ICS

TECHNICAL REPORT

Functional Abstraction of Programmable Embedded Systems

Prabhat Mishra, Jonas Astrom, Nikil Dutt, and Alex Nicolau
{pmishra, astrom, dutt, nicolau}@ics.uci.edu
http://www.cecs.uci.edu/~aces

UCI-ICS Technical Report #01-04
Dept. of Information and Computer Science
University of California, Irvine, CA 92697

January 29, 2001

Information and Computer Science
University of California, Irvine

Functional Abstraction of Programmable Embedded Systems

Prabhat Mishra pmishra@cecs.uci.edu
Jonas Astrom astrom@cecs.uci.edu
Nikil Dutt dutt@cecs.uci.edu
Alex Nicolau nicolau@cecs.uci.edu

Architectures and Compilers for Embedded Systems (ACES) Laboratory
Center for Embedded Computer Systems, University of California, Irvine, CA 92697, USA

Technical Report #01-04
Dept. of Information and Computer Science
University of California, Irvine, CA 92697, USA

January 2001

Abstract

Rapid Design Space Exploration (DSE) of a processor-memory architecture is feasible using automatic toolkit (compiler, simulator, assembler) generation methodology driven by an Architecture Description Language (ADL). While many contemporary ADLs can effectively capture one class of architecture, they are typically unable to capture a wide spectrum of architecture and memory features present in DSP, VLIW, EPIC and Superscalar processors. The main bottleneck has been the lack of a functional abstraction underlying the ADL covering a diverse set of heterogeneous architectures. We present in this report the functional abstraction needed to capture such wide variety of programmable embedded systems. We demonstrate the power of this approach by specifying two very different processor-memory architecture using functional abstraction approach. We outline the automatic software toolkit generation from the given ADL description using functional abstractions. We show initial results of rapid design space exploration of architectures specified using functional abstraction based ADL approach.

Contents

List of Figures

List of Tables

1 Introduction

Contemporary processor architectures vary widely in terms of their architectural features. Program address generation and instruction dispatch features are widely used in DSP processors. VLIW processors use strong compiler support to ensure correct execution of long instruction words. Superscalar processors on the other hand, use hardware scheduling techniques, register renaming etc. Multimedia processors support SIMD operations. Furthermore, each architecture has a different type of branch prediction, different execution style - in-order/out-of-order, different ways of detecting hazards , different way of handling interrupts/exceptions and last but not the least different memory subsystems[16]. Emerging architectures have combined features of classical architectures (DSP, VLIW and Superscalar). For example, the Intel Itanium combines features of VLIW and superscalar; the TI C6x family combines features of DSP and VLIW. In order to allow rapid design space exploration of such heterogeneous processor-memory architectures, we need the ability to capture a wide variety of such architectural features. Moreover, during design space exploration using customized IP cores designers may want to add certain architectural features (some superscalar features to a VLIW processor core for example) to see how it impacts area, power, performance and other important design parameters. Similarly, to find the best match between the application characteristics and the memory organization features (caches, stream buffers, access modes, SRAM, DRAM etc.), the designer needs to explore different memory configurations in combination with different processor architectures, and evaluate each such system for a set of metrics (such as cost, power and performance) [16]. To enable this, designers need (i) a way of specifying wide variety of processor-memory features and (ii) automatic software toolkit generation to enable rapid design space exploration.

In this report we present a functional abstraction based specification technique using the EXPRESSION ADL [4], which is capable of capturing a wide variety of processor-memory architectures. The previous ADL-based approaches have, in general, been targeted towards a specific class of architectures, with limited descriptive facilities for complex memory organizations. EXPRESSION [4], on the other hand, is an ADL designed to capture a wide range of programmable architectures, including DSP, VLIW, and Superscalar, together with their distinct architectural features. This is possible due to the functional abstractions we have developed to support such an ADL-driven approach. Indeed, an ADL such as EXPRESSION critically needs the power of reuse in composing heterogeneous architectures using functional abstraction primitives; this facilitates rapid generation of software toolkits for a wide range of architectures, thus allowing effective design space exploration of heterogeneous processor-memory architectures.

The rest of the report is organized as follows. Section 2 outlines our approach and the overall flow of our environment. Section 3 surveys the contemporary processor-memory architectures. Section 4 presents the functional abstraction needed to capture the wide variety of architectural features and memory configurations. Section 5 illustrates how contemporary example architectures can be described using this functional abstraction. Section 6 shows initial results of design space exploration using this approach. Section 7 concludes the report.

2 Our Approach

Figure 1 shows the flow in our approach. In our IP library based Design Space Exploration (DSE) scenario, the designer starts by specifying the design using the functional abstractions using EXPRESSION ADL.

Section 3 surveys contemporary processor-memory architectures from each architecture domain viz., VLIW, DSP, Superscalar, EPIC, and RISC etc. We have studied the similarities and differences of each architectural feature in different architecture domain. Based on our observations we have defined the necessary generic functions, sub-functions and computational environment needed to capture wide variety of architecture and memory features. These parametric functions and sub-functions are described in a pseudo language. It is important to note that the generation of generic simulation models (our case in C++) is a one-time activity and independent of the processor-memory architecture. Section 4 describes the functional abstraction in detail.

The software toolkit including compiler, simulator, and assembler can be automatically generated from the ADL description using generic simulation models. The input application program is compiled and simulated and the feedback is used to modify the architecture specification.

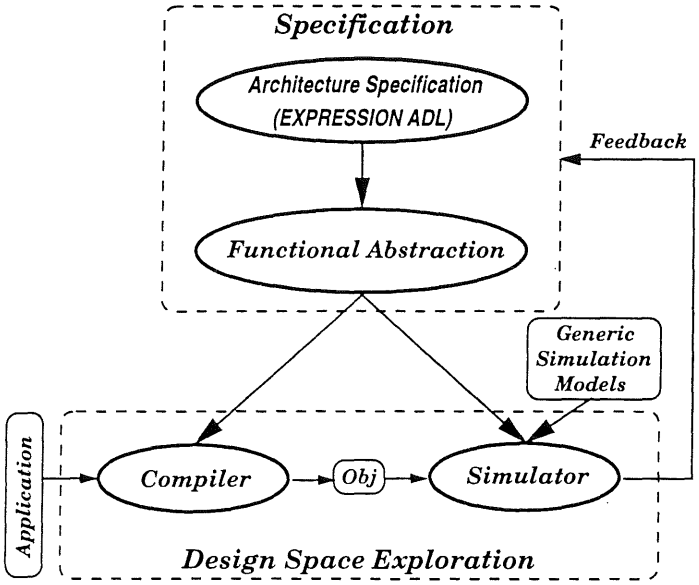

Figure 1. The Flow in our approach

3 Survey of Contemporary Processor-Memory Architectures

We have studied contemporary processor-memory architectures from each architecture domain viz., VLIW, DSP, Superscalar, EPIC etc. In this section we describe in detail the processor and memory features of the Intel Itanium, MIPS R4000, MIPS R10000, TI C6x and PowerPC. Section 3.6 summarizes the processor-memory features for different architectures. We conclude this section by summarizing the similarities and differences of the architectural features available in wide a variety of processor-memory architectures.

3.1 MIPS R4000

The MIPS R4000 [14] is a **RISC** microprocessor. It contains 32 general purpose 64-bit registers. When operating as a 32-bit processor, the general purpose registers are 32-bits wide. All instructions are 32-bits wide. The superpipeline design of the processor results in an execution rate approaching one instruction per cycle. Pipeline stalls and exceptional events are handled precisely and efficiently. The floating-point unit (FPU) is located on-chip and implements the ANSI/IEEE standard 754-1985. The processor block diagram is shown in Figure 2. The processor has eight pipeline stages:

1. IF - Instruction Fetch, First Half
2. IS - Instruction Fetch, Second Half
3. RF - Register Fetch
4. EX - Execution
5. DF - Data Fetch, First Half
6. DS - Data Fetch, Second Half
7. TC - Tag Check
8. WB - Write Back

The R4000 processor uses an on-chip TLB that provides rapid virtual-to-physical address translation. The primary instruction and data caches reside on-chip, and can each hold 8 Kbytes. Architecturally, each primary cache can be increased to hold up to 32 Kbytes. An off-chip secondary cache (R4000SC and R4000MC processors only) can hold from 128 Kbytes to 4 Mbytes. All processor cache control logic, including the secondary cache control logic, is on-chip.

3.2 Intel Itanium

The Intel Itanium [6] belongs to the **EPIC** category. It has combined features of **VLIW** and **Superscalar** processors with out-of-order execution. It maximizes performance via hardware and software synergy. Advanced features, e.g., prediction, speculation etc., enhance instruction level parallelism. It has 6-wide EPIC hardware under precise compiler control. It fetches upto six instructions per cycle and has hierarchy of branch predictors. It has dispersal of upto six instructions on 9 ports and has support for register remapping and register stack engine. It uses register read and bypasses to get the data, uses scoreboard and predicated dependencies. Figure 3 shows the processor block diagram. It has 10-stage in-order pipeline:

1. IPG - Instruction pointer generation.
2. FET - Fetch

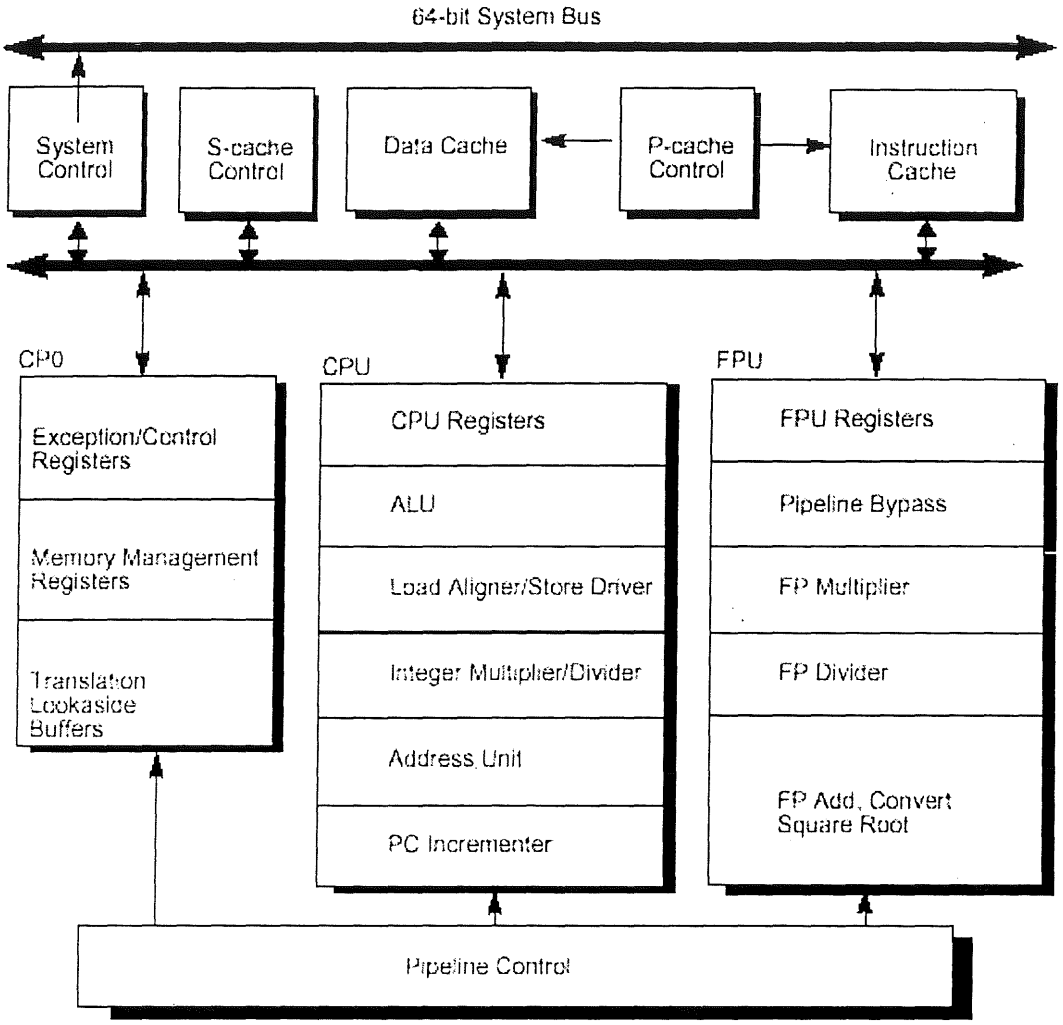

Figure 2. R4000 processor internal block diagram

3. ROT - Rotate
4. EXP - Expand
5. REN - Register rename
6. WLD - Word-line decode
7. REG - Register read
8. EXE - Execute
9. DET - Exception detect
10. WRB - Write back

