Group 2

SMIPS Multimedia Extensions

Myron King
Asif Khan

Project Description

In this project, we designed and implemented a multimedia extension to the SMIPS ISA, concentrating primarily on issues surrounding the datapath and control logic. This extension is similar to Intel's MMX or SSE, both in its semantics and its utility.

Implementing an extension like this affects every aspect of the chip, and the optimization of its performance requires extensive work in the datapath, control logic, and memory. In order to keep the scope of this project within the allowable implementation period (half a semester), it was our original intention to concentrate on issues surrounding the datapath and control logic, spending only as much time on memory issues as was required to ensure satisfactory performance. This later proved to be a little bit too optimistic, and we found ourselves up to our elbows in the data cache before the first week was up! The extension is executed on a SIMD coprocessor which is tightly coupled with the control processor except in cases where the instructions being executed require no interaction between the two processors, be it the result of data dependency or microarchitectural implementation detail.

Further architectural exploration involved the addition of predicated writes, lengthening the vector exposed to the ISA while keeping the same number of lanes on the coprocessor, and the addition of a control register which allows the programmer to specify the vector length as a multiple of four (with an upper limit of thirty-two) and use strip-mining techniques. Exploration into selective ALU implementation and its effect on critical path length and clock period was also undertaken.

The motivation for this project comes from the realization that certain types of programs, many of which are typical to graphical manipulation, physical simulations, and scientific computing in general, contain kernels which are inherently parallel. These experience a significant speedup running on SIMD machines from the addition of lanes, the reduction in instruction count per unit of data throughput, and from the fact that SIMD flow control uses predication which allows us to avoid dynamic branching, thereby improving instruction cache behavior and possible wasted cycles on mispredicted instructions. As demonstrated by the entire vector computing industry, the graphics processor industry, and Intel's success with MMX and then SSE, this is not news, but it has made for an interesting project.

Architectural Sketch

The MIPS architecture provides a convenient way to encapsulate new instructions through the use of the COP2 interface and there are enough bits in the SMIPS ISA to encode a useful number of multimedia operations. The baseline implementation uses a 128-bit wide SIMD pipeline including a separate register file (with 128-bit wide registers) for use by this SIMD engine. The SWC2 and LWC2 instructions move 128 bits of data between the system memory and the COP2 register file, while the MFC2 and MTC2 instructions move 32 bits data between the control processor and COP2 register files. In further architectural

experiments, with the exception of these explicit inter-processor transfer instructions, the size of vectors transferred or operated on by all the cop2 instructions is equal to the vector length exposed to the ISA.

The SIMD unit contains only an execute and a writeback stage; all instructions are decoded and dispatched by the control processor. The coprocessor and the scalar unit use the same clock, though one might imagine this need not always be the case. At the head of the pipeline, there is a single fetch and decode stage which determines whether the current instruction should be executed on COP2 or the control processor. The two processors are decoupled through the use of fifo's so that both processors should be able to maintain a more than 50% throughput (at the very least) due to the bottleneck resulting from the shared instruction fetch and decode logic. Both processors then remove queued instructions from their respective fifo's, decode the instructions, fetch the data from the register file, execute the instructions, and write back the results independently of each other. Apart from data dependency, the only other cases where the two processors need to synchronize are LWC2, SWC2, MTC2, and MFC2, since these require resources from both the processors and failure to synchronize would result in the violation of sequential consistency. To simplify the picture, two steps are taken:

(1) unaligned memory references are prohibited, and

(2) there is no control flow instruction executed on the coprocessor.

Instruction Semantics

Binary Instructions

The implemented binary opcodes include addv and mulv. First, let's establish some nomenclature: If we have a COP2 register r1, which is 128 bits wide, then r1.comp0 refers to bits 0-31, r1.comp1 to bits 32-63, r1.comp2 to bits 64-95, and r1.comp3 to bits 96-127. Using this nomenclature, the semantics of the "c2 addv r1, r2, r3" instruction are as follows.

r1.comp0 = r2.comp0 + r3.comp0
r1.comp1 = r2.comp1 + r3.comp1
r1.comp2 = r2.comp2 + r3.comp2
r1.comp3 = r2.comp3 + r3.comp3

The semantics of "c2 mulv r1, r2, r3" are identical, other than replacing "+" with "*". In the exploratory architectures, the semantics of this inherently parallel instruction are independent of the vector length exposed to the ISA. For an arbitrarily sized vector of length N (DWORDs), the value written to r1.compM for all M < N is r2.compM + r3.compM.

The dispatch stage issues these instructions to the instQ_cop2 only. In the baseline implementation, the execute stage reads two 128-bit registers using the 8 read ports of the cop2

register file. The appropriate arithmetic operation is performed on 32-bit chunks of data, and four 32-bit results are enqueued in the wbQ_cop2. In the writeback stage these results are written to the cop2 register file using the 4 write ports.

Immediate Binary Instructions

These opcodes include andiv and addiuv. They execute in the same manner as the addv and mulv instructions, with the only exception that source2 is replaced by an immediate value that is encoded in the instruction. Like the Binary instructions, these are dispatched to the coprocessor exclusively. Immediate Binary Instructions adapt to different vector lengths much like their simple binary counterparts.

The Swizzle Instruction

This instruction performs the re-ordering of the four 32-bit components of a 128-bit cop2 register. It is important to note that this is a source swizzle, meaning that the read of the source register takes place under the swizzle and every component of the destination register is written once. This instruction is also dispatched to the coprocessor exclusively.

The permute comp0 in the swiz instruction indicates that source.comp0 will be moved to dest.compN where N corresponds to permute comp0 (bits 9-8). Similarly, source.comp1 will be moved to dest.compM where M corresponds to bits 7-6, and so on and so forth.

For example, the instruction "swiz r1, r2, 3210" will perform the following actions:

r1.comp0 = r2.comp3
r1.comp1 = r2.comp2
r1.comp2 = r2.comp1
r1.comp3 = r2.comp0

Unlike the other binary arithmetic instructions, this instruction operates on primitives of four DWORDs. For this reason, all SIMD vector lengths (in DWORDs) exposed to the ISA are restricted to multiples of four. This way, the swizzle instruction repeats itself along the vector every four DWORDs.

