Sheffield Hallam University  
Semester I of academic year 2021/22

**"SoC architectures and FPGA prototyping"**

# **Lab 6 –** Assessing performance of Cortex-M7 based SoC built by STMicroelectronics –use the report template from the Appendix 4

Background to the lab (as this material gave place to more important one in the lecture course):   
everyone paying, say, the double price expects to get around double of something in return. In terms of computing customers usually pay for better performance, i.e., less time taken to complete some compute-intensive task. Unfortunately, the use cases vary significantly, and a computer system that is flying for one task can be felt too sluggish for another. Computer benchmarks aim at comparing performance of different systems for a workload that can be quite specific (for example, computer graphic and networking benchmarks). The entry level SoCs with limited resources (like microcontrollers) cannot be subjected to the benchmarks developed to gauge application processor-class SoCs (like smartphones) but can be subjected to good old benchmarks that were successfully used in the past.   
In my opinion:  
- there is no and cannot be the best computing benchmark across the board  
- proprietary benchmarks tend to be skewed to support some agenda  
- manufacturers of computer systems tend to tune their systems to report best benchmarking scores for the tests popular at the time  
- some people develop their own real world tests that may be or may not be applicable to some systems (for example, Gary Explains Speed Test G for smartphones).  
Some benchmarks are synthetic – they do not relate to a real world computing applications but include a defined number of computing operations (multiply, divide etc) instead. The Whetstone benchmark was devised by profiling a suite of applications running at a floating-point capable computer by using built in operations counters. The original benchmark was written in 1972 in the UK using Algol-60 (ALGOrithmic Language as specified in 1960) to test the floating-point performance but served well long after. At that time the computers were predominantly used for scientific and engineering calculations for which floating point performance was the key metric.  
In 1984 the Dhrystone synthetic benchmark was introduced to measure the general-purpose CPU performance for integer calculations (or system programming – operating systems, compilers etc). This benchmark is sometimes used even nowadays. Originally the output was measured in DMIPS (Dhrystone million instructions per second) but in recent times it is reported as DMIPS/MHz to make the metric independent on the clock frequency of which it depends on linearly.  
At present the most popular MCU benchmark is called CoreMark which is reported as CoreMark score (or CoreMark/MHz for a pure CPU). The test includes real world open source subroutines to operate matrices, linked lists, state machine and calculation of the Cyclic Redundancy Check (CRC) code. This benchmark became more popular than the Dhrystone, but it is more difficult to port because it includes specific clauses to guard its impartiality. It concerns with the integer performance only as most MCUs do not have a built-in floating-point hardware.

***I point to some further references in the Appendix 3.***

# Assignment 1. Assessing performance enhancement, provided by the built-in hardware accelerators

Background: the STM32H7 Nucleo board is equipped with a superscalar Cortex-M7 CPU that has a built in double precision floating point coprocessor and separate instruction and data cache memory, which can be enabled and disabled from the application code, making it possible to explore several “what if?” scenarios.   
( ? Can you think of the reason to switch these off in your application code? )

Navigate to the folder Lab6/lab\_6\_main/MDK-ARM and double click on the project file ***H7\_Whet\_Dhry\_400.uvprojx*** (or select this file from the ARM MDK Keil ***Project/Open project…***)

Build the project

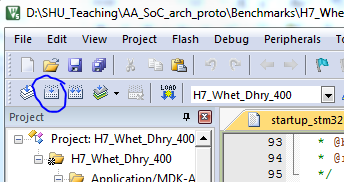


Figure How to build an MDK project

It will take some time (around 3 mins) when you do it first; next time it will be much quicker.

Plug in the H7 Nucleo board into an USB port; ***first time wait until all the drivers are installed***

Click Debug (or Ctrl+F5)

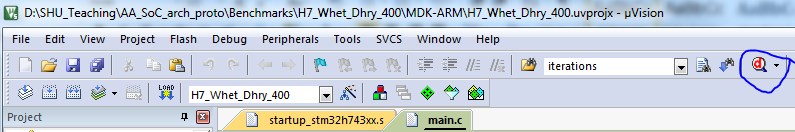


Figure How to start the debug mode

It will take some time to upload the code to the board and switch the MDK to the debug view shown below

