CE 435

Embedded Systems

Spring 2018

Lab 4

SW and HW Optimizations

Introduction

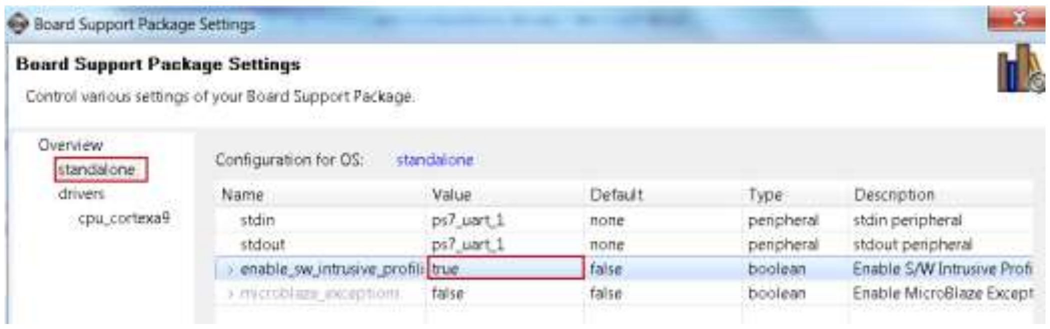
Besides new peripherals, FPGA fabric can be used to implement hardware accelerators to offload computationally demanding tasks from the CPU.
In this lab you will first develop a software application on the ARM Cortex-A9 processor and you will profile the application to assess its performance footprint. You will perform software optimizations to improve its execution time.
However, software running on a processor, no matter how well optimized it is, is slower than a hardware accelerator implementing the same functionality. Based on the outcome of SW profiling you will implement computationally intensive parts of the application as a hardware accelerator to further improve performance. Addional optimizations in hardware can provide additional performance gains.

Step 1 – Software Application Development and Optimization

In this phase you will implement a program that computes Y[i] = A*X[i]*X[i] and execute it on the ARM9 processor. Alternatively, you can implement a larger application (e.g. matrix multiplication) which has more potential for optimizations and is more interesting.

You may start from the hardware you built in one of the previous labs (e.g. lab3), export the hardware HDF file and invoke SDK. Then, you will profile the code to measure its performance and the contribution of each function to the total execution time. Profiling is done using the *gprof* utility as follows:

Select the BSP Package Settings, the *Overview* → *standalone* entry in the left pane, click on the drop-down arrow of the *enable_sw_intrusive_profiling Value* field and select **true**. This will include the profiling drivers when you build your BSP.



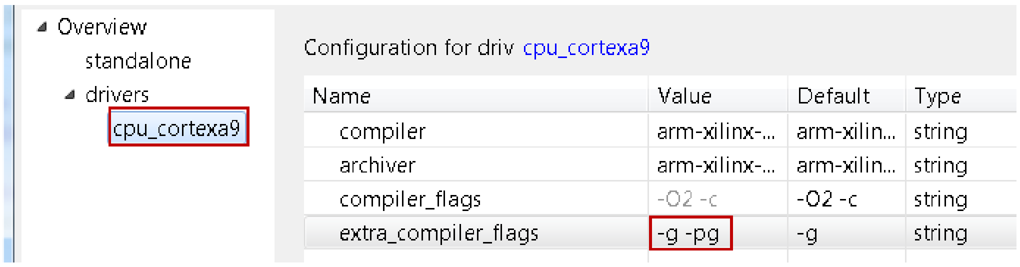
Board Support Package Settings

Control various settings of your Board Support Package.

Configuration for OS: standalone

Name	Value	Default	Type	Description
stdin	ps7_uart_1	none	peripheral	stdin peripheral
stdout	ps7_uart_1	none	peripheral	stdout peripheral
enable_sw_intrusive_profili	true	false	boolean	Enable S/W Intrusive Profi
microblaze_exceptions	false	false	boolean	Enable MicroBlaze Except

Then, select the *Overview* → *drivers* → *cpu_cortexa9* and add **–pg** in addition to the –g in the *extra_compiler_flags Value* field.

