MIPSfpga

by Imagination

Lab 13

Using microAptiv's Performance Counters

Lab 13
Using microAptiv's Performance Counters

1. Introduction

In this lab we explain how to configure and use the performance counters available in MIPSfpga, which are extensively explained in Section 6.2.47 of [1]. Performance counters constitute a valuable resource to test functionalities and find bottlenecks in a system. They allow us to measure different microarchitectural events in a program, such as the number of cycles, number of instructions, number of cache accesses or misses, number of taken branches, number of stall cycles, and many others. Figure 1 shows a simple example where we monitor a simple program that performs matrix addition, accounting for the number of cycles and number of instructions.

```
➔ CONFIGURE COUNTERS FOR MEASURING # OF CYCLES AND # OF INSTRUCTIONS
➔ START COUNTERS
  for (i=0; i<N; i++){
     C[i]=A[i]+B[i];
  }
➔ READ COUNTERS
```

Figure 1. Example of performance counters use.

Performance counters are implemented inside the System Control Coprocessor (CP0) of microAptiv, which provides the register interface to the core for the support of memory management, address translation, exception handling and other privileged operations. Chapter 6 of [1] describes the 32 CP0 registers available in microAptiv, among which **CP0 register $25** is assigned to the performance counters.

2. Performance counters in microAptiv

In this section we describe the performance counters available in microAptiv and explain how we can configure and interact with this resource.

2.1 Performance counters available

MicroAptiv processor provides 2 performance counters (PerfCounter-0 and PerfCounter-1). Each one has 2 CP0 registers associated with it: the *control* register, used for configuring the counter (selection of the mode, event to measure, etc.), and the *counter* register, which stores the value of the accounting event (number of cycles, number of instructions, number of branch instructions, etc.) selected by the *control* register. The specific register is chosen by means of the *select* number as defined in Table 1.

Table 1. Performance counters available in microAptiv

PerfCounter	Select Value	Description
PerfCounter-0	0	Control Register 0
	1	Counter Register 0
PerfCounter-1	2	Control Register 1
	3	Counter Register 1

The *control* registers, which can be selected with *select* numbers equal to 0 and 2 for PerfCounter-0 and PerfCounter-1 respectively (as shown in Table 1), allow the user to configure a wide range of parameters. Figure 2 illustrates the fields conforming the *control* register (you can see the field description in Table 6.55 of [1]). The *Event* field (bits 5 to 10) determines the event measured by the performance counter. Table 2 illustrates the first 24 Performance Counter events (12 events per counter) available in microAptiv and their encoding (the whole list can be viewed in Table 6.56 of [1], and Table 6.57 of [1] describes each event in detail). In this lab and in the labs related with the memory system we will mainly use the *number of cycles* (event 0 in both counters), the *number of instructions completed* (event 1 in both counters) and the *number of D$ accesses* (PerfCounter-0, event 10) and *number of D$ misses* (event 11 in both counters).

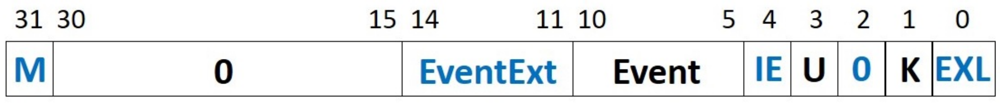

Figure 2. Performance Counter Control Register.

Table 2. First 24 performance counter events

Event number	PerfCounter-0	PerfCounter-1
0	Cycles	Cycles
1	Instructions completed	Instructions completed
2	Branch instructions	Reserved
3	JR (r31) instructions	Reserved
4	JR (not r31) instructions	Reserved
5	ITLB accesses	ITLB misses
6	DTLB accesses	DTLB misses
7	JTLB instruction accesses	JTLB instruction misses
8	JTLB data accesses	JTLB data misses
9	Instruction Cache accesses	Instruction cache misses
10	Data cache accesses	Data cache writebacks
11	Data cache misses	Data cache misses

2.2. Instructions and macros for accessing the performance counters

In this subsection we first describe two instructions available to configure and read the performance counters, and then we explain a more convenient way of working with these instructions by using two macros defined in file *…/Toolchains/mips-mti-elf/2015.06-05/mips-mti-elf/include/mips/m32c0.h*.

