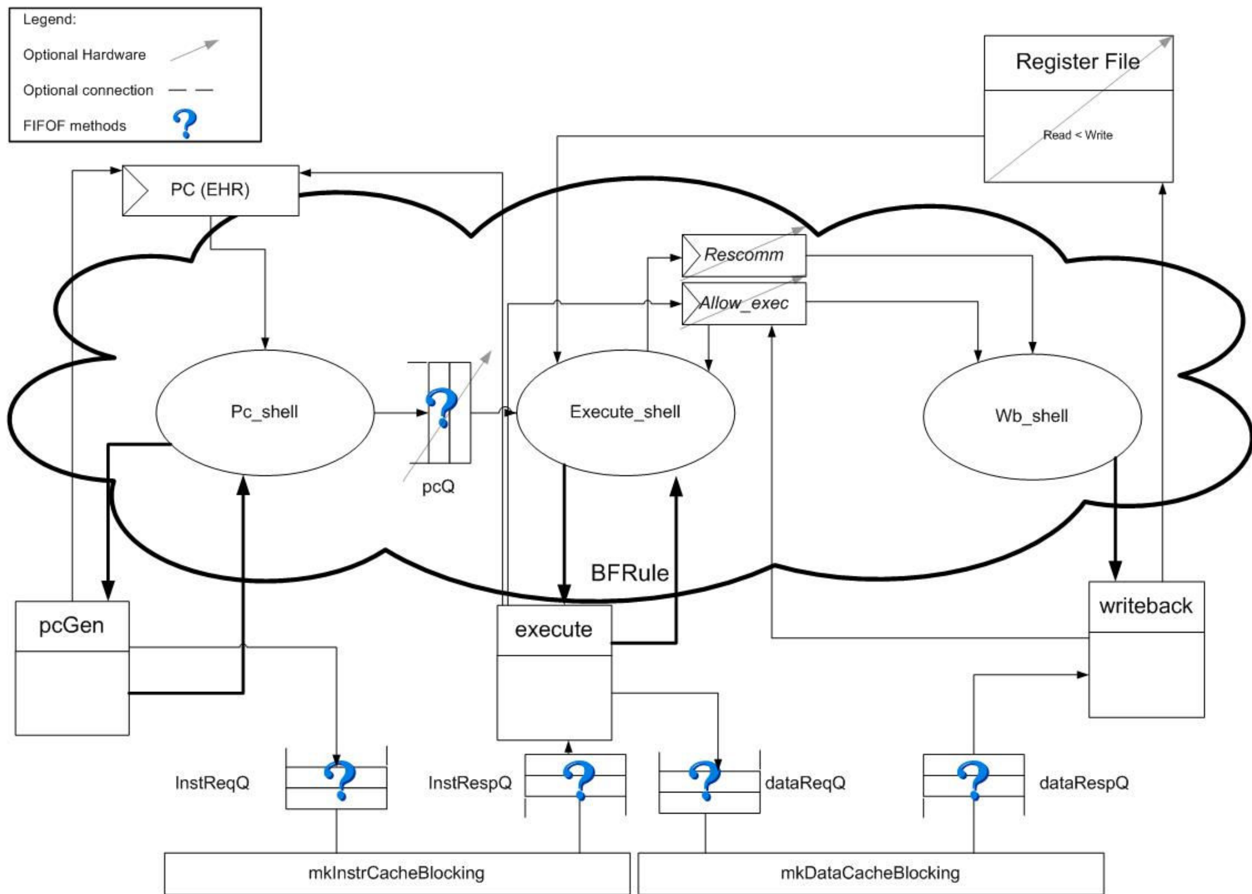


Figure 3: The 2-stage implementation with merged execute and writeback. The EHR register file is substituted with a regular read<write register file. The *rescomm* register allows for single-cycle (ALU instructions) or deferred (LD/ST) communication between the two stages. The Boolean *allow_exec* synchronized the two stages, while allowing single-stage firing.

2.3.3 Combining the merging: 1-stage version

Figure 1 shows a 1-stage version of the processor, which was was obtained by linearly combining the explicit guards and routing paths of both 2-stage implementations. This linear superimposition stands as a demonstration of true and elegant parametrization, because it proves that the two stage parameters are independent. The simplicity in the implementation of the 1-stage version leads a fundamental conclusion for future expansions: with new parameters introduced, code length and complexity increase linearly, while the number of possible processor configurations increase exponentially.

