Extended Split-Issue: Enabling Flexibility in the Hardware Implementation of NUAL VLIW DSPs

Bharath Iyer*, Sadagopan Srinivasan, and Bruce Jacob
Dept. of Electrical & Computer Engineering
University of Maryland, College Park
{iyerb,sgopan,blj}@eng.umd.edu

http://www.ece.umd.edu/~blj/embedded/

ABSTRACT

VLIW architecture based DSPs have become widespread due to the combined benefits of simple hardware and compiler-extracted instruction-level parallelism. However, the VLIW instruction set architecture and its hardware implementation are tightly coupled, especially so for Non-Unit Assumed Latency (NUAL) VLIWs. The problem of object code compatibility across processors having different numbers of functional units or hardware latencies has been the Achilles' heel of this otherwise powerful architecture. In this paper, we propose eXtended Split-Issue (XSI), a novel mechanism that breaks the instruction packet syntax of an NUAL VLIW compiler without violating the dataflow dependences. XSI provides a designer the freedom of disassociating the hardware implementation of the NUAL VLIW processor from the instruction set architecture. Further, we investigate fairly radical (in the context of VLIW) changes to the hardware—like removing an adder, adding a multiplier, and incorporating simultaneous multithreading (SMT)—to show that our technique works for a variety of hardware configurations without compromising on performance. The technique can be used in both single-threaded and multi-threaded architectures to achieve a level of flexibility heretofore unavailable in the VLIW arena.

1 INTRODUCTION

VLIW processors have prevailed in the DSP market because of their simplicity and ability to execute multiple operations per cycle [1]. The regular algorithmic structure and dataflow properties of DSP applications make them ideal for VLIW architectures and compiler targets [2]. This has enabled developers to move from using hand-written assembly code to high-level languages like C, thereby lowering development time and effort. The lower time-to-market and cost overheads are considered to largely outweigh the potential performance disadvantages of compiled code compared to the hand-optimized code. VLIW architectures, however, have been plagued by object-code compatibility issues for a long time. Specifically, an application compiled for one class of processor cannot be executed on another processor with differing latencies or functional units without recompiling. This poses a problem, as it provides a manufacturer only limited flexibility in offering multiple processors at different cost points that can all execute the same object code. Hence, our primary goal is to decouple the underlying ISA from the hardware implementation. In its broadest sense, our technique enables (almost) any VLIW code to be run on (almost) any VLIW processor. The main contribution of this paper is a novel scheme designed to enhance portability and flexibility in VLIW architectures without degrading performance.

1.1 Motivation and Background

In VLIW processors with operations having non-unit latencies, the ISA can be defined in two different ways [3]:

Unit Assumed Latencies (UAL): UAL programs require that all operations in one instruction packet are completed before the next is issued. Thus the actual latencies of the operations are not exposed to the compiler or the programmer.

Non-Unit Assumed Latencies (NUAL): In NUAL programs, for an operation with latency L, the next L-1 instruction packets are issued before the given operation completes. The latency of each operation is exposed to the compiler or the programmer, and the NUAL architecture is thus said to be a *latency-cognizant ISA* [3].

Scheduling instructions for a UAL ISA is a simple task for the compiler or programmer. If the actual latency of all operations is a single cycle, then the hardware can issue instruction packets every cycle. However, if even a few operations have latencies longer than one cycle, the hardware would be unable to issue instruction packets every cycle unless it has interlock/dependency-check capabilities. For example, the StarCore SC140 is a UAL VLIW, where an instruction packet is issued for execution only after all operations belonging to the previous instruction packet have completed execution [4]. Another commercially available UAL VLIW, Analog Devices' TigerSharc, employs interlocking register files which stall a program when a dependency is detected, until the required data becomes available [5]. For VLIW architectures, in which an instruction packet comprises multiple independent operations, such dependency-check mechanisms could be expensive. Also, deeper pipelines would result in large penalties due to such stalls. This may be one of the contributing factors for UAL VLIWs like TigerSharc and SC140 to have shorter 1- or 2-stage execution units.

The advantage of a NUAL ISA is that the functional unit latencies are exposed to the compiler, thereby allowing the dependency check and latency-cognizant instruction-scheduling to be done entirely at compile time. Since there would be no hardware-enforced stalls, the pipeline depth can be increased, leading to faster clock speeds. Thus, NUAL VLIWs can provide better throughput without the hardware

* Bharath Iyer is now a Design and Verification Engineer at AMD. *Address*: Advanced Micro Devices, M/S 625, 5204 E Ben White Blvd., Austin, TX 78741. *Email*: bharath.iyer@amd.com

Copyright © 2004 IEEE. Published in the *Proceedings of the 31st International Symposium on Computer Architecture*, June 19–23, 2004 in München, Germany. Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works, must be obtained from the IEEE. Contact: Manager, Copyrights and Permissions / IEEE Service Center / 445 Hoes Lane / P.O. Box 1331 / Piscataway, NJ 08855-1331, USA. Telephone: + Intl. 908-562-3966.

mechanisms as complex as in UAL VLIWs. The Texas Instruments' TMS320C6000 series VLIW DSPs are examples of commercially available NUAL VLIWs [6].

However, exposing the operational latencies to the compiler enforces a tight coupling between the NUAL ISA and the hardware implementation. Any change in the ISA, or in the hardware, in terms of the assumed latencies or number or arrangement of functional units, would render them incompatible. Thus object-code compatibility across processors with different hardware configurations becomes a problem. To achieve object-code compatibility, the hardware should be able to break the instruction packets and issue operations depending upon the availability of functional units. Note that this breaking of the instruction packet is quite the opposite of what a VLIW ISA expects. The TMS320C6000 compiler schedules instructions with the assumption that all operations in a VLIW instruction packet will be issued for execution simultaneously [7]. The VLIW compiler's data-dependency assumptions could be potentially sabotaged by not issuing all the instructions in an instruction packet in the same cycle. Hence, some mechanism would have to be provided to overcome this violation.

1.2 Our Approach

In this paper, we propose a limited dynamic scheduling technique using a set of delay-buffers that store results temporarily and commit them to the architectural register files at the appropriate cycles. For the rest of this paper, we shall use the term VLIW to refer to NUAL VLIWs unless explicitly stated otherwise. The dynamic scheduling technique proposed here is an extension of the *split-issue* technique proposed by Rau [3], which was intended to provide object code compatibility across VLIW processors with similar arrangements of functional units but different assumed latencies. The split-issue mechanism proposed by Rau essentially allocates a temporary buffer to store an instruction's result, which is committed to the architectural register N cycles later, where (N+1) is the compiler-assumed latency of the given instruction. This mechanism, therefore, permits correct execution of a program even when the actual latencies do not agree with the compiler-assumed latencies. Rau's mechanism requires the compiler-assumed latencies for a program to be conveyed to the hardware in some form—e.g., a field in each operation specifying the assumed latency [8].

We make a key observation that the latencies of operations are with respect to instruction packets and not to machine cycles in a VLIW architecture. We use this observation to extend Rau's split-issue mechanism so that it commits a result from the temporary buffer to the architectural register after N *instruction packets* (not *cycles*), where (N+1) is the compiler-assumed latency of the given instruction. This allows the issue hardware to split instruction packets and issue individual operations within an instruction packet separately and in any order. The issue hardware would also have to signal to the commit logic each time it finishes issuing all instructions in an instruction packet. Note that the hardware schedules operations only within an instruction packet and does not perform out-of-order scheduling *across* different instruction packets. This *extended split-issue (XSI)* mechanism enables the issue of subsets of instructions from instruction packets for execution.

An interesting feature of the hardware design proposed here is the flexibility to vary the number of hardware functional units of any given type. Note that, in the case of a clustered VLIW architecture, reducing the number of functional units could also require appropriate multiplexed connectivity to the register files. The issue hardware, which is capable of splitting the instruction packet, would issue operations based on the availability of functional units. This allows the hardware designer to customize the number of functional units based on the target applications, thereby leading to improved utilisation efficiency, and possibly energy and die-area savings as well. Thus this approach achieves decoupling of the hardware and instruction set architecture in a VLIW architecture. Compatibility across processors with different functional unit latencies is another important feature that can be directly derived from Rau's split-issue technique. To illustrate the strengths of the XSI mechanism, we implemented radical hardware configurations of VLIW to highlight the levels of flexibility that are possible, such as incorporating simultaneous multithreading (SMT). Our performance results confirm that XSI provides flexibility without compromising on performance.

1.3 Related Work

SMT improves the processor throughput, extracting maximum parallelism by issuing as many instructions as possible from multiple threads in any given cycle [9][10]. These improvements have been seen previously with workloads that consist of mutually independent applications or applications that can be parallelized into independent threads by the programmer [11][12] or the compiler.