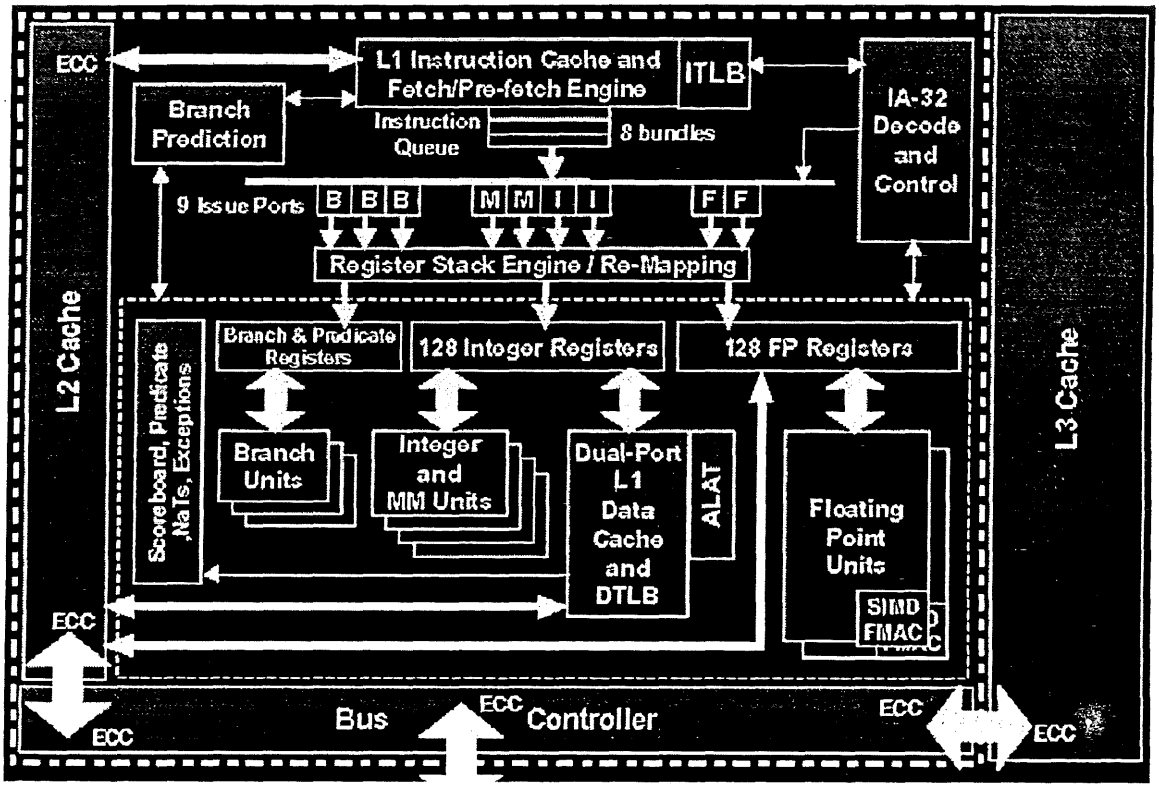

Figure 3. Intel Itanium Processor Block Diagram

It has 128 General registers, 128 Floating-point registers, 64 Predicate registers, 8 Branch registers, 128 Application registers and Instruction Pointer (IP) register. It has separate L1 caches for data and instruction. L2 cache is a combined one. Floating-point units interact directly with L2 cache.

3.3 MIPS R10000

The MIPS R10000 ([17], [20], [21]), is a dynamic, superscalar microprocessor that implements the 64-bit Mips-4 instruction set architecture. It fetches and decodes four instructions per cycle and dynamically issues them to five fully-pipelined, low-latency execution units. Instructions can be fetched and executed speculatively beyond branches. Instructions graduate in order upon completion. Although execution is out of order, the processor still provides sequential memory consistency and precise exception handling. With speculative execution, it calculates memory addresses and initiates cache refills early. It's hierarchical, nonblocking memory system helps hide memory latency with two levels of set-associative, write-back caches. To cope with the complexity of out of order superscalar processing, the R10000 uses a modular design that locates much of the control logic with in regular structures, including the active list, register map tables, and instruction queues.

R10000 fetches and decodes four 32-bit instructions per cycle. If one of these is a branch, its target address is calculated, the branch path is predicted, and instructions are speculatively fetched along the predicted path. Decoded instructions are put into a 32-entry *Active List* and three 16-entry instruction queues. The *Active List* keeps track of the original instruction order. The instruction queues dynamically issue each instruction to the appropriate execution unit after all its operands have become available. The *Floating-point Queue* issues instructions to the floating-point multiplier and adder. The *Integer Queue* issues instructions to two ALUs. The *Address Queue* issues instructions to the Load/Store unit (Address Calculation Unit and TLB) and the *Data Cache*. The *Address Calculation Unit* calculates 44 bit virtual memory addresses and TLB translates them to 40-bit physical addresses. Instructions graduate in order upon completion. Although execution is aggressively out-of-order, the processor still provides sequential memory consistency and precise exception handling.

Figure 4 shows the major blocks in the R10000 processor. Integer and floating-point register files each contain 64 physical registers. The integer register file has seven read ports and three write ports. The floating-point register file has five read and three write ports.

The instruction pipeline continues to fetch and decode instructions as long as there is room in the Active List and queues. When resource conflicts or operand dependencies prevent the queues from issuing instructions in their program order, the queue's dynamic scheduling hardwire tries to find other instructions that can be issued instead. For frequent operations, each execution unit is fully pipelined with a single-cycle repeat rate. The ALUs execute simple integer operations with single cycle latency, so that dependent instructions can be issued on consecutive cycles. The floating-point units has 3-stage pipelines, but special bypass logic reduces latency to only two cycles. Integer operands are loaded from the Data Cache with two cycle latency. Floating-point loads take an extra cycle of latency, because these units are physically farther from the Data Cache.

During instruction decode, integer and floating-point registers are renamed using separate mapping tables. This hardware handles almost any sequence of four instructions, including sequences with dependencies and instructions destined to the same functional units. Renaming maps 32 logical register numbers into 64 physical registers. The physical registers contain both committed and speculative values. When each instruction is decoded, its result is assigned to a physical reg-

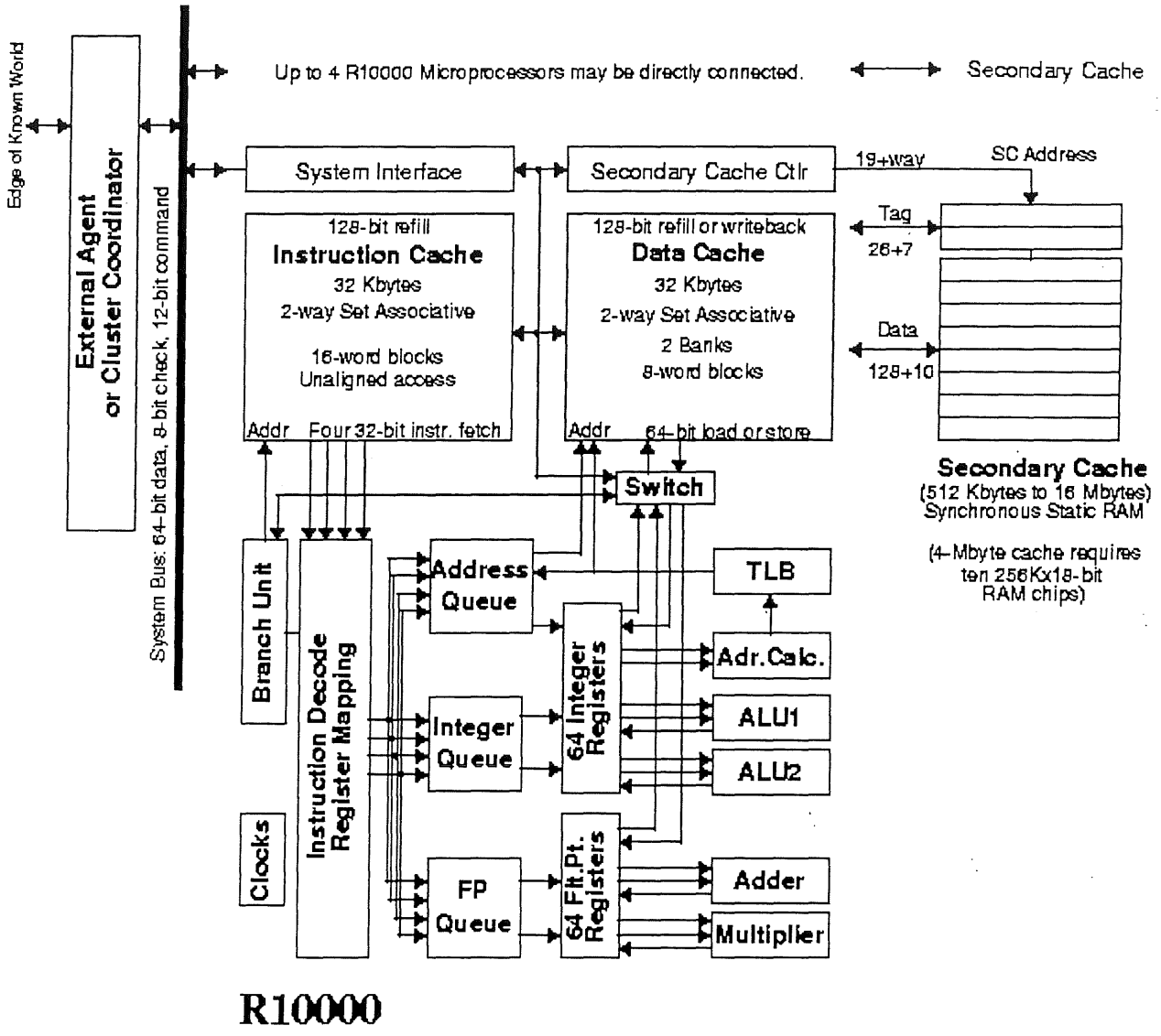

Figure 4. R10000 Microprocessor Block Diagram

ister from a *Free List* of currently unused registers. At graduation, this register contains a new committed value, and the previously assigned physical register is returned to the Free List. Thus, each physical register is uniquely associated with just one value; dependencies can be determined simply by comparing physical register numbers.

The direction taken by a conditional branch is predicted using a 2-bit algorithm, based on a 512 entry Branch History Table. Each prediction is verified as soon as its branch condition is determined. if its prediction was incorrect, all instructions fetched along the mis-predicted path are immediately aborted, and the processor state is restored from a 4-entry Branch Stack. This allows rapid recovery for up to four mis-predicted branches. Fetching along predicted paths may have initiated unneeded cache refills. However, the cache is non-blocking, and the correct path can be fetched while these refills are completed.