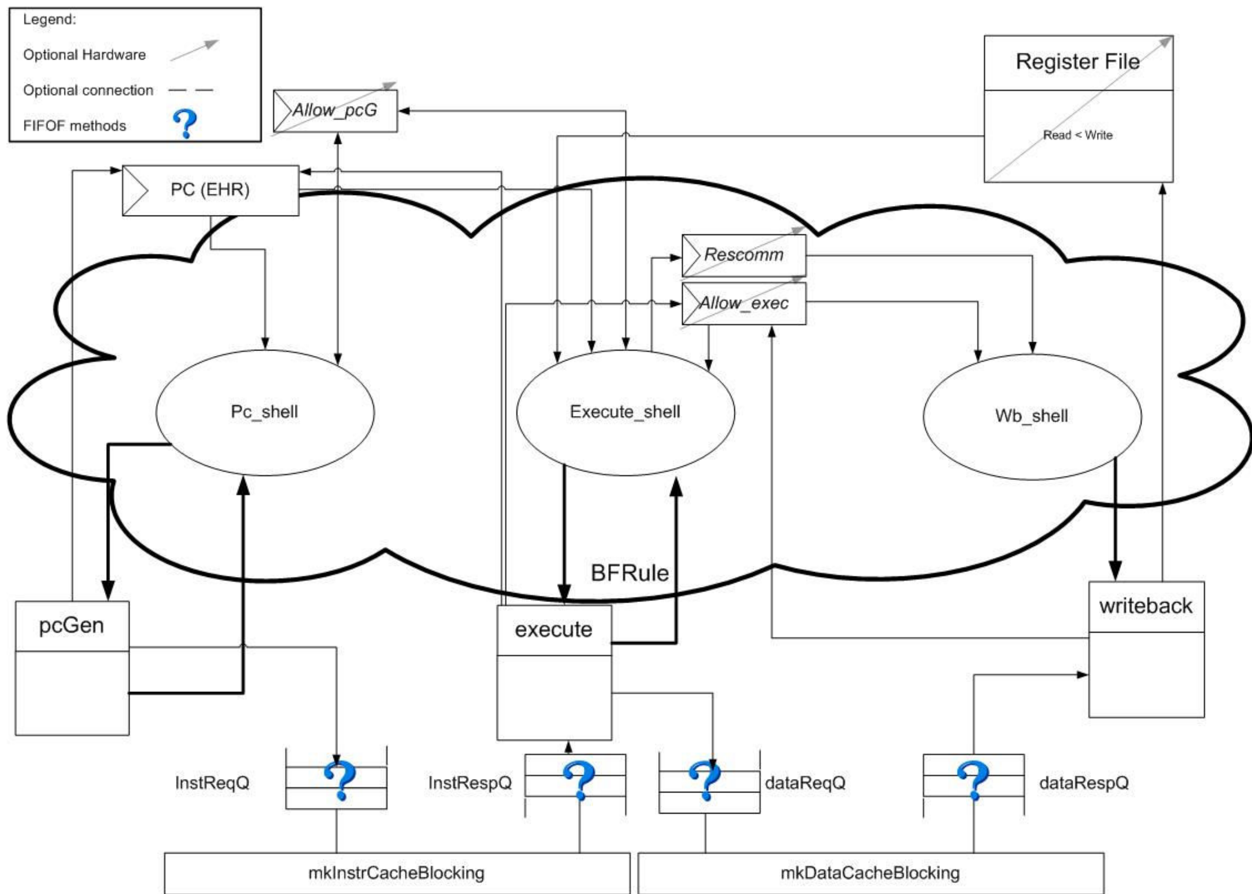


Figure 4: The 1-stage version of the processor. It was obtained by linearly combining the explicit guards and routing paths for both 2-stage versions.

2.4 Parameter Implementation

In implementing the parameters, we decided to use compiler macros to ensure both correctness and area optimizations. The two pipeline parameters are defined initially, and the commands *`ifdef* and *`ifndef* ensure that the compiler can skip parts of the code depending on whether the parameter is defined. We use compiler macros to regulate the composition of explicit guards, the routing paths, and for some hardware instantiations, e.g. the choice of register file type. However, most hardware optimizations, such as stall logic or pcQ/wbQ, is performed automatically by aggressive compiling. Figure 5 shows high-level schematics for our processor, with notiations on hardware that can be optimized out, and connections that are parameter-dependent.