Incorporating SMT capability involves replicating processor context (register files, program counter, etc.) for each thread but retaining the set of functional units and sharing them between the threads. Keckler and Dally [13] have proposed an architecture called *Processor Coupling* where instructions from multiple VLIW threads are scheduled simultaneously to the functional units. The compiler and architecture assume that the operations scheduled in a single VLIW instruction packet need not be executed simultaneously. The hardware employs a scoreboard check mechanism to stall a given operation until all its source register operands are available. Thus, the hardware performs dependency check and resolves conflicts by stalling the processor, unlike a typical VLIW architecture. Later, Kaxiras et al. [14] studied SMT on the StarCore (SC140) VLIW DSP, wherein the issue logic selects VLIW instruction packets from ready threads—as many as it can accommodate, without splitting the VLIW instruction packets—and assigns them to functional units. We shall refer to this model as the *Kaxiras model* in later sections. Özer, Conte and Sharma [15] also propose a SMT VLIW architecture named *Weld*. The *Weld* architecture selects VLIW instruction packets from ready threads and issues them to available functional units. The compiler embeds a *separability bit* in the opcode for each operation, which the hardware uses to decide if it can issue that operation separately, thereby splitting the VLIW instruction packet. The *Weld* architecture is aimed at increasing ILP using compiler-directed (speculative) threads in a multithreaded VLIW architecture. We note here that none of these studies have attempted SMT in a NUAL VLIW architecture.

We extend the idea of SMT VLIW to the next logical step of issuing a subset of instructions from a VLIW instruction packet based on the availability of functional units without any explicit permission from the compiler. Thus, by combining the extended split-issue mechanism and SMT, the hardware schedules more operations to the execution units every cycle, thereby improving throughput, the system's multi-threaded performance.

The remainder of this paper is organized as follows. We describe the dynamic scheduling algorithm used to issue a subset of instructions from a VLIW instruction packet in section 2. Section 3 presents the details of the base TMS320C6201 VLIW architecture. In section 4, we discuss the level of flexibility provided by decoupling the hardware from the instruction set architecture. Section 5 describes the simulation methodology and presents performance results. We conclude in section 6 with an outline of our proposed future work.

2 EXTENDED SPLIT-ISSUE MECHANISM

VLIW architecture, when viewed as a contractual interface between the class of programs and the set of processor implementations, is

Cycle	EP	Operations		
1	1	A5=add(A1,A2),	A5=mul(A5,4),	A5=load(A1)
2	2	A1=mul(A5,A5)		
3	3	A6=sub(A1,A5),	A7=load(A8)	
4	4	…		

Code Listing 1: A VLIW Code Segment

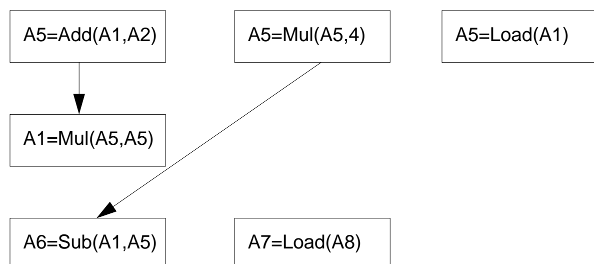


Figure 1: Dataflow Dependences for Register A5

basically an *Independence Architecture* [16]. It specifies a set of operations that are guaranteed to be mutually independent and can be issued simultaneously by the issue hardware without any checks. We shall use the term *execute packet (EP)*, as defined by Texas Instruments [17], to represent such an independent set of operations. These operations could have non-unit latencies. In a NUAL VLIW, the input operands for an operation may not depend on the operations that are in the execute pipeline at the time of issue. The compiler uses this assumption to aggressively reuse registers allocated to long-latency operations, possibly introducing output dependences. Thus any processor implementation would have to respect the flow dependences/independences in both directions: *horizontal* (within a single EP) and *vertical* (across different EPs). Breaking the EP semantics of the compiler and issuing operations from an EP over different cycles would disturb these dependences. In this section, we shall address the question of how to do limited dynamic scheduling in hardware while maintaining the aforementioned flow dependences. The extent of dynamic scheduling would be limited by the ability to issue operations from a single EP over multiple clock cycles. However, in any given cycle, operations that might be issued simultaneously would always belong to the same EP.

Firstly, we shall look at the problems that would arise from issuing operations from a single EP over multiple cycles. Consider the fragment of a VLIW program, shown in Code Listing 1, with operation latencies being 1 cycle for *add* and *sub*, 2 cycles for *mul*, and 5 cycles for *load.* The *cycle* column in the table represents the cycle at which the current EP is issued for execution to the functional units. In EP1, we see that all operations write their results to the register A5. This causes no problem because each of these operations would write their results to A5 in different cycles. The *mul* operation in EP1 should not get its input operand in register A5 from the *add* instruction in the same EP. Also, the *mul* operation in EP2 should get its input operand in register A5 from the *add* operation in EP1 and not from the *mul* or the *load* operations in EP1. Subsequently, the *sub* operation in EP3 should get its input operand in A5 from the *mul* in EP1 and not from the *add* or the *load* in EP1. Also, its input operand in A1 should not be from the *mul* in EP2. The data dependences for register A5 are shown in Figure 1.

The data dependences are respected by the VLIW architecture, in particular by its issuing all operations of an EP simultaneously. However, the same does not hold true if the EPs are broken up: if we issue each of the operations in a different cycle, as shown in Code

Cycle	EP	Operations
1	1	A5=add(A1,A2)
2	1	A5=mul(A5,4)
3	1	A5=load(A1)
4	2	A1=mul(A5,A5)
5	3	A6=sub(A1,A5)
6	3	A7=load(A8)
7	4	…

Code Listing 2: VLIW Code Segment With Each Operation Issued Separately

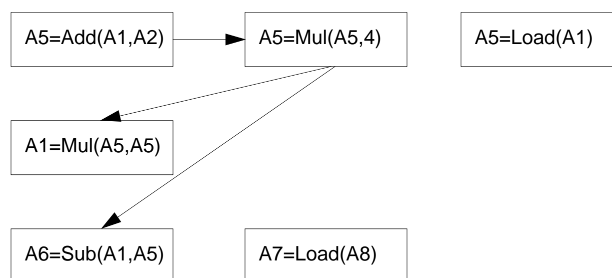


Figure 2: Incorrect Dataflow for Register A5

Listing 2, then the above flow dependences would not hold anymore. Figure 2 shows that the *mul* in EP1 would get the output produced by the *add* operation in register A5. The *mul* in EP2 would get its input from the *mul* in EP1 instead of the *add* in EP1.

To break up the execution packets and yet still maintain the flow dependences, we extend the dynamic scheduling technique proposed by Rau [3]. Rau's technique uses a set of buffers called delay-buffers to store results temporarily. These results are then committed to the architectural registers at the appropriate cycles. Being transparent to both the programmer and the compiler, Rau's technique was originally proposed to guarantee object-code compatibility across VLIW processors with the same ISA but different hardware latencies. It is important to note that Rau's mechanism requires that all operations in an EP be issued in the same cycle. We first illustrate this technique and then describe how it can be extended to handle the problems that would arise from issuing operations from a single EP over different cycles.

In Code Listing 3, the original code fragment is shown augmented with a new set of operations called *phase-2 operations*. These operations are not generated by the compiler; they represent the action of committing the data values from the delay-buffers to the architectural registers and are performed by the hardware at the appropriate cycles. The phase-2 operations specified along with each EP are carried out at the end of the given cycle. In Code Listing 3 and in the rest of this paper, the commit operation from *src* to *dest* is shown as "*dest* <= *src*". The outputs of all operations are stored in dynamically allocated delay buffers. In the example, the output of the *add* instruction in EP1 is stored in the delay-buffer *LA1*. The corresponding phase-2 operation, i.e., committing the data in *LA1* to A5, is scheduled at the end of the same cycle because an ADD is a 1-cycle operation. Similarly, the output of the *mul* in EP1 is stored in delay-buffer *MA1* and then committed to A5 at the end of cycle 2 because a MUL is a 2-cycle operation. There are two phase-2 operations scheduled at the end of cycle 3: the output of *sub* (from EP3) to be committed from *LA1* to A6, and the output of *mul* (from EP2) to be committed from *MA2* to A1. Thus for every program operation, the corresponding phase-2 operations are scheduled after N cycles,

Cycle	EP	Operations				Phase-2
1	1	*LA1*=add(A1,A2),	*MA1*=mul(A5,4),	*DA1*=load(A1)	-	A5<=*LA1*
2	2	*MA2*=mul(A5,A5)			-	A5<=*MA1*
3	3	*LA1*=sub(A1,A5)	*DA3*=load(A8)		-	A6<=*LA1*, A1<=*MA2*
4	4	…			-	
5	5	…			-	A5<=*DA1*

