

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

Prabhat Mishra      Nikil Dutt      Alex Nicolau  
pmishra@ics.uci.edu    dutt@ics.uci.edu    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

# 1 Introduction

Recent work on language-driven Design Space Exploration (DSE) ([1], [2], [3], [4], [6], [7], [9], [11], [12]), uses Architectural Description Languages (ADL) to capture the processor architecture, generate automatically a software toolkit (including compiler, simulator and assembler) for that processor, and provide feedback to the designer on the quality of the architecture.

However none of these ADLs have explicit way of describing hazards and interrupts for wide variety of processors and memory architectures. The nML [6], LISA [5] and RADL [10] processor description languages are closet to our work. We describe in detail the hazard and interrupt specification techniques for these languages.

The RADL [10] processor description language supports interrupts and hazards specification. Hazard/Stall specification is closely tied to the architecture and hence not good candidate for architectural exploration. Moreover, the paper does not demonstrate how to apply this technique for VLIW and Superscalar processors. RADL has one declarative part for interrupt handling. It allows multiple program counters and could specify the program counter in the pipe-line control section declaratively. On a given signal, which indicates an interrupt, it fetches using an alternative PC. However, register file save and restore along with interrupt disable and re-enable was left to be procedural. Moreover, this language mechanism does not correspond to any hardware implementation. So while it leads to cycle and phase-accurate instruction behavior, it may not accurately reflect internal hardware state. 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 [8] figure 3.4. It detects hazard 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)
```

*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, sometimes "kill" construct is 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] 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 already 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. It introduced the 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>
  {
    %ID: (0x7495, 0x0493)
    %cond_code: { %OPCODE1 & 0x7F }
    %dest_address: { %OPCODE2 }
  }

  <schedule>
  {
    BC1(PF, w:ebus_addr, w:pc) |
    BC2(PF, w:pc), BC3(IF) |
    BC4(ID) |
    <if> (condition(cond_code))
    {
      BC5(AC) |
      BC6(PF), BC7(ID), BC8(RE) |
      BC9(EX)
    }
  }
}
```

```

    <else>
    {
        k:NOP(IF), BC10(AC, w:pc) |
        BC11(PF), BC12(ID), BC13(RE) |
        k:NOP(ID), BC14(EX) |
        k:NOP(ID), k:NOP(AC) |
        k:NOP(AC), k:NOP(RE) |
        k:NOP(RB), k:NOP(EX) |
        k:NOP(EX)
    }
}

<operate>
{
    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 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 needs to be re-written. Moreover, the specification technique is not general enough to model hazards or pipeline flushes in contemporary DSP, VLIW and Superscalar architectures. nML has very primitive interrupt specification mechanism which is not powerful enough to model the the interrupts, exceptions and their complex interactions (e.g., handling multiple exceptions) available in contemporary architectures. In other words, existing hazards and interrupts specification techniques are not good candidates for design space exploration of wide spectrum of processor-memory architectures.

Section 2 describes how we specify hazards and stalls in EXPRESSION [7]. The explicit specification of interrupts and exceptions in EXPRESSION is described in Section 3. Section 3 also includes the examples of interrupts and exceptions for contemporary VLIW and superscalar architectures.

## 2 Specification of Hazards and Stalls

There are three classes of hazards:

1. *Structural hazards* arise from resource conflicts when the hardware cannot support all possible combinations of instruction 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 changes the PC.

## 2.1 Data Hazards

We capture the data hazard information of the processor by specifying the functional unit which detects the hazard. It also captures, whenever possible, the operation which causes the hazard. We consider three classes of data-dependent hazards, according to various data update patterns: write after read (WAR) hazards, read after write (RAW) hazards, and write after write (WAW) hazards. Note that read after read (RAR) does not pose a problem, because nothing is changed. The hazard problem can be solved by data forwarding (also called bypassing and sometimes short-circuiting) whenever possible. In general, pipeline gets stalled when a hazard is detected. Stall can be local where only the instruction is stalled. Now this has different implications in different scenarios. In case of in-order execution semantics, stalling a operation means stalling everything if the architecture does not have reservation station anywhere. If it has reservation station in units which 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 operation stalling. Some hazards may not happen for particular architectural style. For example, WAW and WAR is not possible when architecture has register renaming. Detection of a hazard does not mean it would stall operation, functional unit, that particular pipe or the complete architecture. It may not do anything at all and issue the operation. For example, if a architecture supports snooping (reading operands using bypass logic in execution unit) then issue unit can issue the operation even if one or both of its operands are not ready (RAW hazard).

Based on this discussion we classify the stall, due to data hazard, into the following five categories. First four of them belongs to local stall category.

- NO ACTION
- OPERATION STALL
- UNIT STALL
- PIPE STALL, Stalls only that particular pipe, in case of fragmented pipeline
- GLOBAL STALL

We specify the hazard and stall information in control unit. The syntax of this specification is shown below:

```
(Unit Control
  (HAZARDS
    <hazard_list>
  )
)

<hazard_list> := (<hazard type> <hazard_detection_action_list>)
               | <hazard_list>
               | NULL

<hazard type> := RAW | WAR | WAW
<hazard_detection_action_list>
```

```

:= (<function_unit_name> <operation_name> <action>)
   | <hazard_detection_action_list>
<function_unit_name> := /* The unit which handles that hazard */
<operation_name> := /* The operation during which the hazard occurs*/
<action> := "NO ACTION" | "OPERATION STALL" | "UNIT STALL" | "PIPE STALL" | "GLOBAL STALL"

```

Following example shows the hazard specification for a typical architecture using the above grammar. Each type of hazard (RAW, WAR, WAW) may occur in more than one place in the architecture. RAW happens in three issue units, ALU1 and LDST in the example shown below. We specify during which operation (operation read, operation issue, operation graduation, execution etc.) does this hazard occur. The last field specifies the necessary action need to be taken when this hazard is detected.

```

(Unit Control
  (HAZARDS
    (RAW (INTISSUE OPREAD "NO ACTION")
          (FLTISSUE OPREAD "OPERATION STALL")
          (MEMISSUE OPREAD "UNIT STALL")
          (ALU1 OPREAD "PIPE STALL")
          (LDST OPREAD "GLOBAL STALL")
        )
    (WAR (DECODE OPISSUE "GLOBAL STALL"))
    (WAW (COMPLETION GRADUATE "UNIT STALL"))
  )
)

```

## 2.2 Control Hazards

Control hazards due to branches can have different outcome depending on how the branch is handled for that architecture. The actions due to branch mis-prediction can be specified either in the branch opcode or in the functional unit which handles branch operation. It appears more appropriate to specify the actions in the functional unit which handles branch operation. The following piece of code shows the actions taken during branch mis-prediction for a typical processor.

```

If (misprediction )
{
  updatePC;
  updateBTB;
  selectiveFlush;
}

```

UpdatePC and updateBTB are self explanatory. We consider two kinds of flushing here viz., selective flushing and complete flushing. In selective flushing all the instructions ahead (in sequential order) of mis-predicted branch are allowed to graduate and all the instructions came to pipeline after the mis-predicted branch are flushed. In complete flushing all the instructions in the pipeline are flushed.

## 2.3 Structural Hazards

Structural information are provided using reservation tables. During static scheduling (compile time) this resource information is used. In execution time structural hazards leads to different kinds of stalls viz., pipe stall, unit stall, global stall etc. depending on the architecture.

### 3 Specification of Interrupts and Exceptions

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

- External interrupts (reset, power on etc.)
- Software/Hardware interrupts related to functional unit (illegal slot exception etc.)
- Software exceptions related to opcodes (like divide by zero, TLB miss etc.)

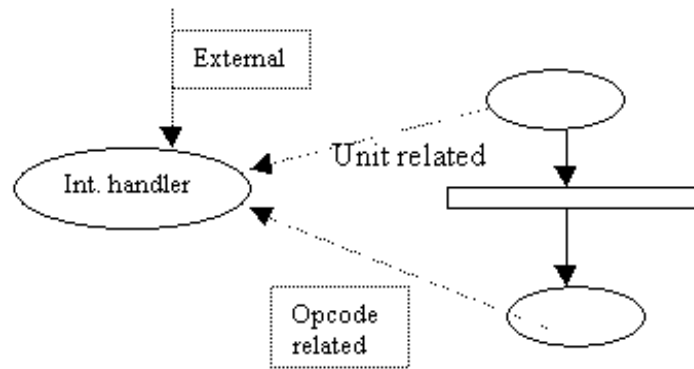


Figure 1. Different types of interrupts

#### 3.1 Opcode related interrupts

It is appropriate to describe opcode related interrupts and its actions inside the opcode specification. For example, 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")
  if (SRC2 == 0) throw D-exception.
)
  
