ICS

TECHNICAL REPORT

Specification of Hazards, Stalls, Interrupts, and Exceptions in EXPRESSION

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

UCI-ICS Technical Report #01-05
Dept. of Information and Computer Science
University of California, Irvine, CA 92697

January, 2001

Information and Computer Science
University of California, Irvine

Specification of Hazards, Stalls, Interrupts, and Exceptions in EXPRESSION

Prabhat Mishra
pmishra@ics.uci.edu

Nikil Dutt
dutt@ics.uci.edu

Alex Nicolau
nicolau@ics.uci.edu

Architectures and Compilers for Embedded Systems (ACES) Laboratory
Center for Embedded Computer Systems
University of California, Irvine, CA, USA
http://www.cecs.uci.edu/~aces

Technical Report #01-05
Dept. of Information and Computer Science
University of California, Irvine, CA 92697, USA

January 2001

Abstract

Recent efforts in language-driven Design Space Exploration (DSE) use Architectural Description Languages (ADL) to capture the processor-memory architecture, generate automatically a software toolkit (including compiler, simulator, assembler) for that architecture, and provide feedback to the designer on the quality of the architecture. While some of these approaches capture the simple cases of hazards and interrupts in ADL, to our knowledge no previous approach has an explicit way of describing hazards and multiple exceptions for a wide variety of processors and memory architectures. In this report, we present a clean and uniform way of specifying hazards and exceptions in EXPRESSION which supports exploration and validation of programmable embedded systems. We present the study of interrupts and exceptions for PowerPC, MIPS R10K, TI C6x and Intel IA-64 architectures.

Contents

1 Introduction 3

2 Related Work 3

3 Specification of Hazards and Stalls in EXPRESSION 6

4 Specification of Interrupts and Exceptions 7
- 4.1 Opcode related exceptions 7
- 4.2 Exceptions Related to Functional Units 8
- 4.3 External Exceptions 9
- 4.4 Interrupt Handler 10

5 Case Study: PowerPC Family 11

6 Conclusion 13

7 Acknowledgments 13

A IA-64 interrupts 15

B TI C6x interrupts 18

C R10K interrupts 19

List of Tables

1 Interrupts and exceptions for the family of PowerPC architecture 12

1 Introduction

The advent of System-on-Chip (SOC) technology has resulted in a paradigm shift for the design process of embedded systems employing programmable processors with custom hardware. Modern system-level design libraries frequently consist of Intellectual Property (IP) blocks such as processor cores that span a spectrum of architectural styles, ranging from traditional DSPs and superscalar RISC, to VLIW and hybrid ASIPs. Furthermore, SOC technologies permit the incorporation of novel on-chip memory organizations (including the use of on-chip DRAM, frame buffers, streaming buffers and partitioned register files), allowing a wide range of memory organizations and hierarchies to be explored and customized for the specific embedded application [12].

Recent efforts in language-driven Design Space Exploration (DSE) ([1], [2], [3], [4], [6], [8], [13], [15], [16]), use Architectural Description Languages (ADL) to capture the processor architecture, generate automatically a software toolkit (including compiler, simulator, assembler) for that processor, and provide feedback to the designer on the quality of the architecture. These approaches extensively address processor and memory [11] features. While some of these approaches ([5], [6], [14]) captures the simple cases of hazards and interrupts in ADL, to our knowledge no previous approach has an explicit way of describing hazards and interrupts for a wide variety of processors and memory architectures [10]. Moreover, the existing approaches are not capable of capturing the interactions between multiple exceptions. In this report, we present a clean and uniform way of specifying hazards and exceptions in EXPRESSION which supports DSE and validation of programmable embedded systems.

Section 2 presents related work addressing specification of hazards and exceptions. Section 3 describes how we specify hazards and stalls in EXPRESSION [8]. The explicit specification of interrupts and exceptions in EXPRESSION is described in Section 4. The study of interrupts and exceptions for the PowerPC family is shown in Section 5. The study of interrupts and exceptions for the MIPS R10K, TI C6x and Intel IA-64 are included in the appendix. Section 6 concludes the report.

2 Related Work

The nML [6], LISA [5] and RADL [14] processor description languages are closest to our work. We describe in detail the hazard and interrupt specification techniques for these languages.