Code Listing 3: VLIW Code Segment With Phase-2 Operations

Cycle	EP	Operations		Phase-2
1	1	*LA1*=add(A1,A2)		
2	1	*MA1*=mul(A5,4)		
3	1	*DA1*=load(A1)	-	A5<=*LA1*
4	2	*MA2*=mul(A5,A5)	-	A5<=*MA1*
5	3	*LA1*=sub(A1,A5)		
6	3	*DA3*=load(A8)	-	A6<=*LA1*, A1<=*MA2*
7	4	…	-	
8	5	…		
9	5	…	-	A5<=*DA1*

Code Listing 4: VLIW Code Segment With Each Operation Issued Separately Augmented By Phase-2 Operations

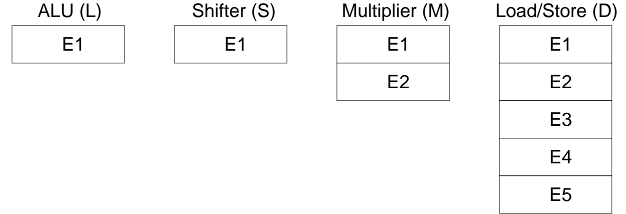


Figure 3: Functional Unit Latencies in the TMS320C6201

Functional Unit	Description	Number of Units	Latency
L	ALU	2	1
S	ALU/Shifter	2	1
M	Multiply	2	2
D	Load/Store	2	5

Table 1: TMS320C6201 Functional Units and Their Latencies

where (N + 1) is the latency of the given operation. Note that the program operations access the register file for their source operands but write their results in the allocated delay-buffers. Conversely, the phase-2 operations read from the delay buffers and copy the data into the appropriate architectural register.

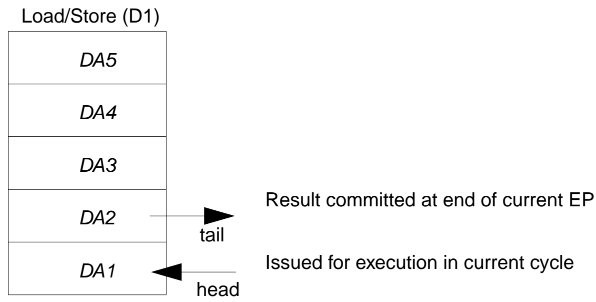
Now we show that by extending the above technique, we can split the operations in an EP and issue them over different cycles without disturbing the flow dependences. To guarantee correct program functionality, *we schedule the phase-2 operations only with respect to the final operation(s) issued from an EP*—i.e., at *EP-boundaries*. Hence, for every program operation, the corresponding phase-2 operations will be scheduled after N *EP-boundaries*, where (N+1) is the latency of the given operation. This is because the flow dependences, horizontal and vertical, are really with respect to EPs and not with clock cycles in VLIW architectures.

We see from Code Listing 4 that the *mul* operation in EP1 does not get its input operand in register A5 from the *add* operation belonging to the same EP, though the *mul* was issued a full cycle later. The *mul* operation in EP2 gets its input operand in register A5 from the *add* operation in EP1 (committed from *LA1*) and not from the *mul* or the *load* operations in EP1. Moreover, the *sub* operation in EP3 gets its input operand in A5 from the *mul* in EP1 (committed from *MA1*) and not from the *add* or the *load* in EP1. The input operand in A1 is not from the *mul* in EP2.

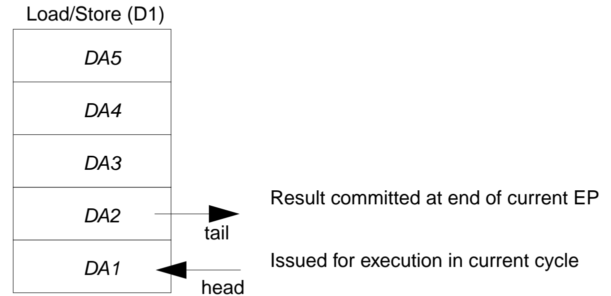
To summarize, we dynamically schedule phase-2 operations for every operation issued at the appropriate *EP-boundary* determined by the latency of the issued operation. This maintains the flow dependences even if all the operations in an EP are not issued in a given cycle.

The total number of delay-buffers required to support this technique is a function of the number of execution units and the maximum latencies of these units. In any given cycle, the number of instructions in the various stages of execution would determine the number of delay buffers required for that unit. In our scheme, the number of buffers corresponds to the hardware latencies, irrespective of the compiler's assumptions; in Rau's scheme, the number of buffers corresponds to the compiler's assumed latencies, irrespective of the hardware latencies. Figure 3 shows the execute stages of the TMS320C6201 pipeline, wherein we see that the multiplier has 2 stages in its execute pipeline, viz., E1 and E2. Therefore in any given cycle, there would be only 2 instructions being executed in the multiplier. This means that there would be at most two outstanding phase-2 operations corresponding to the two instructions in the multiplier. The number of functional units and associated latencies in the TMS320C6201 are given in Table 1. Thus, the total number of delay-buffers required to support limited dynamic scheduling on the TMS320C6201 is (2x1) + (2x1) + (2x2) + (2x5) = 18.

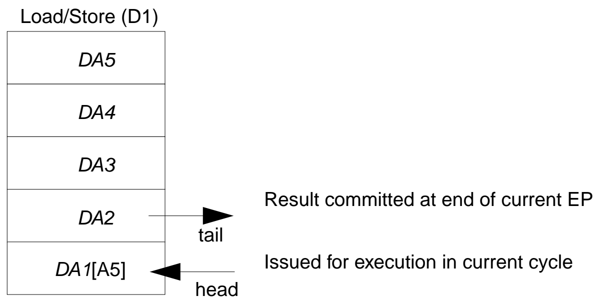
The delay buffers are implemented as circular buffers with a head and a tail pointer. The head pointer points to the buffer where the information about the operation issued for execution in the current cycle is stored, and the tail pointer points to the buffer from which the data is committed to the appropriate architectural register. The pointers are incremented on EP boundaries, not cycle boundaries. To understand this let us look at the delay buffers used by the 5-stage load/store (D1) unit. Figure 4 depicts the circular buffer implementation for the delay buffers of the D1 unit and its behaviour for the code fragment shown in Code Listing 4. The D1 unit has a latency of 5, i.e., the results of an operation issued to this unit will be available to operations in the 5th EP relative to the given operation and later. In Figures 4(a) and 4(b), corresponding to cycles 1 and 2 respectively, we see that the head pointer and tail pointer remain unaltered. The head pointer points to the delay buffer (*DA1*) allocated to the D1 operation in the current EP (i.e. EP 1), and the tail pointer points to the delay buffer whose data is to be committed at the end of the current EP. In cycle 3, when the *load* instruction in EP 1 is issued for execution, its destination register is stored in a field of *DA1*. At the



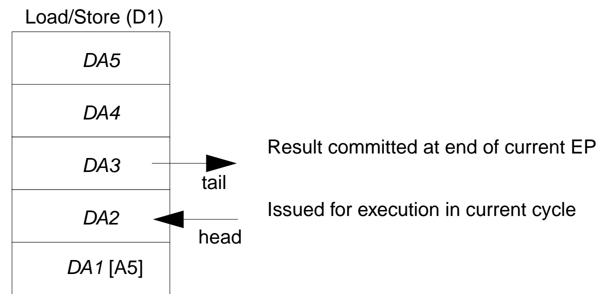
(a) Cycle 1 (EP 1)



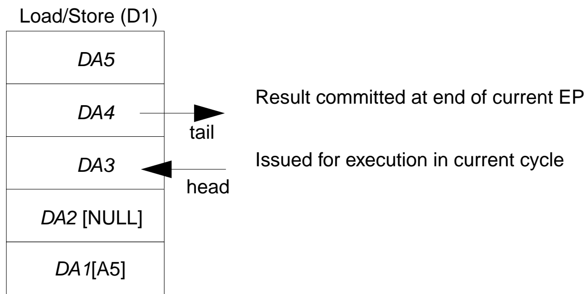
(b) Cycle 2 (EP 1)



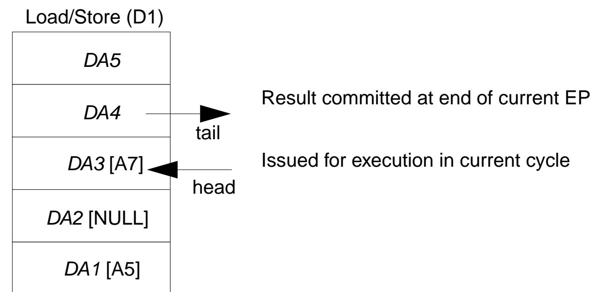
(c) Cycle 3 (EP 1)