`mtc0`

- Format:

31 26	25 21	20 16	15 11	10 3	2 0
Op 010000	MT 00100	Rt	Rd	0 0000000	Sel

```
mtc0 rt, rd, sel
```

- Description: Moves the contents of a general purpose register (*Rt*) to a Coprocessor-0 register, identified by the pair (*Rd*, *Sel*)

`mfc0`

- Format:

31 26	25 21	20 16	15 11	10 3	2 0
Op 010000	MF 00000	Rt	Rd	0 0000000	Sel

```
mfc0 rt, rd, sel
```

- Description: Moves the contents of a Coprocessor-0 register, identified by the pair (*Rd*, *Sel*), to a general purpose register (*Rt*)

Using these instructions directly is one possible way of accessing the performance counter registers. A more convenient option, though, is to use the following macros.

`_m32c0_mtc0(reg, sel, value)`

- Description: Moves *value* to a Coprocessor-0 register identified by the pair (*reg*, *sel*). For instance, the following line configures PerfCounter-1, by writing the *control* register associated to it, to count in kernel mode (*K* field of the *control* register shown in Figure 2) and to measure the *number of instructions completed* (*event* field of the *control* register shown in Figure 2).

```
        _m32c0_mtc0($25, 2, (1 << 5) | (1 << 1));
```

_m32c0_mfc0(reg, sel)

- Description: Assigns the value of a Coprocessor-0 register identified by the pair (*reg*, *sel*) to a variable. For instance, the following line reads the value of the *counter* register of PerfCounter-0 into variable *cntval*.

```
        cntval = _m32c0_mfc0($25, 1);
```

3. Skeleton code

In this section we describe the skeleton code that you will use in this and forthcoming labs. This code is provided in folder *Lab13_PerfCntrs\SimulationSources-Skeleton*. Observe first that we have removed all optimization options in the *makefile* for this skeleton code. Then, open and analyze file **main.c**.

- Two macros used for initiating and reading the performance counters are defined: *INIT_PERF_COUNTS*() and *READ_PERF_COUNTS*() respectively. Note that the performance counters are configured by default for accounting for the *number of cycles* and *number of instructions completed*, using the *_m32c0_mtc0* macro as follows (*C0_PERFCNT* is defined as register $25 in file **m32c0.h**):

```
        _m32c0_mtc0(C0_PERFCNT, 0, (0 << 5) | (1 << 1)); \
        _m32c0_mtc0(C0_PERFCNT, 2, (1 << 5) | (1 << 1)); \
```

  You can easily change the measured events to *number of D$ accesses* and *D$ misses* as follows:

```
        _m32c0_mtc0(C0_PERFCNT, 0, (10 << 5) | (1 << 1)); \
        _m32c0_mtc0(C0_PERFCNT, 2, (11 << 5) | (1 << 1)); \
```

- A struct called *test_result_t* is defined, that stores the two values measured by the performance counters: *event1* and *event2*.
- Then, function **main** is implemented.
  - Initially, the code that we want to test is included (you must substitute comment "*Place your test here, either in C or in MIPS assembly language*" with your program). Note that before that code the performance counters are *initialized* (macro *INIT_PERF_COUNTS()*) and after the invocation they are *read* (macro *READ_PERF_COUNTS(test_result->event1,test_result->event2)*).
  - Then, an infinite *while* loop is implemented that shows the results of the performance counter events on the 7-segment displays. For that purpose, function **writeValTo7Segs**, also implemented at file **main.c**, is invoked. Depending on the value encoded on the switches, *event1*, *event2* or *0* is displayed. Recall that MIPSfpga does not include by

default the support for interacting with the 7-segment displays, thus you first have to modify the MIPSfpga system according to Lab 5.

4. Exercises

In the following exercises you will use the performance counters for evaluating the behavior of several programs. These programs present different hazard scenarios that will be deeply analyzed in Labs 14-18, so you may postpone the resolution of these exercises after those labs.

