SIMD Extensions for the SMIPS core (Group 4)

Mykal Valentine Svilen Kanev

May 13, 2010

1 Project Objective

This project aims to create SIMD extensions to the SMIPS core already designed in the class. Our main goal is to obtain better performance for applications with high level of data-parallelism. The instructions we are planning to implement are general enough to be used in a wide class of applications that benefit from a smaller number of instructions for the same data throughput. Generally speaking, we are targeting multimedia, DSP and scientific applications.

Vector extensions to scalar processors aren't exactly a novel idea. There are quite a few successful commercial products shipping (especially MMX and SSE1-4) and under development (f.e. Intel's Larabee), not to mention the fact the whole GPU segment is based on the same basic principles. But, even if not a groundbreaking new idea, actually implementing the extensions allows us to get a lot of hands-on experience in a field that is going to stay active in the next few years.

On a very high level, our proposed architecture includes a scalar SMIPS core and a vector coprocessor that can exploit data-level parallelism. The coprocessor is more or less another execution unit in our core, which supports variable-length instructions. Synchronization is achieved via main memory and a limited amount of value forwarding. Since we are sharing the front-end of an in-order core, we achieve serial consistency.

2 High-level Design

2.1 Architecture Overview

Choice of extensions The first major choice we had to make was whether to implement an existing set of MIPS SIMD extensions or to create a custom solution. While we are aware of two extension sets for the MIPS architecture, we chose the latter option. From the existing solutions, MIPS-3D has been implemented in commercial processors, but is only focused on 3D operations (clipping, lightning), and MDMX has never gotten implemented and is, therefore, lacking a clear standard. Our aim is to implement a slightly broader set of extensions than MIPS-3D that can be used in a wider class of applications. A prior project for this class [2] has already implemented a similar set of extensions. In order to reuse the infrastructure that has been built, we keep compatibility between the instructions shared by the two projects as much as possible.

Architecture Our architecture includes a SIMD pipeline that operates on vectors of 4 32-bit values. In order to minimize area overheads, we intend to keep the width of the SIMD unit at 32-bits. An overall view of the architecture can be seen in Figure 1. We will go into a much more detailed description in Section 4.

The SIMD instructions follow the general MIPS ISA and are part of the regular instruction stream. They are fetched from the shared fetch unit, decoded and subsequently sent to the vector pipeline. Once such an instruction is sent to the vector unit, the scalar pipeline is stalled until the vector operation is complete. Since it is operating on a different memory size, the vector unit uses a separate 128-bit register file, but shares the memory access port with the scalar pipeline. Consistency is ensured because no pipeline is allowed to execute while the other one is busy.

Initially, the vector pipeline consists of a single 32-bit ALU and a vector register file. Even though in terms of pure computational capabilities this is not significantly more than a regular scalar pipeline, we still expect a moderate performance improvement. The main reason is reducing the stress to both instruction and

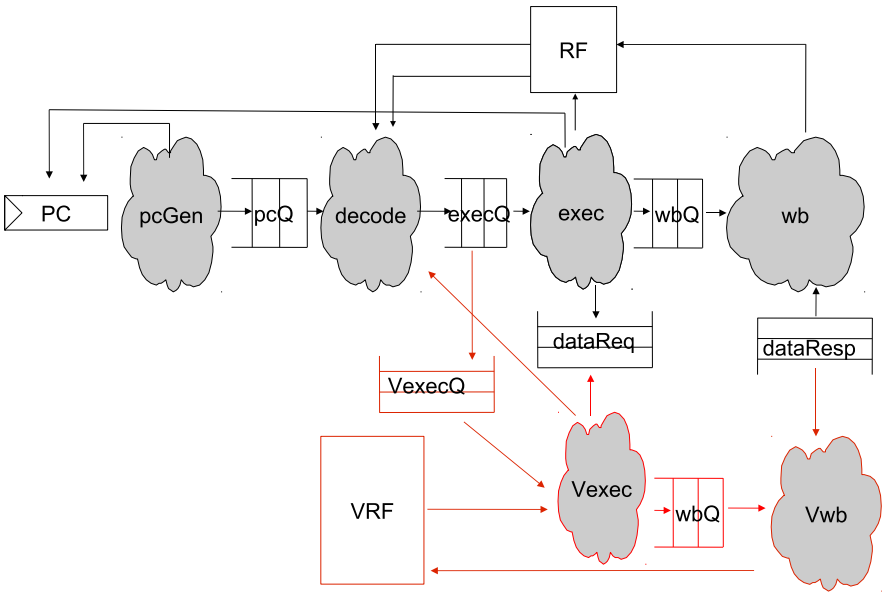


Figure 1: Overall architecture for the SMIPS SIMD machine

data memory - with less instructions per arithmetic operation and more regular (and hence predictable) data memory access patterns. Such an architecture is also able to hide some of the data memory access latency by beginning computation on vector elements before the whole vector is fetched.