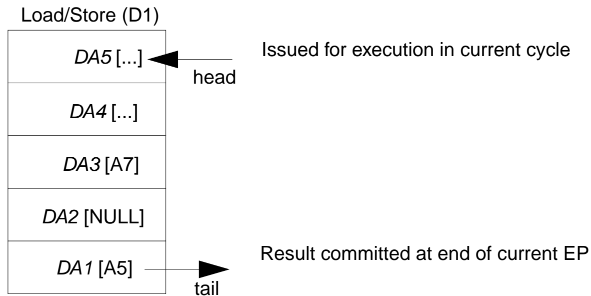
(d) Cycle 4 (EP 2)



(e) Cycle 5 (EP 3)



(f) Cycle 6 (EP 3)



(g) … Cycle 9 (EP 5)

Figure 4: Circular Buffer Implementation of the Delay Buffers for the Load/Store(D1) unit

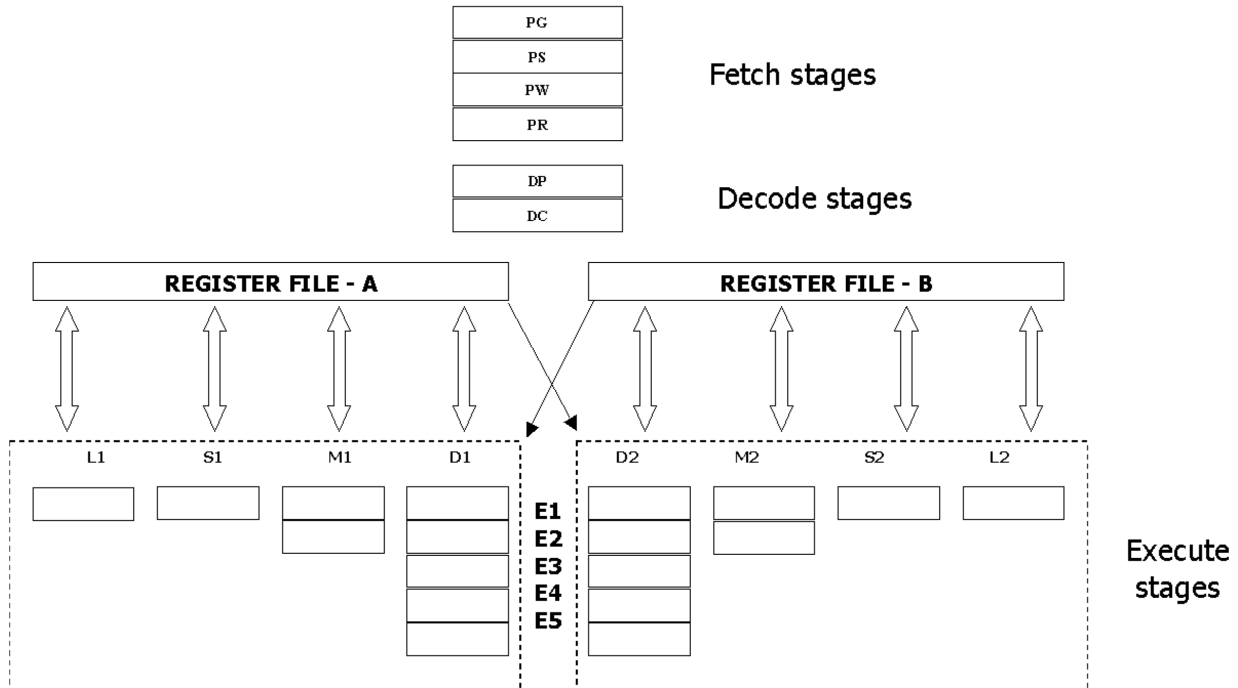


Figure 5: TMS320C201 Base Architecture

end of EP 1, data, if any, from the tail (i.e. *DA2*) is committed to the corresponding architectural register file, and the head and tail pointers shift by one position. This can be seen in Figure 4(d), where the head pointer now points to *DA2* and the tail to *DA3*, for the next cycle. Since there is no D1 operation in EP 2, the destination field in *DA2* contains a NULL entry. Figures 4(e) and 4(f) depict the behaviour for EP 3. We see that as a *load* instruction is issued for execution to the D1 unit in cycle 6, the destination field in *DA3* stores the destination architectural register, i.e. A7. Finally, we can see the data being committed from *DA1* to the architectural register A5 corresponding to the load in EP 1 in Figure 4(g). This corresponds to the EP 5 boundary at cycle 9 as in Code Listing 4.

The delay buffers for any unit can be implemented similarly with a simple circular buffer structure of appropriate size. Functional units with unit latencies like the L and S units would have only one delay-buffer associated with each to hold the data until the end of the EP. Note that the delay buffers are simple circular buffers which are not in the processor's critical path. Also, access to the delay buffers can be pipelined. Hence, they do not degrade the processor's cycle time, one of the chief components of processor performance.

3 TEXAS INSTRUMENTS 'C6201 ARCHITECTURE

We chose Texas Instruments' TMS320C6201 [6][17] for our study as it is a good representative of commercially available NUAL VLIWs. In this section we shall describe its architecture.

The TMS320C6201 is an 8-wide VLIW DSP with an 11-stage pipeline. It has a clustered architecture; i.e., the 8 functional units are divided into 2 sets of 4 each. Two register files A and B, with 16 registers each, are connected to one set of functional units each. Crossovers allow limited use of the A-registers by the B-side functional units and vice versa. Figure 5 depicts the CPU pipeline. The pipeline phases are divided into three stages, viz. fetch, decode and execute. The fetch phase comprises 4 stages: program address generate (PG), program address send (PS), program access ready wait (PW) and program fetch packet receive (PR). A fetch packet (FP) comprises 8 operations (also referred to as instructions) packed together. However, all 8 operations in the same fetch packet need not constitute a single EP. An EP could be made up of any number of operations ranging from 1 to 8, and therefore an FP could comprise of a single EP or up to 8 EPs. The structure of a typical FP is shown in Figure 6.

The decode phase comprises 2 stages: instruction dispatch (DP) and instruction decode (DC). During the DP stage, operations in an EP are extracted from the FP and assigned to the appropriate functional units. Note that if an FP contains more than one EP, the fetch stages stall until all EPs in that FP are dispatched. In the DC stage, the source registers, destination registers, and associated paths are decoded for the execution of the operations in the functional units.

Finally, the execute phase involves a varying number of stages depending on the functional unit. The 4 functional units on each side are a matched set. Each side contains a 40-bit integer ALU (L-unit), 40-bit shifter (S-unit), a 16-bit multiplier (M-unit) and a 32-bit adder, which is also used as an address generator for loads and stores (D-unit). As we can see from Figure 5, the L-units and S-units have an execution latency of 1 cycle, the M-units have a latency of 2 cycles, and the D-units have a latency of 5 cycles. Branches are resolved in the S-units and take effect as delayed-branches with 5 delay slots, i.e., the branch target begins execution (E1 stage) in the sixth cycle after the branch instruction. The branch instruction is executed in the E1 stage of an S-unit, and the target program counter (PC) is fed back to the PG stage. The branch can be said to be a single-cycle latency operation, the output of which is not stored in any register file but in the program counter of the PG stage.