The RADL [14] processor description language supports interrupts and hazards specification. Hazard/Stall specification is closely tied to the architecture and hence is not a good candidate for architectural exploration. Moreover, the paper does not demonstrate how to apply this technique for VLIW and Superscalar processors. The paper does not give any examples of interrupt specification. It provides example for hazard detection and stalling using the simple DLX pipeline as described in Hennessey and Patterson [9]. It detects hazards using its load interlock detection logic and sets the appropriate control signal, *load_raw* (say). The strategy to perform the stall using *load_raw* is as follows:

```
load_raw, ID:stall(NOP)
```

where the *load_raw* signal decides whether the above strategy is applicable. The second element, "ID:", indicates the pipeline stage involved. The third element, "stall(NOP)", indicates that NOP instruction will be inserted into the stage just after the ID stage. The ID stage and all other upstream stages are stalled. The rest of the stages (MEM and WB) will continue to flow smoothly. In RADL, "kill" construct is sometimes used to replace an instruction with stalling upstream stages.

The nML processor description language [6] has explicit way of describing interrupts. The example shown below is given in the paper. The example assumes an interrupt register that may hold a value of 0 or an interrupt number that serves as index into some vector array stored at address 256.

```
mem interrupt_register[1, card(4)] volatile="irq"

op instruction(i:rest_instruction)
action={
        i.action;
        if interrupt_register != 0
        then STORED_PC = PC;
            PC=M[interrupt_register << 2+0x100];
            interrupt_register = 0;
        endif;
       }
```

The interrupt-register is marked as "volatile", i.e., "changing its value". If some non-zero value appears, the PC is stored in some intermediate location (or put on the stack or whatever) and changed to the address found at the index. Of course, on a real machine much more happens: the current CPU state is stored, special mode bits are set, interrupts may be masked etc.

LISA [5] uses Gnatt chart based models to detect structural hazards. In order to detect data and control hazards and perform pipeline flushes it uses extended Gnatt charts by introducing L-charts and operation descriptors. The following example shows two instructions producing a hazard:

```
IF | ID(!w:R0) | IA       | IE(w:R0) |     % instruction \#1
     IF        | ID(r:R0) | IA       | IE  % instruction \#2
```

Instruction #1 reserves register R0 for writing during the ID operation by announcing the write access to register R0 using the resource descriptor !w: and it performs the write during the IE operation (specified by the w: descriptor). Instruction #2 (shown shifted) attempts to read register R0 during the decode operation (the r: descriptor is used). Using the supplied information the data hazard on register R0 can be easily detected and resolved using interlocking, as shown below. The same mechanism is used to describe control hazards and effects of short circuiting.

```
IF | ID(!w:R0) | IA       | IE(w:R0) |
     IF        | nop      | nop      | ID(r:R0) | IA | IE
```

In order to describe pipeline flushing, LISA permits some of the control instructions to explicitly change the sequencing mechanism of the generic machine model, using the kill ("k:") descriptor for operations (e.g., k:03). The kill descriptor is described in the example given below. The example is the LISA machine description of TMS320C54x branch conditional (BC) instruction.

The kill descriptor simply overloads the operation in the specified stage with its own operation, in this case NOP. In this way operation cancellation takes place to stop further propagation (issuing) of the instructions which are supposed to be flushed due to branch mis-prediction.

```
<insn> BC
{
    <decode>                            // Decoding information
    {
        %ID: (0x7495, 0x0493)
        %cond_code: { %OPCODE1 & 0x7F }
        %dest_address: { %OPCODE2 }
    }
                                        // Pipeline stages are
  // PF, IF, ID, AC, RE, and EX
    <schedule>                          // Pipeline scheduling information
    {
        BC1(PF, w:ebus_addr, w:pc) |    // Write e_bus address and pc at program
                                        // fetch (PF) stage in BC1
        BC2(PF, w:pc), BC3(IF) |        // While writing pc occurs in PF stage,
                                        // BC3 can be in instruction fetch (IF)
        BC4(ID) |                       // BC4 in instruction decode stage
        <if> (condition(cond_code))     // Branch taken
        {
            BC5(AC) |
            BC6(PF), BC7(ID), BC8(RE) |
            BC9(EX)
        }
        <else>                              // Branch not taken
        {
            k:NOP(IF), BC10(AC, w:pc) |     // Kill operation in IF stage
            BC11(PF), BC12(ID), BC13(RE) |
            k:NOP(ID), BC14(EX) |           // Kill operation in ID stage
            k:NOP(ID), k:NOP(AC) |          // Kill operation in ID and AC stage
            k:NOP(AC), k:NOP(RE) |          // Kill operation in AC and RE stage
            k:NOP(RB), k:NOP(EX) |          // Kill operation RB and EX stage
            k:NOP(EX)                       // Kill operation in EX stage
        }
    }

    <operate>                           // Behavior
    {
        BC1.control: { ebus_addr = pc++ }
        BC2.control: { ir = mem[ebus_addr]; pc++ }
        BC10.control: { pc = (%OPCODE2) }
    }
}
```