In Section 6 we explore decoupling the two pipelines and allowing the scalar one to run past a long-latency vector operation, as long as there is no memory data dependency. With such decoupling, in theory, both pipelines can execute on each cycle, keeping a throughput close to 1, despite the shared fetch unit. The choice of such architecture is motivated by the fact that most of today's workloads are memory-bound. By effective usage of the instruction memory we are trying to decrease memory workload.

2.2 Instruction set extensions

We are extending the SMIPS ISA trough the COP2 interface described in the ISA definition [1]. A complete list of the extended instruction formats follows in Appendix A.

Based on the instruction format that we chose early on, we have 16 opcodes available for `c2` instructions. We have implemented 12 `c2` instructions, so we are actually close to that limit. In choosing which instructions to implement, we were guided by what our benchmarks require in order to execute. That is why the implementation is missing, for example, a full spectrum of logic operations.

Before describing the actual instructions, we should setup some semantic definitions. For a vector register, rx.compy refers to the y-th component of vector x. Furthermore, all vector operations are masked - that is, results on some elements can be discarded if internal mask bits for those elements are cleared. This is done mainly so that the machine can operate on vectors of length smaller than 4. We have implemented instructions to explicitly deal with the masks.

Also, we extend the SMIPS ISA convention that register $0 always evaluates to 0 to the vector register $0.

2.2.1 Vector memory instructions - lwc2 and swc2

These instructions perform the memory operations for the vector unit. Each of them operates on whole vectors (even though the actual implementation can take multiple cycles to execute). They are the only interface to transfer data between the scalar and vector pipelines.

The address base is stored in the scalar register file. The motivation for that decision comes from the fact that address calculations (meaning address generation by previous instructions, not simply offsetting the address base) are generally simple and scalar, so there is no reason for their results to be stored in the vector

register file. In that case, the full address generation can be done immediately after reading from the scalar register file, which in our pipeline happens in the scalar decode stage.

2.2.2 Two-operand instructions - addv and mulv

These instructions simply add two vectors, or perform per-element multiplication. Put more formally:

$$\texttt{addv rd, rs, rt} \equiv rd.comp[i] = rs.comp[i] + rt.comp[i]$$

$$\texttt{mulv rd, rs, rt} \equiv rd.comp[i] = rs.comp[i] * rt.comp[i]$$

2.2.3 One-operand, immediate instructions - addiuv, andiv, sllvv, srlvv

These instructions perform arithmetic or logic operations on one vector and one scalar immediate value. Due to instruction-length constraints, the immediate value is limited to 11 bits. This is enough for the two shift instructions, but should be taken into account if, for example, `addiuv` is used to initialize vectors to large values.

2.2.4 Mask instructions - seteqv, setnev, setltv

These instructions follow the format `set*v rd, rs, rt, m`. They test for the specific condition each element of the source operands and write 0 or 1 in the respective destination elements. If the `m` immediate is set, they also update the mask bit corresponding to that element index with the comparison result. All other instructions that operate on vectors are aware of the mask bits and don't perform computation on an element if its corresponding mask bit is cleared.

As mentioned above, these instructions can be used to limit the vector length below 4. Their more general use allows conditional operations on individual vector elements that still benefit from the single-instruction format.

2.2.5 Swizzle - swiz

This instruction computes a permutation of the elements of a vector. It takes an immediate value that specifies which source element is assigned to the respective destination position. By definition:

$$\texttt{swiz rd, rs, ijkl} \equiv \begin{cases} rd.comp0 = rs.compi \\ rd.comp1 = rs.compj \\ rd.comp2 = rs.compk \\ rd.comp3 = rs.compl \end{cases}$$

Note that the instruction permutates the source vector elements, not the destination ones. This is done so that the implementation of the instruction can also benefit from register chaining, which we describe in Section 4.

2.2.6 Dot products - dot4, dot4v

The first dot product flavor, `dot4 rd, rs, rt, di` operates on two "horizontal" vectors, performs the per-element multiplications and stores their sum in `rd.di`. Here, by horizontal we are denoting the vectors used so far by all other vector operations.

