FPGA-Based Real-Time Task Profiler and Deadline Monitor

Final Project Report

Teresa Co Handong He Xiao Lu

CSEE 4840 Embedded System Design — Spring 2026
Terasic DE1-SoC, Cyclone V FPGA, ARM Cortex-A9 HPS, Ubuntu Linux 16.04

Contents

1 Project Overview **4**

2 Motivation and Problem Statement **5**

3 Revised Project Proposal **5**

4 System Architecture **6**

5 Hardware Design **8**

5.1 Main Hardware Module: `event_logger.sv` 8

5.2 Event Buffer 9

5.3 Avalon-MM Register Map 10

6 Deadline Monitor Design **10**

6.1 Simple Deadline Example 12

6.2 Why the Deadline Value Is in Cycles 12

6.3 Sticky Flag 13

7 VGA Visualization Design **14**

8 Software Design **14**

9 Qsys and Quartus Integration **16**

10 Build and Deployment Workflow **17**

10.1 Open the Project 17

10.2 Refresh the Qsys Conduit 17

10.3 Verify the Generated Ports 18

10.4 Compile the Quartus Project 18

10.5 Program the FPGA 18

10.6 Connect to the Board Linux System 19

10.7 Compile and Run the C Program on the Board 19

11 Testing and Results **19**

11.1 Test 1: Hardware Sanity Check 19

11.2 Test 2: VGA Color Mapping 19

11.3 Test 3: Real Workload Trace 20

11.4 Test 4: Deadline Violation 21

11.5 Test 5: Slow Visualization 22

12 Comparison with Software Timing **22**

13 Problems Faced and Solutions **23**

13.1 Corruption of `event_logger.sv` 23

13.2 Qsys Did Not Pick Up the New Conduit 23

13.3 VGA Color Was Not Changing 24

13.4 Problems with Serial Transfers 24

13.5 Platform Pivot 24

14 Individual Contributions **24**

14.1 Teresa Co 24

14.2 Handong He 24
14.3 Xiao Lu 25
14.4 Shared Team Work 25

15 Lessons Learned **25**

16 Recommendations for Future Projects **26**

17 Future Improvements **26**

18 Complete Listing of Files Written for the Project **27**

18.1 Hardware Source Files 27
18.1.1 `event_logger.sv` 27
18.1.2 `task_visualizer.sv` 27
18.1.3 `soc_system_top.sv` 28
18.1.4 `event_logger_hw.tcl` 28
18.2 Quartus and Qsys Files 28
18.3 C Source Files 28
18.4 Analysis and Documentation Files 28
18.5 Files Not Written Manually 29

19 Final `.tar.gz` Package **29**

20 Conclusion **30**

1 Project Overview

Our project is the FPGA-based Real-Time Task Profiler and Deadline Monitor System. The aim of our project is to find out the runtime of software tasks and their compliance with a certain deadline. This requirement arises for embedded devices, since many of them require the timely execution of their functions. Thus, a sensor device needs data sampling at a certain frequency; a controller requires an actuator update within the next control interval; video and signal processing devices have a strict deadline in terms of frame or sample processing time.

A developer of an ordinary piece of software can determine the runtime of his/her task by invoking the respective timer function at the beginning and the end of it. However, such method lacks precision when it comes to tasks executing in microseconds. First, there is an overhead of the timer function itself. Second, the task execution can be preempted by the operating system. It means that the measurement will be inaccurate due to the interference of the operating system into task execution.

Our solution to this problem involves the transfer of timestamping to the FPGA hardware. The ARM processor just writes an event marker to the FPGA. Timestamping is performed by a 32-bit hardware counter that increments its value every clock cycle. Because the clock frequency of the hardware counter is 50 MHz, the interval between two consecutive clock cycles is 20 nanoseconds.

There are three primary components in our project:

1. **Hardware Event Logger**: This is implemented in `event_logger.sv`. It receives `TASK_START` and `TASK_END` events from the ARM HPS, records timestamps, stores them in a hardware buffer, and tracks status flags.
2. **Single-Cycle Deadline Monitor**: This is also inside `event_logger.sv`. It compares the actual task runtime with the allowed deadline. If the task takes too long, it sets a sticky `deadline_missed` flag.
3. **VGA Visualization Layer**: This is implemented in `task_visualizer.sv`. It shows the current task using different colors and flashes red when a deadline miss happens.