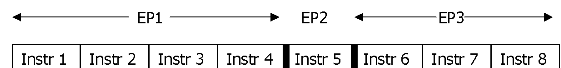


Figure 6: A Fetch Packet in the 'C6000 Architecture

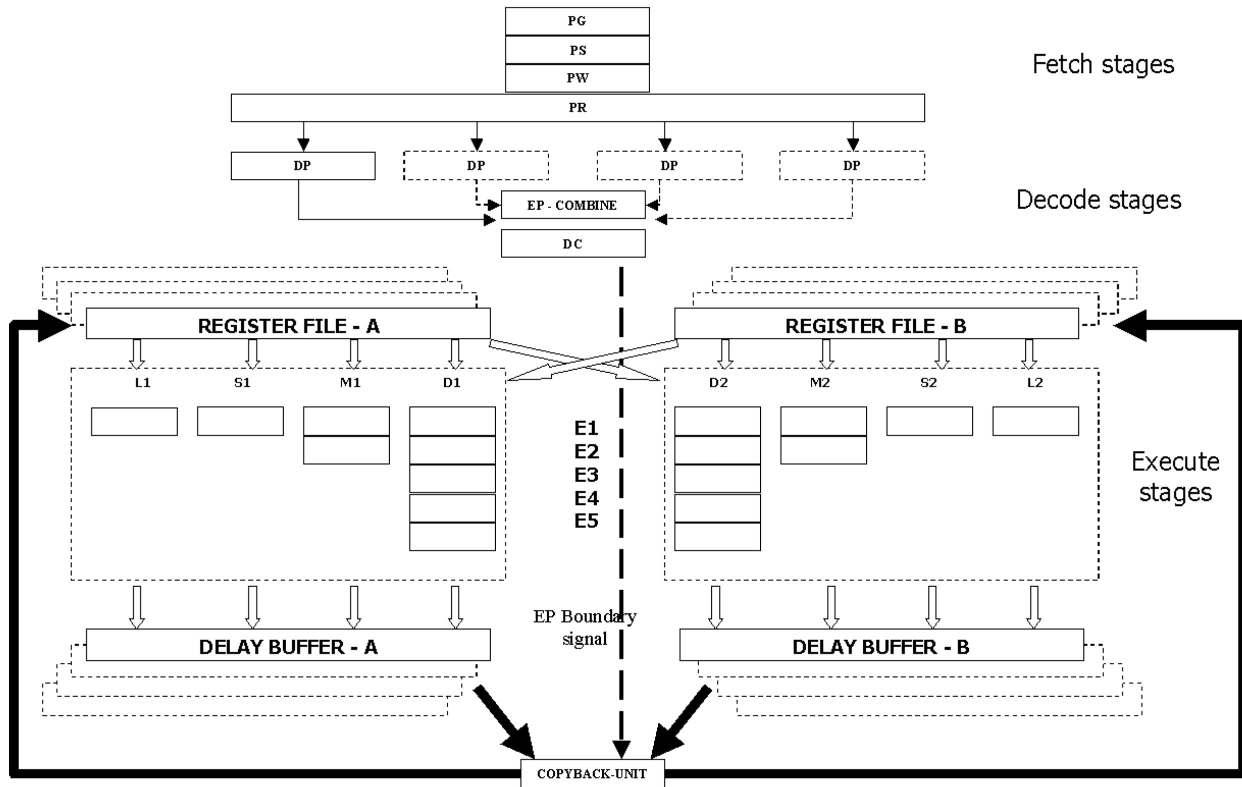


Figure 7: TMS320C6201 VLIW Architecture Extended to Handle SMT

4 DECOUPLING IMPLEMENTATION FROM ISA

Recall that a key feature of the XSI dynamic scheduling technique described in Section 2 is that it provides the leverage to decouple the hardware implementation of the VLIW processor from its instruction set architecture. The compiler schedules instructions based on the ISA, but the hardware need not conform exactly to the compiler's assumptions. We noted earlier that the dataflow dependences in the VLIW architecture are with respect to the EPs and not clock cycles. We have shown that the delay-buffer and copyback logic schedules phase-2 operations so that the results of instructions are committed based only on the *EP-boundary* signal. Hence, the issue logic can afford to break the EPs without disturbing the compiler's intended semantics. In this section we demonstrate the flexibility offered by the technique by showing several possible hardware schemes that can be implemented using XSI.

4.1 SMT VLIW Architecture

Our SMT-VLIW architecture is depicted in Figure 7, which shows an architecture supporting up to 4 threads. The processor context—i.e., register files, program counter, etc.—is replicated appropriately to accommodate 4 threads. It resembles the base architecture in most aspects. It retains the same functional units in both number and type. The functional units get their input operands from the register files, and they store their output in the delay-buffers. The copyback-unit schedules the phase-2 operations to copy the data from the delay-buffers to the appropriate registers upon receiving the *EP-boundary* signal. The DC unit generates a thread-specific *EP-boundary* signal when it decodes the last set of operations from an EP.

To understand the decode phase, let us assume that the PR stage of the fetch phase provides fetch packets from all 4 threads. The DP stage is replicated to provide the ability to dispatch EPs from all 4 threads. The *EP-combine* stage, an additional stage depicted in Figure 7, represents the dynamic scheduling hardware that issues instructions from the EPs of different threads. The dynamic scheduling technique described in Section 2 is used to issue instructions from an EP for a given thread based on the availability of the functional units. The issue policy determines the priority of the threads, and thereby the sequence in which they are to be chosen for issuing instructions. We shall refer to the highest priority thread as thread-1, the thread with the next priority as thread-2, and so on. Thus the DC stage receives up to 8 instructions, possibly from different threads, each tagged with its thread identifier. For the instructions issued, the DC stage then decodes the source registers, destination registers, and associated paths and sends them to the execution units. Each source operand is read from the register file corresponding to the thread to which the instruction belongs. When the DC stage decodes and issues an instruction which is at an EP boundary, it triggers an *EP-boundary* signal for that thread. The copyback unit keeps track of all the *EP-boundary* signals it receives and schedules the phase-2 operations for appropriate instructions based on their latencies in terms of EPs. Again, this is because the flow dependences are with respect to the EPs and not with clock cycles in VLIW architecture.

The functional units store the results in the allocated delay-buffers after executing the instructions. As all instructions are issued and executed in-order, the allocation of delay-buffers is simple and follows the in-order issue of instructions. Therefore, the phase-2 operations corresponding to the instructions are also scheduled in-order by the copy-back unit.

In the PR stage, the FPs are stored in separate sets of buffers corresponding to the thread number. We could extend this procedure to

accommodate any number of threads. For simplicity (and to aid predictability, which is important in many embedded and DSP systems), we implement a static-priority scheme for fetching from threads. That is, thread-1 is unaffected, and the remaining functional units are filled with instructions taken from thread-2, thread-3, etc. Consequently, fetch stalls behave exactly as in the base architecture for the highest priority thread, and SMT happens transparently with respect to the highest priority thread. However, for the lower priority threads this is not the only condition when they cannot fetch an FP. A low priority thread would be unable to fetch and dispatch in the DP stage when the *EP-combine* stage is unable to issue a complete EP from that thread.

In the base architecture, we saw that the fetch phase stalls when an FP containing multiple EPs is being serviced by the DP stage. We can exploit these stall cycles to fetch FPs from other threads. A clear advantage of this is that we can fetch FPs from different threads without requiring additional fetch bandwidth and using only a minimal addition of logic. The fetch stages in the SMT VLIW processor maintain instruction buffers for each thread and fetch the instructions in a multiplexed fashion, fetching an FP from the next available highest priority thread in a cycle. We omit describing the implementation of fetch stages in greater detail due to space constraints. Moreover, given that in any cycle the fetch stages would fetch only an FP from among the different threads, the performance of the processor would not be affected significantly by different implementations.

Finally, we enumerate the expenses incurred in hardware to incorporate SMT over the base VLIW architecture. The pipeline requires a new *EP-combine* stage, and for every additional context we require a set of register files and their datapaths, a set of delay-buffers, pipeline latches for the fetch and decode stages, dispatch logic to extract an EP from an FP, program counter, a set of multiplexers at the *EP-combine* stage to issue operations from different threads, and copy-back logic that commits data from the delay-buffers to the register files on the trigger provided by the *EP-boundary* signal.

We note here that the SMT model proposed by Kaxiras, et al. [14] would also need a similar delay-buffer and copy-back mechanism, if it were implemented on an NUAL VLIW, to maintain the vertical dataflow dependence assumptions of the compiler's schedule. In the Kaxiras model, where the instructions are issued without splitting the EPs, the secondary threads may or may not be able to issue EPs every cycle based on the functional unit availability. Such lack of predictability can disturb the vertical data dependences for instructions in those threads, which would have to be handled with some buffering mechanism. However, since the EPs are issued without splitting, the copy-back logic would not require the DC stage to provide an *EP-boundary* signal and, therefore, would be simpler.

4.2 Different Hardware Configurations

To illustrate how XSI allows different hardware configurations, we consider the 'C6201 VLIW DSP again. Figure 8 shows the average functional unit utilisation statistics for various DSP benchmarks executed on the original architecture. We observe that both the processor's multipliers (M1 and M2) have very low utilisation. Moreover, it is rare that both multipliers are used simultaneously. Thus, one could design an implementation of the 'C6201 with only one multiplier. In any given cycle, this multiplier could function as M1 or M2 as required. However, if any EP has instructions scheduled for both the multipliers, then the issue logic would split the EP and issue one of them in the following cycle. Thus, the program would behave as if one of the multipliers is always being used by a (imaginary) higher priority thread. Such a design requires that this multiplier has datapaths multiplexed to both the register files, thereby allowing it to behave like M1 or M2 as required in a given cycle.

Figure 10 shows that the L and S units are heavily utilised—especially so as the number of threads being executed increases. Given the freedom, a hardware designer could choose to increase the number of such functional units to improve the performance throughput. For example, if there were an additional pair of L units in the processor, the issue logic could accommodate 2 more ALU instructions from the lower priority threads. The hardware could also have just one additional L unit behaving as L1 or L2 as required in a given cycle. As mentioned before, this would require the datapaths to be multiplexed between the 2 register files. It is also important to note that the issue logic would never issue instructions from multiple EPs for a given thread; thus performance enhancements from additional functional units can be exploited only by multithreading. On the other hand, *removing* a functional unit represents a potentially significant cost-performance trade-off even for a single thread.

We also investigated schemes of substituting the multipliers with (high latency) alternate functional units. We studied mechanisms wherein multipliers were replaced with a single adder/shifter combination [18]. Here, this adder/shifter combinational unit alone is multiplexed between the 2 register files like the additional functional units mentioned above. This combinational unit is used to execute multiply instructions whenever there is one in the instruction packet and for adder/shifter operations otherwise.