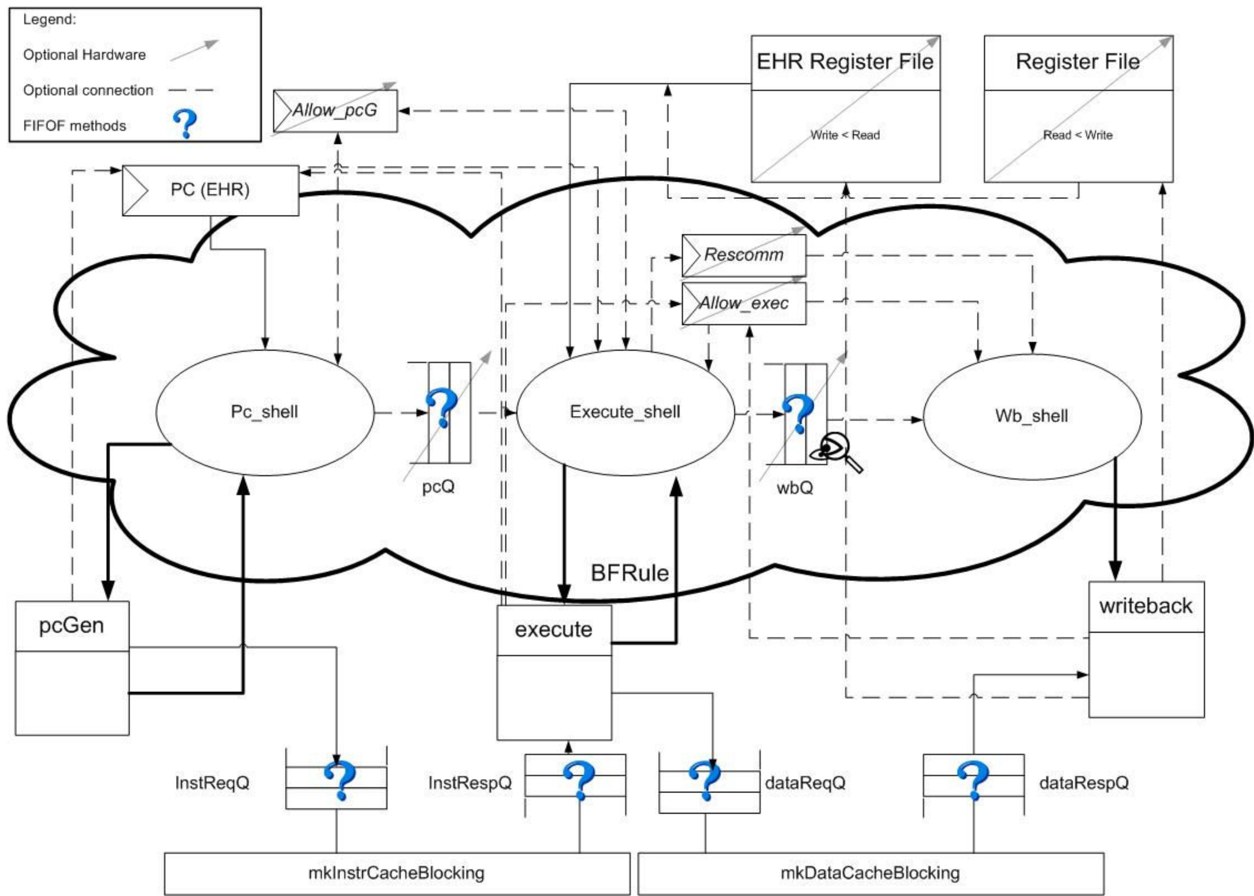


Figure 5: High-level processor view. The notation indicates hardware whose synthesis depends on the processor configuration, and parameter-dependent routing connections.

3. Test Strategy

Our parametrizable processor implements exactly the same instruction sets as the SMIPSv2 processor. That allows us to use the same tests and benchmarks as those used for the SMIPSv2 processor to ensure the correctness of the processor.

However, the primary purposes of designing a parametrizable processor are flexibility and adaptability, which implies the ability to tune area, clock period and power consumption to obtain the most efficient processor for the intended task. Therefore, the way in which the processor synthesizes for each configuration is a major concern for this project. Furthermore, for our work to be useful, we had to make sure that each configuration has some benefits, in term of area or Instruction per second (IPS).

Considering the huge amount of time it could take to manually synthesize and extract data for every processor configuration, we built a set of automated benchmarks, to change the parameters of the processor, synthesize it, and extract all relevant data such as area and clock period for each configuration automatically. Table 1 shows a summary of the characteristics we check for every configuration:

Table 1: Description of test parameters with comparative expected results

Parameter	Expectation	Best with
Post-synthesis total area	Area should increase with the number of pipelined stages	1-stage
Post-synthesis critical path and effective clock period	The effective clock period is expected to decrease with the number of pipelined stages, as we make the critical path shorter	3-stages
Post-place+route total area	Same behavior as post-synthesis total area	1-stage
Post-place+route critical path and effective clock period	Same behavior as post-synthesis effective clock period	3-stages
Instruction per cycle (IPC)	Due to parallel execution and shared resources, IPC should slightly decrease when there is more pipelined stages	1-stage
Instruction per second (IPS)	Even if IPC is a little worst for the maximum stages configuration, IPS should improve thanks to a increased effective clock period	3-stages

Unfortunately, we were not able to obtain power consumption information for the various processor configurations. This was due to the fact that parts of our implementation are described in Verilog, and the BlueSim power consumption simulation requires a full Bluespec work.