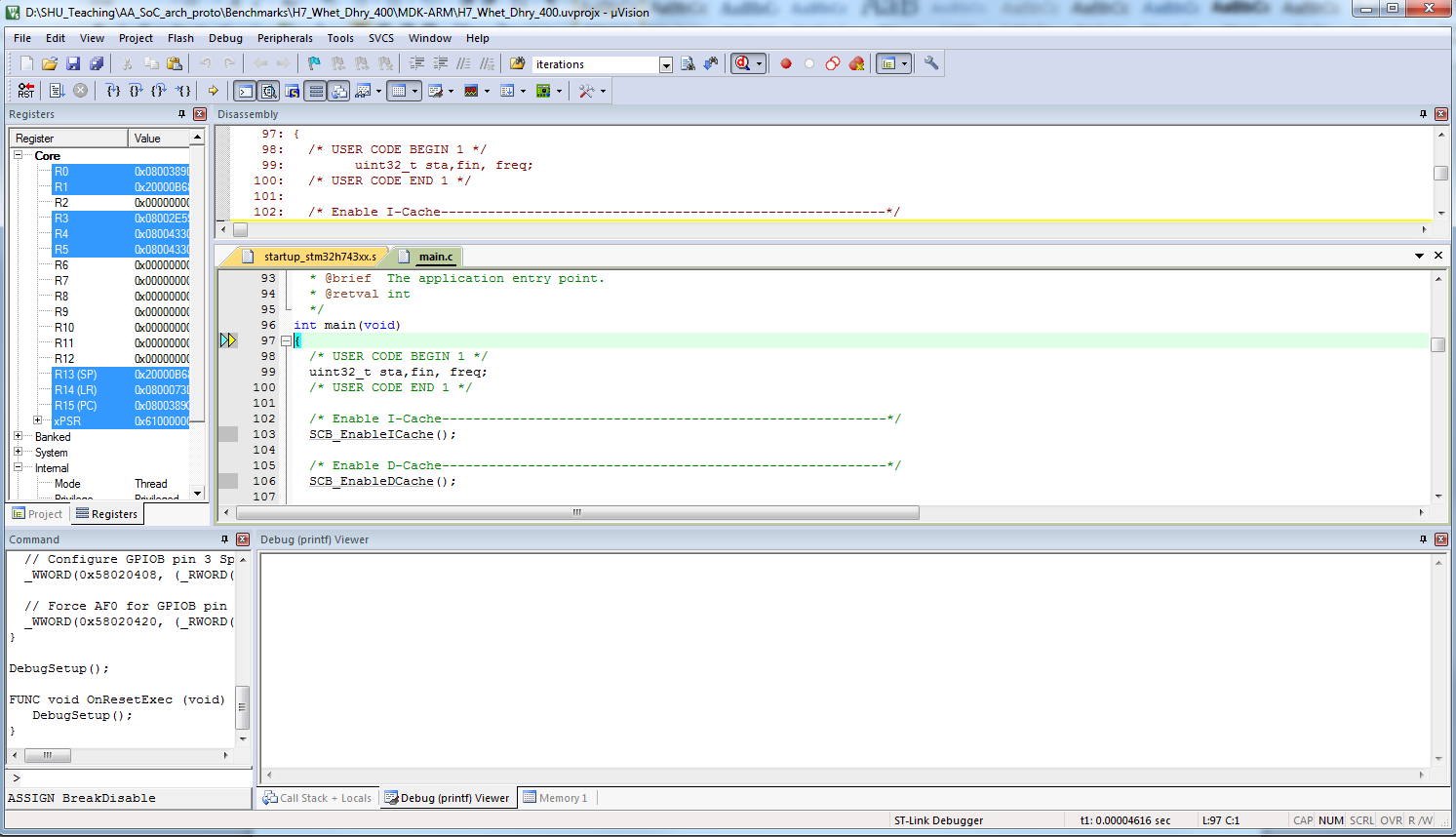


Figure MDK debug view

Click Run button (F5) and note the output of the Debug (printf) Viewer into the report template

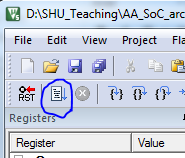


Figure How to run the code

The output shows the performance of the code for the two synthetic benchmarks   
(floating point Whetstone and integer Dhrystone) and a real-world benchmark (CoreMark).