The integer queue issues instructions to the two integer arithmetic units: ALU1 and ALU2. The integer queue contains 16 instruction entries. Up to four instructions may be written during each cycle; newly decoded integer instructions are written into empty entries in no particular order.

The floating-point queue issues instructions to the floating-point multiplier and the floating-point adder. The floating-point queue contains 16 instruction entries. Up to four instructions may be written during each cycle; newly decoded integer instructions are written into empty entries in no particular order. The adders and multiplier are each fully pipelined with single-cycle repeat rate and latency of just two cycles.

The address queue issues instructions to the load/store unit. The address queue contains 16 instruction entries. Unlike the other two queues, the address queue is organized as a circular First-In First-Out (FIFO) buffer. A newly decoded load/store instruction is written into the next available sequential empty entry; upto four instruction may be written during each cycle. The FIFO order maintains the program's original instruction sequence so that memory address dependencies may be easily computed. Instructions remain in this queue until they have graduated; they cannot be deleted immediately after being issued, since the load/store unit may not be able to complete the operation immediately.

R10000 implements a nonblocking memory hierarchy with two levels of set-associative caches. It finds cache misses early, and begins refills in parallel with other useful work. The on-chip caches provide concurrent access for instruction fetch, data load and store, and refill. All caches least-recently-used(LRU) replacement algorithm. The Data Cache is 2-way interleaved with independent tag and data arrays for each bank. These four arrays operate under shared control of the Address Queue and the External Interface. The queue concurrently processes up to 16 load and store instructions in four separate pipelines. The primary cache consists of 2K doublewords. The secondary cache consists of 512K doublewords.

3.4 MPC7450: PowerPC Microprocessor

MPC7450 [9] microprocessors feature a high-frequency **superscalar** PowerPC core, capable of issuing four instructions per clock cycle (three instructions + branch) into eleven independent execution units:

1. Four integer units (3 simple + 1 complex)

2. Double-precision floating-point unit
3. Four Altivec [10] units (simple, complex, floating, and permute)
4. Load/Store unit
5. Branch processing unit

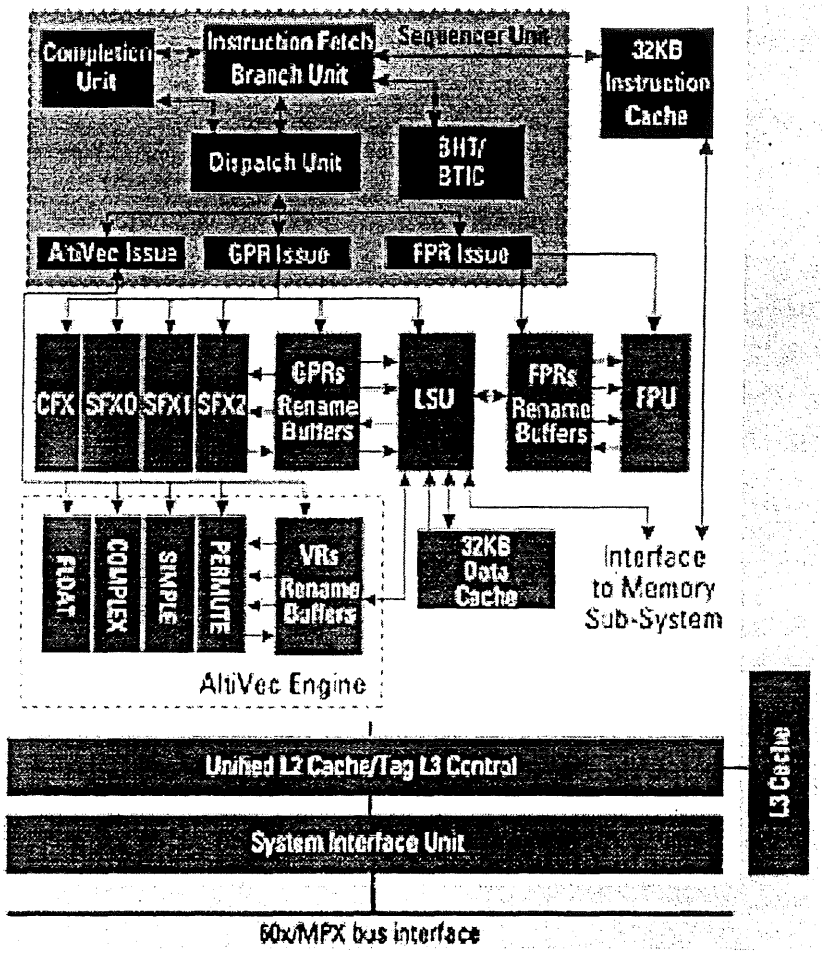

Figure 5. MPC7450 processor block diagram

Figure 5 shows the block diagram of the processor. It has separate 32KB, physically addressed instruction and data caches. Both L1 caches feature cache way locking and are eight-way set-associative. The L2 cache is on-chip with 256-bit interface to L1. This L2 cache is fully pipelined with 256 KB eight-way set-associative. It supports off-chip L3 cache up to 2MB.

3.5 TI C6x

The TI C6x ([19], [12]) has combined architectural features of **VLIW** and **DSP** processors. Figure 6 shows the block diagram of the TMS320C62x/C67x DSPs. It has separate program and data memory. The CPU contains:

1. Program fetch unit
2. Instruction dispatch unit
3. Instruction decode unit
4. Two data paths, each with four functional units
5. 32 32-bit registers
6. Control registers
7. Control logic
8. Test, emulation, and interrupt logic

The program fetch, instruction dispatch, and instruction decode units can deliver up to eight 32-bit instructions to the functional units every CPU clock cycle. The processing of instructions occurs in each of the two data paths (A and B as shown in Figure 7) each of which contains four functional units (.L, .S, .M, and .D) and 16 32-bit general purpose registers. It has on-chip configurable SRAM and off-chip DRAM with page/burst access modes. It can have upto 2 level of cache hierarchy.

3.6 Summary of Architectures Studied

Table 1 summarizes the processor-memory features for different architectures. Each row of the table corresponds to a architectural feature. Each column represents a architecture. We have used processors from different architecture domains - the MIPS R4000 [14] and StrongArm [7] are **RISC** processors; Motorola 56000 and TI C5x are **DSP** processors, TI C6x [12], MAP1000A [1], and Motorola StarCore [8] are **VLIW DSP** processors; MIPS R10000 [11], Motorola MPC7450 [9], Sun UltraSparc IIi [18], and DEC Alpha 21364 are **superscalar** processors; Intel IA-64 [6] and Sony Playstation [2] are **hybrid** processors. The Intel IA-64 architecture has combined features of **VLIW** and **Superscalar** processors with out-of-order execution. The Sony Playstation 2000 has a superscalar CPU core with VLIW co-processors. An entry in this table, TAB[F, A], represents the behavior of an architecture A towards an memory feature F. If an entry is marked x then that feature is supported by that architecture. If an entry is blank then the feature is either not supported or not applicable (or not known) to that architecture. An entry containing an integer number, n, means that features is supported n times. An entry containing a series , n-m, implies that the feature is supported for i times, where ($n <= i <= m$). Similarly, an entry containing a set, n,m, means that the feature is supported either n or m times. For example, the table entry with memory feature

