EEE3096S: Embedded Systems II

LECTURE 4:

BENCHMARKING

Presented by:

Dr Yaaseen Martin

Electrical Engineering
University of Cape Town

SELECTION PROCESS

- **Selection Process:**
  Selecting parts and other things from which you will build your embedded system
- This includes choosing processor(s), hardware components, software and code to incorporate and/or reuse.

THE SELECTION PROCESS

- A significant part of the embedded design process
- Involves selecting the best software and hardware choices for the required **functionality**
- Correct selection is critical to profit $$$
- Can (greatly) lower production costs

  → No money wasted on unused features
- Lower non-recurring engineering costs

  → Development cheaper with good tools
- Better reliability and maintainability

THE ‘GOLDEN MEASURE’

TERM: GOLDEN MEASURE

- This concept originates from the theory of a ‘gold standard measure’* which is:
  - a benchmark available under reasonable conditions. Not necessarily the perfect test, but the best available one that has a standard with known results. This is especially important when faced with the impossibility of direct measurements
- A **golden measure** in computing could be:
  - A solution that may run slowly, isn’t optimized, but *you know* it gives (numerically speaking) excellent results

    e.g., a solution written in MATLAB or Python, verify it is correct using graphs or exhaustive testing, or checking by hand with calculator, etc.

* This term originated more in medicine (although has a much older origin if looking further back) and is also widely applied to data science and computing nowadays.

BENCHMARKING

- **Benchmarking:**
  Process of measuring performance of a system (an embedded system in our case)
- It's a program (or systematic approach) that quantitatively evaluates performance, cost, and computing resources (among other things) of a computing solution
- **Benchmark suite:**
  A sets of benchmark programs designed to get a comprehensive view of the performance of a computer system for executing a variety of representative processing operations.

WALL CLOCK TIME

- Generally the most “accurate”: use a built-in timer, which is directly related to real time (e.g., if the timer measures 1s, then 1s elapsed in the real world)
- Note: To avoid overflow, used unsigned variables


```
start = read_the_timer();
 DO PROCESSING
end = read_the_timer();
time = end-start
```

STDC: GETTIMEOFDAY

gettimeofday

- Very portable, part of the StdC (Standard C) library
- Should be available on any Linux system
- Returns time in seconds and microseconds since midnight January 1 1970
- Uses struct timeval comprising
  - tv_sec : number of seconds
  - tv_usec : number of microseconds*
- Converting to microseconds will use huge numbers, rather work on differences

**** Word of caution: some implementations always return 0 for the usec field!!***
On Cygwin, the resolution is only in milliseconds, so tv_usec in multiples of 1000. Not provided in DevC++ on Windows.

GETTIMEOFDAY EXAMPLE

```
#include <stdio.h>
#include <sys/time.h>
#include <time.h>
struct timeval start_time, end_time; // variables to hold start and end time

int main()  {
   int tot_usecs;  /* may want to use long int on CPU with smaller word length */
   gettimeofday(&start_time, (struct timezone*)0); // starting timestamp

   /*   Do some work */
   int i,j,sum=0;
   for (i=0; i<10000; i++)
      for (j=0; j<i; j++) sum += i*j;

   gettimeofday(&end_time, (struct timezone*)0);  // ending timestamp
   tot_usecs = (end_time.tv_sec-start_time.tv_sec) * 1000000 +
               (end_time.tv_usec-start_time.tv_usec);
   printf("Total time: %d usec.\n", tot_usecs);
}
```

WHAT IS WRONG ABOUT USING ONLY WALL CLOCK TIME?

It can provide a false impression of how effective your solution is – or rather more correctly it at least doesn't give a 'full picture' of performance …

Things to consider:

- Typically do tests after the system has 'warmed up' (cache loaded) by running the same benchmark multiple times*
- Speed improved but accuracy sacrificed?
- Development effort vs. execution speed improvement?
- Resource costs for upgrading vs. costs saved by remaining with the old version?
- Power usage? Does the new solution need more power (per execution, also on average including idle time)
- Maintainability? (e.g. is the new version more complex?)
- Environment impact? (Does the upgrade result in waste that could be environmentally detrimental)

* But note, this could also be a false impression depending how your system is likely to work, if operations are typically not going to be in cache when invoked, this approach could invalidate your benchmarking.