Exercise 1. Analytical and experimental study of Exercise 7.33 in [2]

Exercise 7.33 from [2] asks the student to calculate the CPI of the following code (in this exercise we have changed the number of iterations from 10 to 1000), taking into account the microarchitectural characteristics of the processor explained in that book.

```
add $s0, $0, $0 # i = 0
add $s1, $0, $0 # sum = 0
addi $t0, $0, 1000 # $t0 = 1000
loop:
      slt $t1, $s0, $t0 # if (i < 1000), $t1 = 1, else $t1 = 0
      beq $t1, $0, done # if $t1 == 0 (i >= 1000), branch to done
      add $s1, $s1, $s0 # sum = sum + i
      addi $s0, $s0, 1 # increment i
      j loop
done:
```

Compute analytically the CPI of this program both in the pipelined processor form [2] and in microAptiv. Recall that there are several differences between microAptiv and the processor from [2]. Specifically:

1. MicroAptiv does not stall due to a RAW dependence between an Arithmetic-Logic instruction and a subsequent beq instruction.
2. MicroAptiv implements delayed branches.
3. The Instruction Memory is not ideal in microAptiv, thus, I$ misses introduce some delay.

Once you have performed the analytical study, resolve the same exercise empirically, by means of the performance counters, and compare the results of both approximations. You can follow the next steps:

1. Copy the skeleton code (folder *Lab13_PerfCntrs\SimulationSources-Skeleton*) in a new folder named *OriginalCode*.
2. Go into the new folder and open file **main.c**.
3. Replace the comment "*// Place your test here, either in C or in MIPS assembly language*" for the following lines, that define the code that we are going to evaluate:

```
asm volatile
(
    "   add $s0, $0, $0;"
    "   add $s1, $0, $0;"
    "   addi $t0, $0, 1000;"
    "   loop:"
    "   slt $t1, $s0, $t0;"
    "   beq $t1, $0, done;"
    "   add $s1, $s1, $s0;"
    "   addi $s0, $s0, 1;"
    "   j loop;"
    "   done:"
);
```

4. To compile this program, open a shell (i.e., *cmd.exe* from the *Start* menu), go into the new folder, and type "*make*" in the shell (as we said above, the *makefile* is not using any optimization options). You can see the compiled program in file **program.dis**, if you look for "*<main>:*". Observe that the compiler introduces a `nop` instruction in the *delay slot* of every `branch/jump` instruction. Also, note that, when using Performance Counters, some instructions are added before and after the program as a result of the accounting instructions.
5. Use the provided *bitfile* (*Lab13_PerfCntrs\mfp_nexys4_ddr.bit*) or generate your own *bitfile* by creating a new project in Vivado as explained in Lab 1 – Step 1. As we warned you in Section 3, it is essential to modify the original MIPSfpga system according to Lab 5, in order to expand the capability of the system so that it can write to the 7-segment displays.
6. Compile the project as explained in Lab 1 – Step 3: Click on the **Generate Bitstream** button at the top of the window. Now wait for synthesis, placement, routing, and bitstream generation to complete. This typically takes around 10-20 minutes or more, depending on your computer speed.
7. Program the FPGA board, as explained in Lab 1 – Step 4: Click on **Open Hardware Manager** in the **Flow Navigator** window on the left. Make sure that the Nexys4 DDR FPGA board is turned on and connected to your computer, and click on **Open Target → Auto Connect**. Finally, click on **Program Device → xc7a100t_0**, select the *bitfile* if it is not selected yet, and click on **Program**.
8. Download the program to the board using the script **loadMIPSfpga.bat** as explained in Section 7.5 of the Getting Started Guide. The program executes, thus, if you set the switches to 0 or 1, the results of the events measured by the performance counters (*cycles* and *instructions completed* respectively) should be shown on the 7-segment displays.

Once you have performed this exercise, reorder manually the program shown above, trying to fill the delay slot with useful instructions instead of `nop`s, and redo the same analytical and empirical analysis. For that purpose you must insert directive ".*set noreorder;*" right before your assembly program and directive ".*set reorder;*" right after it. This directive tells the assembler that the programmer is in control and thus it must not move instructions about (i.e. the compiler will not insert `nop` instructions after the `branch/jump` instructions but will maintain the instruction that we place after them).

Finally, use a –O3 optimization option, analyze the assembly program generated by the compiler, and evaluate it in MIPSfpga as explained above, comparing the results with the previous versions.