A designer using our parameterized processor should be able to easily choose the best configuration for his interests, which would certainly be a tradeoff between area and performance, and be able to quantify accurately the pros and cons offered by each configuration.

4. Results

We tested each possible configuration:

- without pcQ, without wbQ (1-stage);
- with pcQ, without wbQ (2-stage);
- without pcQ, with wbQ (2-stage);
- with pcQ, with wbQ (3-stage).

Here is an example of a complete simulation report, automatically generated by our benchmarks Makefile:

```
Configuration: woPCQ_woWBQ
   Post-synthesis total area: 19897.500000
   Post-synthesis effective clock period: 4.00ns - (0.00ns) = 4.00ns
   Post-place+route total area: 324682.6 um^2
   Post-place+route effective clock period: 5.000ns - (-1.056ns) = 6.056ns
   Benchmark: median
      IPC: 0.249868
      IPS: 41259577 /s
   Benchmark: qsort
      IPC: 0.249921
      IPS: 41268328 /s
   Benchmark: towers
      IPC: 0.249962
      IPS: 41275099 /s
   Benchmark: vvadd
      IPC: 0.249751
      IPS: 41240257 /s
   Benchmark: multiply
      IPC: 0.248531
      IPS: 41038804 /s

Configuration: wPCQ_woWBQ
   Post-synthesis total area: 20427.000000
   Post-synthesis effective clock period: 4.00ns - (0.00ns) = 4.00ns
   Post-place+route total area: 351514.2 um^2
   Post-place+route effective clock period: 5.000ns - (-1.701ns) = 6.701ns
   Benchmark: median
      IPC: 0.341072
      IPS: 50898671 /s
   Benchmark: qsort
      IPC: 0.410716
      IPS: 61291747 /s
   Benchmark: towers
      IPC: 0.316339
      IPS: 47207730 /s
   Benchmark: vvadd
      IPC: 0.384429
      IPS: 57368900 /s
   Benchmark: multiply
      IPC: 0.380400
      IPS: 56767646 /s
```

```
Configuration: woPCQ_wWBQ
   Post-synthesis total area: 22406.750000
   Post-synthesis effective clock period: 4.00ns - (0.00ns) = 4.00ns
   Post-place+route total area: 349309.6 um^2
   Post-place+route effective clock period: 5.000ns - (-2.060ns) = 7.060ns
   Benchmark: median
      IPC: 0.249868
      IPS: 35392067 /s
   Benchmark: qsort
      IPC: 0.249939
      IPS: 35402124 /s
   Benchmark: towers
      IPC: 0.249962
      IPS: 35405382 /s
   Benchmark: vvadd
      IPC: 0.249751
      IPS: 35375495 /s
   Benchmark: multiply
      IPC: 0.249350
      IPS: 35318696 /s

Configuration: wPCQ_wWBQ
   Post-synthesis total area: 22823.250000
   Post-synthesis effective clock period: 4.00ns - (0.00ns) = 4.00ns
   Post-place+route total area: 363453.0 um^2
   Post-place+route effective clock period: 5.000ns - (-0.433ns) = 5.433ns
   Benchmark: median
      IPC: 0.392375
      IPS: 72220688 /s
   Benchmark: qsort
      IPC: 0.434285
      IPS: 79934658 /s
   Benchmark: towers
      IPC: 0.419139
      IPS: 77146880 /s
   Benchmark: vvadd
      IPC: 0.453956
      IPS: 83555310 /s
   Benchmark: multiply
      IPC: 0.384702
      IPS: 70808393 /s
```

As the output example shows, we performed synthesis and place+route for all 4 configurations. However, we focused more on the synthesis results, which seems more relevant for several reasons:

- If we can reasonably expect to get every time exactly the same results for the synthesis step for a given code, it is not the case of the place+route step, because it is partially heuristic and very dependant on the synthesized code. Indeed, for insignificant (or even no) changes in Bluespec code, the area and effective clock period can change a lot.

- The place+route process is not ideal: we expect that the default floorplanning may not be optimal for every processor version, and in the worst case, it could even favor the pipelined version since it was primarily designed for this processor configuration. The idea of parametrizing the floorplanning configuration is feasible, but beyond the scope of our project.
- Since the theoretical minimal effective clock period is different for each version, and lower for the fully pipelined processor (since the critical path is shorter), having a fixed target for the effective clock period, whatever the configuration is, may result in certain case in a much bigger area than intended, as Encounter might desperately try to slightly decrease the clock period in cost of much bigger area.

At the contrary, synthesis figures don't really reflect the real area we will eventually get, but are probably more adapted to do comparisons between different architectures, since they are directly proportional to the number of gates and transistors in the circuit.