The implementation was carried out using the Terasic DE1-SoC board. It consists of ARM Cortex-A9 HPS with the Ubuntu Linux 16.04 operating system and Cyclone V FPGA fabric. Communication between HPS and the FPGA is done using the lightweight HPS-to-FPGA bridge with base address `0xFF200000`. FPGA hardware includes an Avalon-MM slave interface, cycle counter, 256 events buffer, deadline comparison, and conduit signals for VGA visualization.

The whole design was synthesized, placed, and routed with Quartus 21.1 with 0 errors. It was programmed to the FPGA and tested using the DE1-SoC board with a pthread workload. The following five demonstration tests were successfully done: sanity check, color mapping, actual workload trace, deadline violation, and slow visualization.

2 Motivation and Problem Statement

The first issue relates to the imprecision of the timing of software in the case of embedded systems. In an embedded system, timing is often a critical factor. It means that the task not only needs to give the correct result but also needs to give this result within a certain time.

For instance, if a task has to complete in 2 milliseconds, the completion of this task in 1 millisecond will be fine for the system, while the completion of this task in 5 milliseconds will not be good. In real-time systems, lateness is as much problematic as the generation of an incorrect result.

Timing of software faces three major challenges.

To start with, there is the issue of timing overhead in software. The function to measure time like $\mathrm{clock}_g ettime() has some overhead to run. A POSIX call can take from 200 ns to 500 ns. For a short task, like the one tak$

Next, there is the problem of interruptions from the Linux operating system. The Linux operating system is not strictly real-time. There will be timer interruptions, scheduling, device interruptions, and kernel operations in the operating system between the two timestamps that mark the execution of the task. In this case, the time taken to perform the task includes not only the actual task but also the overhead introduced by the operating system.

Finally, the software timer has a low resolution of around microsecond level accuracy. This level of resolution is not good enough when you have to measure a time frame that is less than microseconds. In our design, we achieve a 20 ns resolution using our FPGA counter.

For these reasons, we have opted to place the timestamp capturing in the FPGA. Our FPGA only measures task events sent by the software.

3 Revised Project Proposal

The new objective of the project was to create a real-time profiling tool that can profile tasks using hardware assistance. Software will indicate when a task is started and ended, whereas the FPGA will timestamp these events accurately, test deadlines, and display the task behavior via VGA monitor.

The final project goals were:

1. Build a custom hardware event logger in SystemVerilog.
2. Connect the event logger to the ARM HPS through an Avalon-MM memory-mapped interface.
3. Use a 32-bit free-running cycle counter running at 50 MHz.
4. Record every `TASK_START` and `TASK_END` event with 20 ns resolution.

5. Store events in a 256-entry hardware buffer.
6. Add a configurable `DEADLINE` register.
7. On every `TASK_END`, compare the actual start-to-end task runtime against the allowed deadline.
8. Set a sticky `deadline_missed` flag if the task runtime is longer than the deadline.
9. Export `current_task_id` and `deadline_missed` as Qsys conduit signals.
10. Build a VGA visualizer that shows task colors and flashes red on a deadline miss.
11. Write Linux C programs to test the hardware.
12. Package the project into a final PDF report and a `.tar.gz` archive with source files.

The initial design was intended for the DE0-Nano board, with NIOS-V as the soft core processor, and the use of FreeRTOS. But since the actual hardware in the lab was a DE1-SoC board, the design had to be implemented on the ARM HPS and Linux operating system. It was helpful in demonstrating that the `event_logger.sv` IP design is portable. The hardware event logger does not have to rely on a particular processor type.

