Pipeline processor with Cache, TLB and support for exceptions

[1 Introduction 3](#_Toc377484499)

[1.1 Instruction set 3](#_Toc377484500)

[2 Pipeline stages 5](#_Toc377484501)

[2.1 IF stage 5](#_Toc377484502)

[2.2 ID stage 5](#_Toc377484503)

[2.3 EX stage 6](#_Toc377484504)

[2.4 MEM stage 6](#_Toc377484505)

[2.5 WB stage 6](#_Toc377484506)

[3 Important features 7](#_Toc377484507)

[3.1 Hazards 7](#_Toc377484508)

[3.2 Exceptions 8](#_Toc377484509)

[3.3 STALL and NOP control 9](#_Toc377484510)

[4 TLB 10](#_Toc377484511)

[5 Memory system 11](#_Toc377484512)

[5.1 Memory 11](#_Toc377484513)

[5.2 Instruction cache 11](#_Toc377484514)

[5.3 Store buffer 12](#_Toc377484515)

[5.4 Data cache 12](#_Toc377484516)

[5.5 Memory arbiter 13](#_Toc377484517)

[6 Operating system 14](#_Toc377484518)

[7 Assembler 15](#_Toc377484519)

[8 Tests 16](#_Toc377484520)

[9 Possible improvements 17](#_Toc377484521)

# Introduction

Main points about the processor are:

* It supports basic instruction set with all necessary instructions
* It has 5 stages – instruction fetch, instruction decode, execute, memory and write back.
* It has instruction and data cache. Data cache implements store buffer in order to support store operation with 1 cycle in MEM stage.
* It has support for different kinds of exceptions
* It has TLB in IF and MEM stages. The logic is designed in such way, that address translation is done in the same cycle as access to cache.

We have also developed an assembler for proposed instruction set.

## Instruction set

Here, we present instruction set developed for purpose of designing this processor, with support for:

* Special instructions
  + NOP – no operation
  + Special ALU instructions that are used for accessing registers related to exceptions (rmAddress and TrapPC)
  + HALT – stopping the processor
* Memory operations with indexed addressing (load and store)
* Loading immediate value in register (Rx). It can be loaded to high or low 16bit part of the register. Also, signed value of immediate operand can be loaded to register.
* ALU operations
  + Arithmetic operations (ADD, SUB)
  + Logical shift operations (SLL, SRL)
  + Logic operations (AND, XOR, OR, NOT)
  + Operations that write to TLBs (ITLB, DTLB)
* Instructions for program flow control
  + Jump – jumping to immediate value
    - There is also special jump instruction that jumps to contents of rmPC register (consult chapter Exceptions for more details). In this case, immediate value is not used.
  + Branches – conditional jumping (Z - zero, NZ – not zero, C – carry, NC – no carry)

Format of all instructions is given in the picture below. Special fields for bits 25..24 are explained in the next picture.





# Pipeline stages

## IF stage

In IF stage, fetching of instruction is done. For that, it is necessary to know what is the address of the current instruction - PC. The value of PC is calculated by "if\_next\_pc" entity, by using data and control signals from different pipeline stages.

In our design, providers of PC values are:

* ID stage; in case of jump instruction, provides the address for jump.
* EX stage; in case of branch instruction, provides the address for branch.
* WB stage; provides offset that is added to the value of TrapPC (ID stage) to determine the address of jump instruction in exception handler vector (read chapter about Exception for details).

Output of this entity is previous and current value of PC.

Current value of PC is fed to iTLB and iCache.

For information about TLB and iCache, read appropriate chapters.

IF stage can generate three types of exceptions – bad PC alignment, bad access rights and iTLB miss.

## ID **stage**

Main task of ID stage is to decode instruction from IF and create control signals for the rest of the pipeline. This is done in id\_decoder entity.

In ID stage, Register file is located. It is also the responsibility of ID stage to read values from appropriate registers (if needed) and provide them to the next stage.

Register file has two read ports and one write port. Writing is done by WB stage, which provides all necessary signals to ID stage.

ID stage accommodates rmRegisters - rmPC and rmAddress. They are written by WB stage. rmAddress register can be read with special instruction, so ID stage provides hardware for reading from this register.

TrapPC register is also located in ID stage. Its output is driven outside of ID stage, to IF stage (read IF and Exceptions chapters for more details). It can be written to, by WB stage.