In contrast, `dot4v rd, rs, rt, di, i2` operates on one horizontal and one vertical vector. The vertical vector is stored in four consecutive vectors in the register file, at index `i2` of each one. That means that even though the instruction semantics shows a read from `rt`, the instruction also performs reads from `rt+1`, `rt+2` and `rt+3`. The rest of the instruction operation is similar to `dot4`.

We realize that `dot4v` has a rather unusual format that may put more stress on the vectorizing compiler. However, we have found such an instruction extremely useful in computing vector-matrix products without having to store the transposed matrix in memory. The mentioned compiler complexity (making sure that registers after `rt` hold the correct data) can be mitigated by porting the vector-matrix product (including loads of the 5 vectors involved) to a pseudoinstruction, or a small routine that can be inlined.

3 Testing infrastructure

Because this project is combining a scalar pipeline and a vector pipeline, it's important to show that the processor is not only efficient when performing vector operations, but also not slow when performing scalar operations. For this reason, the benchmarks have to not only test the vector operations, but also scalar operations.

3.1 Scalar benchmarks

The scalar benchmarks suite is the same we used in Lab 5 to evaluate our revisions of the SMIPS core.The 5 benchmarks (`median, multiply, qsort, towers, vvadd`) show different characteristic, so, for the scope of this project, we can assume they are a comprehensive suite. Since the vector instructions are only extensions to the ISA, our ISA revision is backwards compatible and the benchmarks don't need to be recompiled. We can also reuse the SCE-MI connection used in Lab 5 and gather performance statistics (IPC, number of cycles, cache references) for the performance evaluation.

3.2 Vector benchmarks

3.2.1 Build infrastructure

Since we are implementing custom extensions to the instruction set, we cannot rely on an off-the-shelf compiler to generate vector instructions. However, we could reuse the infrastructure from [2] with slight modifications for our instruction format. This infrastructure consists of a pre-assembler step that parses the `c2` instructions and inserts their bytecode in the assembly file, which is later processed by regular SMIPS assembly. This pre-assembly step is conveniently integrated in the `smips-gcc` toolchain.

For all vector benchmarks, we have C and assembly versions of the source code. The C code can be compiled with `smips-gcc` and run on the scalar SMIPS code. The assembly versions are hand-written and include vector instructions. They are compiled using the infrastructure described above and run on the vector core. In that way, we can evaluate the benefits from the vector extensions.

3.2.2 multiplyv, vvaddv

These benchmarks are vectorized versions of the `multiply` and `vvadd` benchmarks from the scalar suite. Credit for the vectorization goes to [2]. `vvaddv` does a certain number of back-to-back vector adds in two arrays. It has a relatively small main kernel (≈ 8 instructions) and utilizes the `addv` instruction. `multiplyv` computes multiplications of the elements in an array by a shift-and-add algorithm. The kernel size is ≈ 15 instructions.

3.2.3 geometry

This benchmark is also due to [2]. It performs a fixed set of geometric transformations on coordinate data. Initially, we were planning to implement a benchmark that convolves a 2-D image with a small filter kernel. After looking at the code of `geometry`, we realized that image convolution would result in virtually the same computational requirements, so we decided to use this benchmark instead. The kernel consists of ≈ 15 instructions and uses the `dot4`instruction heavily.

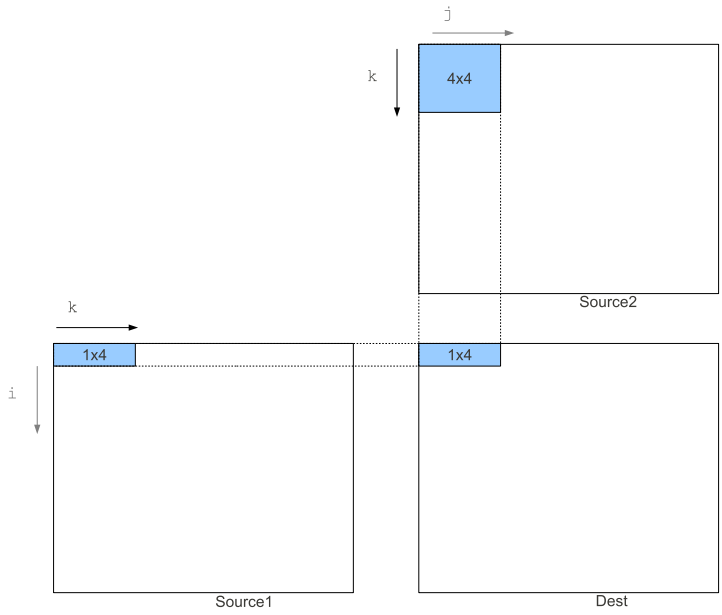


Figure 2: Algorithm for block matrix multiply used in `mmul`

3.2.4 mmul

This benchmark performs block matrix multiplication, which is one of the preferred algorithms for SIMD architectures. It splits the second matrix in 4x4 blocks and computes the vector-matrix product of each block and a vector from the first matrix, accumulating the result. A visualization of the algorithm can be seen at Figure 2.