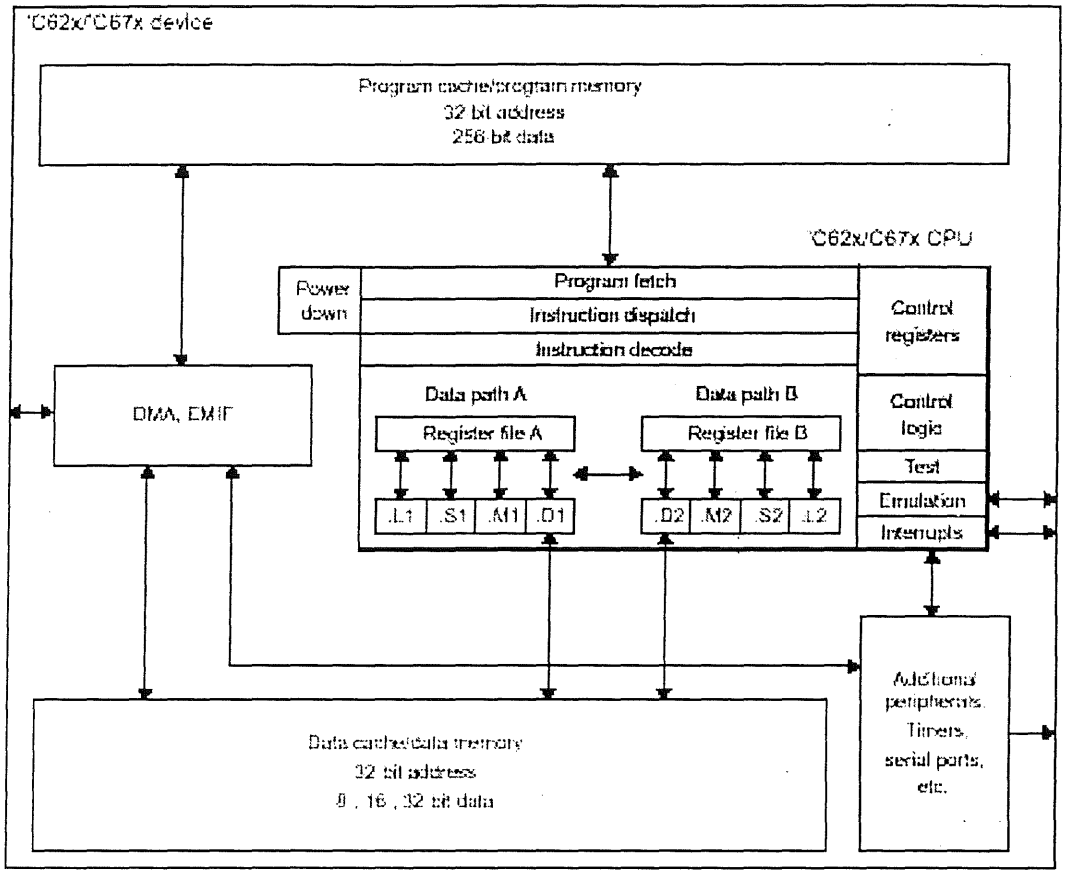

Figure 6. TI C6x processor block diagram

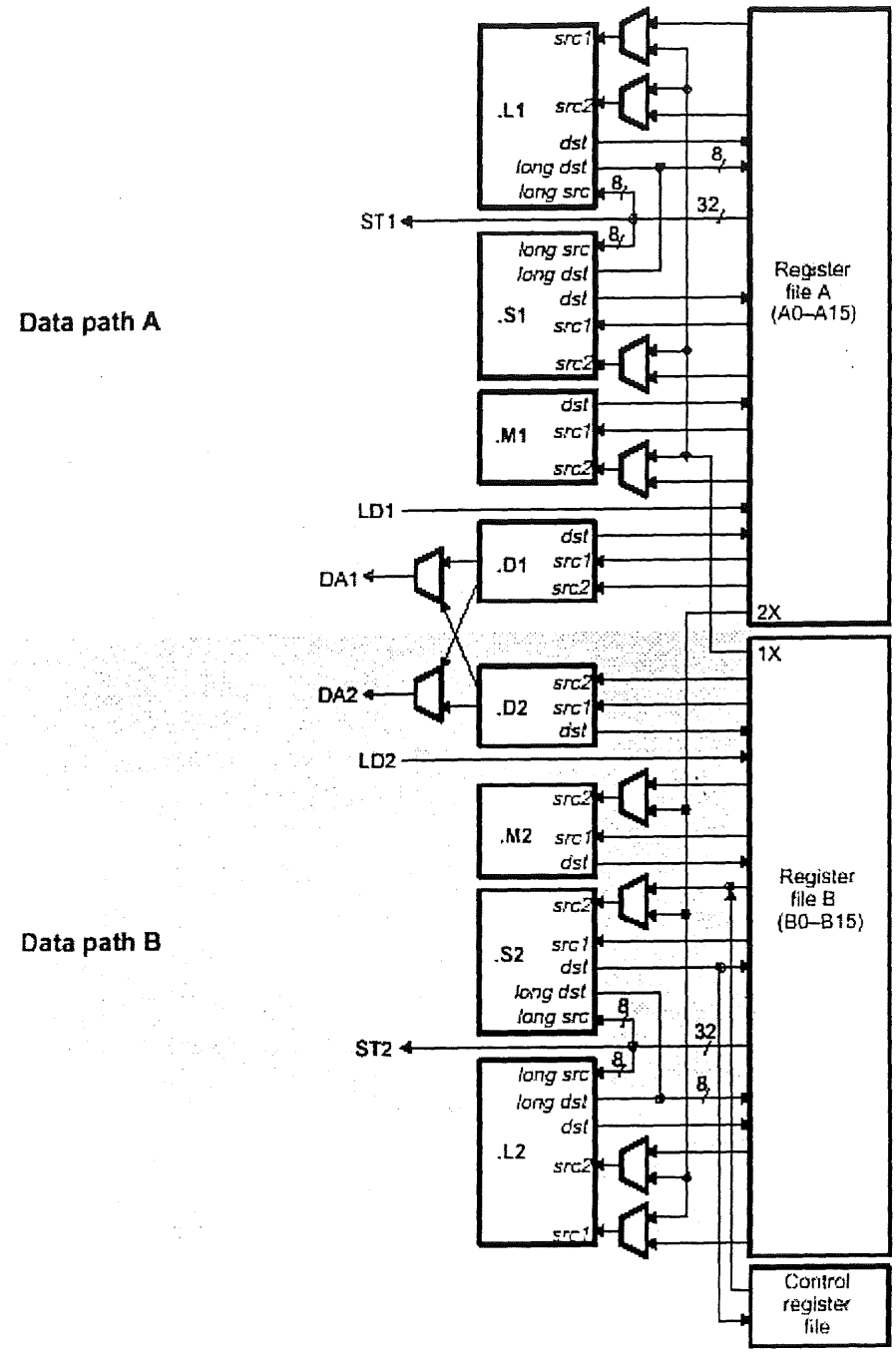

Figure 7. TI C6x CPU data paths

Levels of D-Cache and processor name *IA64* has value 3, this implies IA-64 has 3 levels of data cache. The row corresponding to *operand read in* has four types of values depending on where in the pipeline the operands are read. The values are - (D: Decode stage), (R: Read stage), (I: Issue stage), and (E: Execute stage). The row corresponding to *Branch Prediction* has values which indicate what method of branch prediction is employed in the respective architecture - (2b: 2-bit algorithm using branch history table), (BT: BTB based prediction), and (MA: dynamically choose among multiple algorithms based on local predictor table, global predictor table and branch history table).

3.7 Similarities and Differences

Broadly speaking, the structure of a processor consists of functional units, connected using ports, connections and pipeline latches. Major functional units are the PC unit, fetch unit, decode unit, branch prediction unit, issue unit, load store unit, TLB, execute unit and completion or writeback unit. Similarly, the structure of a memory subsystem consists of SRAM, DRAM, cache hierarchy etc. Although, a broad classification makes the architecture look similar, each architecture differs in terms of the algorithm it employs in branch prediction, the way it detect hazards, the way it handle exceptions etc. Moreover, each unit has different parameters for different architectures (e.g., number of fetches per cycle, levels of cache, cache line size etc.). Program address generation and instruction dispatch features are widely used in DSP processors. VLIW processors use strong compiler support to ensure correct execution of long instruction words. Superscalar processors on the other hand, use hardware scheduling techniques, register renaming etc. Multimedia processors support SIMD operations. The contemporary EPIC architectures uses predication and speculation to increase instruction level parallelism.

Depending on the architecture a functional unit may perform the same operation at different points in time. For example, read-after-write(RAW) followed by operand read happen in the decode unit for some architectures (e.g., DLX [5]), whereas in some others these operations are performed in the issue unit (e.g., MIPS R10K [20]). Some architectures even allow operand read in the execution unit. On the other hand, some architectures do not issue operations if RAW hazard is detected while others issue the operation in spite of RAW hazard (use snooping to read the data at execution stage using feedback paths). In other words, the same functionality is used at different point in the pipeline for different architecture.

We can observe some fundamental differences from the study above; the architecture may use:

1. the same functional or memory unit with different parameters
2. the same functionality in different functional or memory unit
3. new architectural features

The first difference can be eliminated by defining generic functions with appropriate parameters. The second difference can be eliminated by defining generic sub-functions which can be used by different architectures at different point of time. The last one is difficult to alleviate since it is

Architectures	RISC		DSP		VLIW DSP			Superscalar				Hybrid	
Processor-Memory Features	R4K	SA	56K	C5x	C6x	MA	SC	R10	MP	U3	α64	IA64	PS2
# of fetches/cycle	2	1	1	1	8	4	8	4	4	4	4	6	6
# of fetch stages	2	1	1	1	4		3	1	2	1	1	2	1-2
# entries in fetch RS													
# of decodes/cycle	2	1	1	1	8	4		4	3	4	4		6
# entries in decode RS									12			8	
# of issue units								3	3	1	3	3	
# of issues/cycle							6	5	6	4	6	6	6
# entries in issue RS								48	12		35		
# operations/instruction	1	1	1	1	8	4		1	1	1	1		1-16
# of parallel exec units					8	4	6	5	11		6		12
Branch Prediction								2b	BT		MA		BT
Feedback paths		x								x		x	
Operand read in	D	D	E	R	E	E	E	I	I	I	R	I	E
SIMD support						x			x	x		x	x
# entries in completion Q								32	16				
Register Renaming								x	x		x	x	x
Dynamic Scheduling								x	x	x	x	x	x
Speculation											x	x	
Predication						x						x	
# register files	2	1	3	1	3	3	2	2	3		3	5	4
# Coprocessors	3	1											2
# pipeline stages	8	5	3	4	3		5	5-7	7	9	6	10	6,9
Levels of D-Cache	1-2	1			0-2	1	0-2	2	3	2	2	3	1
cache prefetch		x								x		x	
cache hints												x	
On-chip SRAM		x			x	x	x			x			
configurable SRAM					x								
Off-chip DRAM	x	x	x	x	x	x	x	x	x	x	x	x	
page/burst mode		x			x								
Write Buffer		x								x	x		
Read Buffer		x											
Victim Buffer											x		
Stack			x	x			x						
FIFO						x							x
Z Buffer													x
On-chip DRAM													x
DMA		x			x	x	x				x		x
parallel mem transfers	1		2		2		2	1			2	2	
mem pipelining								x			x		

Table 1. Procesor-Memory features of different architectures. *R4K: MIPS R4000, SA: StrongArm, 56K: Motorola 56K, c5x: TI C5x, c6x: TI C6x, MA: MAP1000A, SC: Starcore, R10: MIPS R10000, MP: Motorola MPC7450, U3: SUN UltraSparc IIi, α64: Alpha 21364, IA64: Intel IA-64, PS2: Sony PlayStation 2000*

new, unless this new functionality can be composed of existing sub-functions. Section 4 presents the functional abstraction needed to capture the wide variety of architectural features and memory configurations.

4 Functional Abstraction

Functional abstraction allows the system designer to describe a wide variety of architectures in a hierarchical fashion. In this section we present functional abstraction by way of illustrative examples. We first explain the functional abstraction needed to capture the structure and behavior of the processor and memory subsystem, then we discuss the issues related to defining generic controller functionality, and finally we discuss the issues related to handling interrupts and exceptions.

4.1 Structure of a Generic Processor

Broadly speaking, the structure of a processor consists of functional units, connected using ports, connections and pipeline latches. Major functional units are the PC unit, fetch unit, decode unit, issue unit, execute unit and completion or writeback unit. Although, a broad classification makes the architecture look similar, each architecture differs in terms of the algorithm it employs in branch prediction, the way it detect hazards, the way it handle exceptions etc. Moreover, each functional unit has different parameters for different architectures (e.g., number of fetches per cycle etc.).

We capture the structure of each functional unit using parameterized functions. However, generic functions are not sufficient since each functional unit will perform a different function at different points of time depending on the architecture. For example, operand read and RAW hazard detection happens in the decode unit for some architectures whereas in some others these operations are performed in the issue unit. Some architectures even allow operand read in execution unit. On the other hand, some architectures do not issue operations if RAW hazard is detected while others issue the operation in spite of RAW hazard (use snooping to read the data at execution stage using feedback paths). Hence, there is a need for parametric sub-functions. Based on the observations made in Section 3 we have defined the key set of common functions, sub-functions and appropriate parameters from our study of wide variety of processor-memory architectures. In the following paragraphs we describe briefly some of the generic functions and sub-functions used in functional abstraction.

The program counter latch in the PC unit can be updated in three ways viz., initialization, through the branch unit or by a normal increment operation. Any functional unit may occupy single or multiple pipeline stage. For example, the fetch unit uses four pipeline stages for TI C6x whereas, it uses single stage for the Intel Itanium, MIPS R10K and other processors. The fetch unit may or may not have support for branch prediction. It may or may not have a reservation station. Depending on the architecture it will read a different number of operations per cycle. The number of operations it delivers to decode unit per cycle is also architecture dependent. We capture the structure of each functional unit using parameterized functions. The fetch unit functionality as shown below contains several parameters, viz., number of operations read per cycle, number of operations written per cycle, reservation station size, branch prediction scheme, number of read ports, number of write ports etc. While connecting the units these ports will be used.

In the following specific example, the fetch unit reads n operations from instruction cache using current PC address and writes them to the reservation station. It reads from reservation station m operations and writes them to the output latch (fetch to decode latch). It uses a BTB based prediction mechanism.

```
FetchUnit(......)
{
  address = ReadPC()
  Instructions = ReadInstMemory(address, n)
  WritetoReservationStation(Instructions, n)
  outInst = ReadFromReservationStation(m);
  WriteLatch(decode_latch, outInst)

  pred = QueryPredictor(address)
  IF pred
  {
     nextPC = QueryBTB(address)
     SetPC (next_PC)
  } ELSE
     IncrementPC(x)
}
```

As shown above, the fetch unit is described using sub-functions. Each sub-function is defined using appropriate parameters. The notion of generic sub-function allows the flexibility of specifying the system in finer detail. It also allows reuse of the components. These components can be pre-verified. So the task of verification will reduce to mainly performing interface verification at all levels. The concept of sub-function is necessary because the same functionality can be performed in different units of different architectures as described earlier.

We have defined sub-functions for all common activities e.g., ReadLatch, WriteLatch, ReadOperand, RenameRegister etc. Table 2 provides the list of common activities we have identified. The first column represents the name of the common function, the second column describes the activity, and the last column describes the input and output parameters of the function.