Exercise 2. Analytical and experimental study of a simple loop

The following program (available in folder *SimulationSources_Exercise2*) computes the addition of the elements of an array (*test_array[1000]*) into variable *Addition*. Before executing this program the array is initialized (it is brought into the D$), thus the `lw` and the `sw` instructions will never miss (we can assume ideal instruction and data memories in MIPSfpga).

```
LUI  $t0, 0x8000
ADDIU     $t0, $t0, test_array
SUB $t1,$t1,$t1
SUB $t3,$t3,$t3
ADDI $t4,$0,1000
Loop1:    BEQ $t3,$t4,OutLoop1
          LW $t5,0($t0)
          ADD $t1,$t1,$t5
          ADDI $t0,$t0,4
          ADDI $t3,$t3,1
          B Loop1
OutLoop1: LUI   $t3, 0x8000
ADDIU  $t3, $t3, Addition
SW $t1,0($t3)
```

Assume that the program is executed in the processor from [2]. Build the timing diagram from the beginning of the program until the end of the second iteration of the loop, and also for the end of the program, and then compute the CPI based on that diagram.

After the analysis for the processor from [2], evaluate the behavior of this program in MIPSfpga, first analytically, building the timing diagram and computing the CPI. Then, execute the program on the board, and test it without compiler optimizations, with compiler optimizations (including, for example, the –O3 option), and with manual reordering. Compare and explain all these analysis and experiments.

Observe in file *Lab13_PerfCntrs\SimulationSources_Exercise2\main.c* that we are using the fast debug channel (using *fdc_printf()*) in order to test the solution correctness.

Exercise 3. Analytical and experimental study of an array computation

The following program (available in folder *Lab13_PerfCntrs\SimulationSources_Exercise3*) computes the third part of the array of integers *test_array[300]* based on the first two parts of the same array. Before executing this program the array is initialized (it is brought into the D$), thus the `lw` and the `sw` instructions contained in the loop will never miss.

```
lui     $t6, 0x8000;
addiu   $t6, $t6, test_array;
addi    $t1,$0,100;
LOOP:  lw      $t2,0($t6);
       lw      $t3,400($t6);
       sub     $t2,$t3,$t2;
       sw      $t2,800($t6);
       addi    $t1,$t1,-1;
       addi    $t6,$t6,4;
       bne     $t1,$0,LOOP;
addi    $t1,$t1,100;
```

Evaluate the behavior of this program in the processor from [2] and in MIPSfpga, first analytically and then experimentally. Test the original program in MIPSfpga with and without compiler optimizations and then reorder the code trying to optimize performance. Compare and justify the results.

Check with the fast debug channel (using *fdc_printf()*) if your reordered program obtains the correct solution.

Exercise 4. Analytical and experimental study of an array computation

The following program (available in folder *Lab13_PerfCntrs\SimulationSources_Exercise4*) writes an array (*test_arrayResult[100]*) based on two other arrays (*test_arrayA[100]* and *test_arrayB[100]*). Before executing this program the three arrays are initialized (it is brought into the D$), thus the `lw` and the `sw` instructions contained in the loop will never miss.

```
addi   $t1,$0,100;
lui    $t6, 0x8000;
addiu  $t6, $t6, test_arrayA;
lui    $t7, 0x8000;
addiu  $t7, $t7, test_arrayB;
lui    $t8, 0x8000;
```

```
        addiu  $t8, $t8, test_arrayResult;
        LOOP:  LW   $t2,0($t6);
               ADD  $t2,$t2,$t2;
               LW   $t3,0($t7);
               ADD  $t3,$t2,$t3;
               BEQ  $t3,$0, NoAct;
               SW   $t3,0($t8);
               NoAct: ADDI $t6,$t6,4;
               ADDI $t7,$t7,4;
               ADDI $t8,$t8,4;
               ADDI $t1,$t1,-1;
               BNE  $t1,$0,LOOP;
```

Evaluate the behavior of this program in MIPSfpga, first analytically and then experimentally. Test the original program without compiler optimizations and then reorder the code trying to optimize performance.

Check with the fast debug channel (using *fdc_printf()*) if your reordered program obtains the correct solution.

5. References

[1] “MIPS32® microAptiv™ UP Processor Core Family Software User’s Manual -- MD00942”.

[2] “Digital Design and Computer Architecture”. David Money Harris and Sarah L. Harris. Morgan Kaufmann.