Vary compile (use of the built-in floating-point co-processor) and run time cache options as detailed on the following page to complete the following (the default options are both caches, DP)

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | Floating point HW | Cache options | | | |
| no cache | D$ | I$ | D$ + I$ |
| MWIPS/MHz | Double precision |  |  |  | ? |
| Not  used |  |  |  |  |
| DMIPS/MHz | Not applicable |  |  |  | ? |
| CoreMark/MHz | Not applicable |  |  |  | ? |

Changing the options for the co-processor  
a) click "Options for target"

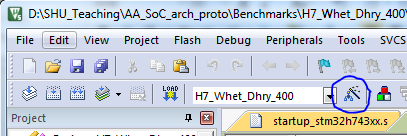


Figure Selecting "Options for target"

b) Select the required options using the dropout list

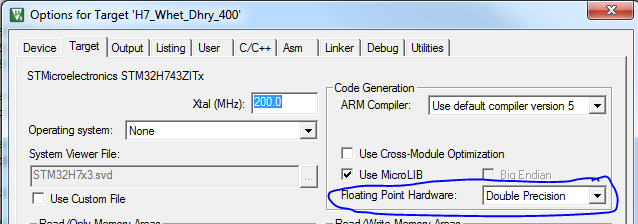


Figure Choosing floating point hardware

Select either “Double Precision” or “Not Used”. The first option will force the compiler to generate double precision instructions for the built-in coprocessor. The other will call subroutines that perform FP calculations using the integer instructions only (FP hardware off to save power).

Changing the options for the cache memory

navigate to the following part of the main.c file:

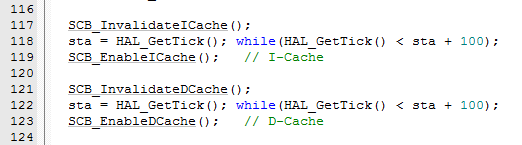


Figure Lines of code that enable instruction and data caches (I-Cache and D-Cache)

Both the I-Cache and D-Cache are enabled in the code. To disable one (or both) simply comment out the relevant lines (117-119 for the I-cache and/or 121-123 for the D-cache).

Re-compile the project by using “Build” not “Rebuild” – this is way faster.

***Make sure to enable caches and set the Double Precision option when you complete this assignment as these settings are required for the 6\_2 and 6\_3.***

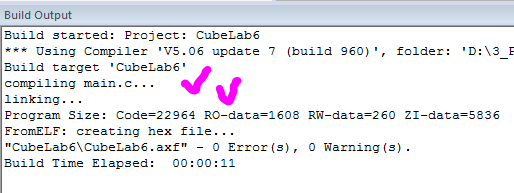
# Assignment 2. Assessing relative efficiency of various ISAs for the same microarchitecture

Background:  
in this exercise you will force the SoC with Cortex-M7 microarchitecture execute only a subset of its instructions (e.g., Cortex-M0 ISA) to evaluate what sort of the performance drop this will result in. (Please do note that you cannot do the other way round – giving a Cortex-M0 CPU Cortex-M7-specific instructions (for example, floating point instructions) will result in hard fault as Cortex-M0 does not have a suitable hardware to execute these.) This may happen if the legacy code is not compatible with a modern C compiler or part of your production code was supplied as binary files by a third party but the supplier went out of business and these files cannot be re-compiled for the better microarchitecture. In order to do this we will compile the benchmarks for the other architectures then include the obtained object binary files (extension ***.o*** for partial binary files) in the same project that you have used for the assignment 1. The details of these ISAs are shown in the Appendix 1, do have a look.

Navigate to the folder Lab6/lab\_6\_main/MDK-ARM and double click on the project file ***H7\_Whet\_Dhry\_400.uvprojx*** (or select this file from the ARM MDK Keil ***Project/Open project…***)

Navigate to the folder Lab6\_400/lab\_6\_2\_o\_files and further down into the folder named after a specific architecture (CoretexM0, CortexM3 or Cortex M4). Each of these folders contain 9 object files (.o files, which represent a full binary code for subroutines that are to be linked to the main.o file to get the final executable). Copy all these files and paste them into the folder Lab6\_400\lab\_6\_main\MDK-ARM\CubeLab6, confirm that you want to overwrite the existing files (originally compiled for the STM32H7453ZI SoC).

Force the IDE to compile the main.c file and build the executable with the newly added .o files. At any place in the main.c code hit space then backspace to indicate that the file changes. Save it and click the Build button (do not click the Rebuild button as it will take a lot of time and restore the .o files you just changed). Observe the following correct output:



Execute the obtained file. Repeat for all the three sets of .o files.