We have defined parameterized functions for all functional units viz., fetch unit, branch unit, decode unit, issue unit, execute unit, completion unit, PC Unit, Latch, Port, Connection etc. using sub-functions:

```
DecodeUnit( number of entries i the reservation station,
            number of input instructions,
            number of instructions decodable per cycle,
            number of operations read each cycle, ......)
{
    Data=ReadLatch(fetch_latch)
    Data=Reorder branch(Data)
    CompletionQInsertOperations(Data)
    RSInsertOperations(Data,m)
    RSReadOperand(r)

    UNTIL (e) instructions are issued or until queue is empty
    {
        Operation=RSReadOperation
        Resource=Unit_receive(Operation)
        IF Resource
        {
```

Table 2. The list of common sub-functions

Function Name	Description	Parameters
ReadLatch	Read a latch for n operations	Latch X, n, Data
WriteLatch	Write data to a latch	Latch, Data
QueryPredictor	Query prediction status	Branch address, status
QueryBTB	Query predicted address	Branch and memory address
UpdateBTB	Send address to branch unit	ID, target address
UpdatePredictor	Update branch predictor	ID, prediction type
BranchOther	Other branch address	ID, Address
IncrementPC	Increase PC with X	X, New PC
SetPC	New PC address X	X, New PC
ReadPC	Get PC	PC address
RSInsertOperation	Add one operation to RS	Operation
RSInsertOperations	Add X operations to RS	Operations, X
RSDeleteOperation	Dequeue operation from RS	ID
RSReadOperation	Read one operation from RS	Operation
RSReadOperand	Read n's operations operands	RS, n, RS
ReadOperand	Read one operand	Address bus, Reg name, Data
ReadOperands	Read all source operands	Operation X
WriteResult	Write operand	Data/Addr bus, Reg name, Data
MarkDestBusy	Mark Register busy	Register name
ReleaseDest	Unmark Register busy	Register name
CheckRAW	Check for RAW	Register name, status
CheckWAW	Check for WAW	ID, status
CheckWAR	Check for WAR	ID, status
IsUnitBusy	Is unit X busy	X, status
IsUnitStalled	Is unit X stalled	X, status
IsOperandRead	Is operand X read	ID, X, status
IsOperandsRead	Are all operands read?	ID, status
MarkOperandRead	Mark the operand as read	ID, X
HasUnitRS	Does unit X have RS?	X, status
SetUnitStalled	Set Stall bit for unit X	X, True/False
SetUnitBusy	Set Busy Bit for unit X	X, True/False
CompletionQDeleteOperation	Remove from completion queue	ID
ReadPredicate	Check predicate register X	Pred reg. X, status
WritePredicate	Set predicate register X to Y	Pred reg., value
ExecuteOperation	Execute an operation	Src1, Src2, func, Result
ExecuteBranchOperation	Execute branch	Src1-2, func, Result, Cmp_reg
MarkOperationDone	Mark operation done in comp queue	ID
IsOperationDone	Query if operation done	ID, status
CompletionQInsertOperation	Add operation to comp queue	Operation
CompletionQInsertOperations	Add operations to comp queue	Operations
CompletionQDeleteOperation	Delete an entry from comp queue	ID
FlushCompletionQ	Remove all operations above ID	ID
IsOperationValid	Query if operation is valid	ID, status
SetValidBit	Set valid bit to X for operation	ID, X
IsBranchAhead	Is there a branch ahead?	ID, status
CheckPredicate	Query ID's predicate	ID, status
IsBranchOperation	Is operation a branch?	ID, status
IsStoreOperation	Is operation a store?	ID, status
IsMapped	Is X in mapping table	Reg X, status
GetPhysicalRegister	For a logical register	Logical, physical reg
GetFreeRegister	Return a free physical reg	Register number
MapRegisters	logical to physical	Logical, physical reg
ComputeBusybit	check if unit is busy	Incoming operations, free entries, cycles left

```
            IF (IsBranchOperation(ID))
            {
                UpdateBTB(ID, dst)
            }

            RSDeleteOperation(ID)
            Add Read bit to instruction (true in case of immediate type)
            Generate ID for instruction

            IF register_renaming
            {
                RenameRegister(ID);
            }
            WriteLatch(Operation,Resource)
            ComputeBusybit()
        }
    }
}

ExecuteUnit(....)
{
    Data=ReadLatch(Decode_latch,m)
    IF (n>0)
    {
        RSInsertOperations(Data,m)
        RSReadOperand(r)
        Operation=RSReadOperation
        RSDeleteOperation (ID(operation))
    }ELSE
        Operation=Data

        IF predication
            IF ReadPredicate(Pred_reg)
                ExecuteOperation (Operation(OP code), Src1,Src2))
        ELSE
                ExecuteOperation (Operation(OP code), Src1,Src2))

    MarkOperationDone (ID)

    IF writeback
        IF !(CheckWAR) AND !(CheckWAW)
        {
            WriteResult(Result, Dst)
            ReleaseDest(Dst)
        }
        ELSE
            WriteLatch(output_latch,Result)

    ComputeBusybit
}


AddressCalculationUnit(....)
{
    Data=ReadLatch(Decode_latch,m)
    IF (n>0)
    {
        RSInsertOperations(Data,m)
        RSReadOperand(r)
        Operation=RSReadOperation()
        RSDeleteOperation (ID(operation))
    }ELSE
        Operation=Data
```

```
    result = AddMemAddressAndOffset(Operation);
    WriteLatch(output_latch,result)
    ComputeBusybit()
}

BranchUnit()
{
    Data=ReadLatch(Decode_latch,m)
    IF (n>0)
    {
        RSInsertOperations(Data,m)
        RSReadOperand(r)
        Operation=RSReadOperation
        RSDeleteOperation (ID(operation))
    }ELSE
        Operation=Data

    Dst,Cmp_reg=ExecuteBranchOpertion (Operation(OP code), Src1,Src2)

    B_result = UpdateBTB (ID, Cmp_reg(OP code))
    MarkOperationDone (ID)
    ComputeBusybit()
}

CompletionUnit(....)
{
    Traverse completion queue until s or b
    {
        IF CheckPredicate(ID)
        SetValidBit=True
        IF IsOperationDone(ID) and IsOperationValid(ID)
        {
            IF IsBranchOperation(ID)
                IF branch flag
                    Flag Flushing
            ELSE
                Activelist_writeback(ID)
            CompletionQDeleteOperation(ID)
            Branch_remove(ID)
        }ELSE
            stop traversing
    }
}

IssueUnit(...)
{
    Data=ReadLatch(Decode_latch,m)
    RSInsertOperations(Data,m)
    RSReadOperand(r)

    UNTIL (n) operations are read
    {
        op=RSReadOperation
        ReadOperands(op)
    }

    Until y instructions are tried to be issued
    {
        Operation=RSReadOperation
        Resource=Unit_receive(Operation)
        IF !!(resource)
        {
            WriteLatch(Resource_latch, operation)
            RSDeleteOperation(ID)
```

```
        }
    }

    ComputeBusybit()
}

WriteBackUnit(....)
{
    Data=ReadLatch(Decode_latch,m)
    IF (n>0)
    {
        RSInsertOperations(Data,m)
        RSReadOperand(r)
    }ELSE
        Operation=Data

    Until y instructions processed
    {
        Operation=RSReadOperation

        IF !(CheckRAW) AND !(CheckWAW)
        {
            WriteResult(Result, Dst)
            ReleaseDest(Dst)
            RSDeleteOperation(ID)
        }
    }
    ComputeBusybit()
}


MemoryController (no. of entries in the RS (Reservation Station) (n),
                  no. of  input instructions (m),
                  no. of load/store per cycle (s)/(l))
{
    Read latch for (m) entries
    Push entries into a list, initialize to _NEW.
    For each entry in the list evaluate the state machine until
    (s) or (l) has been reached:
    {
        _NEW:
            IF store operation
            {
                send write request to memory (value and address)
                IF acknowledged
                    Next state is _Done
                ELSE
                    Next state is _NEW
            }
            IF load operation
            {
                send read request to memory (address)
                IF acknowledged
                    Next state is _RETRIEVE
                ELSE
                    Next state is _NEW
            }

       _RETRIEVE:
            If data is ready
                Next state is _WRITEBACK
            ELSE
                Next state is _RETRIEVE
```

```
        _WRITEBACK
            Check for WAW
            Write received data to appropriate Register (i.e. bank and register)
            Release destination register
    }
}
```

We have also defined a few sub-functions e.g., RenameRegister, GraduateOperation using sub-functions to allow a finer granularity of architectural exploration. We present few sub-functions which are modeled using sub-functions.

```
RenameRegister( Instruction ID )
{
    // Rename Src1
    RenameReg(src1);
    // Rename Src2
    RenameReg(src2);

    // Rename dest
    if (store operation)
        RenameReg(dest);
    else if (not branch operation)
    {
        p_dest = GetFreeRegister();
        MapRegisters(p_dest,dest);
        MarkDestBusy(p_dest);
    }
}

RenameReg(Register src)
{
    IF IsMapped(src)
        p_src=GetPhysicalRegister(src)
    ELSE
    {
        p_src = GetFreeRegister();
        MapRegisters(p_src,src)
    }
}

ReadOperands(Instruction I)
{
    FOR all source operands S in I
    {
        IF  S is not already read and no RAW hazard
        {
            ReadOperand(S)
            MarkOperandRead(I, S);
        }
    }
}

ReservationStationReadOperand(RS)
{
    UNTIL (n) operations are read in RS
    {
        Read one operation I from RS
        ReadOperands(I);
    }
}
```

Now, we discuss few specific points pertaining to defining generic abstractions of different units. The execute unit can be single cycle or multi- cycle. Each execute unit can support different opcode functionalities. The list of opcode functions (described later) supported by a particular execute unit is passed as a parameter for the execute unit function.

Data hazard detection is done using sub-functions e.g., RAW_detect, WAR_detect and WAW_detect. These sub-functions are used by appropriate functional units depending on the architecture. For example, decode unit functionality uses these three sub-functions for DLX architecture whereas in certain architectures read-after-write(RAW) hazard occurs during operand read in issue units, WAW and WAR hazard detection happen in completion or writeback stage using program order buffer. For architectures with register renaming, only RAW hazard is possible.

Branch unit uses branch mis-prediction handler sub-function which specifies the actions to be performed viz., which are the operations to be allowed to graduate and which ones to flush.

The completion unit functionality along with program order completion queue can perform wide varieties of actions depending on the parameters values. Normally, completion queue maintains program order during out-of-order execution. During in-order or out-of-order execution this queue can be used to perform WAW and WAR checks. This program order information can be used for flushing the instructions selectively during branch mis-prediction or interrupts. This queue can further be used for enforcing in-order completion of branches. This may also be used for servicing synchronizing events, e.g., all memory writes are completed, all pending exceptions are reported etc.

The reservation station used in different functional unit can behave differently. For example, during in-order execution FIFO buffers are adequate whereas, for out-of-order execution linked list implementation is needed. This is because when an instruction is deleted, it creates space inside the reservation station. The next incoming instruction is inserted in that place. Now buffer has no order and it needs priority logic for deciding next outgoing instruction. Load Store issue unit or memory controller reservation station buffers are generally cyclic queues since the state of a load/store operations gets modified every cycle. Possible states are issued, address calculation done, TLB accessed, miss, retry etc.

4.2 Behavior of a Generic Processor

The behavior of a generic processor is captured through the definition of opcodes. Each opcode is defined as a function with generic set of parameters which performs the intended functionality. The parameter list includes source and destination operands, necessary control and data type information. We have defined common sub-functions e.g., ADD, SUB, SHIFT etc. The integer addition function is shown below.

```
int
IADD(int x, int y)
{
return (x + y);
}
```

The opcode functions use one or more sub-functions. For example, the MAC (multiply and accumulate) uses two sub-functions. As mentioned in previous section, these opcode functions are used as a parameter for the functional units.

4.3 Generic Controller

We define control in both distributed and centralized manner. While an instruction gets decoded the control information needed to select the operation, the source and the destination operands are placed in the output latch as shown in Figure 8. These decoded control signals pass through the latches between two pipeline stages unless they become redundant. For example, when the value for source1 is read that particular control is not needed any more, instead the read value will be in the latch. We have shown here only the control information of the latch. The latch contains data values and predicate registers (if applicable) as well.