LISA and RADL have similar mechanism for hazard detection and pipeline flushing. These are very much tied to the architecture. For example, in LISA the L-chart for each operation (e.g., BC) describes which operations are to be killed at which particular pipeline stage. During design space exploration where designers want to change pipeline stages, parallelism etc. these techniques are not useful since for every change in the architecture all the operations need to be re-written. Moreover, the specification technique does not appear to be general enough to model hazards and pipeline flushes in contemporary DSP, VLIW and Superscalar architectures. The nML has a very primitive interrupt specification mechanism which is not powerful enough to model the interrupts, exceptions and their complex interactions (e.g., handling multiple exceptions) available in contemporary architectures. In summary, the existing specification techniques of ADL-driven

hazards and interrupts are not good candidates for design space exploration of a wide spectrum of processor-memory architectures.

3 Specification of Hazards and Stalls in EXPRESSION

We can classify hazards into three categories:

1. *Structural hazards* arise from resource conflicts when the hardware cannot support all possible combinations of instructions in simultaneous overlapped execution.

2. *Data hazards* arise when instructions depend on one another in a way that is exposed by the overlapping of instructions in the pipeline.

3. *Control hazards* arise from dependencies on branches and other instructions that change the PC.

A structural hazard or resource conflict occurs on a functional unit if the number of simultaneous data transfers which go through the unit exceeds its capacity. Structural constraints are captured using reservation tables. The Reservation Tables (RTs) are automatically generated using structural (operation timing, pipeline and data transfer paths etc.) and behavioral (opcodes, operands, format etc.) specification of the processor available in EXPRESSION [7]. RTs are tables which specify the list of components which are accessed for each pipe-stage on a per-operation basis. These RTs can be used to detect structural hazards. Data hazards arise when instructions depend on one another in a way that is exposed by the overlapping of instructions in the pipeline. Using the same constraint information (regarding sources and destinations) available in RTs the data hazards (RAW, WAW, WAR) can be detected.

In general, the pipeline gets stalled when a hazard is detected. A stall can be local where only the instruction is stalled. The stalling has different implications in different scenarios. In case of in-order execution semantics, stalling an operation means stalling everything if the architecture does not have a reservation station. If it has reservation stations in units that have space to accommodate incoming operations then the above scenario would mean stalling that particular functional unit which detected the hazard. If it has out-of-order execution semantics then it means only stalling of the operation. Some hazards may not happen for particular architectural styles. For example, WAW and WAR is not possible when the architecture has register renaming. The detection of a hazard does not mean it would stall the operation or functional unit. It may not do anything at all and issue the operation. For example, if a architecture supports snooping (reading operands using bypass logic in the execution unit) then the issue unit can issue the operation even if one or both of its operands are not ready.

A unit can be *stalled* due to external signals or due to conditions arising inside the processor pipeline. For example, the external signal that can stall a fetch unit is *ICache_Miss*; internal condition for stalling of fetch unit can be due to decode stall, hazards, or exceptions. For units with multiple children the stalling condition due to internal contribution may differ. For example, the unit $UNIT_{i-1,j}$ in Figure 1 with q children can be stalled when *any* one of its children is stalled, or when *some* of its children are stalled (designer identifies the specific ones), or when *all* of its

children are stalled; or when *none* of its children are stalled. During specification, designer selects from the set (ANY, SOME, ALL, NONE) the internal contribution along with any external signals to specify stall condition for each unit.