4 System Architecture

The system is divided into two main domains:

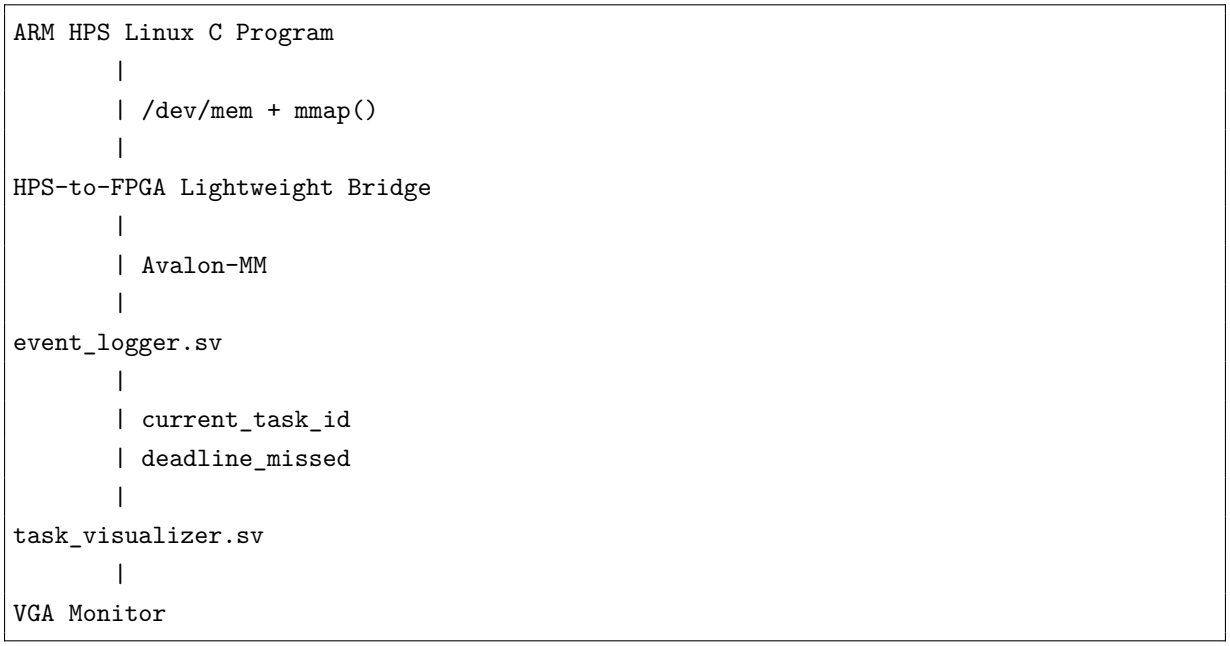
1. HPS software domain
2. FPGA hardware domain

The HPS side contains the ARM Cortex-A9 processor running Ubuntu Linux 16.04. The test programs run in user space. They access FPGA registers using `/dev/mem` and `mmap()`. This allows the C program to read and write physical FPGA addresses from Linux.

The FPGA side contains the custom hardware modules. The main module is `event_logger.sv`. It is connected as an Avalon-MM slave. It receives memory-mapped writes from the HPS. It stores event timestamps, checks deadline violations, and sends visualization signals to the VGA module.

The second FPGA module is `task_visualizer.sv`. It receives `current_task_id` and `deadline_missed` from the event logger through Qsys conduit signals. It uses those signals to generate VGA output.

The high-level system path is:



The measurement flow is:

1. Software task begins.
2. C program writes `TASK_START` to `EVENT_WRITE` register.
3. FPGA captures the current `cycle_counter` value.
4. Task runs.
5. C program writes `TASK_END` to `EVENT_WRITE` register.
6. FPGA captures the current `cycle_counter` value again.
7. FPGA computes elapsed cycles from start to end.
8. FPGA compares elapsed cycles with `deadline_cycles`.
9. If elapsed cycles are greater than `deadline_cycles`, `deadline_missed` becomes 1.

The visualization flow is separate:

1. `event_logger` updates `current_task_id`.
2. `event_logger` exports `current_task_id` through conduit.
3. `task_visualizer` receives the task ID.
4. VGA background color changes based on task ID.
5. If `deadline_missed` is 1, VGA flashes red.

The key point is that the visualization path does not go through software. The path from `event_logger` to `task_visualizer` to VGA pins is entirely inside the FPGA.

5 Hardware Design

5.1 Main Hardware Module: `event_logger.sv`

The most important hardware file is `event_logger.sv`. This module implements the event logger, the cycle counter, the event buffer, the register map, and the deadline monitor.