SET Instructions

The opcodes include seteqv, setnev, and setltv. If bit 4 of a set instruction is set, the mask bit of the corresponding lane will be set to the result (0 or 1) of the comparison. Otherwise, the instruction writes 0 or 1 to a component of the destination register, based on a comparison of corresponding components from the two sources. These instructions are also dispatched to the coprocessor exclusively. Like the binary instructions described above, the Set instructions naturally extend to any vector length.

Load and Store Instructions

The Opcodes LWC2 and SWC2 transfer 128 bit chunks of data between the coprocessor and memory. When the dispatch stage encounters the "LWC2" instruction, it issues the instruction to the instQ, and issues a LWC2_tag to the instQ_cop2. The control processor then issues a 128-bit memory request for the provided address, and discards the instruction in its writeback stage. When the coprocessor encounters the LWC2_tag instruction, the execute stage passes the instruction down to the wbQ_cop2. The writeback stage then collects the 128-bit memory response from the dataRespQ and writes it back to the appropriate cop2 register. Although it is possible for the control processor to write directly into the coprocessor's register file, we use the LWC2_tag to ensure sequential consistency.

When the dispatch stage encounters the "SWC2" instruction, the instruction is issued to the control processor, and SWC2_tag is issued to the coprocessor. The coprocessor enqueues the 128-bit data in the c2sQ. The control processor stalls the SWC2 until it receives the data from the c2sQ. It then enqueues a 128-bit store request in the dataReqQ. The writeback stage of the coprocessor then dequeues the store responses from the dataRespQ.

The data cache differentiates between the load and store responses for the control processor and those for the coprocessor through the use of tags. As the ISA vector length grows, the chunks of memory transferred by LWC2 and SWC2 grow too.

Explicit Data Transfer Instructions

The MTC2 and MFC2 (CTC2 and CFC2, alternatively) opcodes function in a similar manner, the dispatch stage always issuing the instruction to the control processor and the coprocessor. By using c2sQ and s2cQ for communication between the processors, and normal hazard checking within each processor to avoid RAW hazards, this scheme guarantees sequential consistency. Transferring to the coprocessor will always write 1 DWORD of data to comp0 of the specified destination coprocessor register, regardless of ISA vector length. Similarly, transferring from the coprocessor will always read 1 DWORD of data from comp0 of the specified source. As the ISA vector length changes, different numbers of cop2 microarchitectural registers are used to implement one ISA register. The cop2 register specified in any of these instructions always refers to a specific microarchitectural register. This is done for ease of implementation and is one of the more unseemly aspects of this multimedia extension.

The Dot4 Instruction

In the baseline implementation, this opcode computes a dot product of the two vectors specified in the instruction as the source registers. The result is written to the component of the destination register indicated by the destination swizzle. Like the Swizzle instruction, this operates in primitives of four DWORDS. Similar to the explicit data transfer instructions, for all ISA vector lengths greater than four the second source of this instruction refers to a

microarchitectural, not an ISA register. In this case, the decision was made as a short-cut to improve code density and data-packing in the geometry benchmarks. This short-cut actually lends itself perfectly to graphics applications since it is easy to see how loading the transformation matrix into a single ISA register and being able to access each row explicitly would be helpful. As the vector length of the ISA grows by multiples of four, the destination swizzle indicates the destination position in each 4-DWORD sub-component of the destination register.

The ADDH Instruction

Like Swizzle and Dot4, this instructions operates on 4-DWORD primitives, adding all four components of the source register and writing them to the component of the destination register specified by the destination swizzle. As the ISA vector grows, the semantics of the destination swizzle change in as described for the Dot4 instruction.

The Write Mask

All writes to a cop2 GPR by ALU instructions executed on the coprocessor are predicated by the mask bits corresponding to that particular lane. These mask bits can be used to implement the SIMD flow-control. The write mask is large enough to guarantee one bit for each DWORD in the largest exposed ISA vector length.

Organization of Bluespec Modules, Rules and Interfaces

All modules represented in the architectural diagram mkProc, mkCoProc, mkInstCacheBlocking, mkDataCacheBlocking, and mkMemArb are submodules of mkCore, much like the original SMIPSv2 implementation from lab3. Apart from the requirements of statistics gathering infrastructure, the only Interface methods in these modules are FIFO's used for transferring data and instructions. All of the FIFO's are connected using the mkConnection module provided by the Bluespec library. For simplicity the diagram represents these connections as FIFO's between the modules. The core module and the instruction cache remain largely unchanged from lab3, so only the implementation of the coprocessor, the control processor, and the data cache will be discussed below. The baseline coprocessor implementation a register file of eight 4-DWORD registers.

The mkProc Module (implements IProc)

```
interface IProc;

  // Interface for testrig
  interface Put#(Bit#(8)) testrig_fromhost;
  interface Get#(Bit#(8)) testrig_tohost;
```