5 EXPERIMENTAL METHODOLOGY & RESULTS

In this section, we describe the experimental setup and present the performance evaluations. Our choice of benchmarks spans three suites, viz. MediaBench [19], MiBench [20] and the UTDSP benchmark suite [21] and is based on the availability of appropriately ported versions. The chosen benchmarks (ADPCM, G723, Pegwit, MPEG decode, FFT, FIR, Matrix Multiply, etc.) do encompass the general behaviour of DSP applications. Due to lack of space, we present only the averaged behaviour of these benchmarks (full results are shown in the Appendix). We compiled the benchmarks using the Texas Instruments 'C6000 C compiler [7] with important optimisations like loop unrolling, software pipelining, loop optimizations, and loop rotation enabled.

For our base DSP architecture, we used an in-house cycle-accurate TI TMS320C6201 [22] simulator. We extended this simulator to incorporate the XSI mechanism and SMT capability as described in Sections 2 and 4. The benchmarks were executed for 50M cycles at 200MHz representing a workload of 0.25sec [23]. We assume a perfect memory model for these studies as we focus on the flexibility of the XSI mechanism.

5.1 Base Architecture Performance

We first present the utilisation statistics of the TMS320C6201. Figure 8 shows a stacked chart representing the proportion of time each functional unit is utilised on average. The lower portion of each stacked bar indicates the utilisation of a functional unit of each given type (unit 1), the top portion indicates the utilisation of its complementary functional unit (unit 2), and the intermediate portion indicates the simultaneous utilisation of both the units. Please note that the sum of both units' utilisation numbers can total 200%, not 100% (i.e., if both units are completely utilised). We observe that the utilisation of the functional units is so low that each functional unit is idle for nearly 75% of the time on average. Figure 9 shows the statistics corresponding to the number of parallel or independent operations scheduled by the compiler every cycle. It shows the proportion of time for which 0 operations (NOPS) are scheduled, the proportion of time only 1 operation is scheduled, the proportion of time only 2 operations are being scheduled, and so on. These statistics show that the compiler does make an attempt to extract parallelism and, at times, is able to schedule up to 6 parallel operations in a given cycle. However, this does not happen very often, and the functional units remain under-utilised for most of the cycles.

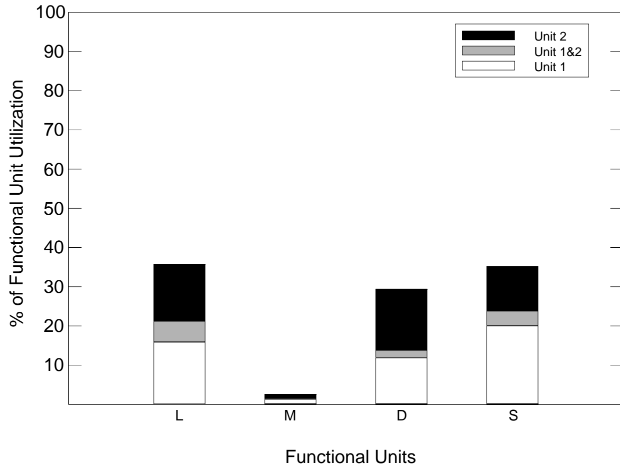


Figure 8: Functional Unit Utilisation for Base Architecture

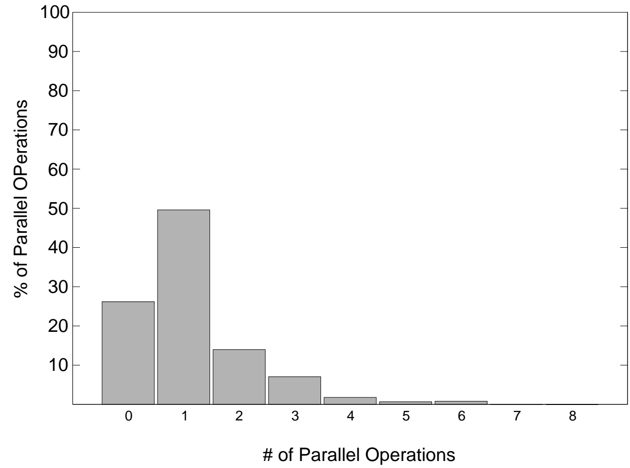


Figure 9: Number of Parallel Operations scheduled in Base Architecture

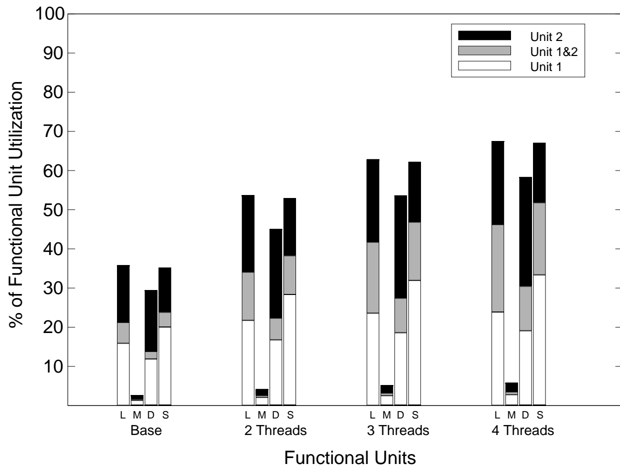


Figure 10: Functional Utilisation for Various Threads

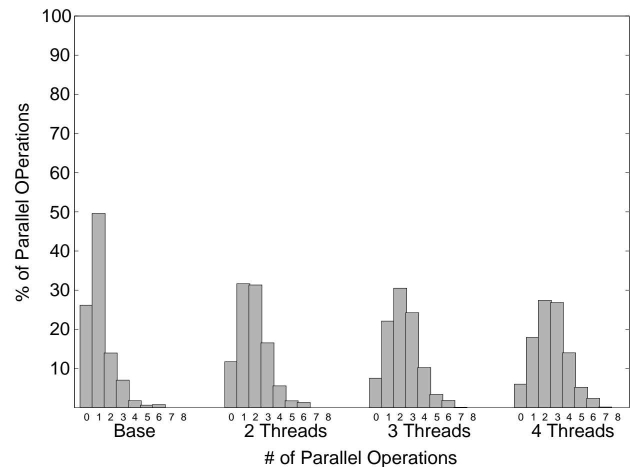


Figure 11: Number of Parallel Operations for Various Threads

5.2 SMT and Hardware-ISA Decoupling

We now examine the effect of SMT on the performance of the TMS320C6201. As mentioned, for our SMT results we have used a fixed-priority issue policy, whereby each thread is assigned a priority for the entire period of its execution. Thus, after issuing the EP from thread 1, based on functional unit availability, operations from the EPs of thread 2, 3, and 4 (in that order) are chosen and issued. For the experiments, multiple instances of the same program are executed as separate threads.

Figure 10 shows the average functional unit utilisation for the benchmarks with an increasing number of threads. We see that as the number of threads increases to 4, the utilisation of the L, S, and D units increase significantly, and the proportion of time for which the pairs of L, S, and D units are simultaneously utilised increases as well. Figure 11 shows the proportion of time taken for a given number of parallel operations issued for execution every cycle on an average. For a single thread (Base), for about 75% of the time, we see that *at most 1 operation* is issued in a single cycle. This distribution changes, and more operations are issued every cycle, as the number of threads are increased—this is expected. For 4 threads, we see that for 75% of the cycles, *at least 2 operations* are issued every cycle, making much better use of the available functional units.

Figure 12 shows the average processor throughput (IPC) for our model of SMT VLIW for 2, 3, and 4 threads in comparison with the Kaxiras model (labeled *2 Threads-K, 3 Threads-K,* and *4 Threads-K*) where the instructions are issued without splitting the instruction packets. (*The detailed graphs for all benchmarks are presented in Figures A1 and A2 of the appendix*). The lowest part of each stacked bar represents the IPC of thread 1, the part above indicating the IPC of thread 2, and so on. We observe that our model performs at least as well as the Kaxiras model and on average 2%, 4%, and 5% better for 2, 3, and 4 threads respectively. More importantly, compared to the base architecture, both the SMT configurations show substantial gains in IPC. The average gains in IPC for our SMT model are about 60%, 100% and 120% for 2, 3, and 4 threads, respectively.