The module contains:

```
- Avalon-MM slave decoder
- 32-bit free-running cycle counter
- EVENT_WRITE register logic
- READ_TIMESTAMP register logic
- READ_INFO register logic
- STATUS register logic
- CONTROL register logic
- DEADLINE register logic
- timestamp buffer
- event info buffer
- read pointer
- write pointer
- entry count
- overflow flag
- buffer full flag
- last_start_ts register
- start_outstanding flag
- deadline_cycles register
- deadline_missed sticky flag
- current_task_id conduit output
- deadline_missed conduit output
```

The cycle counter increments every FPGA clock cycle. Since the FPGA clock is 50 MHz:

```
1 second = 50,000,000 cycles
1 cycle = 1 / 50,000,000 seconds
1 cycle = 20 ns
```

This means every event timestamp has 20 ns resolution.

The event logger receives events when software writes to the `EVENT_WRITE` register. The write data format is:

```
bits [15:8] = event_type
bits [7:0] = task_id
```

The two main event types are:

```
0x01 = TASK_START
```

```
0x02 = TASK_END
```

For example:

```
0x00000101 = TASK_START for task_id 1
0x00000201 = TASK_END for task_id 1
```

When a valid event write arrives, the hardware stores:

```
timestamp = current cycle_counter
event info = event_type and task_id
```

A simplified version of the hardware logic is:

```
if (write_event && !buffer_full) begin
    buf_ts[wr_ptr] <= cycle_counter;
    buf_ev[wr_ptr] <= writedata[15:0];
    wr_ptr <= wr_ptr + 1;
    entry_count <= entry_count + 1;
end
```

This means one software event write creates one hardware log entry.

5.2 Event Buffer

The event logger uses a 256-entry buffer. Each event needs to store:

```
timestamp
event type
task ID
```

The project uses separate arrays for timestamp and event information:

```
buf_ts[256] = timestamps
buf_ev[256] = event type and task ID
```

This makes the design simple. The timestamp buffer stores the 32-bit cycle counter value. The event buffer stores the smaller event information.

The buffer has:

```
write pointer = points to next entry to write
read pointer = points to next entry to read
entry_count = number of stored events
```

When the buffer is full, the hardware sets the `buffer_full` flag. If software keeps writing events when the buffer is full, the design sets an overflow flag.

5.3 Avalon-MM Register Map

The hardware/software interface has six word-addressed registers at base address `0xFF200000`.

Word Offset	Byte Address	Name	Access	Description
0	`0xFF200000`	`EVENT_WRITE`	Write	Write event type and task ID. This triggers timestamp capture.
1	`0xFF200004`	`READ_TIMESTAMP`	Read	Read the timestamp at the current read pointer.
2	`0xFF200008`	`READ_INFO`	Read	Read event type and task ID at the current read pointer.
3	`0xFF20000C`	`STATUS`	Read/Write	Read status bits. Writing advances the read pointer.
4	`0xFF200010`	`CONTROL`	Write	Clear all state or clear only deadline flag.
5	`0xFF200014`	`DEADLINE`	Write	Set deadline in clock cycles. 0 disables the monitor.

The `STATUS` register contains:

```
bits [8:0] = entry_count
bit [16] = overflow_flag
bit [17] = buffer_full
bit [18] = deadline_missed
```

6 Deadline Monitor Design

This is the most important part of the project.

The deadline monitor compares:

```
actual task runtime
```

with:

```
allowed deadline time
```

More specifically, it compares:

```
TASK_END timestamp - TASK_START timestamp
```

with:

```
deadline_cycles
```

So the exact comparison is:

```
(cycle_counter - last_start_ts) > deadline_cycles
```

Here is what each value means:

```
cycle_counter = current timestamp when TASK_END happens
last_start_ts = saved timestamp from TASK_START
cycle_counter - last_start_ts = actual task runtime
deadline_cycles = maximum allowed runtime set by software
```

So the hardware is comparing how long the task actually took against how long the task was allowed to take.

