Design document: Handling exceptions and interrupts

This document describes what happens on pipeline in case of an exception or interrupt or a combination of these.

Definitions

an interrupt

> An external/internal device requests CPU time. This is the normal way to interrupt the processor in a multitasking system. Interrupt request can originate for example from a timer or an external IO –device, coprocessor etc.

An exception

> An instruction causes a violation. An exception is considered to be abnormal condition. Software originated exceptions can be synthesized using trap – instruction.

General philosophy (crap?)

From the hardware point of view, exceptions require immediate attention and actions cannot be delayed even one clock cycle. This is because hardware has to make sure that the instruction causing the exception does not modify the state of the processor: flags, register or memory contents. If it does, it will most probably cause following instructions to fail also.

From software point of view, the processing of an exception in one thread can be delayed if some other thread currently needs CPU time. It is enough to halt the thread where the exception occurred and invoke a handler routine whenever the execution of the violating thread should continue. Information about exception is passed to the handler.

From software point of view, interrupts should be serviced immediately. Especially systems which have strict real time requirements do not allow servicing to be delayed too much. Anyway ‘immediately’ has a slightly different meaning for software than for hardware: Typically switching to an interrupt routine means executing many instructions before the actual processing of the interrupt event (anything up to hundreds of instructions). In hardware, switching takes typically less than five clock cycles!

From hardware point of view, interrupts are not an error condition and as such do not require immediate attention if something with higher priority is processed. In all cases COFFEE core will pass control to an interrupt service routine as soon as possible. In practise the interrupt response time will be predictable. In fact response will be delayed only in these cases: cache miss causes stall cycles, interrupts are disabled by software, an interrupt with a higher priority is in service or an exception occurs simultaneously with detecting an interrupt request or during a context switch.

Processing of interrupts

Signalling an interrupt

The latency from asserting an external interrupt signal to the moment when control of the core detects the signal is multiple cycles. The latency also depends on the mode of operation of the interrupt interface. Latency is calculated from the falling edge of the EXT_INTERRUPT –signal. Latency from signalling to context switch in different cases is shown in the table below. After edge detection stage, the request is saved in PEND register for further processing. If interrupts are disabled, a request will be pending until interrupts are again enabled. As soon as the core acknowledges the pending request it will be visible in the SERV –register after which a new request from the same source can be accepted.

Table 1, Interrupt signalling latency

mode	**signal synchronization**	**edge detection**	**priority check and masking**	**total cycles**
asynchronous (EXT_HANDLER low)	2 clock cycles	1 clock cycle or less depending on timing of the EXT_INTERRUPT –signal.	1 clock cycle	4
synchronous (EXT_HANDLER high)	-		1 clock cycle	2

Deciding return address

Table 2, deciding the return address

case	**explanation**	**return address/source**	**notes**
0	One of the following instructions in stage 1: **bc, bnc, begt, belt, beq, bgt, blt, bne, jal, jalr, jmp, jmpr, retu** or **scall.**	Calculated jump target address if the branch is taken or the address of the instruction following branch slot instruction if branch is not taken.	All of the listed instructions cause execution to branch somewhere. Slot instruction is executed.
1	**swm** -instruction in stage 1 or 2	Address of the instruction following the two required nop –instructions.	There has to be two nop instructions after a swm.
2	**mulu**, **muli, muls** or **mulus** in stage 1	Address instruction itself.	One of these and a following mulhi –instruction is atomic. => Cannot be executed separately.
3	**reti** –instruction in stage 1, 2 or 3	Address on top of the hardware stack.	In practise this means that, an interrupt service routine is interrupted by a higher priority request (nested interrupts)
4	All other cases	Address of the instruction being fetched, that is the current PC value.	

notes:

1. The return address will be written to PC before saving it to hardware stack, which means that it will be visible to the instruction cache even though the instruction

pointed to is not executed. The only exception to this is case 3 in the above table: If the needed return address is already on top of the stack, it is not popped to PC.

2. If an exception happens during context switching, it takes priority and the interrupt request is left pending.

Switching to an interrupt routine

Switching to an interrupt service routine takes multiple clock cycles. The number of clock cycles depends on the contents of the pipeline and possible stalls caused by cache memory misses or data dependencies.