Next, we present in Figure 13 the average behaviour of the benchmarks for different hardware configurations, viz. base and SMT (referred to as *normal*), removing a multiplier (-M), removing a multiplier and adding an ALU (-M+L) and removing a multiplier and adding an ALU and Shifter (-M+L+S). No cycle time penalties were assumed in multiplexing the datapaths to the register files. (*Figures*

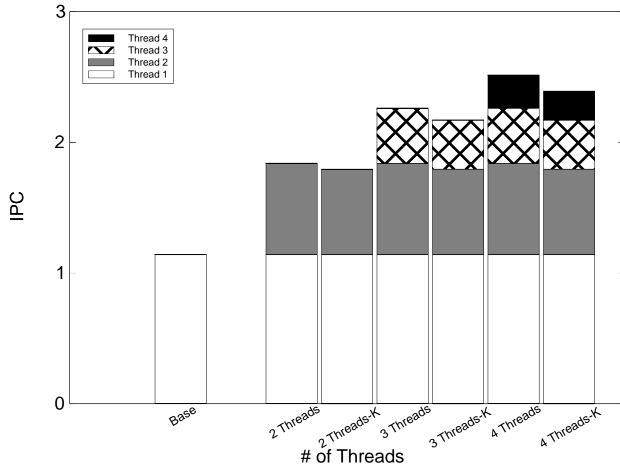


Figure 12: Processor Throughput (IPC) of SMT VLIW (Comparison of Our Model Vs. Kaxiras)

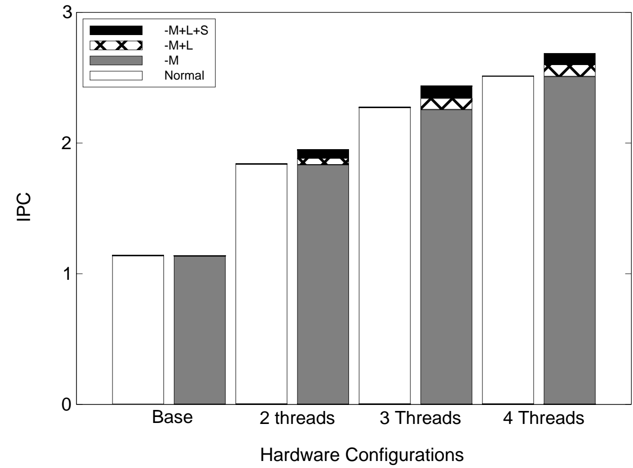


Figure 13: Processor Throughput (IPC) of SMT VLIW for Different H/W Configurations

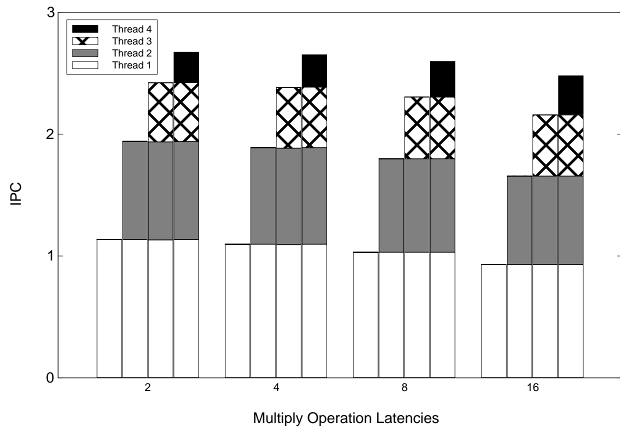


Figure 14: Processor Throughput (IPC) of SMT VLIW for different multiply latencies

A3 and A4 in the appendix show detailed results over all benchmarks.) We chose to add additional L and S units, as we observed from Figure 10 that the L and S units were highly utilised—especially so as the number of threads was increased. We see that for a single thread the decrease in performance by removing a multiplier is insignificant. However, as the number of threads increases, though removing a multiplier has no significant impact on the performance, one can replace that multiplier with other things (an additional adder and/or shifter). Adding an ALU in place of a multiplier (-M+L) compensates well and boosts the throughput. Further gains in throughput are seen in the -M+L+S hardware configuration. Interestingly, for a few benchmarks, namely ADPCM_decode and GSM_Decode (*see Figure A4*), the -M+L+S configuration produces slightly greater throughput for 3 threads as compared to executing 4 threads in the baseline configuration of the processor. This is due to the fact that the parallelism in these applications is limited in a multithreaded scenario by the number of specific functional units (L&S in this situation). Increasing the number of threads does not result in improved performance. Therefore, the second and third threads in these applications efficiently utilise the additional units and give a better throughput than the normal configuration with four threads.

Figure 14 shows the average behaviour of the benchmarks for multipliers replaced by adder/shifter combinations of different latencies, taking advantage of the fact that the adder/shifter combination need not be as fast as the original multiplier unit. The compiler-assumed latency for multiply operations is 2 cycles, and we vary the hardware latency from 2–16 cycles. This demonstrates that the XSI mechanism can handle situations where the compiler-assumed latency is less than the actual latency. This is done by latency stalling as suggested by Rau, wherein a thread is stalled after the lapse of N cycles of multiply operation where N is the compiler assumed latency. Alternate mechanisms for dependence checking such as scoreboarding, Tomasulo, etc. are expensive in this scenario as the hardware logic becomes more complex.

We carried out some experiments on the real-time behaviour of the DSP workloads. Figure 15a shows the arrangement of benchmarks GSM_Encoder, Decrypt (Pegwit), GSM_Decoder, and MPEG_Decoder. Figure 15b shows the behaviour of these benchmarks with GSM_Encoder, Decrypt and GSM_Decoder working at the rate of 50 frames per second and the MPEG_Decoder at 3 frames per second. In the base architecture, for every period of 20ms (4M cycles), the workload is arranged in such a manner that the benchmarks are executed in a serial fashion in the following sequence:

1: GSM_Encoder
2: Decrypt
3: GSM_Decoder
4: MPEG_Decoder (consumes any remaining available cycles)

For the SMT version, for every period of 20ms (4M cycles), the applications are run in the same order as above, but with each application executing as an independent thread. As soon as GSM_Encoder (thread 1) completes producing 1 frame of output, the thread re-schedules itself for the next frame and exits, the priorities are reassigned (promoting each remaining thread one priority level), and another instance of MPEG_Decoder is spawned as thread-4. After Decrypt completes producing 1 frame of output, the thread re-schedules itself and exits, GSM_Decoder and the two instances of MPEG_Decoder are promoted, and another instance of MPEG_Decoder is spawned as thread-4. After GSM_Decoder completes 1 frame, four simultaneous threads of MPEG_Decoder are left to consume the remaining cycles. We can see from Figure 15b that there is a significant increase in the processing throughput in the SMT version (4 threads) as compared to the base processor. Also, we ran these simulations for different configurations of the hardware

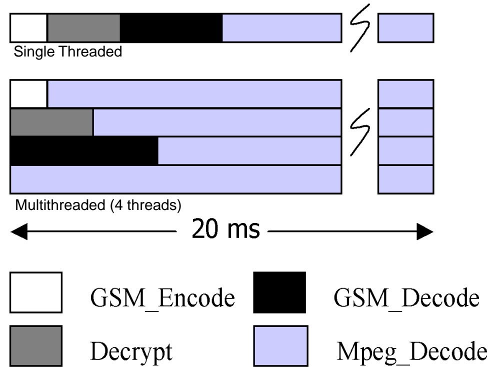


Figure 15a: Real Time Load Arrangement

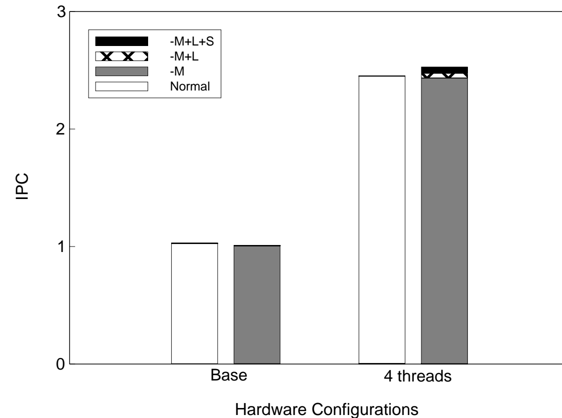


Figure 15b: Real Time Load Performance

(-M, -M+L, and -M+L+S), shown as the stacked bars on top of the 4-thread histogram. These results show that the throughput gains in the various design configurations of the SMT VLIW architecture hold their promise for workloads comprising of a combination of DSP applications as well.

6 CONCLUSIONS

We present an *eXtended Split-Issue (XSI)* mechanism which provides NUAL VLIW architectures with flexibility in choosing any desired hardware configuration by decoupling the *hardware–ISA* association. Through a wide range of experiments, we have shown that our scheme enables incorporating SMT in VLIW processors and thus can potentially increase the throughput. We have also shown that the actual underlying hardware can be totally independent of the compiler assumptions in terms of number and configuration, availability, or latency of functional units.

Thus the XSI mechanism expands the hardware design space along two dimensions. The first allows VLIW architectures to support SMT, and the second allows varying the number of functional units depending on utilisation.

Using a commercial NUAL VLIW DSP architecture, the Texas Instruments TMS320C6201, we show that incorporating SMT can yield rich benefits in terms of processing throughput. The average gain in IPC over the base processor was seen to be 120% for our SMT VLIW processor supporting 4 threads. This can be traded off for reduction in energy consumption by scaling the operating frequency/voltage accordingly in DSPs, where embedded-systems requirements typically make minimization of energy consumption more important than gains in throughput.