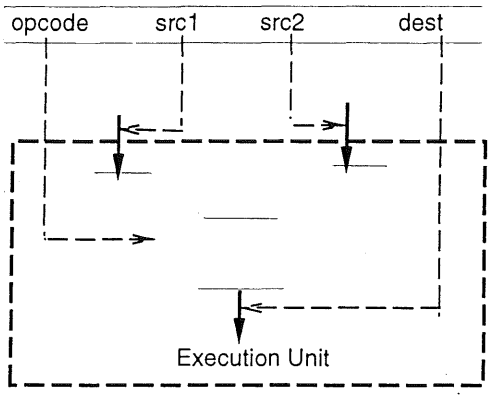

Figure 8. Example of distributed control

The centralized control is maintained by using a generic control table. The number of rows in the table is equal to the number of pipeline stages in the architecture. The number of columns is equal to the maximum number of parallel units present in any pipeline stage. Each entry in the control table corresponds to one particular unit in the architecture. It also contains information specific to that unit e.g., busy bit (BB), stall bit (SB), list of children, list of parents, opcodes supported etc.

The control table captures all the necessary details to perform selective or complete stalling of the pipelines. Stalling happens due to three kinds of hazards viz., structural hazards, data hazards and control hazards. In the following sections we briefly describe how we handle these three kinds of hazards using the control table.

1. **Structural Hazard**: Each unit marks the busy bit in the control table when the following conditions occur.

   - If it does not have sufficient space in reservation station to accommodate the number of incoming instructions possible in next cycle.
   - If it does not have a reservation station and executes a multi-cycle operation which is yet to complete.

In other words, a unit marks itself busy when it can not take any instruction from its parent unit in pipeline. A unit sets its stall bit (which means it will not be executed in the next cycle) when it does not have a reservation station and one of its children is busy or stalled.

2. **Data Hazard**: The detection of different data hazards viz., RAW, WAR and WAW occurs in different functional unit depending on the architecture as described in Section 4.1. The hazard detection sub-functions set the appropriate bits in control table.

3. **Control Hazard**: The branch unit sets the appropriate bits in control table when branch mis-prediction is detected. The sub-function for mis-prediction handling performs the necessary actions as described in Section 4.1

Pipeline stalling happens at the end of the cycle by the control unit in a bottom-up fashion, starting with the leaf level units and proceeding up to the fetch unit. This algorithm terminates when it reaches the fetch unit or when it reaches any stage where none of the units are busy or stalled. The same stalling algorithm resets the stall bit to zero for a particular unit when the stall condition does not hold anymore. A simplified version of the generic controller is shown below.

```
Control unit (...)
{
    IF startup
    {
        Write initial PC address to PC
    }
    ------------------------------------------------
    -- Stall and unstall mechanism--
    ------------------------------------------------
    Start at the leafs and traverse the tree upwards
    {
        IF no reservations station
            IF any child is stalled or busy
                Stall unit
        ELSE
            Not stall
    }
    --------------------------------
    -- Stall mechanism END--
    --------------------------------
    ------------------------------------------------
    --Flushing                                  ---
    ------------------------------------------------
    IF flushing flag is TRUE                // i.e. a branch is mispredicted
    {
        Query branch unit for other Branch address
        Update PC to other branch address
        Reset Busybit RF
        Reset Control table

        IF register_renaming
            Reset Mapping table
        IF free_list
            Add all registers to free list

        IF memorystore_buffer
            Reset memory buffers
        Update buffered memory
```

4.4 Interrupts and Exceptions

In this section we briefly describe the abstraction needed to capture the wide varieties exceptions and interrupts possible. Each exception is captured using appropriate sub-function. Opcode related exceptions (e.g., divide by zero), are captured in opcode functionality. Functional unit related exceptions (e.g., illegal slot exception), are captured in functional units. External interrupts (e.g., reset, debug exceptions), are captured in control unit functionality.

The interrupt handler unit services these exceptions. It has information regarding the priority of interrupts and which exceptions generate what interrupt. The generic interrupt handler has a parameterized priority table. The interrupt handler unit generates one particular interrupt based on priority. Before execution of that particular interrupt service routine, context saving and complete/partial flushing occurs. The specific types of flushing is decided by the semantics of that interrupt. Complete flushing clears the entire pipeline. Partial flushing means flushing only the instructions behind the interrupted instruction and allowing the previous instructions to continue using the program order information available in completion queue. Again, these actions are part of parametric sub-functions that allow a finer grain of microarchitechtural exploration.

4.5 Structure of a Generic Memory Subsystem

The memory represents a major bottleneck in modern embedded systems. Each type of memory module viz., SRAM, cache, DRAM, SDRAM, stream buffer, victim cache etc., is modeled using a function with appropriate parameters. For example, the cache function has parameters: cache size, line size, associativity (zero associativity implies direct cache), word size, replacement policy, write policy, read/write access times etc. These functions also have parameters for specifying pipelining, parallelism, access modes (normal read, page mode read, burst read etc.) etc. Again, each function is composed of sub-functions. For further details on generic memory subsystem, please refer to [15].

4.6 DMA Controller

The direct memory access (DMA) controller transfers data between regions in the memory map without intervention by the CPU. The DMA controller allows movement of data to and from internal memory, internal peripherals, or external devices to occur in the background of CPU operation. DMA controller function has following generic parameters:

- Block transfer: For each block transfer starting address of source memory, starting address of destination memory and the size of the block. Each block transfer can consist of multiple frames of a programmable size. Once a block transfer is complete, a DMA channel can automatically reinitialize itself for the next block transfer.

- Number of channels: The number of independent block transfers. Each channel can be one way or can be used to perform both the receive and transmit element transfers from or to a peripheral simultaneously, effective acting like two DMA channels. Each channel can be independently configured to transfer data values of different width (e.g., bytes, 16-bit halfwords, 32-bit words etc.)

- Programmable priority: Each channel has independently programmable priorities versus the CPU.

- Programmable address generation: Each channel's source and destination address registers can have configurable indexes for each read and write transfer. The address can remain constant, increment, decrement, or be adjusted by a programmable value. The programmable value allows an index for the last transfer in a frame distinct from that used for the preceding transfers.

- Events and Interrupts: Each read, write, or frame transfer may be initiated by selected events. On completion of each frame transfer or block transfer as well as on various error conditions, each DMA channel can send an interrupt to the CPU.

4.7 Coprocessor

The coprocessor is used to perform certain functionality which processor is is not able to handle or coprocessor might be optimized for that. The coprocessor function has similar parameters as generic execution unit function e.g., opcodes supported, single cycle or multi-cycle, single stage or multistage, instruction format supported, etc. It has few parameters which are unique to coprocessor. For example, it may have local memory. As a result before any computation the necessary data needs to be brought in using DMA controller and external memory interface (EMIF). Similarly, at the end of the computation the result needs to be written back to main memory using DMA controller and EMIF. The coprocessor can operate on local or main memory.

5 Contemporary Example Architectures

Using the functional abstraction approach outlined above, we have been able to describe the DLX, TI c6x, MIPS R4000, MIPS R10K, Itanium and PowerPC architectures representing a diverse set of processor-memory styles. In this section we describe how we capture two architectures having different processor and memory styles using our functional abstraction approach. MIPS R10K is a superscalar processor with two level of cache hierarchy. TI C6x is a hybrid processor containing both DSP and VLIW features with a novel memory organization (partitioned register file, cache hierarchy and configurable scratch pad SRAM).

5.1 MIPS R10K Architecture

The MIPS R10000 is a dynamic, superscalar microprocessor that implements the 64-bit Mips-4 instruction set architecture. It fetches and decodes four instructions per cycle and dynamically

issues them to five fully-pipelined, low-latency execution units. Instructions can be fetched and executed speculatively beyond branches. Instructions graduate in order upon completion. Although execution is out of order, the processor still provides sequential memory consistency and precise exception handling. With speculative execution, it calculates memory addresses and initiates cache refills early.

Figure 9 shows a simplified version of the R10K architecture. For illustration, we do not show control unit, completion queue (Active List), memory hierarchy and interrupt handler, branch predictor and few connections for clarity. Small rectangular boxes are pipeline latches. Each large labeled box is a functional unit, register file or memory. Small square boxes are ports and lines are connections. In this section we outline how we specify MIPS R10K using the functional abstraction described in Section 4.

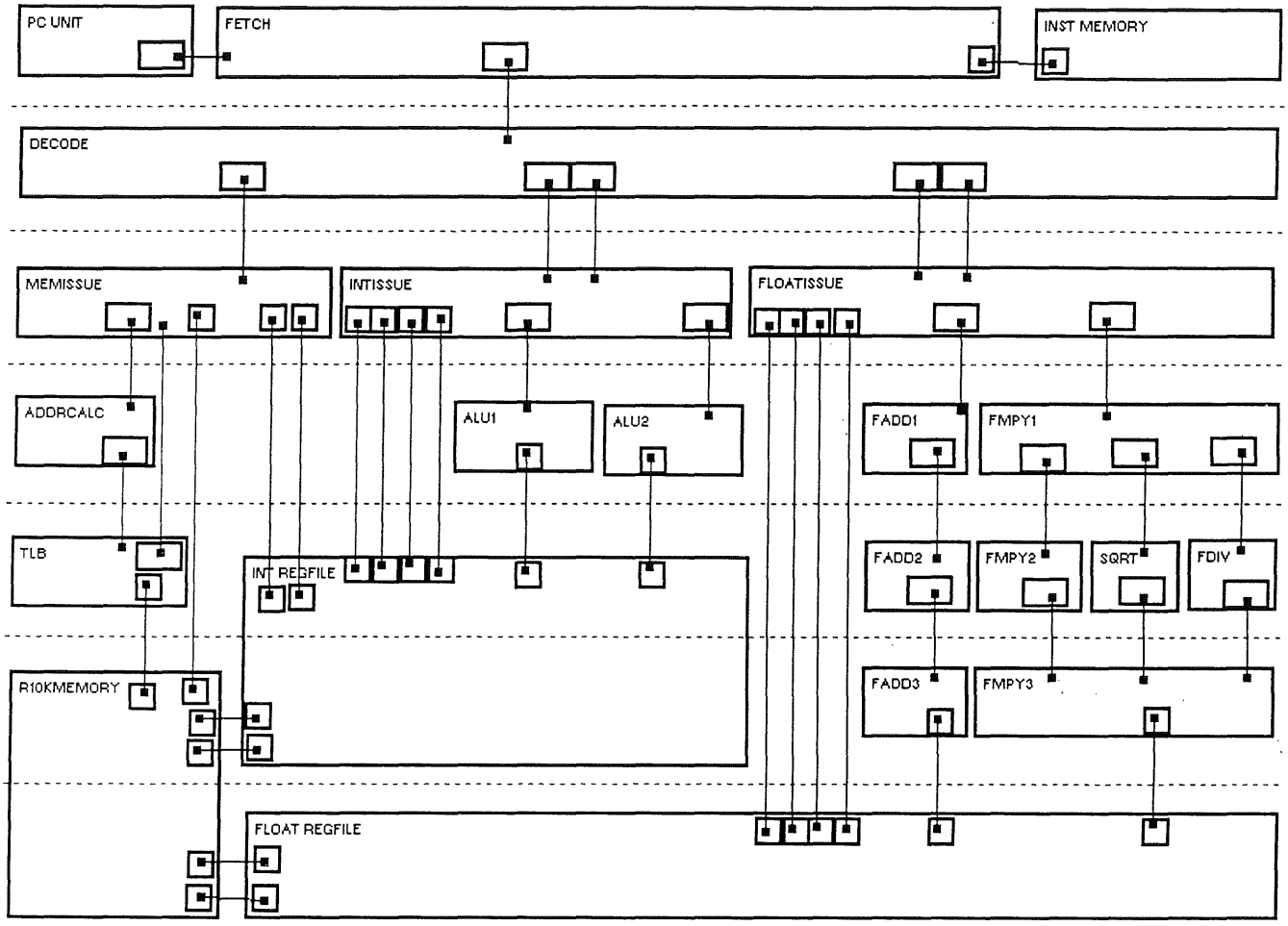

Figure 9. Simplified R10K architecture

The fetch unit function is invoked with three connections initialized viz., input from PC Latch, input from instruction memory and output to decode latch. Both the number of instructions fetched per cycle and number of instructions sent to decode stage per cycle are set to four. The number of

entries in reservation station is set to zero. The number of entries in completion queue (Active List in R10K terminology) is set to 32.