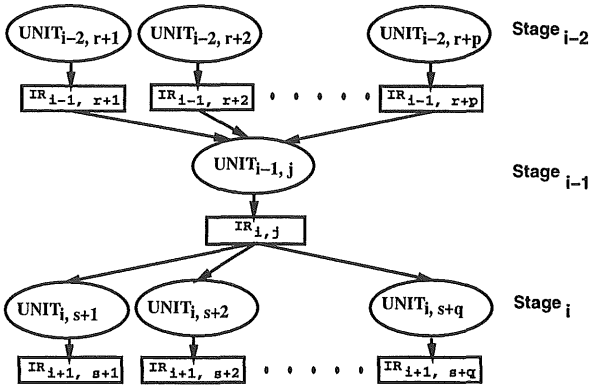

Figure 1. A fragment of a processor pipeline

Control hazards due to branches can have different outcomes depending on how the branch is handled for that architecture. The actions due to branch mis-prediction can be specified in the branch opcode as a set of kill operations that inserts no operation (NOP) in the specific stages of the pipeline [5]. However, this technique is not useful during design space exploration of wide varieties of architectures where for every modification of the processor pipeline the branch action has to be re-written. In our framework we treat branch-misprediction as an exception and the necessary flushing can happen during exception handling as described in Section 4.

4 Specification of Interrupts and Exceptions

We classify exceptions into three categories. This classification is motivated from the ease of specification point of view.

- Opcode related exceptions (e.g., divide by zero etc.)
- Exceptions related to functional units (illegal slot exception etc.)
- External exceptions (reset, power on etc.)

4.1 Opcode related exceptions

It is appropriate to describe opcode related exceptions and their actions inside the opcode specification of EXPRESSION. The syntax of the specification is shown below. Please note that the opcode specification exists in EXPRESSION. We only add the last line i.e., the exception generation statement.

```
(OPCODE <opcode_name>
     (OP_TYPE <opcode_type>)
```

```
      (OPERANDS <list_of_sources_and_destinations>)
      (BEHAVIOR <execution_behavior>)
      (EXCEPTIONS <exception_list>)
)

<opcode_name> := /* as described in EXPRESSION */

<opcode_type> := /* as described in EXPRESSION */

<list_of_sources_and_destinations> := /* as described in EXPRESSION */

<execution_behavior> := /* as described in EXPRESSION */

<exception_list> : = <exception>
                   | <exception_list>
                   | NULL

<exception> := (IF <exception_cond> THROW <opcode_related_exception>)

<exception_cond>     := /* conditional expression involving operands */

<opcode_related_exception> := DIVIDE_BY_ZERO
                            | INVALID_OPERAND
```

For example, the modified DIV opcode is shown below after adding the exception information. Please note that the last line is newly added; the remaining three lines exist in the original EXPRESSION description.

```
(OPCODE DIVW
     (OP_TYPE DATA_OP)
     (OPERANDS (_SOURCE1_ gpr) (_SOURCE_2_ gpr) (_DEST_ gpr))
     (BEHAVIOR "_DEST_=SRC1/SRC2")
     (EXCEPTIONS
        (if (SRC2 == 0) throw DIV_BY_ZERO)
     )
 )
```

4.2 Exceptions Related to Functional Units

Functional unit related exceptions should be defined in EXPRESSION's functional unit specification. The syntax of the exception specification is shown below.

```
(Unit <functional_unit_name>
     ..............  /* other specifications */
     (EXCEPTIONS <exception_list>)
)

<functional_unit_name> := /* as described in EXPRESSION */

<exception_list> : = <exception>
                   | <exception_list>
                   | NULL

<exception> := (IF <exception_cond> THROW <unit_related_exception>)

<exception_cond>     := /* conditional expression involving architectural
                           features */

<unit_related_exception> := ILLEGAL_SLOT_INSTRUCTION
```

```
                                | UNSUPPORTED_OPCODE
                                | DTLB_MISS
                                | ITLB_MISS
                                | SYSTEM_CALL
                                | DCACHE_MISS
                                | ICACHE_MISS
                                | SEGMENT_MISS
                                | OVERFLOW
                                | INVALID_ADDRESS
                                | INVALID_DATA
```