The kernel is larger than the other benchmarks ($\approx$ 30 instructions) and mixes vector and scalar instructions. The scalar instructions are needed for address computations, since the algorithm cannot be broken down to simple sequential access of the matrices and needs data from different rows. It uses the `dot4v` instruction to perform the vector-matrix products.

4 Microarchitecture

4.1 Vector unit

The vector pipeline is in a separate module than the scalar SMIPs processor. This module is responsible for executing the instructions that the SMIPS decode rule sends to it. The module is instantiated with references to the shared structural units (execute queue and memory ports) and can access their methods. This way the only thing the vector unit has to expose is a busy signal, whereas the scalar unit exposes `DataReqQ, DataRespQ, execQ`.

```
interface VectorProc;
        method Action enq(VEXResult intr);
        method Bool Busy();
        method Action clearExQ();
endinterface
```

When a vector instruction is encountered, the decode stage of the scalar pipeline proceeds normally and enqueues it in the execute queue. From there, if appropriate, it is dequeued by the vector unit, which asserts a busy signal, indicating a long-latency operation. The scalar pipeline is aware of that signal and doesn't dequeue scalar operations at the head of the execute queue to ensure consistency.

For the multicycle instructions, the execute stage should not enqueue anything into the writeback's queue until it is finished its computation. In order to prevent anything from being added to the execute stage, the execute stage's queue is not dequeued until the computation is finished. A state variable is used to keep track of the computation so that the execute stage can know where in the computation it is.

The interface for the register file is a modified form of the register file used for the scalar pipeline. The data is still read in 32-bit chunks, but in order to access or write registers, a index needs to be supplied. The implementation of the register file is a vector of 4 regular register files. Each register file has 16 32-bit registers. The index supplied to the vector register file is used to select which register file to operate on.

Bypassing through this vector register file is also changed from the implementation in the scalar pipeline. A tuple with just the register and the data that was written is no longer enough. The tuple needed to have a third piece of data, the index of the register that had been changed.

```
interface Rfile;
        method Action wr(VRindx rindx, Vindx vindx, Bit#(32) data);
        method Bit#(32) rd1(VRindx rindx, Vindx vindx);
        method Bit#(32) rd2(VRindx rindx, Vindx vindx);
endinterface
```

4.2 Vector Rules

The Vector coprocessor only contains two rules, since all instructions are fetched and decoded in the scalar pipeline. The rules are responsible for executing and writebacking the vector instructions. Each rule has multiple stages.

4.2.1 Execute rule

The execute rule has a 2-bit stage register associated with it. On each cycle that the execute rule is active for, it increments the stage register. The stage register indicates which index of the register the execute stage is currently operating on. For most instructions, this is used to read from the register file.

For ALU and memory instructions, each execute stage enqueues the result of the calculation on that index to the writeback queue, along with the index it was doing the calculation on. When the execute rule is acting on the last index of the vector, the rule dequeues the current instruction.

This is different for the `dot4` and `swiz` instructions, however. For example, if the read register and the destination register are the same, then the execute wouldn't be able to properly calculate the result for the whole register if parts of the register are written before the `swiz` instruction finishes. As for the `dot4` instructions, there is no result to enqueue into the writeback queue until the data for the last index of the vector has been read. Thus, we need accumulator register for these instruction types.

For the `dot4` instruction, an extra stage was added in order to avoid doing a multiplication and an addition operation on the same data on the same cycle. The product of the indexes of two registers is added to the accumulator one cycle after the multiplication takes place. While a `dot4` instruction is executing, it enqueues the writeback with NULL, which the writeback rule will ignore. This was done in order to solve an issue with the pipeline enqueuing a dependent instruction into the data of the first element of the vector from the writeback stage being written. It also has the side effect of decreasing the critical path which happens to be in the datapath for that instruction.

4.2.2 Writeback rule

Because the execute stage enqueues something into the wbQ for each index of a vector, the writeback rule behaves in a similar way to the scalar writeback rule for ALU and memory instructions, with an addition of receiving the index from the execute rule. For the instructions that use an accumulator and writes to the whole vector, it acts differently.