The decode functionality is instantiated with read connection from fetch latch and write connections to MemIssue, IntIssue and FloatIssue units. It uses the register renaming sub-function while decoding instructions. It inserts the decoded instruction in the completion queue (ActiveList) which maintains the program order. The decode logic decides where to dispatch (MemIssue, IntIssue or FloatIssue) a particular instruction based on the opcode supported by those issue units. As mentioned in Section 4, the control table has the information regarding the supported opcodes by a particular unit. Table 3 shows the control table for a simplified MIPS R10K architecture as shown in Figure 9, where the rows indicate the pipeline stages and columns represent parallel functional units. For example, the table entry for the third row and the second column corresponds to IntIssue unit with both busy bit and stall bit value zero.

Table 3. Control table for R10K architecture

		Fetch BB:0 SB:0		
		Decode BB:0 SB:0		
MemIssue BB:0 SB:0	IntIssue BB:0 SB:0		FloatIssue BB:0 SB:0	
AddrCalc BB:0 SB:0	ALU1 BB:0 SB:0	ALU2 BB:0 SB:0	FADD1 BB:0 SB:0	FMPY1 BB:0 SB:0
TLB BB:0 SB:0	FADD2 BB:0 SB:0	FMPY2 BB:0 SB:0	SQRT BB:0 SB:0	FDIV BB:0 SB:0
	FADD3 BB:0 SB:0		FMPY3 BB:0 SB:0	

The IntIssue, FloatIssue and MemIssue functions are instantiated with a reservation station size of 16 entries. Each issue unit performs operand read and RAW hazard detection (using appropriate sub-functions) before performing out-of-order issue. The reservation stations are of different nature for the different functional units. The MemIssue unit uses a circular buffer as its reservation station whereas the IntIssue and FloatIssue uses a buffer where instructions are inserted in empty slots (not in any order) and retrieved using priority logic.

Execution units are instantiated using appropriate opcode functionalities. The Address Queue (MemIssue unit) reads data and tag using virtual address while the physical address is computed. It checks whether load is a hit or miss once the physical address is available. This is different than the conventional way of hit or miss detection. In conventional architectures the load request is done using physical address and hit or miss detection is done inside the memory subsystem. Due to our sub-function based abstraction approach, we are able to re-use the hit or miss detection sub-function in the processor side (this remains conventionally in the memory).

The integer and floating-point register files are instantiated using generic register file with data widths and data type parameters. The memory hierarchy consists of two levels of cache. The parameters for the primary cache are: associativity - 2, cache size - 2K double-words, line size - 4 , word size - 64 bits, replacement - LRU, write policy - write back, number of lines - 512, access time - 1 cycle. Similarly secondary cache functionality is instantiated with the appropriate parameters viz., associativity - 2, cache size - 512K double-words, line size - 16, word size - 64 bits, replacement - LRU, write policy - write back, number of lines - 32K, access time - 2 cycles.

Each functional unit invokes the appropriate sub-functions to capture exception conditions. The interrupt handler function captures the priority table of 19 interrupts. In this manner we are able to concisely capture a state-of-the-art dynamic superscalar architecture using functional abstraction approach.

5.2 TI C6x Architecture

We now demonstrate the ability to capture a hybrid VLIW/DSP architecture using our functional abstraction technique. Figure 10 shows a simplified model of the TI C6x architecture. Small rectangular boxes are pipeline latches. Each large labeled box is a functional unit, register file or memory. Small square boxes are ports and lines are connections.

The fetch functionality consists of four stages viz., program address generation, address send, wait, and receive. Each of the four stages is modeled using respective sub-functions with appropriate parameters. The architecture fetches one VLIW instruction (eight parallel operations) per cycle.

The decode function decodes the VLIW word and dispatches upto eight operations per cycle to eight execution units. Each execution unit performs operand read and hazard checks (using sub-functions). At the end of computation each execution unit writes back (using sub-function) the result to register file. Each execution unit is instantiated with appropriate opcode functions as parameters. For example, L unit (L1 and L2) uses 32/40-bit arithmetic and compare operations for fixed-point mode, and arithmetic and conversion operations for floating-point mode.

The functional units, L1, S1, M1 and D1 are connected to the "A" part of the partitioned register file whereas the remaining functional units viz., L2, S2, M2, D2 are connected to the "B" of the register file. Two cross paths, viz., 1X and 2X, are used for transferring data from the other part of the partitioned register file. Each register file is instantiated using generic register file with 16 32-bit registers.

The TI C6x architecture has a novel memory organization (Figure 11, comprised of a 2-level cache hierarchy and a programmable SRAM space. The L1 program cache is 4K bytes, direct mapped with line size 64 bytes. The L1 data cache is 4K bytes, associativity 2 and line size 32 (bytes). The L2 cache is 64K bytes and depending on the mode of configuration, the memory space is divided between SRAM and associative cache. Memory modules are instantiated with appropriate parameters for capturing the memory subsystem.

Each functional unit invokes the appropriate sub-functions to capture exception conditions. The interrupt handler function captures the priority table of 14 interrupts. Reset and NMI has higher priority than INT4 to INT15 interrupts. Thus we are able to capture a hybrid DSP/VLIW architecture using our functional abstraction approach.

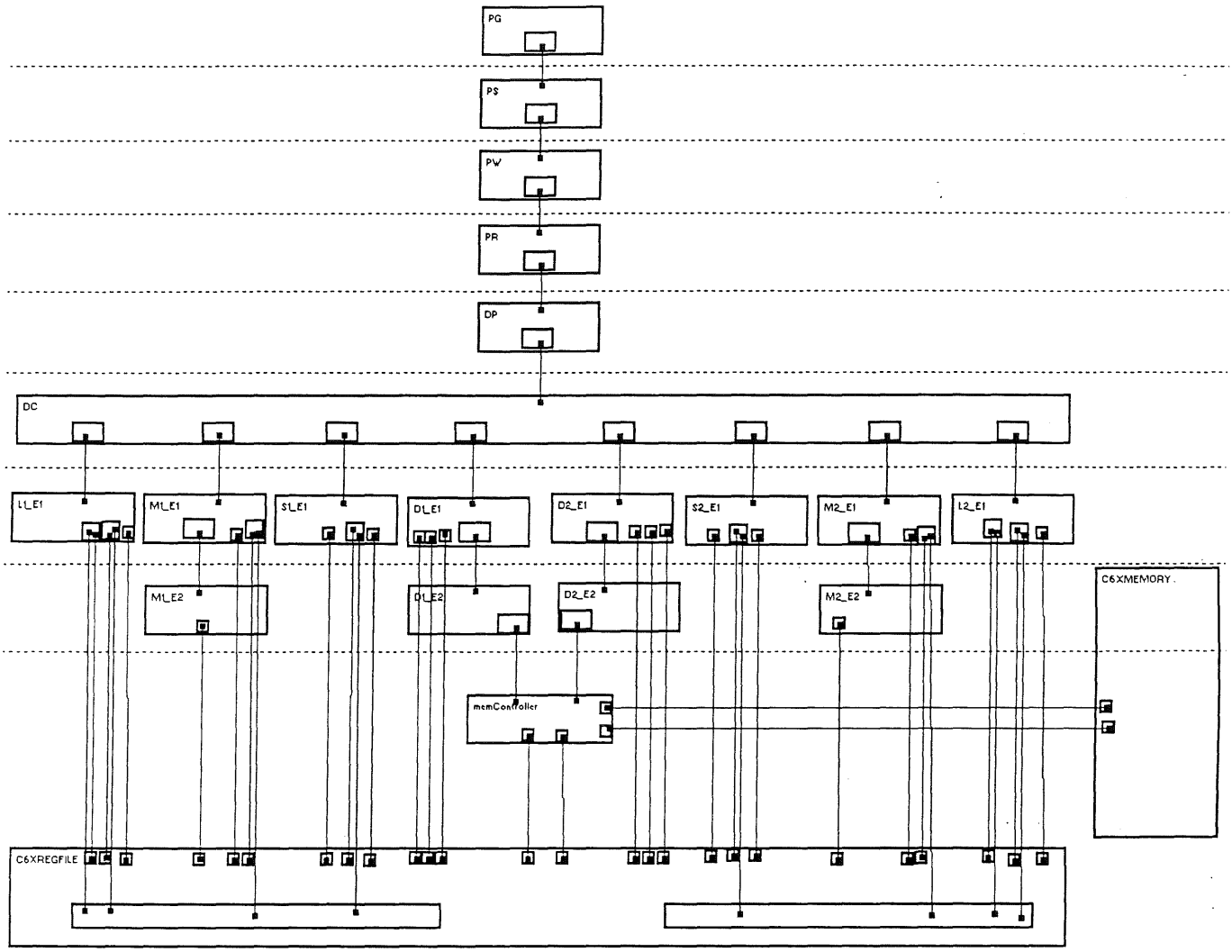

Figure 10. Simplified TI C6x architecture

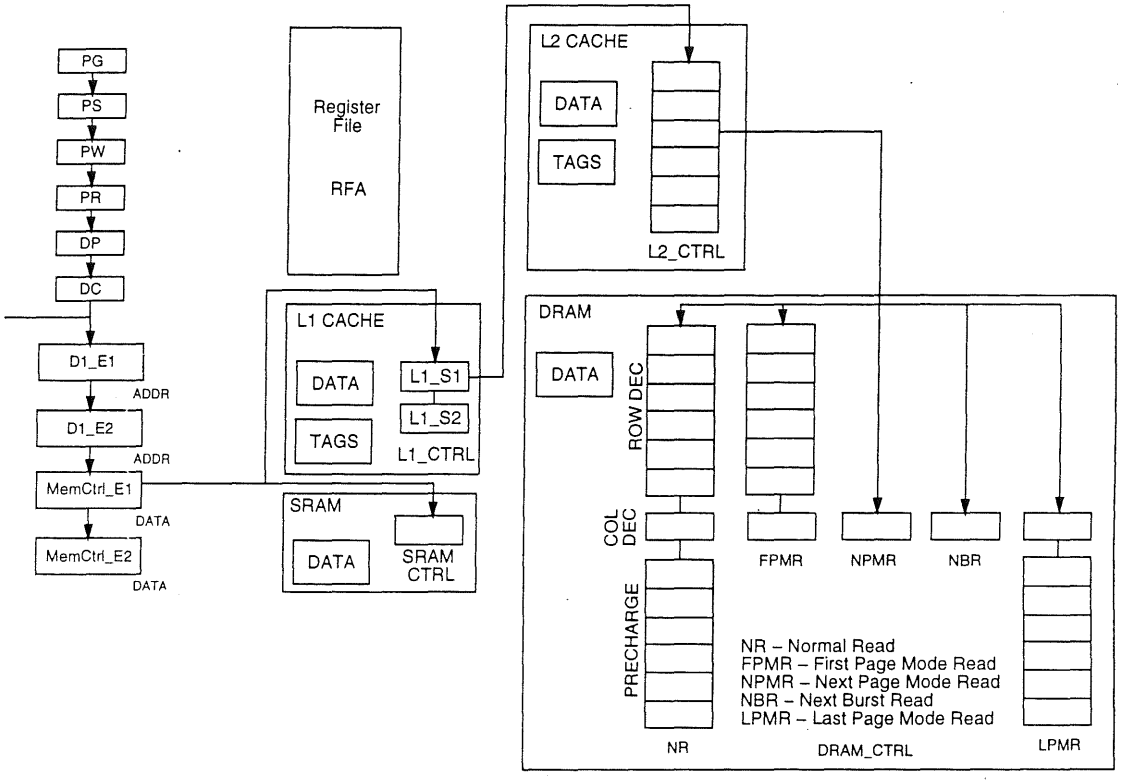

Figure 11. Simplified TI C6x architecture with novel memory organization

6 Experiments

In this section we show the utility of the functional abstraction scheme by performing design space exploration of a TI C6x based architecture. We vary several architectural features, including memory configurations. We describe our initial design space exploration results by generating a retargetable software toolkit from using the functional abstraction approach. Based on feedback from design space exploration results, designers can modify the original specification to reduce bottlenecks. These modifications can be quite drastic, for instance the original VLIW-like architecture can become superscalar after few iterations. This is possible only due to the fact that an ADL such as EXPRESSION can capture wide spectrum of processor-memory architectures using functional abstractions.