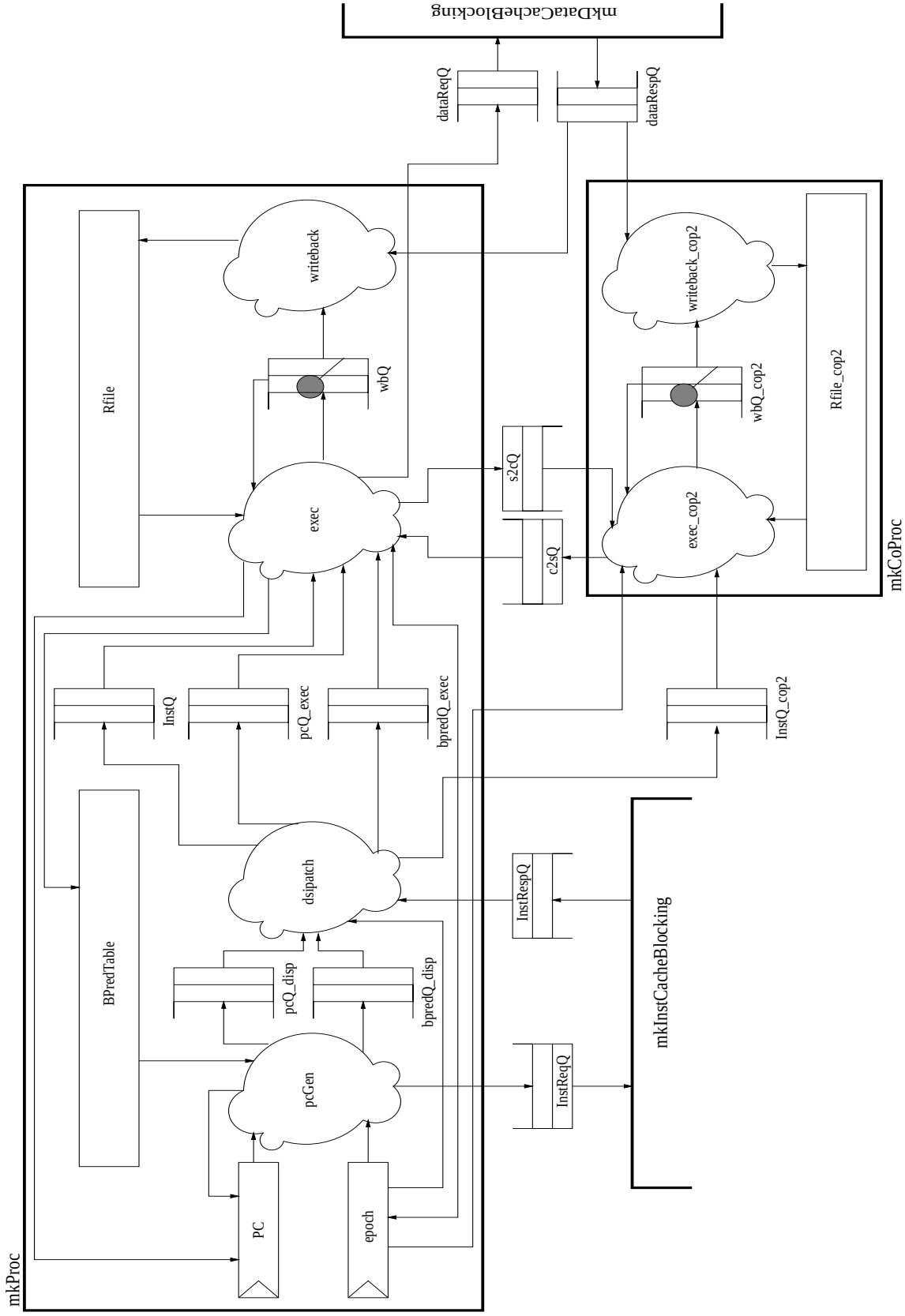

Figure 1: Microarchitecture

```
  // Interface between control processor and caches
  interface Client#(DataReq,DataResp) dmem_client;
  interface Client#(InstReq,InstResp) imem_client;

  // Interface for enabling/disabling statistics in other
  // submodules of mkCore
  interface Get#(Bool) statsEn_get;

  // Interface for Data transfer between control processor
  // and coprocessor
  interface Get#(Data) data_to_cop2;
  interface Put#(Data_cop2) data_from_cop2;

  // Interface for sending instructions to the coprocessor
  interface Get#(InstResp) inst_to_cop2;

  // Interface to send epoch to coprocessor in the event of a
  // mispredicted branch.
  interface Get#(Bit#(8))  epoch_to_cop2;

endinterface
```

The pcGen Rule

The semantics of this rule have been modified slightly from the SMIPSv2 implementation of lab3. In addition to issuing a load request to the instruction cache, it also searches the branch prediction table for pc+4. If a valid entry is found, pc is written with this value, if not, pc gets pc+4. The pc register is an EHR register. The queue bPred_disp FIFO then gets a TRUE or FALSE depending on how the branch was predicted. This value will be used in the exec rule when deciding whether to increment the epoch. Instruction requests are tagged with the epoch for use by the dispatch and execute phases.

The dispatch Rule

This rule dequeues the load responses from the instruction cache and dispatches them to either the control processor (within the same module) or the coprocessor (different module) or both. To see which instructions are dispatched to which processor, see the section on Instruction Semantics. Since it is possible for mispredicted instructions to enter this phase of the pipeline, the dispatch has a counterpart rule named throwaway which fires when the epoch register does not match the dequeued instruction's tag.

The exec Rule

This rule executes the instructions dispatched to it, as well as evaluates the success of branch prediction in the pcGen stage. It updates the epoch register as well as the branch

prediction table if a misprediction occurs. The execution of all the instructions inherited from lab3 (control processor only) remains the same, but the new instructions involving interaction with the coprocessor have more wide-ranging side-effects. See the Instruction Semantics section for more detail on this. The exec rule also has a counterpart called discard in the same stage of the pipeline which discards mispredicted instructions.

The writeback Rule

This rule functions exactly as it did on the SMIPSv2 implementation for lab3. It dequeues the wbQ and writes the data into the appropriate register of the register file. In case of load and store instructions it receives the load and store responses from the data cache and dequeues the dataRespQ.

Firing order: writeback < exec < pcGen

The mkCoProc Module (implements ICoProc)

```
interface ICoProc;

  // Interface to receive load responses from the data cache
  interface Put#(DataResp_cop2) dcache_data_put;

  // Interface for Data transfer between control processor
  // and coprocessor
  interface Get#(Data_cop2) data_to_proc;
  interface Put#(Data) data_from_proc;

  // Interface for receiving instructions from the control processor
  interface Put#(InstResp) inst_from_proc;

  // Interface for receiving the epoch from the control processor
  // to be used to detect mispredicted branches.
  interface Put#(Bit#(8))  epoch_from_proc;

  // set statsEn register for statistics gathering
  interface Put#(Bool) statsEn_put;

endinterface
```

The exec_cop2 Rule

This rule is identical to its scalar counterpart, including the corresponding discard rule called discard_cop2 in the same phase of the pipeline to discard mispredicted instructions. The exact behavior of the instructions executed in this rule and the rule's interaction with the control processor are discussed in detail in the Instruction Semantics section.

The writeback_cop2 Rule

This rule is more or less the same as its scalar counterpart. It dequeues the wbQ_cop2 and writes the 128-bit data into the appropriate register of the register file. In case of lwc2 and swc2 instructions it receives the load and store responses from the data cache and dequeues the dataRespQ.