When writing the whole vector, the writeback doesn't dequeue from its queue until its finished writing to the last index of the vector. When using an accumulator to write to a whole vector, it selects a 32-bit section of the accumulator, depending on the stage register that the writeback rule increments, and writes that to the index corresponding to that stage.

4.2.3 Chaining

Since each pass of the execute rule enqueues something in the writeback stage(except for `dot4` and `swiz`), when the last 32-bit piece of data for an instruction is entered into the writeback queue, three parts of the vector have already been written to the vector register file. This means, when a new dependent instruction is entered into the vector coprocessor, the data for the first element of the vector has already been written. This allows the instruction in the execute stage to be able to read the most recent value of the register at that index that is currently being executed.

4.3 Scalar pipeline modifications

We decided that it didn't make much sense to have the execute stage also be responsible for the dispatch to the vector unit, so we decided to create a decode stage. We also thought that this would help us increase the clock, but the longest path would actually be in the vector unit.

The fact that the scalar register file is accessed before passing control to the vector pipeline allows us to forward data to it without a specific instruction. This possibility can prove useful for any scalar data used by the vector unit. Two examples are the base address of a memory operation, or the scalar from vector-scalar addition.

4.3.1 Decode rule

The case statement for decoding the instructions is more or less the same as the lab5 execute code except with the addition of the vector operations. Unlike the lab5 execute, however, the decode rule does not compute any of the values. It just reads data from the registers depending on the instruction and enqueues the instruction into the exQ.

Because some instructions compute the same functions, even though the data being operated on may be different, when they are entered into the exQ, they will have the same tag, since as far as execute is concerned, they are the same. This reduces the amount of decoding that the execute stage has to do to compute the functions.

4.3.2 Stalling

The decode rule also needed a couple of new stall instructions both because of the new exQ and also because of the vector coprocessor. The stall function for the exQ looks much like the stall function used for the wbQ in lab 5. It needed to be changed to allow for bypassing values that the execute stage.

In order to do this, a new method was added to the register file called `execWr`. When called, execWr writes a register and data tuple to an `RWire`. When a register read is performed, this wire is checked first. This method is called at the end of the `exec rule` with the data that the execute stage has just calculated. The execute stage also writes to an `RWire` so that the stall function for exQ can check to see if the execute stage had written to a register that it would otherwise be stalling because of.

5 Implementation evaluation

5.1 Methodology

The main purpose of our evaluation is to compare the vector unit performance relative to the scalar core. Since we are comparing two different architectures, IPC is not the correct metric to use. We are running the same workload (the amount of useful work indeed is the same, even though the vector version uses fewer instructions) and it makes sense to compare the time it requires to execute. To estimate that, we run our workload and get the number of cycles it needs to execute from the SCE-MI bridge infrastructure. We multiply that result by the minimum cycle time the design can run on, based on synthesis results. Our workloads are short enough that running in simulation is feasible. For ease of debugging we used simulation results for the number of cycles, after validating the initial values with the synthesized design on the `Virtex5` board.

5.2 Scalar performance

We were expecting changes in scalar performance because we introduced a significant change in the scalar pipeline by adding a decode stage, where the register reads are performed. Our initial results showed a very significant decrease in scalar execution times of the new core - on the order of 50%. While some fraction of that can be attributed to the lower frequency that the longer pipeline is running at because of the vector unit (we address that issue in 5.4), it was clear that there was an architectural reason for the low performance. To address that, we looked into better branch prediction and adding a scalar forwarding unit.

5.2.1 Branch prediction

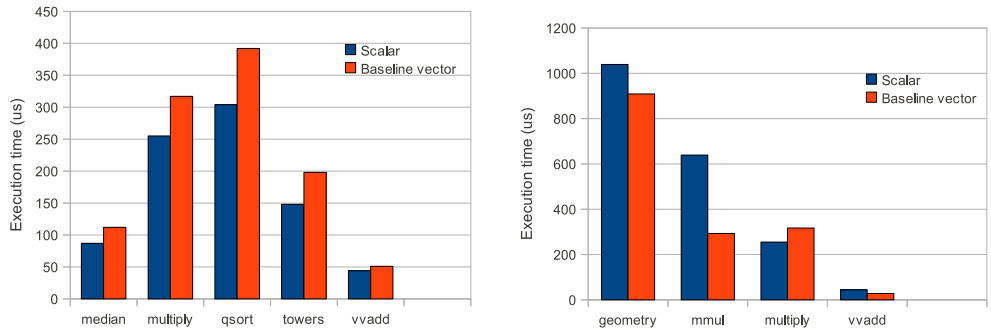
Initially, we assumed that the loss of performance was due to the fact that, by adding a decode stage to the front-end, we had moved the branch resolution point closer to the back-end, thus increasing the branch misprediction penalty. We hoped to compensate for that effect by improving branch prediction accuracy. Thus, we implemented better-performing branch predictors - a simple bimodal scheme and a more complex neural-net based one. While we saw a definite increase in prediction rates in the bimodal scheme relative to the baseline, the IPC of the pipeline virtually didn't change. This indicated that there was a more serious bottleneck than branch prediction.