The sequence is:

1. Software writes a value to the `DEADLINE` register.
2. Software writes `TASK_START`.
3. Hardware saves the current `cycle_counter` into `last_start_ts`.
4. The task runs.
5. Software writes `TASK_END`.
6. Hardware uses the current `cycle_counter` as the end timestamp.
7. Hardware calculates `cycle_counter - last_start_ts`.
8. Hardware compares that elapsed value with `deadline_cycles`.
9. If elapsed is greater than `deadline_cycles`, hardware sets `deadline_missed_reg`.

A simplified code version is:

```
if (in_event_type == EVT_TASK_START) begin
    current_task_id <= in_task_id;
    last_start_ts <= cycle_counter;
    start_outstanding <= 1'b1;
end
else if (in_event_type == EVT_TASK_END) begin
    if (start_outstanding && deadline_cycles != 32'd0) begin
        if ((cycle_counter - last_start_ts) > deadline_cycles)
            deadline_missed_reg <= 1'b1;
    end
    start_outstanding <= 1'b0;
end
```

6.1 Simple Deadline Example

Suppose the timestamps are:

```
TASK_START timestamp = 1000 cycles
TASK_END timestamp = 1300 cycles
deadline_cycles = 250 cycles
```

The hardware calculates:

```
actual task runtime = 1300 - 1000 = 300 cycles
```

Then it compares:

```
300 cycles > 250 cycles
```

This is true, so:

```
deadline_missed = 1
```

That means the task missed its deadline.

Another example:

```
TASK_START timestamp = 1000 cycles
TASK_END timestamp = 1200 cycles
deadline_cycles = 250 cycles
```

The hardware calculates:

```
actual task runtime = 1200 - 1000 = 200 cycles
```

Then it compares:

```
200 cycles > 250 cycles
```

This is false, so:

```
deadline_missed = 0
```

That means the task finished on time.

6.2 Why the Deadline Value Is in Cycles

The `DEADLINE` register stores a number of FPGA clock cycles. Since the FPGA clock is 50 MHz:

```
1 cycle = 20 ns
```

So:

```
100 cycles = 2 us
100,000 cycles = 2 ms
50,000 cycles = 1 ms
```

This makes it easy to set a deadline in hardware.

For example, in the real workload test:

```
deadline_cycles = 100,000
100,000 cycles * 20 ns = 2 ms
```

In the forced deadline violation test:

```
deadline_cycles = 100
100 cycles * 20 ns = 2 us
```

Then the software creates a task that takes about 1 ms:

```
1 ms = 50,000 cycles
```

So the comparison becomes:

```
50,000 cycles > 100 cycles
```

This is true, so the hardware detects a deadline miss.

6.3 Sticky Flag

The `deadline_missed` flag is sticky. This means once it becomes 1, it stays 1 until software clears it.

This becomes relevant considering that the miss of a deadline can be very fast. If there is no sticky flag, then it might be possible for the software to miss out on reading the status register correctly at the correct time.

The sticky flag appears in two places:

```
STATUS[18] = deadline_missed
deadline_missed conduit output = VGA red flash signal
```

So both software and VGA can see the deadline violation.

Software can clear only the deadline flag by writing to `CONTROL[1]`.

7 VGA Visualization Design

The VGA visualization module is `task_visualizer.sv`. Its job is to make the hardware state visible on a monitor.

The module receives:

```
current_task_id[7:0]
deadline_missed
```

It outputs VGA signals for a 640 by 480 display.

The color mapping is:

Task ID / State	VGA Output
0	Dark grey / idle
1	Blue
2	Green
3	Orange
4 or higher	Purple
Deadline missed	Flashing red override

The visualizer also renders a big white number at the center of the screen. The number is rendered using an 8 by 8 bit map ROM, and it is zoomed up to 128 by 128 pixels. This helps determine which task number is running.

The VGA rendering engine is frame bufferless. This implies that there is no need for storing a whole image within the memory space. On the contrary, it calculates the color of each pixel based on the present pixel location, the present task number, and the deadline signal.

Red flashing has precedence over anything else. When `deadline_missed = 1`, the visualizer ignores the usual task color and number display and shows a red screen. It is useful for the demo since the deadline missed condition will be apparent.