There is actually a huge number of possibilities and design exploration that are possible for every single processor configuration (1 to 3-stages), such as tweaking FIFOs size, the register file type, or implementing additional hardware for branch predictor and so on. Considering that, and also the huge amount of time required to benchmark each modification (4 to 5 hours), we choose to present our results for two main variations of our design, to show how optimization for each configuration could impact the performances of the processor.

We found that an EHR register file was actually not necessary for the 2 configurations with a merged execute and writeback, and that we could replace it by a regular register file with negligible IPC loss, but significant decrease in area, as it can be seen in our results. This is only one example, but it is important since it confirms the relevance of our project.

4.1 Area

As expected, the area of our processor increases when we add decoupling with pcQ and wbQ FIFOs. Turning the EHR register file to a regular one effectively decrease the area. If the area is a primary concern for the designer, the no-pcQ and no-wbQ (1-stage) version of the processor is the smallest.

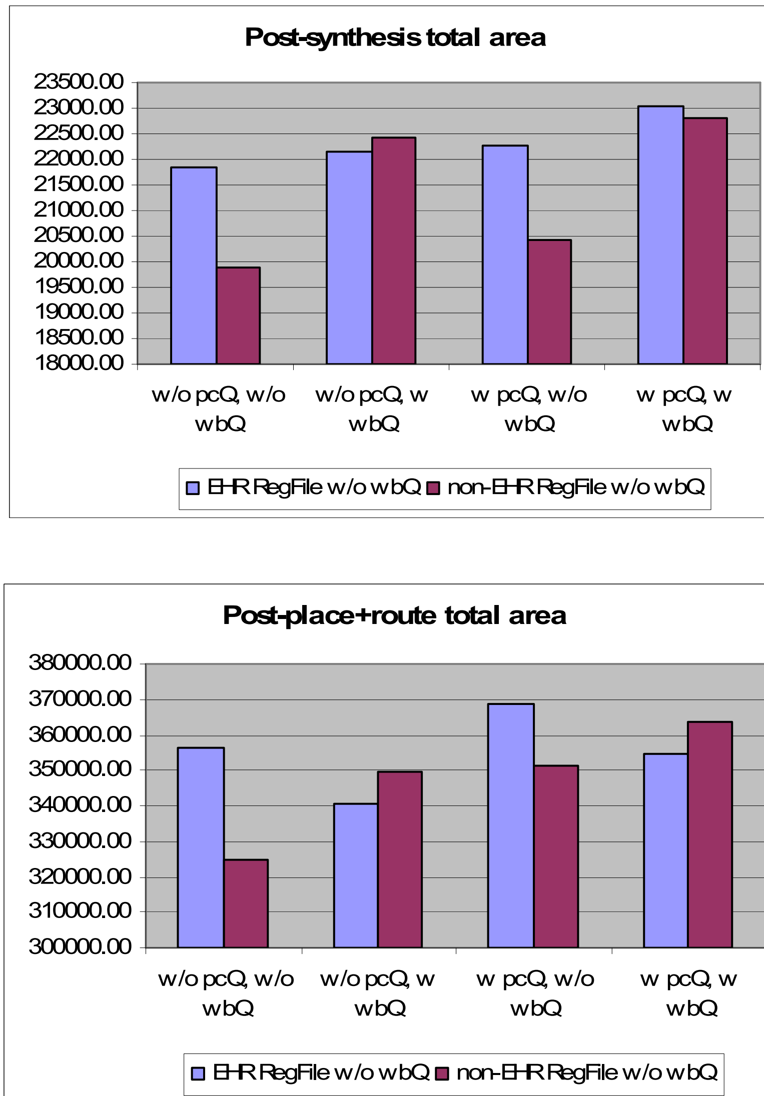


Figure 6: Post-synthesis and post-place+route ares results, for EHR and non-EHR versions

Place+route area is very different from the synthesis area, and leads to the previous discussion about the relevance of theses numbers. However, the general trend is still confirmed for the non-EHR version.

4.2 Instruction per Cycle (IPC)

IPC is important as it reflect the degree of parallelism of the processor. For the 1-stage processor, IPC is comparable to the baseline Bluespec processor, and increase with parallelism. Of course, a lot of effort could still be done in order to increase it, especially for the 3-stages version, mainly by introducing a basic branch predictor. It is important to notice that, for our set of benchmarks, the elimination of the pcGen-execute pipeline causes a dramatic decrease in IPC. This happens because the instruction memory is accesses every cycle, and the pcQ FIFO plays a role decoupling the two stages, thus hiding the memory cycle latency.