5.2.2 Exec → decode forwarding

We identified that a portion of the problem arose from back-to-back dependent instructions. Since the initial architecture only had a writeback → decode forwarding unit, the second of a pair of consecutive instructions with a RAW dependency had to stall until the first instruction reaches the writeback stage. The solution was simple - adding a exec → decode forwarding unit and giving it priority over writeback forwarding (so that the third instruction in a 3-long RAW dependency chain receives the correct value). This forwarding path is noted in Figure 1.

5.2.3 Results

Benchmark results for scalar performance can be seen at Figure 3a. Execution time increase is 25% on average, but 16% can be attributed to the decreased frequency. This data is not shown, but the different branch predictors still result in virtually the same performance. This means that there is still a more serious bottleneck with our scalar design. Our final hypothesis for the cause of tat loss is that increasing the length of the front-end requires a larger instruction memory request queue to effectively hide the longer pipeline latency. We couldn't verify that assumption mainly because of the limited time (we couldn't quickly come up with a multi-element bypassing FIFO), but since the focus of the project is the vector unit, we continued on to evaluating its performance.



(a) Scalar performance

(b) Vector performance

Figure 3: Vector and scalar execution time compared to scalar SMIPS core

5.3 Vector performance

We compared the scalar and vector versions of the vector benchmark suite. First, it is worth noting that the vector versions of the becnhmarks use a substantially smaller number of instruction for the same workload. The relative decrease in instruction count is smallest for `vvaddv` - 9x - and largest for `geometry` - 29x. This results in massively smaller instruction cache utilization, which we do not include in our results, since all benchmarks include an initial cache priming run before actual statistics are gathered.

For the test runs, the scalar versions were unsurprisingly run on the original scalar core. The results are shown in Figure 3b. The average performance improvement over the scalar core is 20%.

Even though the performance improvement isn't as drastic as initially expected, it is a substantial increase, especially taking into account that this scenario can be implemented without adding any more pure computational power. The vector pipeline uses the exact same 32-bit ALU that is used in the scalar pipeline and, given that the two units never execute simultaneously, it is possible to implement an architecture where the two units share the ALU (we didn't go along that route mainly because in Section 6 we explore a scenario where the two units do can execute at the same time).

Given the lack of additional computational power, this design offers an improvement exactly because it relieves some of the stress on the shared front-end by reducing the number of instructions executed. This is further confirmed by the behavior of the `multiplyv` benchmark, where we see a small performance loss. Upon closer inspection, `multiplyv` sustains and IPC really close to 1 on the scalar processor, so obviously it is not limited by the front end and wouldn't benefit from the smaller number of instructions.

	Slice registers	Slice LUTs
Scalar	5175 (7%)	8518 (12%)
Vector	5893 (8%)	11412 (16%)

Table 1: Device utilization on synthesized design

5.4 Synthesis results

In order to calculate execution times, we synthesized all iterations of our design. Area requirements for the final version are shown in Table 1. We can see that the overall overhead compared to the scalar core is ≈20%. However, this number doesn't account for the SCE-MI link which both designs share and which can account for a large fraction of the overall design area.

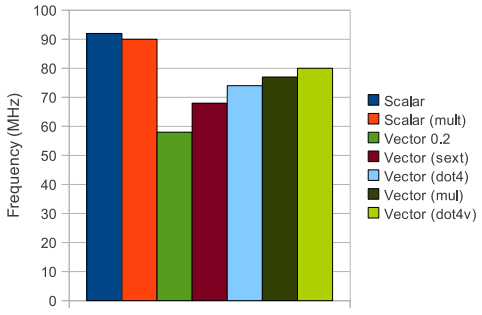


Figure 4: Refinement of clock frequency in different design revisions

Figure 4 shows the iterative improvements we made to decrease the clock cycle, compared to the maximum frequency for the scalar core. It is worth noting that the adding a combinational multiplier to the scalar core virtually didn't affect the critical path length. From this, we concluded that we don't need to implement a multicycle multiplier in the vector pipeline, as was our initial intention.