The total amount of cycles from the falling edge of the EXT_INTERRUPT/COP_EXC –signal to the moment when the address of the first instruction of an interrupt service routine is on the I_ADDR –bus *is from 3 to n cycles.* N depends on pipeline stalls and contents and the interrupt status of the processor.

Before switching to an interrupt service routine, instructions already on pipeline are executed to ‘safe’ state. See *Table 8, Instructions and their safe states* in document *‘Instruction execution cycle times’*.

Figure below illustrates context switching logic for both exceptions and interrupts.

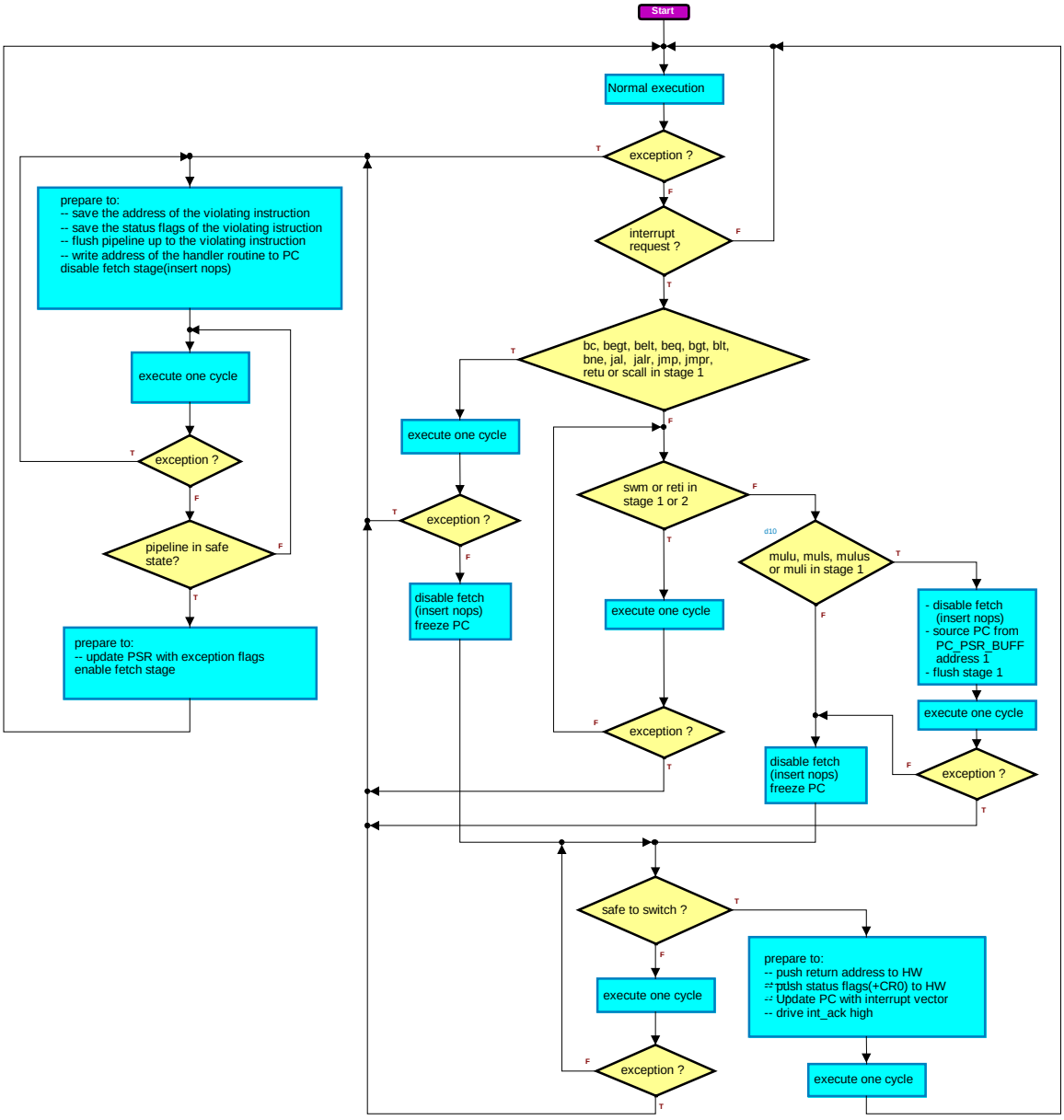

Figure 1, Interrupt & exception logic

Returning from an interrupt service routine

Safe return is guaranteed by executing reti –instruction in stage 2 instead of stage 1. When reti is in stage 2, it can be seen if preceding instructions will cause exceptions. If not, context can be safely restored. Return address will be on memory bus when reti is in stage 4 of the pipeline.