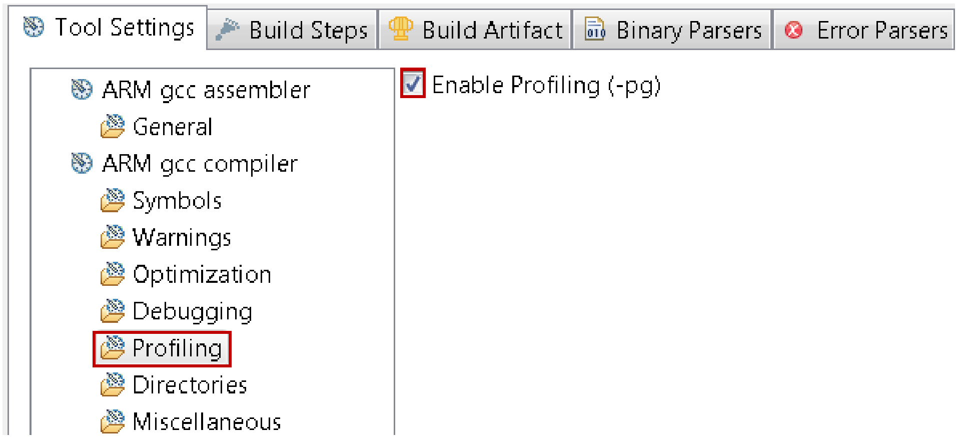


- Overview
  - standalone
  - drivers
    - cpu_cortexa9

Configuration for driv cpu_cortexa9

Name	Value	Default	Type
compiler	arm-xilinx-...	arm-xilin...	string
archiver	arm-xilinx-...	arm-xilin...	string
compiler_flags	-O2 -c	-O2 -c	string
extra_compiler_flags	-g -pg	-g	string

Under the *ARM gcc compiler* group, select the *Profiling* sub-group, then check *the Enable Profiling* box, and click OK.

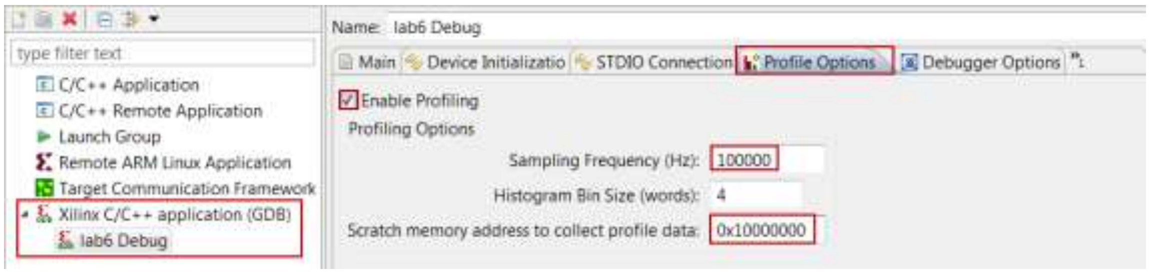


- ARM gcc assembler
  - General
- ARM gcc compiler
  - Symbols
  - Warnings
  - Optimization
  - Debugging
  - Profiling
  - Directories
  - Miscellaneous

[✓] Enable Profiling (-pg)

Select *Run* → *Run Configurations*… and double-click *Xilinx C/C++ Application* to create a new configuration. A configuration refers to the settings and flags used for a specific run of the executable. Defining multiple configuration for a program allows the user to quickly switch between various scenarios without recompilation.

Select the Profile Options tab. Click on the *Enable Profiling* check box, enter 1000000 (1 MHz) in the Sampling Frequency field, enter 0x10000000 in the scratch memory address field, and click Apply.



Name: lab6 Debug

[✓] Enable Profiling

Profiling Options

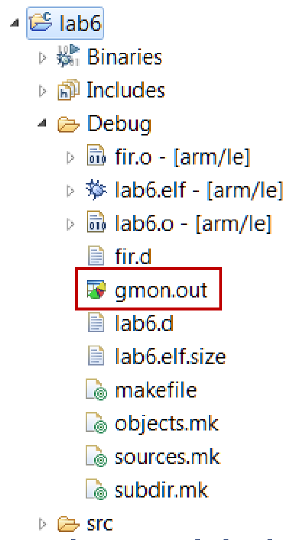
Sampling Frequency (Hz): 100000

Histogram Bin Size (words): 4

Scratch memory address to collect profile data: 0x10000000

Click the *Run* button to download the application and execute it.

When program execution has completed, a message will be displayed indicating that the profiling results are being saved in *gmon.out* file directory. Click OK, expand the *Debug* folder project in the Project Explorer view, and double click on the *gmon.out* entry.



```
lab6
  Binaries
  Includes
  Debug
    fir.o - [arm/le]
    lab6.elf - [arm/le]
    lab6.o - [arm/le]
    fir.d
    gmon.out
    lab6.d
    lab6.elf.size
    makefile
    objects.mk
    sources.mk
    subdir.mk
  src
```

The *Gmon File Viewer* dialog box will appear showing *lab4.elf* as the corresponding binary file.

Click OK. Click on the Sort samples per function button ( ). Click in the %Time column to sort in the descending order.

Besides the profiler, you should also use the timers of lab3 to measure the execution time of your program.

ELF sections and Linker

The *lab4.elf* file generated by the ARM gcc compiler consists of a number of sections, which contain various parts of the program. Open linker script file */src/lscript.ld* and analyze the output to display the sections of the section headers. The most important sections of an elf object file are the instructions of the program (.text section), the static data (.data section), the heap (.heap section), the stack (.stack section), etc.