Initially, we had a significant decrease in maximum frequency (first green bar in Figure 4. We identified that the critical path was exacerbated by sign extending a 32-bit value to the 128-bit shared accumulator registers. Such sign extension required a larger fanout than the FPGA board could provide and a sequence

of buffers was generated on the critical path. We were seeing this issue because we were incorrectly sharing the accumulator register between the `swiz` and `dot4` instructions. Using a smaller accumulator for the dot product (which is naturally the longest path because it requires a multiply-add-accumulate sequence) fixed this issue.

The rest of the optimizations we made in order to get higher frequency were fairly straightforward. We identified the current critical paths and broke them up into more pipeline stages. This strategy allowed us to reach a maximum frequency of 80 Mhz, which is relatively close to the maximum we obtained for the scalar core (90 Mhz). The current critical path is in the datapath for the `dot4v` instruction and the overhead is associated with calculating the vector index before accessing the register file. We believe that, given more time, this overhead can be reduced with further pipelining.

6 Design exploration - non-blocking pipeline

```
add $1, $1, $15
lwc2 $v2, 0($1)
add $1, $1, $15
lwc2 $v3, 0($1)
add $1, $1, $15
lwc2 $v4, 0($1)
add $1, $1, $15
lwc2 $v5, 0($1)
```

(a) Mixing of instructions

```
loop:  addi $1, $1, -16
       lwc2 $v2,0($1)
       c2 mult $v1, $v2, $v3
       bge $1, $0, loop
```

(b) Short kernel

Figure 5: Code snippets that would benefit from non-blocking vector operations

6.1 Motivation

During the performance evaluation, we noticed that some benchmarks were hindered by the fact that the scalar pipeline stalls while long-latency vector instructions are executed. Two representative cases of such code are shown in Figures 5a and 5b. The code at Figure 5a is taken from the `mmul` benchmark. It consists of consecutive address calculations and loads from the respective locations. The bolded scalar instructions are blocked by the loads, each of which takes at least 4 cycles. This does not only stop the address calculations, but also prevents pipelining the 4 load instructions in the vector unit. In the second case (Figure 5b), taken from `vvaddv`, the kernel is relatively short and pipeline performance cannot be achieved for the vector unit. The common feature of these code sequences is the fact that the the blocking is artificially created - there is no direct dependency between the blocked scalar instruction and the vector one that is in execution.

6.2 Implementation

In a large fraction of the cases, such blocking can be removed without breaking the simple model of serial consistency. Since the only synchronization point of the two pipelines is main memory, blocking should not be removed for overlapping memory accesses. This can be achieved by maintaining a structure that keeps track of in-flight memory accesses from the vector core and checking against it when a scalar memory operation is about to be dispatched for execution. We chose to implement a much simpler model, where all scalar memory requests are blocked if the vector unit is busy.

The other instruction type that can break the non-blocking model are branches. Dispatching a mispredicted branch and clearing all pipes on detecting the misprediction can discard an old vector instruction from the correct execution path. We solved this issue by extending the simple epoch scheme to the vector unit

as well - prior to entering writeback, if a vector operation is older than the current epoch, it does not enter writeback. Since our front-end (maximum 2-3 stages) is shorter than the vector execution stage (minimum 4 stage), this ensures that the misprediction is resolved before the vector instruction reaches writeback.

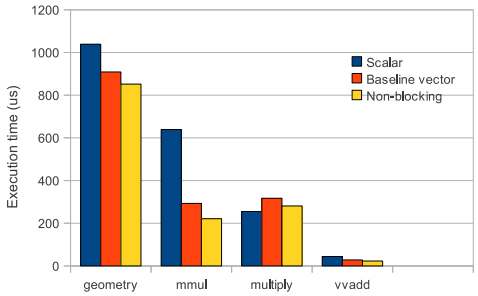


Figure 6: Vector performance with a non-blocking vector pipeline

6.3 Results

Results from evaluating our implementation of a non-blocking pipeline are shown in Figure 6. We can see that the average improvement over the scalar core is 30%, 10% better than the blocking implementation. This is almost a win-win situation because no additional hardware was generated - simply artificial constraints were removed. However, since the two pipelines can now execute simultaneously, sharing a single ALU is not possible for this configuration.