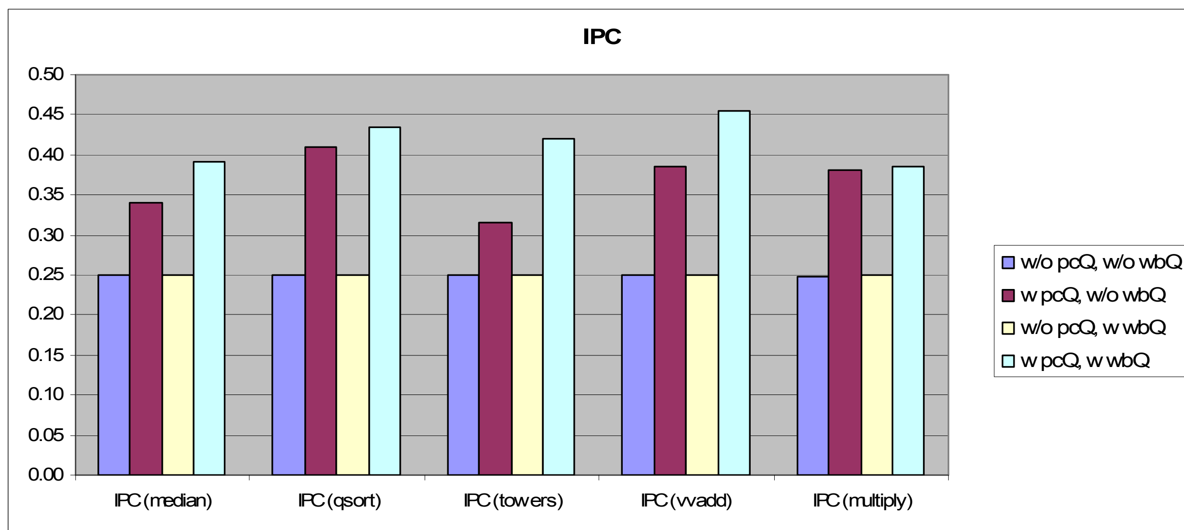


Figure 7: Instructions per Cycle results, for various benchmarks and configurations

4.3 Performance

The effective clock period significantly decrease when the three stages are decoupled, as expected, since the critical path gets shorter. However, it doesn't shrink systematically with the number of stages, since the clock period for the 2-stages processor is pretty much the same as the 1-stage version, suggesting that the ALU is the main latency source. It has to be noticed that the effective clock period increase with the non-EHR even with wbQ (which should not change since the non-EHR version only concern the no-wbQ versions), so it is not very relevant and should be considered for a given the area.

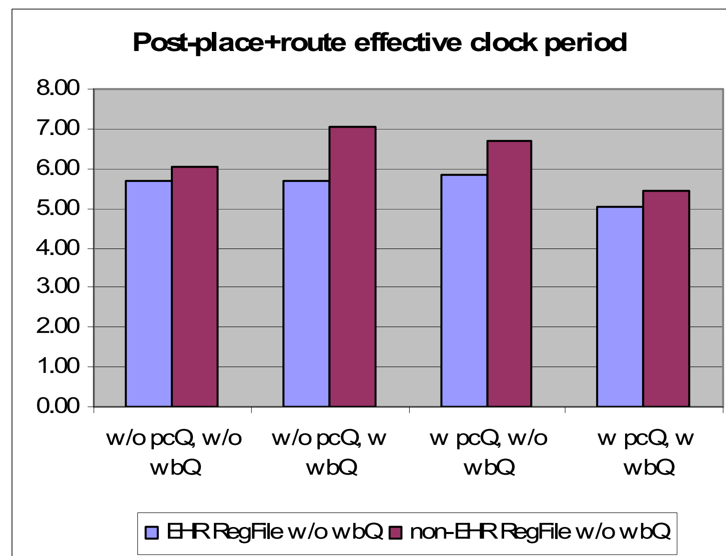
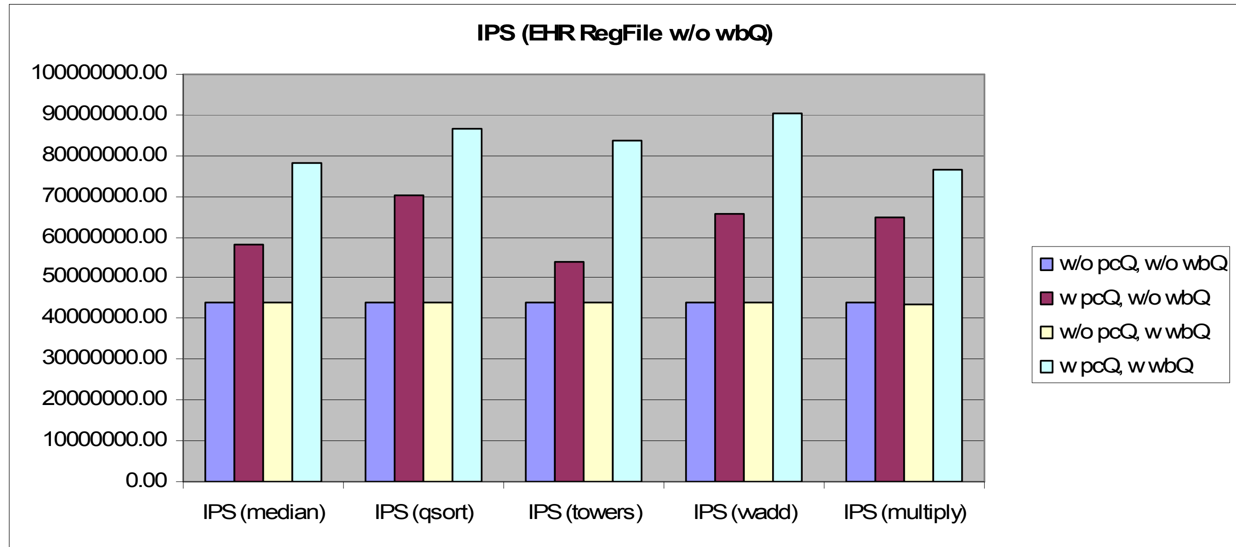