Linker scripts can be used to statically place sections of the program to specific memory locations, for example in the On Chip Memory (OCM) or the external DDR memory. Using the linker script, the programmer can re-direct the linker to place any section of the executable file in any memory location within the memory address space. This is particularly useful if we want to place frequently executed code or frequently accessed data to a low-latency SRAM memory instead of the main memory.

Moreover, the user can modify the size of each section to accommodate larger data structures being stored in that section. For example, if you have a very large automatic data structure defined in a function, you should increase the size of the stack. If you require large dynamic memory allocations (malloc), you will need to increase the size of the heap.

For step1, you may have to increase the size of the stack and/or the heap in case you have defined a very large array in your code. Otherwise, your code may get stuck.

Step 2 – Hardware Accelerator Development and Integration

In this step, you should design, implement in Verilog and test the hardware accelerator that performs the same functionality as the software program of step 1. Then, you will integrate your design to the ARM-based SoC. Finally, you will profile the code in new design and compare with the SW-based solution.

Step 2.1 – Hardware Design, Implementation and Testing

Design and implement an accelerator that computes Y[i] = A*X[i]*X[i] and use the Vivado simulator to verify its correct functionality. The accelerator will receive the input array X[.] and will store the input to an SRAM in the accelerator. Then, it will receive a trigger signal from the testbench to start computing the output function and store all elements to another SRAM. Finally, it will transmit the results from the second SRAM to the output array Y[.].

One possible implementation strategy is to use two dual-ported SRAMs, one for the input X[.] and one for the output Y[.]. You may use a multi-state FSM to control the I/O and the processing in the accelerator. Note that the computation of Y[.] goes through discrete phases:

a) at the beginning, the accelerator is in idle (or RESET) state waiting for the first inputs to appear.
b) The accelerator receives the size of the input (and output) arrays N.
c) The accelerator receives the input array X[.] and stores the elements of the array to the input SRAM. This phase will take N cycles to complete.
d) Once all input data arrive, the accelerator waits for the trigger signal to start computation.
e) Once triggered, the accelerator reads each element X[i], computes Y[i] = A*X[i]*X[i] and stores the result to the output SRAM.
f) When finished, the accelerator reads the N Y[.] data from the output SRAM and presents them to the output port. At the same time, it asserts an output enable signal to show to the testbench that the data are available.

Step 2.1 requires that you provide a block diagram of the FSM and the whole accelerator, and implement the accelerator and a testbench in Verilog to evaluate correct functionality. As a last step, you should place the accelerator in the IP-XACT repository to reuse it in the following steps as part of the ARM-based SoC.

SRAM memories

Some implementation hints regarding the SRAMs[1]: there are different ways to instantiate the BRAMs available in the Xilinx FPGAs and use them in your design. You can infer the use of a memory by using the following definition in Verilog:

```
reg [31 : 0] memA [0 : 1023];
```

This creates a 1024-entry memory named memA, each entry being 32 bits. You can assign a value to the ith memory entry using the assignment statement:

```
memA[i] <= 0xAABBCCDD;
```

[1] Refer to Xilinx Synthesis Technology (XST) manual in the course website. Page 383 discusses RAM styles.

In order to specifically infer a BRAM hard IP from your Verilog code, you will need to qualify this definition using a RAM STYLE directive as follows:

```
(*ram_style* = “block”) reg [31 : 0] memA [0 : 1023];
```

This Verilog directive instructs the synthesis tool to use a BRAM hard core to implement `memA` instead of using the LUTs (distributed RAM).

Besides this directive, you should also read from `memA` only at the edge of a clock within an always statement. Otherwise, the synthesizer thinks that the memory output is always available, thus, creating a distributed RAM ciruit.

Step 2.2 – Integration of the Hardware Accelerator to the SoC

Save an older project (e.g. from lab3) to a new directory as project *lab4_hw_simple*. Using the procedure explained in lab3 create a new AXI4-Lite peripheral with the appropriate number of interface registers.

Instantiate the hardware created in step 2.1, in template that was generated automatically (similar to lab3, step 3) and create the new accelerator.

Besides providing the hardware of the accelerator, you may have to provide the software drivers for the following tasks: a) send an input array and size N to the accelerator, b) trigger the computation, and c) read the data from the accelerator to the CPU memory.

Test your design in the FPG by writing the output data in the monitor.

Step 2.3 – Profiling of the Hardware Accelerator to the SoC

Similar to step 2.1, you should profile the hardware-based design and compare the two approaches (SW and HW). Can you think of some SW and HW optimizations? If you have time, you may think of optimizations such as increasing the bandwidth of the accelerator, or even using a DMA engine.