One of the design choices was to retain `current_task_id` even after `TASK_END`. At the start, the design allowed clearing the task ID at `TASK_END`. But since the time duration for which the tasks will run is from 6 to 28 microseconds, whereas the duration of one VGA frame is 16.7 ms, clearing the task ID at `TASK_END` will result in a loss of color even before the monitor can display it. To avoid such an issue, the idea was to retain the latest task ID until the next `TASK_START`.

8 Software Design

The application is developed for an ARM Cortex-A9 HPS device running the Ubuntu 16.04 operating system. The application code is programmed in C.

Accessing FPGA register memory through `/dev/mem` and `mmap()` makes it possible to map

the physical addresses of the HPS-FPGA lightweight bridge interface into user-space memory addresses.

The general software setup is:

```
int fd = open("/dev/mem", O_RDWR | O_SYNC);

void *map = mmap(NULL, LW_BRIDGE_SPAN,
                 PROT_READ | PROT_WRITE,
                 MAP_SHARED,
                 fd,
                 LW_BRIDGE_BASE);

volatile uint32_t *logger_base = (volatile uint32_t *)map;
```

After this, software can write to the FPGA like this:

```
logger_base[EVT_REG_EVENT_WRITE] =
    (EVT_TASK_START << 8) | task_id;
```

and:

```
logger_base[EVT_REG_EVENT_WRITE] =
    (EVT_TASK_END << 8) | task_id;
```

The main software program is `test_logger.c`. It does the following:

1. Opens `/dev/mem`.
2. Maps the FPGA lightweight bridge.
3. Clears the logger.
4. Sets the deadline register.
5. Creates two pthread worker tasks.
6. Each worker writes `TASK_START` before doing work.
7. Each worker writes `TASK_END` after finishing work.
8. The program waits for the threads to finish.
9. It reads the event buffer.
10. It prints timing results and status flags.

The real workload uses two pthread tasks:

```
Task 1:
- 1000 loop iterations
- 500 ms sleep period
- expected VGA color: blue

Task 2:
- 5000 loop iterations
- 1000 ms sleep period
- expected VGA color: green
```

Each task runs 10 iterations. Each iteration creates two events:

```
TASK_START
TASK_END
```

So the total event count should be:

```
2 tasks * 10 iterations * 2 events = 40 events
```

The project also includes helper programs:

File	Purpose
`test_logger.c`	Main pthread workload test
`test_color.c`	Manually verifies task colors
`test_slow.c`	Slowly toggles blue/green visualization
`test_miss.c`	Forces a deadline miss and red flash
`clear.c`	Clears logger state
`demo.c`	Runs the full interactive demo
`analyze_events.py`	Parses event logs and computes timing

9 Qsys and Quartus Integration

The project uses Intel Platform Designer, also called Qsys, to connect the HPS, the bus system, and the custom event logger.

The event logger is packaged as a custom Avalon-MM slave. The HPS lightweight bridge acts as the master. The event logger also exports conduit signals for visualization.

The custom IP description file is:

```
event_logger_hw.tcl
```

This file defines:

```
- clock interface
- reset interface
```

```
- Avalon-MM slave interface
- viz conduit interface
```

The `viz` conduit includes:

```
current_task_id[7:0]
deadline_missed
```

In Platform Designer, this conduit must be exported as:

```
event_logger_0_viz
```

After HDL generation, `soc_system.v` includes ports like:

```
event_logger_0_viz_current_task_id
event_logger_0_viz_deadline_missed
```

Then `soc_system_top.sv` connects those signals to `task_visualizer.sv`.

The final compile had:

```
0 errors
+2.110 ns setup slack
+0.077 ns hold slack
```

This means the design met timing on the FPGA.

10 Build and Deployment Workflow

To reproduce the project, the workflow is below.

10.1 Open the Project

On the lab machine:

```
cd ~/Downloads/event_logger_de1soc
quartus soc_system.qpf &
```

This opens the Quartus project.

10.2 Refresh the Qsys Conduit

Open Platform Designer from Quartus:

```
Tools -> Platform Designer
```

Then:

1. Open `soc_system.qsys`.
2. Right-click `event_logger_0`.
3. Click Edit.
4. Click Finish.
5. In the Export column for `viz`, type `event_logger_0_viz`.
6. Generate HDL.
7. Close Platform Designer.