Firing order: writeback_cop2 < exec_cop2

The mkDataCacheBlocking Module (implements IDCache)

```
interface IDCache#( type req_t, type resp_t );

  // Interface from processor to cache
  interface Server#(req_t,resp_t) proc_server;

  // Interface from coprocessor to cache
  interface Get#(DataResp_cop2) cop2_data_get;

  // Interface from cache to main memory
  interface Client#(MainMemReq,MainMemResp) mmem_client;

  // Interface for enabling/disabling statistics
  interface Put#(Bool) statsEn_put;

endinterface
```

This cache uses an in-order single address FIFO to ensure sequential consistency. All load and store requests are issued by the control processor. A tag attached to the request directs the response to the data response queue of either the control processor or the coprocessor. Apart from this, the things which differentiate this cache from the one used by SMIPSv2 implementation for lab3 are wider memory bandwidth (128-bits) and more cache lines to support single-cycle 128-bit memory accesses.

Implementation notes

To begin with, we implemented the coprocessor instructions in the same module as the control processor, this led to abysmal compile times along with all the other ills associated with large unmodularized lumps of code. Moreover, in the initial implementation memory references required four cycles for all memory transfers to and from the coprocessor, i.e. 32 bits per cycle.

So, we widened the memory ports, allowing 128-bit loads and stores from the coprocessor. At some point we got frustrated with unwieldy lumps of code, and decided to modularize our design.

The last additions were a branch predictor, a write-mask (and associated set instructions) as well as more than twice the number of instruction than originally planned, when we realized we would need them to run any meaningful benchmarks.

The branch predictor uses a simple direct-mapped branch-target-buffer (BTB). It contains an 8-entry table where each entry in the table is a <pc+4,target> pair. It only predicts branch instructions.

Hazard detection occurs in the execute stage of the pipeline for both the control processor and the coprocessor. Searchable fifo's between the execute and writeback stages allow the checking of pending register-writes, thus stalling when a RAW hazard is detected. With the addition of the coprocessor, the only necessary additions to the interlock logic needed to preserve sequential consistency, involve the transfer of data between the coprocessor and memory, and the coprocessor and the control processor. By executing these instructions on both the processors, and enforcing synchronization in the execution phase, we keep the two processors in sync only when necessary and allow the two processors to progress independently if there is no chance of memory incoherency or if there is no violation of sequential consistency. The microarchitecture achieves memory coherence through the use of a blocking cache, a single read-port, and an in-order address port to memory.

The datapath, control logic, and modularization are illustrated in Figure1.

Please refer to group2/readme.txt for instructions on building the simulator and running the tests and benchmarks, and generating the Bluespec scheduling reports.

Tests and Benchmarks

The tool pre-asm.pl has been written and placed in the local test directory. It has also been integrated into the tool chain, so all local tests have the "c2" macro expanded prior to assembly by the smips-assembly tool. All unit tests and benchmarks written in assembly are compiled using this process.

The more sophisticated benchmarks are written in C, while their kernels (the code which actually benefits from using the SIMD unit) are written in assembly, assembled using pre-asm.pl and smips-assembler, and linked to the other object files compiled by smips-gcc.

We have unit tests written for all the cop2 instructions and functionality (mask bits etc.). Instructions were added after analysis of our benchmarks, thus determining which ones would be required.

For the initial benchmarks, we ported the multiply (using shift and add) and vvadd benchmarks to the coprocessor to demonstrate and quantify the speedup gained by using the coprocessor to perform inherently parallel tasks. A more sophisticated benchmark consists of a kernel which performs a spacial transformation on a wire-frame model and finally executes backface culling of occluded geometry. The profile of this kernel is a long series of matrix multiplies, which has been implemented using dot products, ended by the predicated output of the geometry and is typical of GPU benchmarks.

Architectural Explorations

The Branch Predictor

We experimented with four and eight entry branch prediction tables for all our different architectures. The improvement in IPCs after implementing the branch-predictor were remarkable. Doubling the size of the table from four to eight produced yet another significant improvement in the IPC of the multiply benchmark. The improvements were similar to those seen in lab3, so we decided to include this feature as part of the baseline architecture.

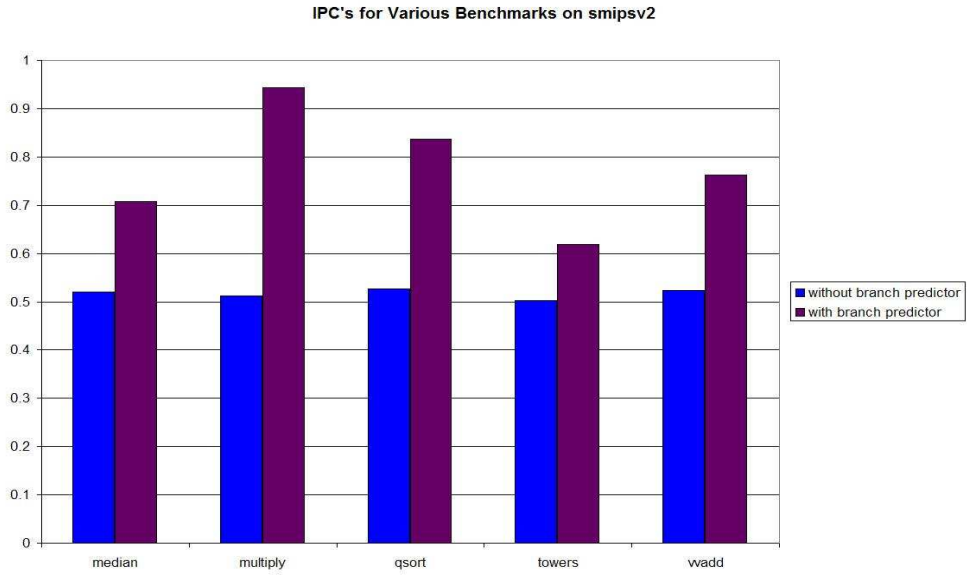

Figure 2: IPC gains with branch prediction

Figure 2 shows the improvement in performance gained through the addition of a branch predictor. Since the branch predictor has a negligible effect on the clock period and chip size, it is obviously a win for all cases tested.