ID stage is responsible for control of pipeline in case of LOAD-ALU hazard. Read chapter about hazards for more details.

ID stage can generate one exception - wrong instruction opcode.

## EX stage

Main purpose of EX stage is to perform given ALU operation over two operands that are passed from ID stage. To resolve data hazards, EX stage contains logic for MEM-EX and EX-EX bypasses. For more information about bypasses, read chapter about hazards.

EX stage also calculates branch condition in case of branch instructions. It provides necessary signals for branching to the IF stage (address of jump and control signals).

## MEM stage

Main responsibility of MEM stage are memory operations - read and write. For that purpose, it accommodates dTLB and dCache.

It has simple logic for solving MEM-MEM hazard.

It generates similar exceptions as IF stage (aligns, access and tlb miss).

For more details about dTLB and dCache, read appropriate chapters.

## WB stage

Main purpose of WB stage is to provide proper data, address and control signals for writing values to register file.

Apart from that, WB stage is responsible for handling exceptions and calculating offset in exception handler vector. It also provides signals for writing to rmPC and rmAddress registers. Read chapter about exceptions for more details.

In WB stage, we also added performance counter registers. They count total cycles and active cycles in the pipeline, so IPC (and CPI) can be easily calculated (in simulation). Number of total cycles is equal to number of cycles since the last reset. Number of active cycles is equal to number of committed instructions. Committed instruction is the one that is not NOPed (killed, flushed). Each instruction has isValid bit set to 1 in IF stage. If for some reason it gets NOP-ed in the pipeline, this bit is set to 0. When this bit is active, ActiveCycles counter is increased.

It is possible to add instructions for manual resetting of the counters and also for reading their values. This does not require much changes in hardware.

# Important features

## Hazards

In our pipeline, structural hazards do not exist.

Control hazards occur when executing branch instructions.

For jump instruction, we calculate the jump address in ID stage and pass it, with control signals, to IF stage, which in the same cycle reads from this address. Therefore, we do not loose cycles in this case.

For branch instruction, address is also calculated in ID stage, but decision whether to jump or not is made in the EX stage. We pass those signals from EX to IF, which fetches appropriate instruction in the same cycle. In order not to lose a cycle in every branch instruction, it was decided to make static branch prediction with always-not-take policy. In case of branch-taken, we lose one cycle. In case of branch not taken, we do not lose any cycles.

Adding a branch prediction (for example branch target buffer) could improve performance for branches.

We have several data hazards. Let us observe all possible situations that can result in data hazard (NOP represent instruction that is not important for hazard).

Example 1.1 (EX-EX)

ADD R1,R2->R3 F D E M W

ADD R3,R4->R5 F D E M W

In Decode stage, second instruction reads R3, but first instruction is in EX stage (in WB stage it will write to R3), so second instruction reads wrong value. This is resolved by passing the value from EX to EX.

Example 1.2 (MEM-EX)

ADD R1,R2->R3 F D E M W

NOP F D E M W

ADD R3,R4->R5 F D E M W

Similar as previous problem. It is resolved by passing the correct value from MEM to EX.

Example 1.3

ADD R1,R2->R3 F D E M W

NOP F D E M W

NOP F D E M W

ADD R3,R4->R5 F D E M W

Here, there is no hazard. Registers are written in the middle of the stage, and are read in the second half.

Example 2.1 (MEM-MEM)

LD 0(R1)->R2 F D E M W

ST R2->0(R3) F D E M W

Second instruction does not have correct value of R2. By passing from MEM to MEM, we solve this problem.

Example 2.2 (MEM-EX)

LD 0(R1)->R2 F D E M W

NOP F D E M W

ST R2->0(R3) F D E M W

Here, we pass from MEM to EX to give third instruction the correct value of R2

Example 3.1 (stall + MEM-EX)

LD 0(R1)->R2 F D E M W

ADD R2,R3->R4 F D E M W

In second instruction, we need R2 in EX stage, but it will be available at the end of MEM stage of the first instruction. Here, it is necessary to stall the second instruction one cycle, and then use MEM->EX bypass. In our implementation, we detect in ID, if previous instruction is LOAD and current one uses the register in ALU. If that happens, ID sends NOP instruction to EX (bubble) and stalls IF stage. Chronogram looks like this:

LD 0(R1)->R2 F D E M W

ADD R2,R3->R4 F D D E M W

NEXT\_INS F F D E M W

Example 3.2 (MEM-EX)

LD 0(R1)->R2 F D E M W

NOP F D E M W

ADD R2,R3->R4 F D E M W

Here, we pass from MEM of first instruction to EX of the third. In case there are two instruction between hazarding ones, we have safe situation - register gets written by first, before being read by second instruction.