Using the functional abstraction approach and the generic simulation models as shown in Section 1 we have generated a software toolkit, including a compiler and simulator, for the TI c6x architecture. In this section we demonstrate the design space exploration capability using different memory configurations, starting from the base TIC6211 processor architecture, with the goal of studying the trade-off between cost and performance.

6.1 Experimental Setup

The memory organization of the TIC6211 is varied by using an L1 cache, L2 cache, an off-chip DRAM module, and an on-chip SRAM module. The L1 cache is a 2-way set associative cache with line size of 4 words and word of 4 bytes. The L2 cache shares a total of 2K on-chip SRAM memory with the direct mapped on-chip SRAM.

We used a set of benchmarks from the multimedia and DSP domains, and compiled them using the generated EXPRESS compiler[3]. We collected the statistics information using the generated SIMPRESS [13] cycle-accurate structural simulator, which models both the TI6211 processor and the memory subsystem.

The configurations we experimented with are presented in Table 4. The numbers in Table 4 represent: the size of the memory module (e.g., the size of L1 in configuration 1 is 128), the cache/stream buffer organizations: $num\_lines \times num\_ways \times line\_size \times word\_size$, the latency (in number of processor cycles), and the replacement policy (LRU or FIFO).

The configurations in Table 4 are presented in increasing order of the cost in terms of area. The first configuration contains the L1 cache and an on-chip direct mapped SRAM of 2K. Some of the arrays in the application are mapped to the SRAM. Due to the reduced control necessary for the SRAM, it has a small latency (of 1 cycle), and the area requirements are small. The second configuration contains L1 and L2 caches with FIFO replacement policy. Due to the control necessary for the L2 cache (of size 2K), the cost of this configuration is larger than the configuration 2 containing the SRAM. Configuration III is the same as configuration 3, but with LRU replacement policy for the L1 and L2 caches. Due to the more complex control required by the LRU policy, the cost of this configuration is larger than configuration II. Configuration IV contains an L1 cache, an L2 cache of size 1K and a direct mapped SRAM of size 1K. Due to the extra busses to route the data to the caches and SRAM, this configuration has a larger cost than the previous one. All the configurations contain the same off-chip DRAM module with a latency of 20 cycles.

Table 4. The memory subsystem configurations

Config	L1 Cache	L2 Cache	SRAM	DRAM
I	4x2x4x4 lat=1 (LRU)	-	2K lat=1	lat=20 cycle
II	4x2x4x4 lat=1 (FIFO)	16x4x8x4 lat=4 (FIFO)	-	lat=20 cycle
III	4x2x4x4 lat=1 (LRU)	16x4x8x4 lat=4 (LRU)	-	lat=20 cycle
IV	4x2x4x4 lat=1 (FIFO)	32x1x8x4 lat=4 (FIFO)	1K lat=1	lat=20 cycle

6.2 Results

Figure 12 presents a subset of experiments we ran, showing the total cycle counts (including the time spent in the processor) for the set of benchmarks for different memory configurations attached to the TIC6211 processor. From the experiments we performed, we chose a representative set of benchmarks, which show the different trends in the cost versus performance trade-off. Even though these benchmarks are kernels, we observed a significant variation in the trends shown by the different applications.

For instance, in FirstMin and Linear the first configuration even though has the lowest cost performs the best (lower cycle count means higher performance), due to the fact the most frequently used data fits in small SRAM. The expected trend of higher cost - higher performance was apparent in the applications 2DPartPusher, compress, and lowpass. In this manner, we are able to use our functional abstraction based Design Space Exploration approach to obtain design points with varying cost and performance. The designer is also able to explore the effects of employing hybrid architectural features.

7 Conclusion

This report proposed a functional abstraction based design space exploration methodology which is capable of capturing a wide variety of processor and memory architectures. Rapid design space exploration can be performed by generating the software toolkit automatically. Functional abstraction can be extracted from an ADL description as well. Generic function based design space exploration allows designers to make fast design decisions and reuse the generic components. Hence the problem of design verification in this methodology reduces to interface verification, since each generic function can be pre-verified.

Our ongoing work targets the use of this functional abstraction based design space exploration by generating synthesized hardware automatically. Furthermore, we plan to extend this specification technique to generate FSM automatically and perform property checking during rapid design space exploration driven by EXPRESSION ADL [4].

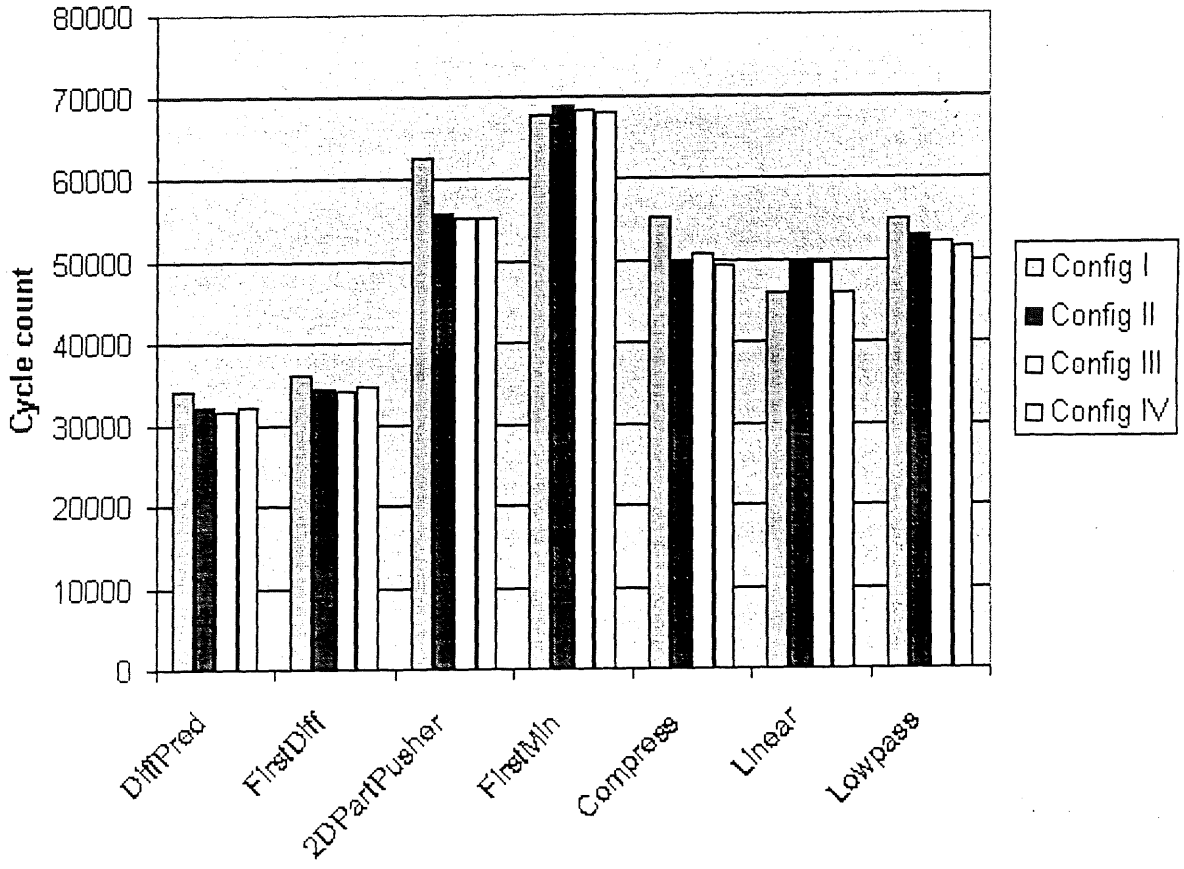

Figure 12. Cycle counts for the memory configurations

8 Acknowledgements

This work was partially supported by grants from NSF (MIP-9708067), DARPA (F33615-00-C-1632) and a Motorola fellowship. We would like to gratefully acknowledge Ashok Halambi, Peter Grun, Srikanth Srinivasan, and all other EXPRESSION team members for their contribution to the functional abstraction work.

References

[1] J. S. O. Chris Basoglu, Woobin Lee. *The MAP1000A VLIW Mediaprocessor*, 2000.

[2] K. Diefendorff. Sony's emotionally charged chip. *Microprocessor Report*, 13(5):1–7, 1999.

[3] A. Halambi, N. Dutt, and A. Nicolau. Customizing software toolkits for embedded systems-on-chip. In *DIPES 2000*, 2000.

[4] A. Halambi, P. Grun, V. Ganesh, A. Khare, N. Dutt, and A. Nicolau. EXPRESSION: A language for architecture exploration through compiler/simulator retargetability. In *Proc. DATE*, Mar. 1999.

[5] J. Hennessy and D. Patterson. *Computer Architecture: A quantitative approach.* Morgan Kaufmann Publishers Inc, San Mateo, CA, 1990.

[6] http://developer.intel.com/design/ia-64/architecture.htm. *IA-64 Architecture.*

[7] http://developer.intel.com/design/strong/sa1100.htm. *StrongARM Processors.*

[8] http://www.lucent.com/micro/Starcore. *Starcore, Next Generation DSPs.*

[9] http://www.motorola.com/SPS/PowerPC. *MPC7400 PowerPC Microprocessor.*

[10] http://www.motorola.com/SPS/PowerPC/Altivec. *Altivec: Motorola's high-performance vector parallel processing expansion to the PowerPC^{TM} architecture.*

[11] http://www.sgi.com/processors/r10k. *MIPS R10000 Microprocessor.*

[12] http://www.ti.com/sc/docs/products/dsp/C6000/index.htm. *TMS320C6000^{TM} Highest Performance DSP Platform.*

[13] A. Khare, N. Savoiu, A. Halambi, P. Grun, N. Dutt, and A. Nicolau. V-SAT: A visual specification and analysis tool for system-on-chip exploration. In *Proc. EUROMICRO*, 1999.

[14] MIPS Technologies, Inc. *MIPS R4000 Microprocessor User's Manual*, 1994.

[15] P. Mishra, P. Grun, N. Dutt, and A. Nicolau. Memory subsystem description in EXPRESSION. Technical Report UCI-ICS 00-31, University of California, Irvine, 2000.

[16] P. Mishra, P. Grun, N. Dutt, and A. Nicolau. Processor-memory co-exploration driven by an architectural description language. In *Intl. Conf. on VLSI Design 2001*, Bangalore, India, 2001.

[17] SGI - MIPS R10000 Superscalar Microprocessor. *http://www.sgi.com/processors/r10k.*

[18] SUN Microsystems. *UltraSPARC IIi User's Manual*, 1997.

[19] Texas Instruments. *TMS320C6201 CPU and Instruction Set Reference Guide*, 1998.

[20] N. Vasseghi, K. Yeager, E. Sarto, and M. Seddighnezhad. 200-mhz superscalar risc microprocessor. *IEEE Journal of Solid-State Circuits*, 31(11):1675–1686, November 1996.

[21] K. Yeager. The mips r10000 superscalar microprocessor. *IEEE Micro*, 16(2):28–40, April 1996.