16-DWORD Vectors

We lengthened the vector size exposed to the ISA, while keeping the same 4 lanes in our coprocessor. The changes to the baseline implementation were non-trivial and involved the addition of new rules to both the control processor and the coprocessor as well as a more sophisticated hazard checking mechanism. In order to support the larger vectors and retain the same number of ISA registers, the register file needed to be enlarged to contain 24 4-DWORD registers. This alone accounts for most of the increase in size and is a trend for all the exploratory architectures.

In the control processor, only the exec rule required changes. All exclusive control processor instruction still require one cycle in the execute phase and one in writeback. Since the explicit data transfer instructions move 1 DWORD of data regardless of ISA vector length, the execution of these remained unchanged as well. As discussed in the section on instruction semantics, the LWC2 and SWC2 instructions execute simultaneously on both the control and the coprocessor. Because of this, new rules (exec_lwc2 and exec_swc2) were added to the mkProc module to add extra stages for LWC2 and SWC2 exclusively. When the exec rule first sees an LWC2 instruction, it enqueues a data request for the first 128 bits and sets a state-register exec_lwc2_can_fire to true. The predicate of exec has been changed so it can only fire when this register is set to zero. This flag allows exec_lwc2 to fire and a counter ensures it only fires 3 more times, enqueuing 3 requests for the remaining 384 bits of data. The execution of SWC2 in the control processor is almost identical, except instead of enqueuing load requests, it dequeues data from c2sQ and enqueues appropriate tagged store requests. All store and load responses are handled by the coprocessor so neither of these instructions enter the writeback phase of the control processor.

In the coprocessor, the exec_cop2 rule was broken up into exec_cop2 and exec_cop2_special. The rule exec_cop2_special executes all the cop2 instructions which require only one stage (cycle) to complete. These are the explicit data transfer instructions. All the remaining opcodes are executed by exec_cop2, which fires four times (adds four stages) for each 4 DWORD quarter of the 16 DWORD ISA vector. A trip counter causes the instruction to be dequeued from the cop2_instQ once the last quarter has been executed.

Apart from changing the execution phases of both the processors, the other major change in this experimental architecture involved the discarding of mis-predicted branches. The single epoch scheme imported from SMIPSv2 falls apart since coprocessor instructions take multiple cycles to execute and during that execution, the control processor runs ahead, executes an independent branch instruction, increments the epoch register and causes a valid instruction in the process of being executed to be thrown away in error. The fact that the control processor runs ahead on independent scalar instructions is a desirable side-effect of decoupling the two processors, so in order to keep that, we needed to add a second epoch register which is incremented by the dispatch rule in the control processor every time a branch instruction is dispatched. A corresponding second tag is added to instructions dispatched to the coprocessor. If the tag of an instruction being executed on

the control processor fails the original epoch test, it will be discarded only if the second tag is greater than the second epoch register, indicating that it was dispatched AFTER the critical branch instruction. This feature required the addition of an interface method to both IProc as well as ICoProc for the transfer of the second epoch value.

Variable Length Vectors

A control register was added to hold a value specifying the length of the ISA vector. The programmer can set this register using the CTC2 instruction. This is really just a more generalized version of the 16-DWORD architecture discussed above, allowing us to experiment with multiple vector lengths on the same microarchitecture. It is conceivable that efficient code could be written to make use of multiple vector lengths in a particular stream of computation (doing strip-mining for example), but our benchmarks didn't get that sophisticated. Further adding to the size of this microarchitecture, the register file was expanded to 32 4-DWORD registers. Even with a register file of this size, we were not able to implement some of the benchmarks which required more registers at the longest vector lengths.

Apart from some additional state required when you don't have a hard-coded vector length, the Bluespec (and hence, the microprotocol) for this is quite similar to its 16-DWORD hard-coded counterpart.

ALU changes for clock speed improvement

We recognized that the dot4 instruction is the longest combinational path, contributing to a slow-down in clock speed. In an attempt to increase the clock speed, we removed the dot product, adding instead the addh (horizontal add) instruction which when combined with a mulv instruction, accomplished the same arithmetic, except over twice the number of cycles. This contained the same number of cop2 GPRs as the baseline design.

This change required only trivial changes to the exec_cop2 rule in the coprocessor.

Numbers, Graphs, and Analysis

Table 1 and Table 2 demonstrate the differences in area and effective clock period among the various architectural variations. The increase in size from SMIPSv2 to the Baseline Coprocessor implementation is quite predictable as we are adding an additional module with a significantly sized register file. The increase in size as the vector length increases (from the 16-length to the variable-length with a max of 32) is due primarily to the larger register files. The drop in effective clock from the baseline to the alternate ALU implementation reported by the Synthesis tool is due to the fact that the critical path no longer contains the dot4 instruction. That this is not reflected in the numbers generated by Encounter

Table 1: Numbers Generated by Synthesis Tool

Microarchitecture	Area (abstract units)	Effective Clock Period (ns)
smipsv2 implementation	28,837	4.26
baseline implementation	87,413	5.50
16-dword vectors implementation	147,652	5.50
variable length vectors implementation	172,251	5.84
alternate ALU implementation	104,651	5.00

Table 2: Numbers Generated by Encounter

Microarchitecture	Area (abstract units)	Effective Clock Period (ns)
smipsv2 implementation	464,849	7.174
baseline implementation	1,415,025	9.453
16-dword vectors implementation	2,466,711	14.520
variable length vectors implementation	2,799,400	14.889
alternate ALU implementation	1,708,809	10.782

is perplexing and due most likely to the fact that we weren't able to provide the correct directives. With more work this could likely be fixed.

Table 3 shows that the instruction counts for benchmarks which do not use the coprocessor remain unchanged over all the architectural variations, which comes as no surprise. For those benchmarks which do use Cop2 instructions, the instruction count decreases as the vector length increases since fewer vector instructions are required to do the same amount of work. This puts less strain on the instruction cache and should increase the performance of this module. Table 4 shows similar trends in cycle counts for the same reasons. If a lower cycle count outweighs the corresponding increase in clock period, the benchmark will run in less time.

The trends in IPC's shown in Table 5 are not unexpected. As the vector length increases, more cycles will be required to execute one instruction, therefore lowering this particular ratio though not necessarily the overall data thruput of the pipeline. The run-times shown in Table 6 are more easily interpreted graphically.

Figure 3 compares the IPC's for each benchmark among all the different architectural configurations. Perhaps the most important graph in this section, Figure 4 shows that using a SIMD coprocessor is actually a win for kernels which are inherently parallel.