Example 4.1 (stall + MEM-EX)

LD 0(R1)->R2 F D E M W

ST R3->0(R2) F D E M W

Here, for second instruction, EX stage needs value of R2 in order to calculate destination address in memory, but it will be available in next cycle, at the end of MEM stage of the first instruction. This is LOAD-ALU hazard. It is resolved in the same way as described example 3.1

## Exceptions

Processor supports 9 kinds of exceptions (sorted by descending priority):

* bad alignment of PC (instruction is 4 bytes wide, so value of pc has to be divisible by 4)
* bad instruction access - trying to execute data which is not marked as executable. Detected by TLB
* iTLB miss - there is no translation for requested virtual address in TLB
* wrong instruction opcode. Detected in ID stage.
* ALU overflow - detected by ALU unit in EX stage
* bad memory align. If reading half-word (16b) or word (32b) from memory, address should be aligned - divisible by 2 and 4, respectively.
* bad data access - writing to non-writable location, or reading from non-readable location in memory. Detected by dTLB.
* dTLB miss - same as iTLB miss

Only iTLB and dTLB miss exceptions are recoverable. All other should halt the processor. Otherwise, faulting instruction will get executed again, which will result in infinite loop.

To enable precise exceptions, the decision for jumping to exception handler is done in WB stage. When exception occurs in some stage, appropriate bit in exception vector is set to 1 (exception vector has a bit for every type of exceptions). This exception vector is passed through whole pipeline, to WB stage. When WB stage detects exception in exception vector, it determines which entry in exception handlers vector is dedicated to the occurring exception. One instruction can raise multiple exceptions. However, only top priority one is served (usually, it means that lower priority exceptions are invalid).

Exception handler vector is vector of jump instructions. Each of these jump instruction jumps to handler function for a given exception. Beginning of this vector is stored in trapPC register. There is instruction for writing to this register.

After the faulting instruction, first, we jump to appropriate jump instruction in this vector. This is done in IF stage, by adding offset calculated by WB stage to the value of TrapPC register. After that, jump instruction in the vector will jump to appropriate exception handler function.

Also, when exception is detected in WB stage, PC of the faulting instruction is written to rmPC register (for TLB misses, faulting address is written to rmAddress register). This step is necessary, so control can be returned to faulting instruction after exception handler function is finished. These registers are in ID stage, and WB stage provides all necessary signals for writing to them.

Adding support for new exceptions is easy - in hardware, it is necessary to expand exception vector. In software, it is necessary to add one entry to exception handler vector and, if necessary, instruction for handling the exception.

## STALL and NOP control

In our implementation of pipeline, several stages can request stalling or nop-ing any set of stages. To make this control simpler, we designed a centralized point to which all requests for stall/nop go, from all stages. This control unit, then, decides which stages to stall/nop, based on which request arrived. For that purpose, each output of STALL/NOP control unit has equation that depends on inputs (requests). All mentioned logic is combinatorial.

This control unit is also responsible for halting the processor. Halting is basically equivalent to stalling all stages. When a HALT instruction arrives to WB stage (commits), signal is sent to this control unit and all stages are stalled as the result of this signal. Also, since we want to maintain halted data, we use D-flip-flop to remember this signal. The flip-flop is reset on reset signal.

# TLB

Our processor has two TLB, one in IF (iTLB) and one in MEM (dTLB) stage. iTLB translates PC, whih contains virtual address, to physical address and forwards it to iCache. dTLB translates virtual address for accessing the memory (dCache) and forwards it to dCache. Both TLBs are based on the same VHDL code. Each TLB has 4 entries with associative mapping. Each entry has the following entries:

* virtual page number (31..11 bits of the address – total 21 bits)
* physical page number (31..11 bits of the address – total 21 bits)
* valid bit
* access bits - read, write and execute. (3 bits)

First entry in iTLB and the last entry in dTLB are reserved for pages that contain instructions and data for handling TLB miss. User pages can be stored in second or third entry.