We have also shown that the efficiency of the VLIW DSP can be increased by a processor architect choosing the number of functional units of each type based on their utilisation statistics. For example, removing a multiplier yields no significant performance degradation, yet the elimination of multiplier can have significant impact on die area and power consumption. This gives the hardware designer a wider variety of choices to build an efficient NUAL VLIW processor without modifying the compiler or ISA.

As part of future work, we will explore the design space with precise interrupt considerations. The behaviour of real-time applications is another aspect that needs detailed investigation.

ACKNOWLEDGEMENTS

The authors would like to thank Brinda Ganesh, Arunchandar Vasan and the anonymous reviewers for their valuable feedback.

The work of Sadagopan Srinivasan is supported in part by NSF grant EIA-9806645 and NSF CAREER Award CCR-9983618. The work of Bruce Jacob is supported in part by NSF CAREER Award CCR-9983618, NSF grant EIA-9806645, NSF grant EIA-0000439, DOD MURI award AFOSR-F496200110374, the Laboratory of Physical Sciences in College Park MD, and by Compaq.

REFERENCES

[1] P. Faraboschi, G. Desoli and J.A. Fisher. The Latest Word in Digital and Media Processing. In *IEEE Signal Processing Magazine*, pages 59-85, March 1998.

[2] J.D. Mellot and F. Taylor. Very Long Instruction Word Architectures for Digital Signal Processing. In *Proceedings of the IEEE Conference on Acoustics Speech and Signal Processing*, pages 583-586, April 1997.

[3] B.R. Rau. Dynamically Scheduled VLIW Processors. In *Proceedings of the 26th International Symposium on Microarchitecture*, pages 80-90, December 1993.

[4] A Technical Evaluation by the Staff of Berkeley Design Technology, Inc. *Excerpts from Inside the StarCore SC140*, 2000.

[5] J. Fridman, Z. Greenfield. The TigerSHARC DSP architecture, *IEEE Micro*, Volume 20, Issue 1, January/February 2000.

[6] J. Turley and H. Hakkarainen. TI's New 'C6x DSP Screams at 1,600 MIPS. *The Microprocessor Report*, 11:14-17, February 1997.

[7] Texas Instruments. *TMS320C6X Optimizing C Compiler User's Guide*, March 2000.

[8] M.R. Thistle and B.J. Smith. A Processor Architecture for Horizon. In *Proceedings of Supercomputing*, Vol. 1, pages 35-41, November 1988.

[9] D.M. Tullsen, S.J. Eggers and H.M. Levy. Simultaneous Multithreading: Maximizing On-Chip Parallelism. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 392-403, June 1995.

[10] D.M. Tullsen, S.J. Eggers, J.S. Emer, H.M. Levy, J.L. Lo and R.L. Stamm. Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pages 191-202, May 1996.

[11] A. Bilas, J. Fritts and J. P. Singh. Real-Time Parallel MPEG-2 Decoding in Software. In *Proceedings of the 11th International Parallel Processing Symposium*, pages 197-203, Apr 1997.

[12] M.A. Bayoumi, editor. *Parallel Algorithms and Architectures for DSP Applications*. Kluwer Academic Publishers, September 1991.

[13] S.W. Keckler and W.J. Dally. Processor Coupling: Integrating Compile Time and Runtime Scheduling for Parallelism. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 202-213, May 1992.

[14] S. Kaxiras, G. Narlikar, A.D. Berenbaum and Z. Hu. Comparing Power Consumption of an SMT and a CMP DSP for Mobile Phone Workloads. In *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, pages 211-220, December 2001.

[15] E. Özer, T.M. Conte and S. Sharma. Weld: A Multithreading Technique Towards Latency-tolerant VLIW Processors. In *Proceedings of the 8th International Conference on High Performance Computing*, pages 192-203, December 2001.

[16] B.R Rau and J.A. Fisher. Instruction-Level Parallel Processing: History, Overview and Perspective. *The Journal of Supercomputing*, Volume 7, No. 1, pages 9-50, January 1993.

[17] Texas Instruments. *TMS320C6000 CPU and Instruction Set Reference Guide*, January 2000.

[18] N. H.E. Weste and K. Eshraghian, *Principles of CMOS VLSI Design*. Addison-Wesley Publishers, 1998.

[19] C. Lee, M. Potkonjak, W.H. Mangione-Smith. MediaBench: a tool for evaluating and synthesizing multimedia and communications systems. In *Proceedings of the 30th annual IEEE/ACM international symposium on Microarchitecture*, pages 330-335, 1997.

[20] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, R. B. Brown. MiBench: A free, commercially representative embedded benchmark suite. In *Proceedings of IEEE 4th Annual Workshop on Workload Characterization*, pages 3-14, December 2001.

[21] C. Lee, *UTDSP Benchmark Suite*, University of Toronto, Canada, 1992.

[22] P. Kohout. Hardware Support for Real-Time Operating Systems. *Master's Thesis*, University of Maryland (College Park), August 2002

[23] Texas Instruments. *TMS320C6201 Datasheet*, August 1998.

APPENDIX

In this section we present the results of all the benchmarks for both the SMT and variable hardware configurations.

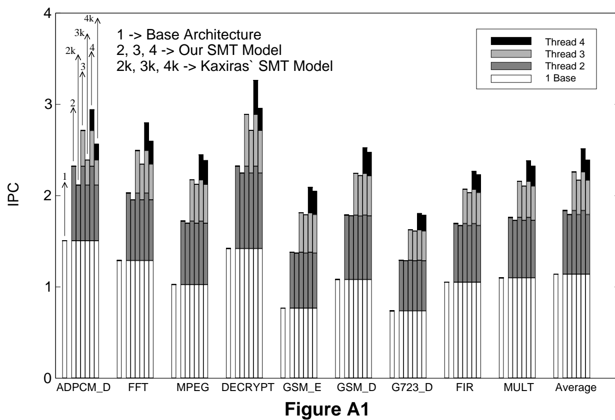


Figure A1

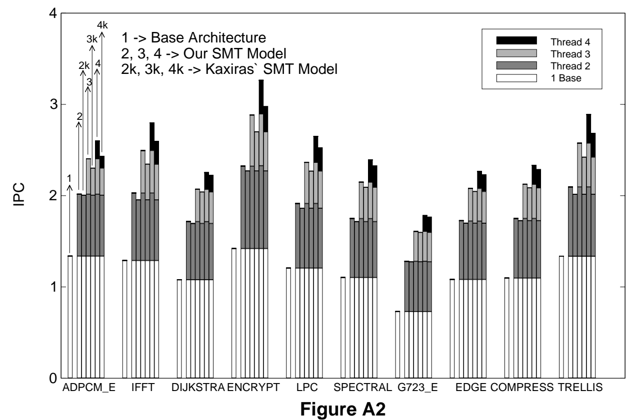


Figure A2

Processor Throughput (IPC) of SMT VLIW. Our Model vs. Kaxiras for Various Benchmarks

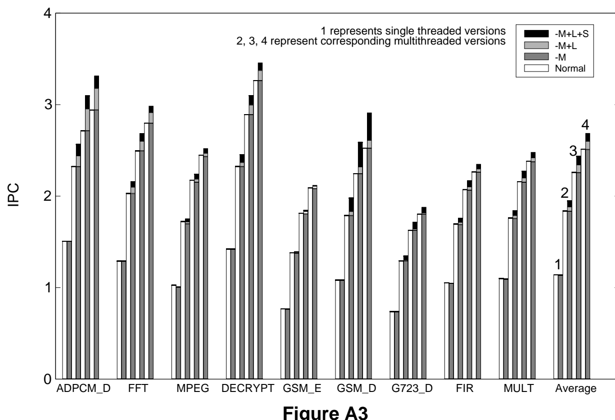


Figure A3

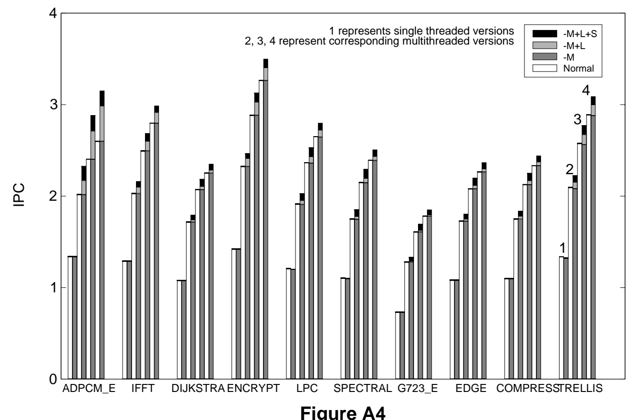


Figure A4

Processor Throughput (IPC) of SMT VLIW for Different Hardware Configurations