```

#### 3.2 Interrupts Related to Functional Units

Functional unit related interrupts should be defined in functional unit specification. For example, illegal slot instruction can be described in decode unit.

```

(Unit Decode
  (CAPACITY 2)
  (TIMING (all 1))
  (OPCODES all)
  (PORTS Obj3 Obj5 Obj36 Obj2)
  if (SLOT4 opcode != LDST_type) throw illegal slot instruction
)
  
```

### 3.3 External Interrupts

External interrupts can be specified in the control unit.

### 3.4 Interrupt Handler

Interrupt handler will have a priority table and will be able to accept  $n$  number of exception/interrupt requests and generate only one interrupt per cycle.

There may not be one interrupt associated with each exception. A class of exceptions may give rise to one interrupt, in that case architecture implementation should ensure only one exception from that class happens at a time. If each exception corresponds to a particular interrupt then actions for each interrupt can be described where exception is described viz., in opcode or in unit. In general, one interrupt corresponds to more than one exception. We specify the interrupts and exceptions in control unit. 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, ....)
  (OPERANDS ....)
  (BEHAVIOR "SelectiveStall;
            Save state;
            SetPC(INT1 address);
            ExecuteISR1(...); // Updates TLB
            Restore state"
  )
)
```

### 3.5 Multiple Exceptions

As we discussed earlier, interrupt handler is responsible for handling multiple exceptions. Each exception sets one particular bit of interrupt service register (ISR) in interrupt handler unit. Interrupt handler decides the highest priority interrupt using interrupt priority table. Now the question



remains how do we specify explicitly, what happens to the remaining interrupts. This is system specific. In general, external interrupts are serviced one after the other in the priority order (unless masked by earlier one) before resuming the program execution. The software interrupts caused due to program execution (which got flushed) can be ignored since they will be generated again once execution is resumed. Masking information for each interrupt need to be captured explicitly. In general, interrupts are not allowed to interrupt an interrupt service routine. However, NMI in c6x is allowed to interrupt the execution of lower priority interrupts. NMI saves the state, completes execution and then earlier interrupt execution is resumed. The family of PowerPC architectures execute all the interrupts based on priority and gurranted to report in program order. Only excep- tion is in the case of multiple synchronous imprecise interrupts where synchronizing event ensures all previously unreported exceptions are reported. An interrupt of an exception class masks the interrupts of the classes having equal or lower priority. This may not mean an exception, which might have caused an interrupt otherwise, is lost. It could be stored in temporary register and later, when that particular type of interrupt is enabled, might be serviced.

In our scheme we can mask all the equal or lower priority interrupts. Service all the external interrupts (not generated due to program execution). Enable all the pending interrupts. Now these pending interrupts were generated due to program execution which are expected to generate while execution resumes. If they don't get generated again, have two possibilities, viz.,

- The program code segment which generated this interrupt have been completed successfully.
- The exception is not valid any more. For example., TLB miss processing for one load oper- ation can suppress the TLB miss exception for the other one.

An synchronizing event should ensure (if necessary) that all the pending interrupts are enabled.