BENCHMARKING: WHAT TO TEST

- Compiler
  - Converts High Level Language to Assembly Language thus we benchmark compiler efficiency (such as how efficient is the generated assembly code?)
- Processor
  - Code in hand-crafted/inspected assembly (to make comparisons fair)
- Operating System
  - Interrupt latencies, overhead of operating system calls, limits on devices, kernel size, availability of services and facilities such as support for virtual memory and paged memory.
- Platform
  - Scalability of memory. Peripheral limits. Interfaces supported. Power use. Power saving features. OS's supported.
- Applications

BENCHMARKING: WHAT IS USUALLY MEASURED

- For embedded and DSP systems we typically benchmark:
  - Cycle count
  - Data and Program memory usage
  - Execution time
  - Power consumption (has recently become a common thing to report on)

PROCESSOR PERFORMANCE

- This is usually talked about the most, and is often the most important design decision
- *Particularly:* will the proposed processor/controller/DSP be fast enough?
- Performance measuring tools are available to analyse this, at least roughly, particularly:
  - MIPS (Million Instructions Per Second)
  - Dhrystone Benchmark
  - FLOPS (Floating Point Operations Per Second)

An important point to note: that processor manufacturer might optimise their toolchains to get better benchmark results using these common tools, so it's usually best to use some more representative tests for your particular application.

EXAMPLE OF STANDARD BENCHMARK

- The Dhrystone benchmark
  - Developed in 1984 by R. Wecker
  - Tests integer performance only
    - No I/O, no OS calls, no floating point
- Good for estimating integer performance
- Dhrystone score is the number of times a small program can run per second
- Like all benchmarks, applicability of Dhrystone varies (may be totally irrelevant to your specific application)

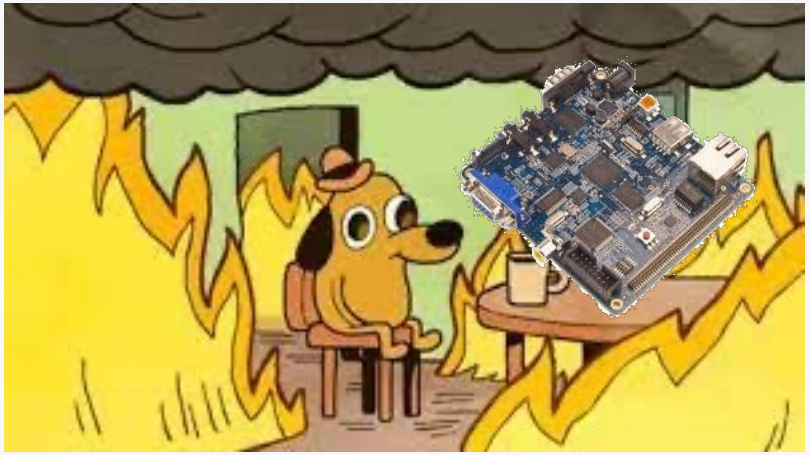
… so don't get to comfortable using a common very 'general purpose' benchmarking tool like Dhrystone!

MEANINGFUL BENCHMARKING

- How will the processor perform with our application?
- Including complex factors
    - Interrupt handling
    - Task switching
    - Memory performance
- Even more important for Real Time applications
    - Context switching time is critical
    - Save registers, change mode, etc.

RUNNING CUSTOM BENCHMARKS

- Typically use evaluation boards
  - Save costs of designing support hardware
- Evaluation boards
  - Single board computers
  - Include a selection of common peripherals
- Some manufacturers sell **Hot Boards**
  - Designed to boost benchmarks
  - Watch out for these boards



BAD NEWS ABOUT BENCHMARKS

- Benchmarks do not consider all design factors which have an effect on performance, such as:
  - Power consumption
  - Physical size
  - Board layout
- Do not rely entirely on benchmark results for processor selection

WHAT IS A CODE REVIEW?

- It is a practice (e.g. meeting or regular team-link up) of reviewing code developed for a project
- It follows (generally) a systematic practice
  (formulate your own initiative method for now!)
- It's about managing quality of code and ways to assess this, e.g. consider:
  - How to spot and log any defects
  - How many or common are the defects
  - Methods to use, such as (not necessarily all of these, not necessarily used in every review) are:
    - Unit testing
    - Function testing
    - Integration testing
    - How can this be improved?