Figure 8: Effective clock-period results

To actually measure the effective speed of our processor, we have computed the number of instructions per second (IPS) for our benchmarks. It is expected that the IPS increases with the number of pipelined stages, as more parallelism is achieved. Given our results, IPS is more dependants on IPC than the effective clock period, which is quite constant. This is interesting, since it tell use that we should focus on improving IPC without worrying too much on critical path.



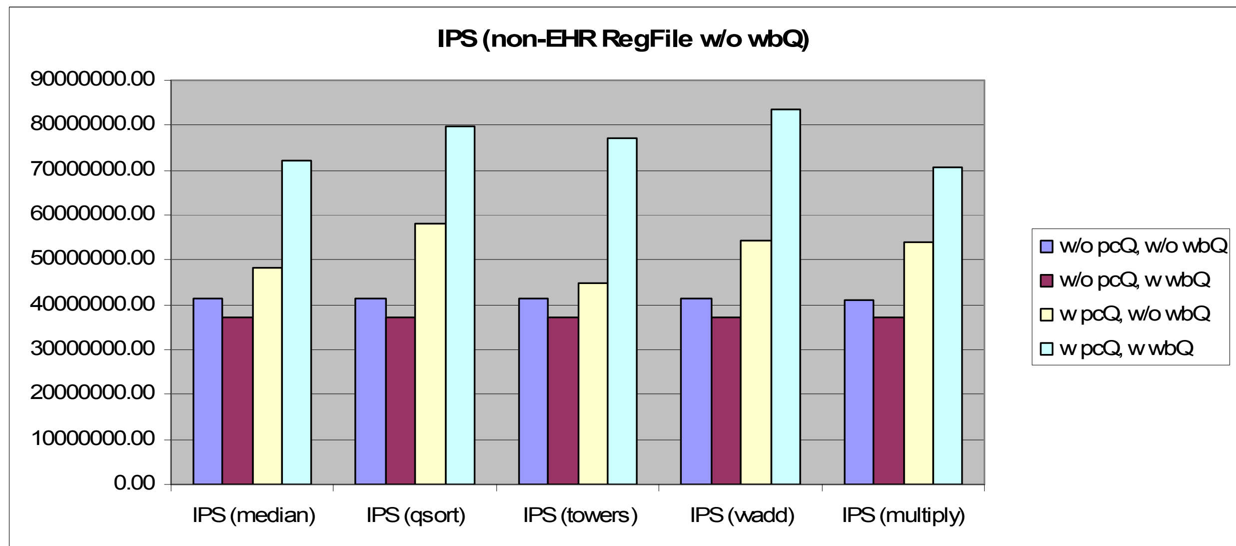


Figure 9: Instructions per Second results for various benchmarks and configurations

We can see that the IPS don't increase at all by pipelining execute and writeback stage without pcQ FIFO. This seems relatively easy to explain, as decoupling writeback stage is really useful only for read or write into the data memory, because the memory has some latency. But since pcgen and execute stages are not decoupled, the throughput is actually limited by the instruction memory latency.

4.4 Summary

For our parameterizable processor to be useful, we should take care that the gain in speed is compensated by the increase in area, and vice versa. To estimate the overall performance of each configuration, we introduced a “figure of merit” coefficient, which takes into account the tree main preoccupations of a designer: speed, in term of IPS, area and power:

$$FOM \propto \frac{IPS}{Area \times Power}$$

Since the power is hard to estimate with accuracy, especially in regards of our considerations about the place+route process, we just made the very rough assumption that the power is proportional to the number of gates, therefore to post-synthesis area. Our therefore becomes $FOM \propto \frac{IPS}{Area^2}$, which is probably fair enough for a first comparison between each variation of our processor.