Complete the following table for the additional Cortex-M options and use Cortex-M7 data from the assignment 1:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | Measured  MWIPS/MHz | ARM official  DMIPS/MHz | Measured  DMIPS/MHz | ARM official  CoreMark/MHz | Measured  CoreMark/MHz |
| Cortex M0 |  | 0.83 |  | 2.33 |  |
| Cortex M3 |  | 1.25 |  | 3.34 |  |
| Cortex M4 |  | 1.25 |  | 3.42 |  |
| Cortex M7 | (from the  assignment 1) | 2.14 | (from the  assignment 1) | 5.01 | (from the  assignment 1) |

# Assignment 3. Assessing efficiency of various compiler optimisations

Background:  
the C compilers became well optimised over time and using high level of optimisation may yield significant boost in performance, especially for the synthetic benchmarks. Please note that compiler optimisation modes are geared towards either getting better performance or using less memory or some mix of the both, and use these options accordingly.

Navigate to the folder Lab6/lab\_6\_main/MDK-ARM and double click on the project file ***H7\_Whet\_Dhry\_400.uvprojx*** (or select this file from the ARM MDK Keil ***Project/Open project…***)

Confirm that the project was set up using the O0 optimisation (no optimisation) option:

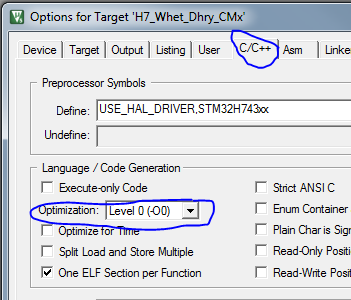
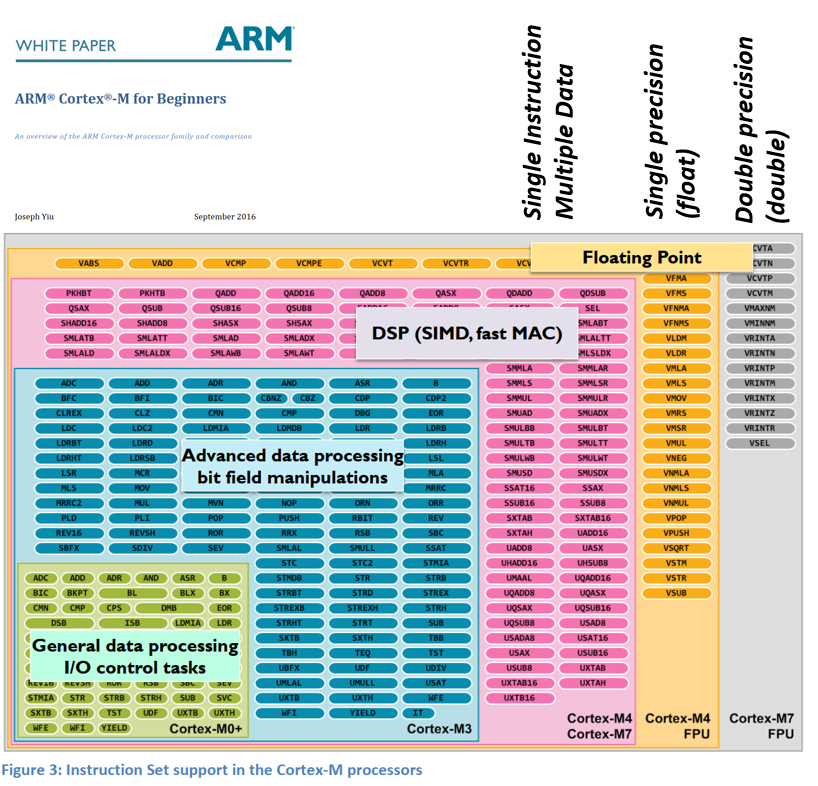


Figure Changing the compiler optimisation level

Rebuild the project three times using the O1, O2, O3 optimisation options, run it and fill in the table

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Compiler option => | O0 | O1 | O2 | O3 |
| MWIPS/MHz | From the assignment 1 |  |  |  |
| DMIPS/MHz | From the assignment 1 |  |  |  |
| CoreMark/MHz | From the assignment 1 |  |  |  |

**Appendix 1**



Please note that the Cortex-Ms with bigger numbers can execute all the instructions shown for the Cortex-Ms with the lower numbers but not the other way around.

Small ovals depict instructions with the length of 16 bits, longer ovals denote 32 bits instructions.

Cortex-M0+ is another microarchitecture for the same Cortex-M0 ISA thus it is not shown. (Cortex-M1 adheres to the same ISA; its Verilog code was optimised for the FPGA implementation.)