If TLB detects that it does not have translation for given address, it raises TLB miss exception. However, in case of dTLB, if the instruction is not of memory type, this exception is ignored. Also, if wrong access is detected (reading from non-readable page, writing to non-writable page, or executing from non-executable page), exception is risen, but is ignored if instruction is not of memory type. Wrong access exception has bigger priority than TLB miss exception.

In case of wrong access exception, OS (exception handler function) will halt the processor, as it is impossible to recover from it.

TLB miss exception, as every other exception, calls the operating system (read chapter about exceptions for more details). Faulting virtual address (and PC of faulting instruction) is stored in special register. There is an instruction for moving the contents of that register to a general purpose register.

Function for handling this exception, first, saves registers to stack. Next, it finds, in page table, appropriate translation for the given virtual address. It also decides in which TLB entry to write the new translation. In current implementation, it is done either on first or second position (in Round-Robin fashion). Counters for that purpose (one for each TLB) are stored in memory. Information about destination TLB entry is encoded together with information about physical page and access rights. This and faulting virtual address are written to TLB using special instruction (ITLB and DTLB), which copies contents of two general purpose registers to TLB.

Logic of TLB itself it simple and consists of two parts:

* translating the address and checking the access rights (and raising the exception(s) if necessary)
* writing the translation to TLB

# Memory system

## Memory

The memory subsystem is composed of a decoder, a ROM and a RAM (both memories have a size of 4KB and are addressable, and only allow aligned access to 4 bytes). The decoder generates the CS signals for the other two circuits with the memory map:

----------------

|0000-0FFF | ROM |

----------------

|1000-1FFF | RAM |

----------------

Since the size of the pages in the system is 2KB, the system has 2 physical pages of ROM and 2 physical pages of RAM.

It is also implemented other memory system only suitable for simulation, which take all the values from a file and doesn't difference between ROM or RAM.

## Instruction cache

The iCache is a simpler version of the circuit used in dCache, it is known that all the access are going to be word access because iCache is only used for fetching instructions ,so it is not needed any special byte handling mechanism. Also, iCache is also a read only cache, so it is not needed implement eviction mechanism. The organization of the cache is direct mapping with lines of 8 words.

The size of the cache is 4 lines, so the size of the data array is 4 multiply by the size of a line, that is 8 words, and the result is 256 bytes. Thus 2 bits are used for the line number and 3 bits for the column number and 25 bits for the tag:

tag row col

| | x x | x x x | 0 0 |

31 - 7 6-5 4-2 1-0

It is also needed a valid array which indicates what lines are valid and the tag array, when is stored the tag of every line.

Two addresses are used in the iCache, the virtual address which comes from the pipeline, and the physical address that comes from the iTLB. The physical address is used only in the end of the cycle to check that the tag stored in the cache line is the same that the pipeline needs, so the time used in the iTLB is not a problem because both, iTLB and iCache, are lookup in parallel.

When a line is not valid, or the tag is different to the tag from the iTLB a miss is produced and the physical address is sent to the memory arbiter, who will read the line from the main memory and will put fill line to 1.

## Store buffer

The store buffer is organized as a FIFO array, where the older store is in the first position of the array. When there is a cycle without memory operations these value (the first position of the array) is written in cache and all the elements of the array are shifted one position. When a new store arrives to the array it must to be stored in the first free position, so it is needed an array of valid bits and a priority encoder, who signals what is the first free position in the array.

The processor supports byte and 16 bits operations, so it is not enough with the valid bit, it is also needed a mask for every byte of the word stored in the buffer. When a load arrives to the buffer, it is possible that some bytes of the result come from a position of the buffer, and other bytes of the result come from other position of the array, so it is needed to give priority for every byte to the last position in the array where is stored data of the result and the mask bit is 1. It also implies that it is needed a output mask, because maybe not all the bytes of the result are in the store buffer, and some bytes have to be read from the cache line.

When a data is written in the store buffer the first step is to determine what is the mask for the input data, so using the 2 low bits of the address and the type of memory access (lines sel8, sel16 and sel32) is calculated the mask for every byte of the input. Once that the mask is known, each byte is set in the correct position of the word that is written in the buffer (for example a byte write in position 3 will have the correct data in the bits 7 to 0 of the input, but they have to be stored in the bits 31 to 24 of the store buffer).

When a store arrives to the store buffer and there is no free space in the buffer, then the full line is set to 1, and the control logic has to stall the pipeline, and one position of the store buffer is written in the cache.