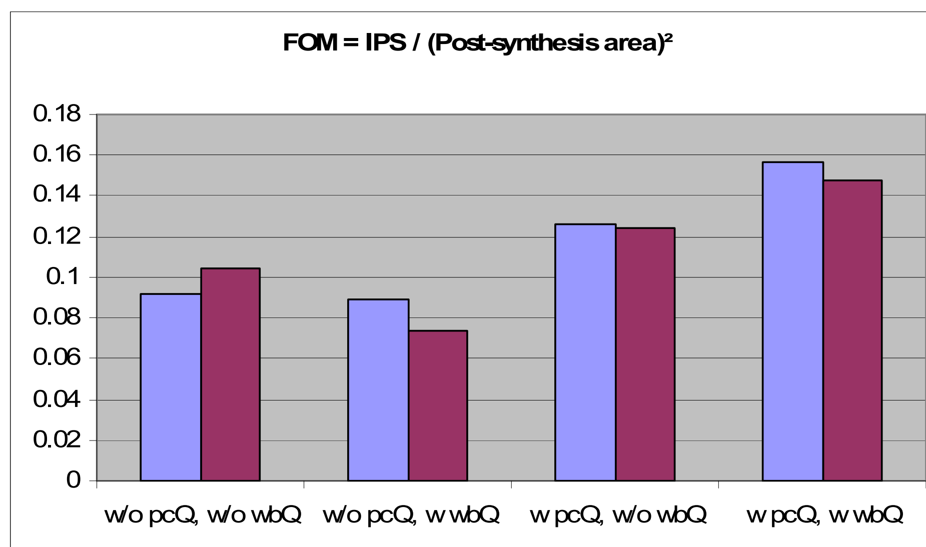


Figure 10: Figure of Merit tradeoffs for various configurations

Currently, without further optimization, the 3-stage configuration has the best FOM. However, the figures are relatively well-balanced for each processor configuration. It is safe to say that no version truly dominates the others, therefore the parameters provide a balanced tradeoff.

Results show that both versions with merged pcGen and execute stages display a dramatic decrease in performance, whose has been explained. These results, however, do not exclude those versions as purely dominated, since the performance decrease depends on the type of benchmarks, and on the baseline processor configuration. It can be shown that, under a different configuration and benchmark code, those two implementations can show improved performance. For example with branch prediction, and vector-based code with a high-number of load/store operations, there would be high branching, low instruction memory access, and high data memory access, making for a feasible no-pcQ version.

Each configuration could probably be optimized further, especially for the place+route step, with clever floorplanning for instance. But our results already demonstrate the viability of a parametrizable processor, which could be worthwhile particularly for specialized multi-core processors, where every core doesn't need to run at the same speed.

5. Conclusions

We managed to design and implement in Bluespec a parametrizable processor where the number of pipeline stages, and thus the degree of parallelism, is defined using a single parameter. Our final design includes 4 configurations consisting of a 1-stage, two 2-stage and one 3-stage processor.

The key to the usefulness and value of our project was ensuring that there is something to gain from each configuration. Otherwise, if one processor version were superior to all the other versions in all respects (area/performance/power) then there would be no need to ever use the other configurations. Namely, we had to ensure that, on going to lower levels of parallelism, we gained in other dimensions such as area and/or power consumption. By optimizing each configuration, this turned out to be the case. As we go to lower degrees of parallelism, we reduce the area of the processor by primarily eliminating bulky FIFOs. Secondly, we can remove complicated combinational logic that served to protect against data hazards that only occur due to concurrency of pipeline stages. Additionally, at low levels of parallelism, a simpler non-EHR register file was sufficient to ensure correctness and optimal performance, thus saving even more area on the processor chip.

Further studies that we would have liked to perform, given more time, include power analysis and comparison between the different processor configurations. We feel that, similar to area, power consumption would show a decrease with decreasing levels of parallelism, again due to the absence of FIFOs, less supporting combinational logic (such as the stall function) and a simpler register file. Secondly, we would have liked to commit some time to floorplanning; our area analysis is based mainly on post-synthesis results. This is because post place-and-route results are highly heuristic and were optimized for the three-stage version of the processor. If we had time to experiment with floorplanning, we feel that the results would have been just as good, if not better, than the post-synthesis results; that lower degrees of parallelism allow for smaller processor area.

Finally, we were happy to see that different parameter configurations were linearly super-imposable. While our parameter-regulated processor certainly took more engineering effort than four separate processors, we are confident that, in future expansions, the addition of more parameters will be linear growth in complexity, and an exponentially higher number of possible configurations, thus making our processor a valuable tool for the SOC designer.

6. Acknowledgements

We would like to thank the entire 6.375 staff for their support and suggestions, in particular our TA Myron King. We would also like to thank the 6.375 student community, who were always willing to share suggestions and possible solutions for commonly encountered problems.