This forces Qsys to reread `event_logger_hw.tcl` and expose the new visualization conduit.

10.3 Verify the Generated Ports

Run:

```
grep "event_logger_0_viz" soc_system/synthesis/soc_system.v
```

Expected result:

```
The generated soc_system.v should include the current_task_id and deadline_missed
    conduit ports.
```

10.4 Compile the Quartus Project

Run:

```
quartus_sh --flow compile soc_system
```

This runs synthesis, fitting, assembly, and timing analysis.

10.5 Program the FPGA

Run:

```
quartus_pgm -m jtag -o "p;output_files/soc_system.sof@2"
```

This downloads the bitstream onto the DE1-SoC FPGA.

10.6 Connect to the Board Linux System

Connect through serial:

```
screen /dev/ttyUSB0 115200
```

10.7 Compile and Run the C Program on the Board

On the board:

```
gcc -O2 -o test_logger test_logger.c -lpthread
./test_logger > event_log.txt
cat event_log.txt
```

This runs the real workload and saves the output to `event_log.txt`.

11 Testing and Results

The final demo has five tests.

11.1 Test 1: Hardware Sanity Check

The first test clears the logger and reads the `STATUS` register.

Expected result:

```
entry_count = 0
overflow_flag = 0
buffer_full = 0
deadline_missed = 0
```

The VGA screen should show:

```
dark grey background
digit 0
```

This means the FPGA is programmed, the Avalon-MM register interface works, the conduit signal is alive, and the VGA output is working.

11.2 Test 2: VGA Color Mapping

The second test writes task IDs 1 through 4 and checks the screen colors.

Expected output:

```
task_id 1 = blue
task_id 2 = green
task_id 3 = orange
task_id 4 = purple
```

This verifies that this path works correctly:

```
software write -> event_logger -> conduit -> task_visualizer -> VGA
```

11.3 Test 3: Real Workload Trace

The third test runs the real pthread workload.

The setup is:

```
Task 1:
1000 loop iterations
500 ms period
blue

Task 2:
5000 loop iterations
1000 ms period
green

Deadline:
100,000 cycles = 2 ms
```

Since both tasks are much shorter than 2 ms, no deadline miss should happen.

The measured results were:

```
Task 1 execution time = 296 cycles = 5.92 us average
Task 2 execution time = 1372 cycles = 27.44 us average
40 events captured
```

The ratio between task 2 and task 1 is 4.6 times more. It is expected since task 2 has 5000 loops, while task 1 has only 1000 loops. The difference should be fivefold, and it is quite close to the actual result.

In another version of presentation, we can see:

```
40 events captured
0 buffer overflows
0 deadline misses
0.014% period error
Task 1 execution time: 5.84 us
```

```
Task 2 execution time: 27.4 us
Task 1 period: 500.07 ms
Task 2 period: 1000.13 ms
Buffer fill: 40 / 256
```

These results show that the logger captured the correct number of events and the buffer had enough capacity.

11.4 Test 4: Deadline Violation

The fourth test intentionally causes a deadline miss.

The test sets:

```
deadline = 100 cycles
```

Since:

```
1 cycle = 20 ns
```

then:

```
100 cycles = 2 us
```

Then the program makes a task take about:

```
1 ms = 50,000 cycles
```

So the hardware compares:

```
actual runtime = about 50,000 cycles
deadline = 100 cycles
```

The comparison is:

```
50,000 > 100
```

This is true, so the hardware sets:

```
deadline_missed = 1
```

The VGA screen displays in red color. The console reports a missed deadline notification. When the status value is `0x00040002`, this indicates that the 18th bit is enabled for missed deadlines, and there are two entries recorded.

This test validates the functionality of the hardware deadline monitor.

11.5 Test 5: Slow Visualization

The fifth test involves the slow toggling of task 1 and task 2 every 800 ms, making it easier to view on the VGA screen.

Expected Output:

```
blue
green
blue
green
...
```

The experiment was conducted only for demonstration purposes. Normal activities would be too fast to notice with the naked eye; hence, the need to visualize everything slowly.