7 Design exploration - Branch prediction in decode stage

One of the ways we tried to do was to add a second level of branch prediction to the processor in the decode stage. This branch predictor would predict if the branch predictor in the instruction fetch stage actually made a good prediction. For this predictor, we tried first to use a predictor based on a method using neural nets.

The neural net method requires keeping a set of global history bits plus a set of local history bits for each line in the cache used for branch prediction. It also keeps a list of weights to associate with these bits (although, for the local history bits, there is only one set of weights, not a set of waits for each line in the cache.)

To predict whether a branch is taken or not, the history bits are turned into 1 or -1 depending if the past branch was taken or not and then a dot product is performed. If the result is more than 0, it predicts taken, otherwise it predicts not taken. The predictor also needs to be able to learn. When the predictor makes a wrong prediction, the weights that corresponds to the history bits that disagree with the prediction are adjusted. The implementation we tried to create used 8 bit signed integers for the weights.

An obvious other addition is to check to see if the instruction is a branch at all. If the instruction isn't a branch and the BTB in the first stage decided to branch, it is obviously wrong.

7.1 Motivation

Because we increased the pipeline length, the cost of a misbranch was also higher compared to the lab5 implementation. By putting another, more accurate predictor in the decode stage, we can cut down on the penalty of a misprediction by flushing the pipeline early.

We chose the neural net method because its cost in memory is linear with the number of weights used. This is good, since prediction algorithms seem to generally perform better when the amount of history information increases.

7.2 Results

Unfortunately, we had difficulty in writing a neural net branch predictor that bluespec would actually compile when used in our design, so we never got to see the effects it would have had on the simulation time. Even if it did work and was correct, it most likely would have been at the expense of clock speed. The predictor we were trying to simulate held 32 global history bits and 16 local history bits per line. This seems as though it might become the longest path in the architecture.

Even when we tested the decode branch predictor scheme with a simple branch predictor, things did not go so well. It's most likely that when we were trying to implement it, we created the possibility for a situation to arise where neither the decode rule or the exec rules can be fired because of something to do with the epoch registers. Unfortunately, we were unable to fix this in time and it was cut from the design.

References

[1] 6.375. SMIPS ISA documentation. `http://csg.csail.mit.edu/6.375/6_375_2010_www/handouts/other/smips-spec.pdf`.

[2] King, M., and Khan, A. SMIPS Multimedia Extensions. `http://csg.csail.mit.edu/6.375/6_375_2006_www/projects/group2-report.pdf`.

A ISA Extensions

Based on the initial format in [2]; modified to follow the scalar SMIPS format closer for ease in decoding.

31 26	25 21	20 16	15 11	10 6	5 0	
110010	rbase	vrdst	signed offset			`lwc2 vrdst, off(rbase)`
111010	rbase	vrsrc	signed offset			`swc2 vrsrc, off(rbase)`

010010	1	0001	vrdst	vrsrc1	vrsrc2	000000			`c2 addv vrdst, vrsrc1, vrsrc2`
010010	1	0010	vrdst	vrsrc1	vrsrc2	000000			`c2 mulv vrdst, vrsrc1, vrsrc2`

010010	1	0100	vrdst	vrsrc1	imm				`c2 addiuv vrdst, vrsrc1, imm`
010010	1	0101	vrdst	vrsrc1	imm				`c2 andiv vrdst, vrsrc1, imm`
010010	1	0110	vrdst	vrsrc1	imm				`c2 sllvv vrdst, vrsrc1, imm`
010010	1	0111	vrdst	vrsrc1	imm				`c2 srlvv vrdst, vrsrc1, imm`

010010	1	1011	vrdst	vrsrc1	vrsrc2	0	di	000		`c2 dot4 vrdst.di, vrsrc1, vrsrc2`
010010	1	1100	vrdst	vrsrc1	vrsrc2	0	di	d2	0	`c2 dot4v vrdst.di, vrsrc1, vrsrc2.i2`

010010	1	1001	vrdst	vrsrc1	vrsrc2	m	00000	`c2 seteqv vrdst, vrsrc1, vrsrc2, m`
010010	1	1000	vrdst	vrsrc1	vrsrc2	m	00000	`c2 setnev vrdst, vrsrc1, vrsrc2, m`
010010	1	1010	vrdst	vrsrc1	vrsrc2	m	00000	`c2 setltv vrdst, vrsrc1, vrsrc2, m`

010010	1	0011	vrdst	vrsrc	0	swizzle	00	`c2 swiz, vrdst, vrsrc, swizzle`

Table 2: Instruction format for vector extensions to SMIPS