For example, an illegal slot instruction can be described in the decode unit.

```
(Unit Decode
      (CAPACITY 2)
      (TIMING (all 1))
      (OPCODES all)
      (PORTS ....)
      (EXCEPTIONS
         (if (SLOT4 opcode != LDST_type) throw ILLEGAL_SLOT_INSTRUCTION)
      )
)
```

4.3 External Exceptions

External interrupts can be specified in the control unit specification. The syntax of the exception specification is shown below.

```
(Unit Control
     ..............  /* other specifications */
     (EXCEPTIONS <exception_list>)
)

<exception_list> : = <exception>
                   | <exception_list>
                   | NULL

<exception> := (IF <exception_cond> THROW <external_exception>)

<exception_cond>     := /*  conditional expression involving external
                            signals */

<external_exception> := MACHINE_RESET
                        | MACHINE_CHECK
                        | INIT
                        | BREAKPOINT
                        | DEBUG_EVENT
```

For example, an machine reset exception can be described in the control unit. We assume the *RESET* is an external interrupt which generated the internal exception *MACHINE_RESET*.

```
(Unit Control
    (EXCEPTIONS
      (if (RESET) throw MACHINE_RESET)
    )
)
```

This specification may appear redundant since we could have used external interrupt "RESET" instead of generating exception "MACHINE_RESET" from the interrupt. The exception "MACHINE_RESET" is used to define the corresponding interrupt. We have used this approach for

external interrupts for uniformity. Any given external interrupt will be mapped to the external exceptions already defined in EXPRESSION. For example, the COLD_RESET interrupt of the MIPS R10K can generate the exception MACHINE_RESET.

There may not be one interrupt associated with each exception. A class of exceptions may give rise to one interrupt, in that case the architecture implementation should ensure that only one exception from that class happens at a time. In general, one interrupt corresponds to more than one exception. We specify the interrupts and exceptions in the control unit of EXPRESSION. The syntax of this specification is shown below.

```
(Unit Control
    ( INTERRUPTS <interrupt_list> )
    )

<interrupt_list> :=  <Interrupt>
                   | <interrupt_list>
                   | NULL

<Interrupt> := ( INTERRUPT <interrupt_name>
                    ( EXCEPTIONS <exception_list> )
                    ( OPERANDS <operand_list> )
                    ( BEHAVIOR <behavior of ISR> )
               )

<interrupt_name> := /* name of the interrupt */

<exception_list> := /* The list of exceptions which give rise to
                        that particular interrupt. */

<behavior of ISR> := /* Behavioral description of the interrupt
                        service routine for the interrupt <interrupt_name> */
```

For example, interrupt INT1 is described below. INT1 gets generated due to any memory failure during memory operation e.g, ITLB miss, DTLB miss etc.

```
(Interrupt INT1
    (EXCEPTIONS ITLB_MISS, DTLB_MISS, ....)
    (MASKS INT2, INT7, ...)
    (OPERANDS ....)
    (BEHAVIOR "SelectiveStall;
              Save state;
              SetPC(INT1 address);
              ExecuteISR1(...); // Updates TLB
              Restore state"
    )
)
```

4.4 Interrupt Handler

We model the interrupt handler using a priority table that can accept n number of exception/interrupt requests and generate only one interrupt per cycle.

In our EXPRESSION ADL based DSE framework we handle multiple exceptions in a simple and uniform manner. The length of the interrupt service register (ISR) in the interrupt handler unit is equal to the number of interrupts possible in that architecture. One entry in the ISR corresponds to an interrupt. Control unit defines the class of exceptions which generates a particular interrupt. Each exception (opcode related, functional unit related, or external) sets one particular bit in the

ISR of the interrupt handler. Interrupt handler decides the highest priority interrupt using the interrupt priority table. Depending on the masking information the highest priority interrupt masks the appropriate bits in ISR. The process of selecting highest priority interrupt continues until there are no bits set in ISR.

5 Case Study: PowerPC Family

Table 1 shows all the interrupt and exception categories possible in the family of PowerPC architectures. It also shows what category does each exception belongs to viz., asynchronous, synchronous precise, synchronous imprecise, and critical.

We can classify the the exceptions shown in Table 1 in the three major categories mentioned earlier depending on where we want to capture them in EXPRESSION. It does not have exception for the opcode category.