**Appendix 2**



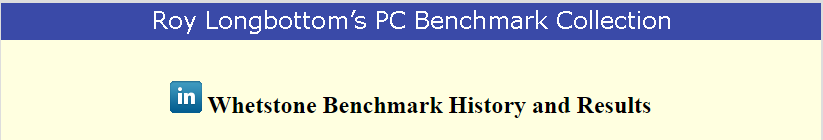
First three columns represent the same ISAs but three different microarchitectures (that is why their scores differ).

Armv8-M instruction set architecture is no longer compatible with the Armv7-M thus cannot be tried on the STM32H743 SoC.

Note that the performance scores quoted above were obtained in the ideal conditions for the simulated pure Verilog core (no memory access delays, no external bus delays, no interrupts, no concurrency in the bus matrix etc).

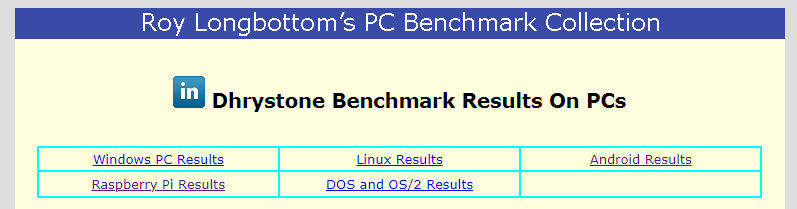
**Appendix 3**

<http://www.roylongbottom.org.uk/whetstone.htm>



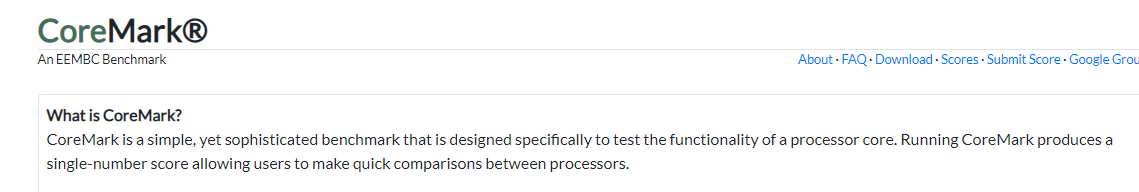
Can you find the place for the STM32H743 SoC in the personal computers of 1990-ies <http://www.roylongbottom.org.uk/whetstone.htm#anchorPC1> ?

<http://www.roylongbottom.org.uk/dhrystone%20results.htm>



Can you compare the SoC with the original Raspberry Pi?

Official website <https://www.eembc.org/coremark/>



<https://en.wikichip.org/wiki/coremark-mhz> - additional information and scores

**Appendix 4**

***Lab 6***

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Assignment 1.  Benchmark scores for various combinations of the built-in hardware accelerators   |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | |  | Floating point HW | Cache options | | | | | no cache | D$ | I$ | D$ + I$ | | MWIPS/MHz | Double precision |  |  |  | ? | | Not  used |  |  |  |  | | DMIPS/MHz | Not applicable |  |  |  | ? | | CoreMark/MHz | Not applicable |  |  |  | ? |   Comment on the measured numbers  …  Compare the relative efficiency of the data and instruction caches  … |
| Assignment 2.  Benchmark scores for various Cortex-M instruction sets   |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | |  | Measured  MWIPS/MHz | ARM official  DMIPS/MHz | Measured  DMIPS/MHz | ARM official  CoreMark/MHz | Measured  CoreMark/MHz | | Cortex M0 |  | 0.83 |  | 2.33 |  | | Cortex M3 |  | 1.25 |  | 3.34 |  | | Cortex M4 |  | 1.25 |  | 3.42 |  | | Cortex M7 | from the  assignment 1 | 2.14 | (from the  assignment 1) | 5.01 | (from the  assignment 1) |   Comment on the measured figures and why they are different from the ARM official DMIPS scores  …  Comment on the differences in ISAs and microarchitectures of the three compared CPU cores  … |
| Assignment 3.   |  |  |  |  |  | | --- | --- | --- | --- | --- | | Compiler option => | O0 | O1 | O2 | O3 | | MWIPS/MHz | From the assignment 1 |  |  |  | | DMIPS/MHz | From the assignment 1 |  |  |  | | CoreMark/MHz | From the assignment 1 |  |  |  |   Comment on the measured numbers  … |