Figure 5 shows an obvious sweet-spot, indicating that with the existing memory configuration, the variable-length microarchitecture is optimally configured with a vector length of either four or twelve. Perhaps the second most important graph in this section is Figure 7, which shows that without changing the memory configuration, increasing the vector length is useless.

Table 3: Instruction Counts For Selected Benchmarks

Microarchitecture	multiply	vvadd	vvaddv	multiplyv	geometry
smipsv2	21194	3012	-	-	-
Baseline	21194	3012	690	6690	6019
16-dword Vectors	21194	3012	186	1617	1516
Variable Length Vectors (8-dword)	21194	3012	359	3221	3019
Variable Length Vectors (12-dword)	21194	3012	242	2153	2014
Variable Length Vectors (16-dword)	21194	3012	188	1619	1518
Variable Length Vectors (20-dword)	21194	3012	152	1352	1218
Variable Length Vectors (24-dword)	21194	3012	134	-	1023
Variable Length Vectors (28-dword)	21194	3012	116	-	-
Variable Length Vectors (32-dword)	21194	3012	98	-	-
Alternate ALU	21194	3012	690	6690	7619

Table 4: Cycle Counts For Selected Benchmarks

Microarchitecture	multiply	vvadd	vvaddv	multiplyv	geometry
smipsv2	22452	3946	-	-	-
Baseline	23212	3635	867	6875	11663
16-dword Vectors	23212	3635	528	4815	11079
Variable Length Vectors (8-dword)	23212	3635	619	4876	11066
Variable Length Vectors (12-dword)	23212	3635	534	4829	10982
Variable Length Vectors (16-dword)	23212	3635	546	4817	10981
Variable Length Vectors (20-dword)	23212	3635	498	5009	10969
Variable Length Vectors (24-dword)	23212	3635	638	-	11018
Variable Length Vectors (28-dword)	23212	3635	600	-	-
Variable Length Vectors (32-dword)	23212	3635	411	-	-
Alternate ALU	23212	3635	867	6875	13263

Table 5: IPCs for Selected Benchmarks

Microarchitecture	multiply	vvadd	vvaddv	multiplyv	geometry
smipsv2	0.9440	0.7633	-	-	-
Baseline	0.9131	0.8286	0.7958	0.9731	0.5161
16-dword Vectors	0.9131	0.8286	0.3523	0.3358	0.1368
Variable Length Vectors (8-dword)	0.9131	0.8286	0.5800	0.6606	0.2728
Variable Length Vectors (12-dword)	0.9131	0.8286	0.3443	0.4458	0.1834
Variable Length Vectors (16-dword)	0.9131	0.8286	0.4532	0.3361	0.1382
Variable Length Vectors (20-dword)	0.9131	0.8286	0.3052	0.2699	0.1110
Variable Length Vectors (24-dword)	0.9131	0.8286	0.2100	-	0.0928
Variable Length Vectors (28-dword)	0.9131	0.8286	0.1933	-	-
Variable Length Vectors (32-dword)	0.9131	0.8286	0.2384	-	-
Alternate ALU	0.9131	0.8286	0.7958	0.9731	0.5745

Table 6: Runtimes for Selected Benchmarks

Microarchitecture	multiply	vvadd	vvaddv	multiplyv	geometry
smipsv2	161070.6	28308.6	-	-	-
Baseline	219423.0	34361.7	8195.8	64989.4	110250.3
16-dword Vectors	337038.2	52780.2	7666.6	69913.8	160867.1
Variable Length Vectors (8-dword)	345603.5	54121.5	9216.3	72598.8	164761.7
Variable Length Vectors (12-dword	250271.8	39192.6	5757.6	52066.3	118407.9
Variable Length Vectors (16-dword	345603.5	54121.5	8129.4	71720.3	163496.1
Variable Length Vectors (20-dword	345603.5	54121.5	7414.7	74579.0	163317.4
Variable Length Vectors (24-dword	345603.5	54121.5	9499.2	-	164047.0
Variable Length Vectors (28-dword	345603.5	54121.5	8933.4	-	-
Variable Length Vectors (32-dword	345603.5	54121.5	6119.4	-	-
Alternate ALU	250271.8	39192.6	9348.0	74126.3	143001.7

