Unit II: Code Optimization

Signal Processors Lab

Markus Quaritsch
markus.quaritsch@tugraz.at

SS 2017
Insitute for Technical Informatics

The topic of this second unit of the DSP lab is *code optimization for the C6713 architecture*. Starting with the dot-product, different methods are applied to the assembler code to improve the runtime step-by-step. The goal is to understand how the compiler optimizes the code for speed for finally getting better results with manual optimization. The optimization methods which are applied are the following: (i) filling delay slots, (ii) executing instructions in parallel, (iii) applying word-wide optimization, and (iv) applying software pipelining.

1 Tasks

The project *unit2* is provided for solving all tasks of this unit. A prototype and a skeleton in C and assembler is already provided. Implement all tasks step by step, and try to reuse as many code as possible from the previous steps.

Task 1 – Dot-Product in C: Implement the dot-product function in C and compute the dot-product of the two vectors defined in *uebung2.c*.

$$\text{sum} = \sum_{i=1}^{count} a_i \cdot x_i = 11480$$

(a) Measure the runtime of your implementation (cycle count) in *debug* and *release* mode.

Task 2 – Different memory regions: The build process and the memory layout are not configured for optimal speed. The goal of the second task is to accelerate the C-implementation by changing compiler and linker parameters.

(a) Move the code segment (`.text`) and the global data section (`.bss`) from external RAM (SDRAM) to internal RAM (ISRAM). The mapping from memory section to the physical memory is defined by the linker script (`c6x13dsk.cmd`), which is part of the project. Measure the clock cycles again.

(b) Configure the compiler to optimize the code for speed. The level of optimization can be set in the project's build options (Project→Build Options). Measure the runtime again for different configurations.

Task 3 – Assembler Implementation: In tasks 3 – 7 the assembler implementation of the dot-product is optimized, step-by-step using different methods. In each step, the code of the preceding optimization step is improved by using an additional method.

Implement the dot-product in scheduled-assembler without optimization and pay attention on the required delay slots. For executing the assembler function adapt the call to it in `main()` properly. Validate and evaluate your implementation. The result must correspond with the result of the C implementation, and with the equation given above.

(a) Make sure your implementation still computes the correct result.

(b) Measure the runtime of your implementation (cycle count).

Task 4 – Filling delay slots: : The NOPs as used so far guarantee that the destination register of an instruction is not used before it contains the final result of the operation. Furthermore the NOPs after a branch guarantee that all pipelines are cleared before the branch is executed. But as all functional units are equipped with their own pipeline, NOPs can be removed as long as no data dependency between instructions exist (1 NOP per instruction).

The goal of this task is to optimize your implementation by removing NOPs. For that, you should reorder instructions (especially branches) and you should reduce data dependencies using a larger set of registers.

(a) Make sure your implementation still computes the correct result.

(b) Measure the runtime of your implementation (cycle count).

Task 5 – Parallel execution: The C6713 core consists of 8 functional units (L1, S1, M1, D1, L2, S2, M2, D2) supporting code execution in parallel. The units are divided into two datapaths (A and B), each path having a L-, an S-, an M- and a D-unit. The instruction sets of the units are different – some instructions are available on a single type only (e.g., `LD` and `ST` on D-unit only), some are available on multiple units (e.g., `ADD` on L- and S-units). The datapaths are basically separated from each other which means that registers A can only be used by units of datapath A, and registers B can only be used by unit of datapath B – except one *data crosspath* per VLIW-instruction in each direction.

Goal of this task is to make use of parallel execution to improve the runtime of your implementation. Try to find as many parallelism as possible. Draw a dependency graph to get an idea about data dependencies. Test your implementation and measure the runtime. Instructions are executed in parallel by using two pipe symbols `||` between the istructions. Furthermore

an instruction can be assigned to a functional unit by adding the units name, e.g., .D1, to the instruction:

```
1       MV      .S1     A6,A1
2  ||   MV      .D1     A4,A6
3  ||   ZERO    .L1     A4
```

(a) Make sure your implementation still computes the correct result.

(b) Measure the runtime of your implementation (cycle count).

Task 6 – Word wide optimization: The C6713 implements instructions for packed data processing (e.g., ADD2), which means that sets of two 16-bit or four 8-bit operands can be processed in a single instruction. For load-store, there is also a 64-bit variant available.

The goal of this task is to make use of those packed instructions for both, arithmetic and memory operations. The instruction set reference document gives an overview of all instructions (some in the C62/C64/C67 chapter, most in the C67 chapter).

(a) Make sure your implementation still computes the correct result.

(b) Measure the runtime of your implementation (cycle count).

Task 7 – Software pipelining: The final step in optimizing the dot-product in this unit is to use software pipelining. The idea of software pipelining is to optmize the scheduling of the functional units in such a way, that all units are utilized all the time. Use the scheduling table which is provided for optimizing the dataflow and for utilizing all functional units.

(a) Draw the data flow graph containing the instructions, functional units, and instruction latency (delay slots)

(b) Fill the scheduling table: prolog, kernel, epilog.

(c) Implement your software-pipelined solution.

(d) Make sure your implementation still computes the correct result.

(e) Measure the runtime of your implementation (cycle count).

Task 8 – Optional Task: Give the compiler some hints and thus allow more aggressive optimization.

Add `#pragmas` to your C code such that the compiler has additional information about the arguments and thus can more aggressively optimize the generated code

Implement the dot-product in linear assembler (word-wide data processing). Use asembler directives to provide information about the arguments such that the compiler can better optimize the code.

(a) Make sure your implementation still computes the correct result.

(b) Measure the runtime of your implementation (cycle count).

(c) Inspect the generated code. Can you identify a structure in the generated code?

(d) What happens if you break the asserted constraints about the function arguments (e.g., odd number for `count`, etc.)?

2 Questionaire

Task 1 – 7

1. In the protocol show the optimized source code for every task.
2. For every optimization step measure the execution time for computing the dot-product of the two vectors.
3. Does your implementation pose any constraints on the arguments of the `dotp`-function