## **3.6 Example architectures: interrupts and exceptions**

### **3.6.1 PowerPC Family**

Table 1 shows all the interrupt and exception category possible in the family of PowerPC architec- tures. It also shows what category does each exception belong to viz., asynchronous, synchronous precise, synchronous imprecise. Table also shows the critical interrupts.

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

### **3.6.2 IA-64 interrupts**

1. Machine reset
2. Machine check
3. Initialization interrupt
4. Platform management interrupt
5. External interrupt
6. IR unimplemented data address fault
7. IR data nested TLB fault
8. IR alternate data TLB fault
9. IR VHPT data fault
10. IR data TLB fault
11. IR data page not present fault
12. IR data NaT page consumption fault

13. IR data key miss fault
14. IR data key permission fault
15. IR data access rights fault
16. IR data access bit fault
17. IR data debug fault
18. IA-32 instruction breakpoint fault
19. IA-32 code fetch fault
20. Alternate instruction TLB fault
21. VHPT instruction fault
22. Instruction TLB fault
23. Instruction page not present fault
24. Instruction NaT page consumption fault
25. Instruction key miss fault
26. Instruction key permission fault
27. Instruction access rights fault
28. Instruction access bit fault
29. Instruction debug fault
30. IA-32 instruction length  $\geq$  15 bytes
31. IA-32 invalid opcode fault
32. IA-32 instruction intercept fault
33. Illegal operation fault
34. Illegal dependency fault
35. Break instruction fault
36. Privileged operation fault
37. Disabled floating-point register fault

38. Disabled instruction set transition fault
39. IA-32 device not available fault
40. IA-32 FP error fault
41. Register NaT consumption fault
42. Reserved register/field fault
43. Unimplemented data address fault
44. Privileged register fault
45. Speculative operation fault
46. IA-32 stack exception
47. IA-32 general protection fault
48. Data nested TLB fault
49. Alternate data TLB fault
50. VHPT data fault
51. Data TLB fault
52. Data page not present fault
53. Data NaT page consumption fault
54. Data key miss fault
55. Data key permission fault
56. Data access rights fault
57. Data dirty bit fault
58. Data access bit fault
59. Data debug fault
60. Unaligned data reference fault
61. IA-32 alignment check fault
62. IA-32 locked data reference fault

63. IA-32 segment not present fault
64. IA-32 divide y zero fault
65. IA-32 bound fault
66. IA-32 streaming SIMD extension numeric error fault
67. Unsupported data reference fault
68. Floating point fault
69. Unimplemented instruction address trap
70. Floating-point trap
71. Lower-privilege transfer trap
72. Taken branch trap
73. Single step trap
74. IA-32 system flag intercept trap
75. IA-32 gate intercept trap
76. IA-32 INTO trap
77. IA-32 breakpoint trap
78. IA-32 software interrupt trap
79. IA-32 data breakpoint trap
80. IA-32 taken branch trap
81. IA-32 single step trap

### **3.6.3 TI C6x interrupts**

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).

### 3.6.4 R10K interrupts

The priority of the exceptions are shown below. 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)

## References

- [1] ARC Cores. <http://www.arccores.com>.
- [2] G. G. et al. CHES: 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] 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.

- [8] J. Hennessy and D. Patterson. *Computer Architecture: A quantitative approach*. Morgan Kaufmann Publishers Inc, San Mateo, CA, 1990.
- [9] V. Rajesh and R. Moona. Processor modeling for hardware software codesign. In *International Conference on VLSI Design*, Jan. 1999.
- [10] C. Siska. A processor description language supporting retargetable multi-pipeline dsp program development tools. In *Proc. ISSS*, Dec. 1998.
- [11] Tensilica Incorporated. <http://www.tensilica.com>.
- [12] Trimaran Release: <http://www.trimaran.org>. *The MDES User Manual*, 1997.



**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
	Uncond debug event			x	x