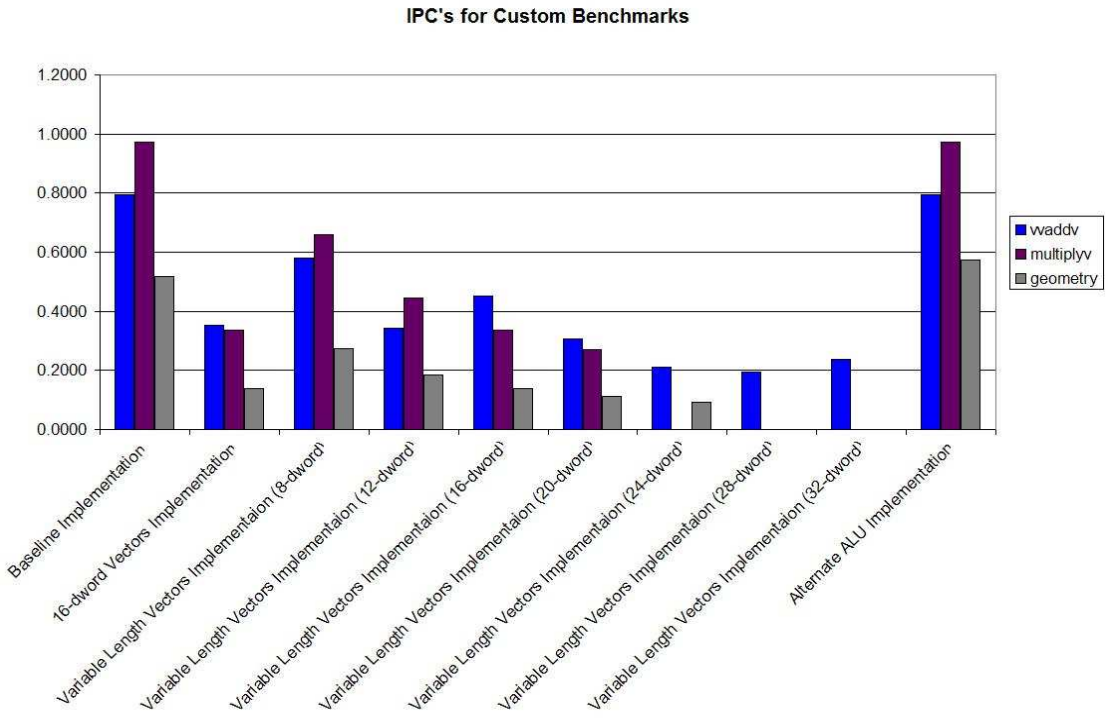

Figure 3: IPCs

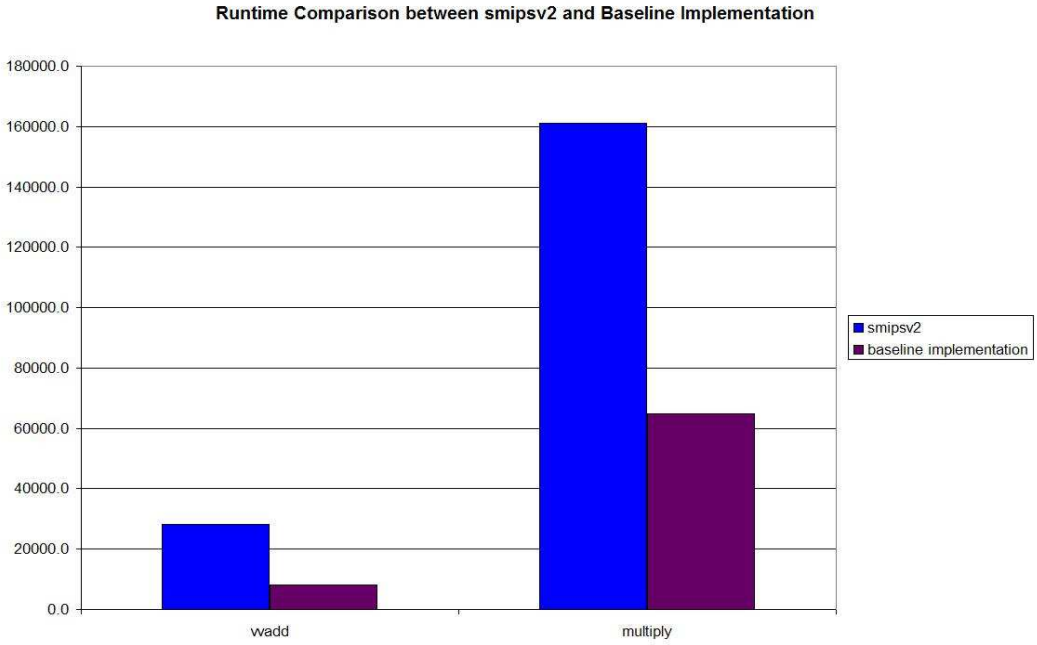

Figure 4: Baseline vs. SMIPSv2

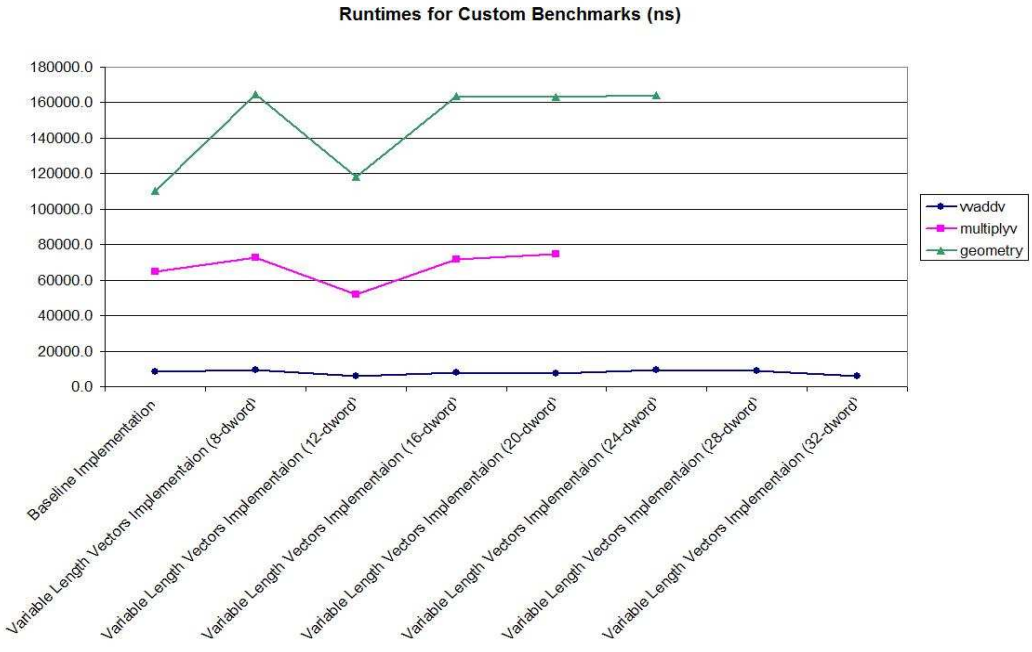

Figure 5: Runtimes

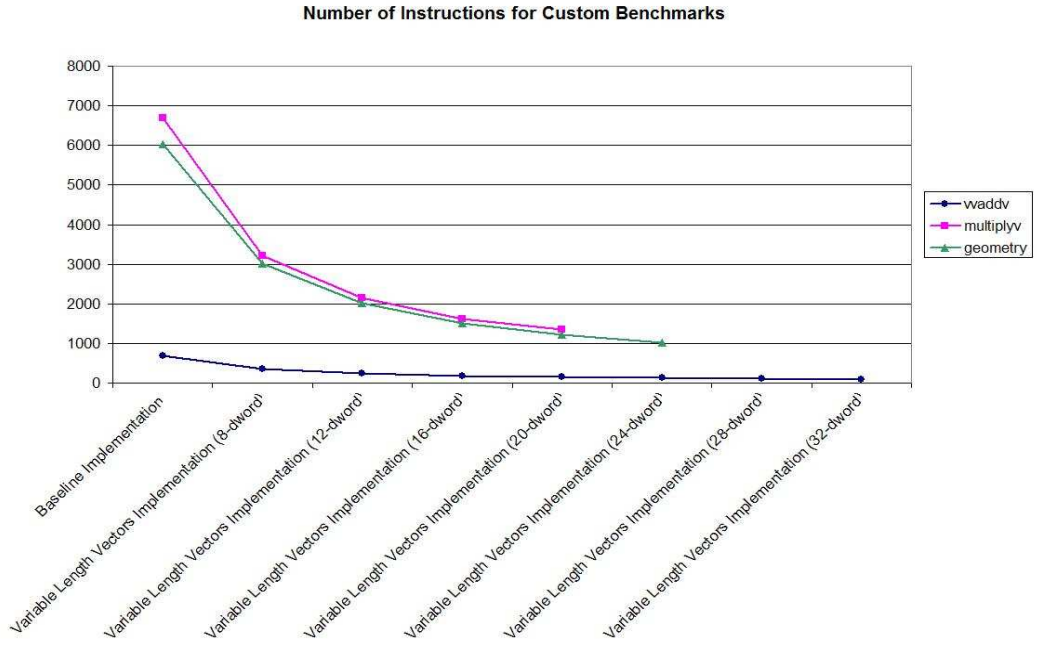

Figure 6: Number of Instructions

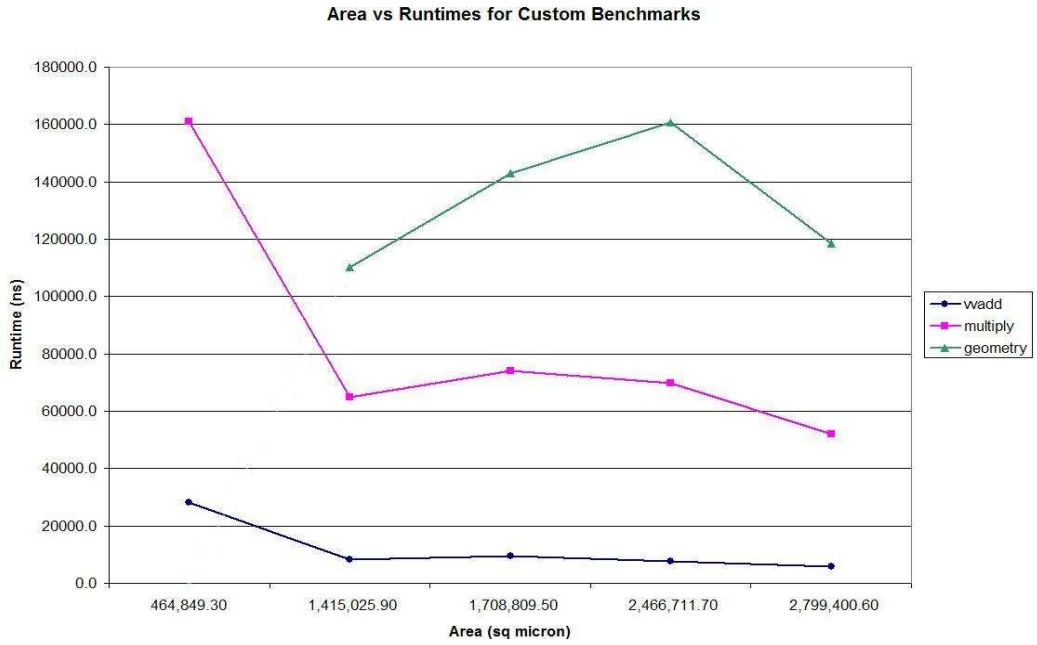

Figure 7: Area vs. Runtimes

Conclusion

The baseline implementation is a win, but this comes as no surprise. If we hadn't expected this, we would never have attempted this project. We also expected the exploratory architectures to be a greater success, but there a number of possible explanations as to why they didn't perform as we had hoped they would.

As the ISA vector length increases, we require an increasingly larger register file to support the same number of ISA registers. At some point the coprocessor register file became so large that we decided it was not worth enlarging it further. This put greater register pressure on the benchmarks, even to the point where register spilling scuttled any performance gain completely. Another problem we ran into was the memory bottleneck. Without widening the memory bandwidth between the cache and the processors, Lengthening the ISA vector can only do so much before memory inhibits any further performance gains.

Given more time, we would have liked to do more with floor-planning and other aspects of physical design. This failing is demonstrated by the numbers generated by Encounter. Also a few more benchmarks would have been helpful. To produce better performance numbers with the exploratory architectures, we would have liked to add registers, enabling us to implement software pipelining, thus mitigating the impact of the memory bottleneck.