- External (describe in Control unit)
    1. Critical Input
    2. Machine Check
    3. External Input
    4. Alignment
    5. Decrementer
    6. Fixed-interval timer
    7. Watchdog timer
    8. Debug
- HW/SW exceptions (describe in functional unit)
    1. Data Storage
    2. Instruction Storage
    3. Program
    4. FP Unavailable
    5. System Call
    6. AP Unavailable
    7. Data TLB error
    8. Instruction TLB error

Table 1. Interrupts and exceptions for the family of PowerPC architecture

Interrupt	Exception	Type			
		Async	Sync precise	Sync Imprecise	Critical
Critical Input	Critical Input	x			x
Machine Check	Machine Check				x
Data Storage	Read access control		x		
	Write access control		x		
	Byte ordering		x		
	Cache locking		x		
	Storage synchronization		x		
Instruction Storage	Execute access control		x		
	Byte ordering		x		
External Input	External input	x			
Alignment	Alignment		x		
Program	Enabled		x	x	
	Illegal instruction		x		
	Privileged instruction		x		
	Trap		x		
	Unimplemented operation		x		
FP Unavailable	FP unavailable		x		
System Call	System Call		x		
AP Unavailable	AP unavailable		x		
Decrementer		x			
Fixed-interval timer		x			
Watchdog timer		x			x
Data TLB error	TLB Miss		x		
	Large address error		x		
Instruction TLB error	TLB Miss		x		
	Large address error		x		
Debug	Trap	x	x		x
	Inst Addr Compare	x	x		x
	Data Addr compare	x	x		x
	Instruction complete		x		x
	Branch taken		x		x
	Return from interrupt		x		x
	Interrupt taken			x	x
	Uncond debug event			x	x

6 Conclusion

This report proposed a clean and uniform way of specifying hazards, stalls, interrupts and exceptions in EXPRESSION, which supports ADL driven DSE and validation of programmable embedded systems.

Our ongoing work targets the exploration and validation of processor-memory architectures in the presence of multiple exceptions.

7 Acknowledgments

This work was partially supported by grants from NSF (MIP-9708067), DARPA (F33615-00-C-1632) and Motorola Inc. We would like to gratefully acknowledge the EXPRESSION team members for their contribution to the specification work.

References

[1] ARC Cores. *http://www.arccores.com.*

[2] G. G. et al. CHESS: Retargetable code generation for embedded DSP processors. In *Code Generation for Embedded Processors.* Kluwer, 1997.

[3] G. H. et al. ISDL: An instruction set description language for retargetability. In *Proc. DAC*, 1997.

[4] R. L. et al. Retargetable generation of code selectors from HDL processor models. In *Proc. EDTC*, 1997.

[5] V. Z. et al. LISA - machine description language and generic machine model for HW/SW co-design. In *IEEE Workshop on VLSI Signal Processing*, 1996.

[6] M. Freericks. The nML machine description formalism. Technical Report TR SM-IMP/DIST/08, TU Berlin CS Dept., 1993.

[7] P. Grun, A. Halambi, N. Dutt, and A. Nicolau. RTGEN: An algorithm for automatic generation of reservation tables from architectural descriptions. In *ISSS*, San Jose, CA, 1999.

[8] 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.

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

[10] P. Mishra, J. Astrom, N. Dutt, and A. Nicolau. Functional abstraction of programmable embedded systems. Technical Report UCI-ICS 01-04, University of California, Irvine, 2001.

[11] 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.

[12] 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.

[13] V. Rajesh and R. Moona. Processor modeling for hardware software codesign. In *International Conference on VLSI Design*, Jan. 1999.

[14] C. Siska. A processor description language supporting retargetable multi-pipeline dsp program development tools. In *Proc. ISSS*, Dec. 1998.

[15] Tensilica Incorporated. *http://www.tensilica.com.*

[16] Trimaran Release: http://www.trimaran.org. *The MDES User Manual*, 1997.

A IA-64 interrupts

The complete list of IA-32 and IA-64 interrupts are shown below grouped according to type (aborts, interrupts, faults and traps), and listed in priority order.