## Data cache

dCache is more complicated than iCache because the processor can write/read 8, 16 and 32 bits, and there is also a store buffer. The size of the cache is the same than in instruction case, and the organization of the cache is the same (direct mapping with lines of 8 words), so the size of the cache is 4 lines, and the size of the data array is 4 multiply by the size of a line, that is 8 words, and the result is 256 bytes. Thus 2 bits are used for the line number and 3 bits for the column number and 25 bits for the tag:

tag row col

| | x x | x x x | 0 0 |

31 - 7 6-5 4-2 1-0

As in the same case of the iCache, it is needed a valid bit for every line, but it is also needed a dirty bit for every line, so when the row of a line is used for a new line, if the dirty bit is 1 then the line must be written to memory before of reading the new line.

When data is read, it is needed to generate a byte enable signal, which indicates what bytes of the cache are going to be used in the output signal, but it is also needed check that the store buffer does not have a valid byte for this position, so the mask array of the store buffer has to be checked.

When dCache generates a miss, then it is possible that an old line will be written to memory, so the store buffer has to be written into the cache before the eviction. In the process of writing in cache the input address of the cache is changed, because we need the address stored in the store buffer, and since miss is a combinational signal which depends of the address it is needed some mechanism to avoid a combinational loop. It is solved using a register bit which is set to 1 in the end of cycle where the miss is produced, and it is reset to 0 when the store buffer is empty. The output miss signal is produced using an or between this bit and the combinational signal, so even is the address is changed due to the store buffer writing, the output miss signal is stable until the new line is written into the cache.

It is also important notice, that when a word is written from the store buffer, maybe not all the bytes are valid, so it is needed check the mask of the store buffer and write a word which is a mix between the data of the cache and the data from the store buffer. It is important that store buffer only writes one word, so the rest of the line comes from the cache, so the store word must be placed in the correct position of the line written in cache. When a write is done in cache from memory, then the line written is taken only from the data that come from memory.

## Memory arbiter

iCache and dCache generate access to the memory system, so it is needed a arbiter who decides what cache access to the memory system in a determinate moment. This arbiter has two input ports for the address and data (one for every source) and only one output port for the data read from the memory system. It also generates two fill signals, so every source can detect that a data present in the output bus of the arbiter is ready for it or for the other source. The arbiter always gives precedence to the iCache.

The memory system works with words, but caches work with lines, so arbiter is the circuit which transforms between them. When a source read a line from memory, the arbiter generates the low bits of the address using a counter modulo eight, so all the words of the line are read from memory. A temporal buffer is needed to store each word until whole line is read. This same buffer will register the value of the line until the data is read from the source. So the time used in an access to main memory is 10 cycles (1 for the miss, 8 for reading the line and 1 for the fill signal).

When dCache stores a line in memory (iCache only fetches lines and not stores any data in memory) the process is similar, the counter generates the low part of the address for every word of the line, but the word written in memory is selected from the input line of the arbiter.

# Operating system

In order to support exception handling (with accent to TLB misses), we wrote a code that serves as a small operating system for our processor.

Every program starts executing at label BOOT. Here is located the program that does following operations:

* Writes the address of exception handlers vector to trapPC.
* Writes translate data for the last RAM page to TLB (location for stack and TLB counters)
* Initializes the values for TLB entry counters (read TLB for more details about these counters).
* Jumps to MAIN function.

It is important to note that writing the translation for last RAM page to dTLB is done before writing to this page (counter initialization). Before this point, system is not ready to handle any TLB miss, because the last page of RAM not accessible (virtual address is not in TLB).

We also specify the exception handlers vector (more information about this vector is available in Exceptions chapter). The code contains functions for handling all types of exceptions. All exceptions other than TLB misses result in HALTing the processor.

The handler for TLB miss is described in detail in TLB chapter.

Operating system assumes that main program starts at label MAIN, so that has to be taken into account when writing the main program.