Appendix A: Instruction Encoding

There are 25 bits to encode the coprocessor function (copfunc). The tool pre-asm.pl assembles the c2 macros to the hexadecimal representation required by the smips assembler. The highest 5 bits will contain the opcode while the encoding of the lower 20 bits will depend on the instruction. Currently implemented cop2 opcodes are:

addv = 1
mulv = 2
swiz = 3
addiuv = 4
andiv = 5
sllv = 6
srlv = 7
setnev = 8
seteqv = 9
setltv = 10
dot4 = 11
addh = 12

Encoding of the addv and mulv instructions:

24-20 - opcode
19-15 - destination register
14-10 - source1
09-05 - source2
04-00 - unused

Encoding of the swizzle instruction:

24-20 - opcode
19-15 - destination register
14-10 - source
09-08 - permute comp0
07-06 - permute comp1
05-04 - permute comp2
03-02 - permute comp3
01-00 - unused

Encoding of the addiuv, addiv, andiv, luiv, sllv and srlv instructions:

24-20 - opcode
19-15 - destination register
14-10 - source

09-00 - immediate operand

Encoding of the seteqv, setnev and setltv instructions:

24-20 - opcode
19-15 - destination register
14-10 - source1
09-05 - source2
04-04 - update mask bits
03-00 - unused

Encoding of the dot4 instruction:

24-20 - opcode
19-15 - destination register
14-10 - source1
09-05 - source2
04-03 - dst_swiz
02-00 - unused

Encoding of the addh instruction:

24-20 - opcode
19-15 - destination register
14-10 - source
09-08 - dst_swiz
07-00 - unused