1. Aborts
    - Machine reset
    - Machine check
2. Interrupts
    - Initialization interrupt
    - Platform management interrupt
    - External interrupt
3. Faults
    - IR unimplemented data address fault
    - IR data nested TLB fault
    - IR alternate data TLB fault
    - IR VHPT data fault
    - IR data TLB fault
    - IR data page not present fault
    - IR data NaT page consumption fault
    - IR data key miss fault
    - IR data key permission fault
    - IR data access rights fault
    - IR data access bit fault
    - IR data debug fault
    - IA-32 instruction breakpoint fault
    - IA-32 code fetch fault
    - Alternate instruction TLB fault
    - VHPT instruction fault
    - Instruction TLB fault
    - Instruction page not present fault
    - Instruction NaT page consumption fault
    - Instruction key miss fault
    - Instruction key permission fault

- Instruction access rights fault
- Instruction access bit fault
- Instruction debug fault
- IA-32 instruction length > 15 bytes
- IA-32 invalid opcode fault
- IA-32 instruction intercept fault
- Illegal operation fault
- Illegal dependency fault
- Break instruction fault
- Privileged operation fault
- Disabled floating-point register fault
- Disabled instruction set transition fault
- IA-32 device not available fault
- IA-32 FP error fault
- Register NaT consumption fault
- Reserved register/field fault
- Unimplemented data address fault
- Privileged register fault
- Speculative operation fault
- IA-32 stack exception
- IA-32 general protection fault
- Data nested TLB fault
- Alternate data TLB fault
- VHPT data fault
- Data TLB fault
- Data page not present fault
- Data NaT page consumption fault
- Data key miss fault
- Data key permission fault
- Data access rights fault
- Data dirty bit fault
- Data access bit fault
- Data debug fault

- Unaligned data reference fault
- IA-32 alignment check fault
- IA-32 locked data reference fault
- IA-32 segment not present fault
- IA-32 divide by zero fault
- IA-32 bound fault
- IA-32 streaming SIMD extension numeric error fault
- Unsupported data reference fault
- Floating point fault

4. Traps

- Unimplemented instruction address trap
- Floating-point trap
- Lower-privilege transfer trap
- Taken branch trap
- Single step trap
- IA-32 system flag intercept trap
- IA-32 gate intercept trap
- IA-32 INTO trap
- IA-32 breakpoint trap
- IA-32 software interrupt trap
- IA-32 data breakpoint trap
- IA-32 taken branch trap
- IA-32 single step trap

B TI C6x interrupts

The list of the TI C6x exceptions are shown below in the decreasing order of priority.

1. Reset, highest priority
2. NMI
3. INT4
4. INT5
5. INT6
6. INT7
7. INT8
8. INT9
9. INT10
10. INT11
11. INT12
12. INT13
13. INT14
14. INT15, lowest priority

Reset is used to halt the CPU and return it to a known state. Non-maskable interrupt (NMI) is used to alert the CPU of a serious hardware problem such as imminent power failure. The remaining twelve interrupts viz., INT4 to INT15, can be associated with external devices, on-chip peripherals, software control, or not be available.

C6x programmer guide explains how to interrupt a function always or a particular number of times by using the pragma in C program as shown below.

```
#pragma FUNC_INTERRUPT_THRESHOLD(func, 1); // Always
#pragma FUNC_INTERRUPT_THRESHOLD(func, threshold);
```

To generate interrupt service routine (ISR) the *Interrupt* keyword should be used. Alternatively to define a existing function as an ISR pragma can be used as shown below.

```
Interrupt void int_handler()    OR #pragma INTERRUPT(func)
{
    unsigned int flags;
       ....
}
```

Enabling and disabling interrupts is done through control status register (CSR).

C R10K interrupts

The list of exceptions of the MIPS R10K are shown below in the decreasing order of priority. Each exception is handled ("processed") by hardware and then serviced by software.

1. Cold reset (highest priority)
2. Soft reset
3. Non-maskable interrupt (NMI)
4. Cache error - instruction cache
5. Cache error - data cache
6. Cache error - secondary cache
7. Cache error - system interface
8. Address error - instruction fetch
9. TLB refill - instruction fetch
10. TLB invalid - instruction fetch
11. Bus error - instruction fetch
12. Integer overflow, trap, system call, breakpoint, reserved instruction,
13. unusable, floating-point exception
14. Address error - data access
15. TLB refill - data access
16. TLB invalid - Data access
17. TLB modified - data write
18. Watch
19. Bus error - data access
20. Interrupt (lowest priority)