Here, we show the page table – mapping virtual pages to physical pages, along with access rights:

|  |  |  |
| --- | --- | --- |
| **Virtual page** | **Physical page** | **R W X** |
| **0** | 0 - ROM 0 | 101 |
| **1** | 0 - ROM 0 | 001 |
| **2** | 1 - ROM 1 | 001 |
| **3** | 3 - RAM 1 | 110 |
| **4** | 1 - ROM 1 | 110 |
| **5** | 2 - RAM 0 | 110 |
| **6** | 4 - Out of boundaries | 110 |
| **7** | 2 - RAM 0 | 110 |

# Assembler

Due to the complexity of witting the exception handling routine directly in binary, an assembler was written. If the assembler is executed with -h parameter the next help is printed in the standard error:

$ as -h

as [-h][-o output.bin][-l [file.lst]][-i [file.lst]][input.as]

-h Show this help

-o Output file

-l Output listing file (labels.lst by default)

-i Input listing file (labels.lst by default)

input Input file

User can supply an input file, which must have the extension '.as', otherwise it will read from standard input. The output is written in a file with the same name that the input, but with the extension '.bin'. If the input was the standard input, then the output is written in the standard output. The user can change where the result is stored with the -o switch. The switch '-l' enables to write an output listing file, where value of all the labels are written (the name labels.lst is used if no parameter is supplied with the option) and -i reads into the symbol table the symbols present in a listing file (with name labels.lst is no parameter is supplied with the option). Examples of invocation can be:

- as file.as

- as < file.as > file.bin

- as -o file.bin < file.as

- as -l -o file.bin < file.as

The format of the assembler is:

LABEL: OP PARAMETERS ;COMMENT

Everything that begins in column 0 is supposed to be a label, and the ':' is mandatory. The maximum size of a label is 8 characters. The input of the assembler must be in upper case (it accepts lower and upper case in some cases, but for example the instructions must be always in upper case). The assembler supports the directives:

DATA Defines a word with data. More of one word can be included

using ','. Ex:

TBL: DATA 0,1,0,3,4

EQU Defines the value for the label in the line. Ex:

NEXCEP: EQU 8

ORG Set an offset for the instruction pointer. Ex:

ORG 100H

FILL Defines a number of data words with the same value. Ex:

TBL: FILL 30,0

Numbers can be leaded with '-' or '+', and can be written in decimal and in hexadecimal format. If a number is written in hexadecimal format it must end with a 'H' letter. It also important that all the numbers have to begin with a digit, so sometimes is needed an extra '0' in some hexadecimal numbers (A000H must be written as 0A000H or it will be considered a label).

The format of the output file is:

Address binary representation of the opcode disassemble of the opcode

-------- -------------------------------- --------------------------

00000000 00101000000011110011111111111100 ;280F3FFC LDI 3FFC,R15

00000164 00000000000000000000000000000001 ;DATA 00000001

The format of the listing file is:

LABEL1: VALUE1

LABEL2: VALUE2

...

# Tests

All tests use small operating system in order to support TLB miss handling (and other exception handling), initializing stack pointer, etc.

The first instruction of the test arrives to IF stage at 3600 ps.

With report, we provided wave.do file, where all important signals for the processor are displayed. It can be loaded in ModelSim, after starting simulation, by File->Load->Macro File.

In the directory tests there is a Makefile which automates the testing process. If you execute make in the directory then the assembler will be compiled, and all the tests will be assembled, generating one '.bin' file and one '.bin.lst' for each test. The '.bin' file is the rom that is loaded in the design, and the '.lst' file is the listing file where all the labels are printed with its value. It is also possible use the Makefile to run vsim, but you need to put in the Makefile where is located your copy of vsim (for example '/opt/altera/13.0sp1/modelsim\_ase/bin/vsim'), and is also needed a vsim project in the root directory of the sources. In this case the Makefile can be used with the command line:

make testxxx

where xxx is the test number (that must be included in the test.lst file), and the Makefile will copy the correct '.bin' file to the root directory with the name 'rom.bin' (the name used in the design for the rom content) and it will run vsim with the wave.do file, which automatically executes the simulation until the time 35000ps, which is enough for all of the tests.

# Possible improvements

It is possible to add following modifications:

* Branch prediction – Branch Target Buffer with 2 prediction bits.
* In current design, there is long critical path between EX stage (determining if we should branch) and IF stage. With this design, we have 0 lost cycles on branch prediction hit (static always-not-taken) and we lose 2 cycles on miss predict (when branch is taken). But, as a result, we get longer cycle.
* Interrupt. With current design, adding support for this is relatively easy, since we have reserved entry in exception handler vector. Also, exceptions system is scalable for adding new exceptions (which is true for adding interrupts, also).
* It is possible to add instructions for manual resetting of the counters and also for reading their values. This does not require many changes in hardware.