Processing of exceptions

Table 3, Exception types and codes.

pri	code	name	description
10	00000000	instruction address violation [3]	While in user mode, instruction is fetched from memory address not allowed for user.
6	00000001	unknown opcode	Version 1.0 of COFFEE RISC does not have any unused opcodes which makes this obsolete.
7	00000010	Illegal instruction	While in 16 bit mode, trying to execute an instruction which is valid only in 32 bit mode or trying to execute a superuser only instruction in user mode.
3	00000011	miss aligned jump address [4]	Calculated jump target is not aligned to word(32 bit mode) or halfword(16 bit mode) boundary.
2	00000100	jump address overflow	A PC relative jump below the bottom of the memory or above the top of the memory.
9	00000101	miss aligned instruction address [1]	Instruction address is not aligned according to mode. This can be caused by: - External boot address was not aligned to word boundary - An interrupt vector is not properly aligned or interrupt mode is not correctly set - Exception handler entry address is not aligned to word boundary (this will lock the core by causing an eternal loop!) - System entry address is not aligned to word boundary
8	111xxxxx	trap [2]	processor encountered a trap instruction
5	00000110	arithmetic overflow	The result of a signed arithmetic operation exceeds 2^{31}-1 or falls below -2^{31}.
0	00000111	data address violation	While in user mode, a data address refers to memory address not allowed for user.
1	00001000	data address overflow	Trying to index data below of the bottom or above of the top of the memory
4	00001001	Illegal jump	Trying to jump to protected instruction memory area while in user -mode.
x	00001010 ... 00011111		Reserved for future extensions

Table 4, Exception signalling stages

name	violating instruction in stage
unknown opcode	2
Illegal instruction	2
miss aligned jump address	3
jump address overflow	
Illegal jump	
instruction address violation	1
miss aligned instruction address	
trap	2
arithmetic overflow	3
data address violation	4
data address overflow	4

Priorities

Priority 0 means most urgent and 10 means the lowest priority. Priorities come directly from the order of execution. When two or more instructions cause exception in different parts of the pipeline, the one with the highest priority is taken into account.

Switching to exception handler routine

The offending instruction and all following instructions in the pipeline (instructions which follow the violating one in the order of execution) are flushed. The address of the violating instruction is saved along with status flags (PSR), which were valid when decoding the instruction. Also a cause code is saved. See CCB registers. The remaining instructions on the pipeline (instructions which precede the violating one in order of execution) are executed until the pipeline is in safe state, which means that no more exceptions can take place(and processor state does not change). New instructions are not fetched during this pipeline clean operation. If during pipeline clean another exception occurs, the pipeline is flushed up to that instruction and exception data corresponding to the violating instruction is saved (in EXCEPTION_CS, EXCEPTION_PC and EXCEPTION_PSR). After this the cleaning of pipeline will continue until it's safe to switch to the exception handler routine.

When the pipeline is clean, PSR will be updated with default handler flags shown below and execution from address defined in CCB register EXCEP_ADDR is started.

RESERVED	IE	IL	RSWR	RSRD	UM
xxx	0	1	1	1	0

This kind of operation guarantees that an exception is always catched and instructions which preceded the violating one are executed properly. Instructions which follow the violating one are not executed.

Offending instructions are not able to modify the state of the processor or contents of the memory or registers. Note that Exception data registers inside CCB (EXCEPTION_CS, EXCEPTION_PC and EXCEPTION_PSR) will be overwritten immediately. If an exception happens in an exception handler routine (little hope for the software to recover!) the handler routine is restarted and the link to the original context might be lost depending on the handler routine.

Figure 1 (previous chapter) illustrates the exception logic.