12 Comparison with Software Timing

In a regular software timing process, the following procedure takes place:

```
read software timer
run task
read software timer again
subtract end time - start time
```

This is simple, but it has problems:

```
- timer calls have overhead
- Linux can interrupt the task
- resolution is limited
```

With our hardware timing, the program does this:

```
write TASK_START event
run task
write TASK_END event
```

The FPGA does the timestamp capture:

```
TASK_START timestamp = hardware cycle counter
TASK_END timestamp = hardware cycle counter
actual runtime = end timestamp - start timestamp
```

That does not prevent Linux from interrupting the process. Linux may still interrupt the process between `TASK_START` and `TASK_END`. The timestamp capture will be much more accurate, however, since it is carried out by the FPGA hardware.

The hardware technique yields:

```
20 ns resolution
single-cycle deadline detection
low event logging overhead
live hardware visualization
```

The software-only solution cannot achieve the same level of timing accuracy and hardware-based deadline flag.

13 Problems Faced and Solutions

13.1 Corruption of `event_logger.sv`

The first problem was that `event_logger.sv` got corrupted due to an accidental overwrite by the output of a Quartus programming operation. It was necessary to rebuild `event_logger.sv` from scratch.

In this process, the `DEADLINE` register, sticky `deadline_missed` flag, clear control signal for the deadline flag, internal deadline condition, and conduit signals were included.

Lesson:

```
Always be careful with command-line redirection.
Do not redirect output into important source files.
Keep backups of source files.
```

13.2 Qsys Did Not Pick Up the New Conduit

After `event_logger_hw.tcl` was updated, Qsys did not automatically expose the new conduit signals. The generated `soc_system.v` did not include the new ports.

The fix was:

1. Delete old generated folders if needed.
2. Open Platform Designer manually.
3. Right-click `event_logger_0` and choose Edit.
4. Finish the component edit so Qsys rereads the TCL file.
5. Export the `viz` conduit as `event_logger_0_viz`.
6. Generate HDL.
7. Compile again.

13.3 VGA Color Was Not Changing

However, at some stage, when running `test_logger`, the VGA screen stayed dark grey although `test_color.c` could manipulate colors manually. It meant that the conduit path was functioning properly, but the real workload went faster than one could see.

It turned out that `current_task_id` had to be cleared at `TASK_END`. However, tasks took only 6 to 28 microseconds while a VGA frame is equal to 16.7 ms; hence, the color was cleared very fast. Therefore, it became clear that the color had to be kept until the next `TASK_START` by stopping clearing `current_task_id` at `TASK_END`.

13.4 Problems with Serial Transfers

Yet another problem was related to transferring large C files using serial connection. By pasting large files through `screen /dev/ttyUSB0 115200`, one risked to have their files truncated or with some escape codes in them. The possible solution included base64 encoding, file splitting, etc.

13.5 Platform Pivot

While the initial plan was to build on top of DE0-Nano with NIOS-V and FreeRTOS, the implementation relied on DE1-SoC with ARM HPS and Linux. In order to fix this problem, a software access approach to `/dev/mem` and `mmap()` had to be adopted. Fortunately, there was no need for large modifications to the SystemVerilog code for event logging, indicating portability of the IP.

14 Individual Contributions

14.1 Teresa Co

Teresa contributed to the organization of the final project direction, testing, integration, report structure, explanation for the final presentation, and project documentation. Specifically, she helped clarify the motivation, overall system architecture, behavior of the deadline monitor, VGA visualization of the data, testing results, and the final submission process.

14.2 Handong He

Handong was more involved in the implementation and the tracking of its progress. This involved reworking `event_logger.sv`, including the `DEADLINE` register, the `STATUS[18]` register for the sticky deadline, `CONTROL[1]` for clearing the deadline bit, the deadine comparison state

machine, and the export of the visualization signal conduit.

Handong also assisted in Quartus compilation, FPGA programming, board testing, and demo validation.

14.3 Xiao Lu

Xiao assisted in the implementation, integration, testing, and preparation for the final demonstration. Xiao also participated in the verification of proper interaction between the hardware and software and the validation of the demo behavior.

14.4 Shared Team Work

In our group, we focused on debugging, testing, documentation, and delivery. Debugging and testing include SystemVerilog design, C software, Linux memory mapping, Qsys, Quartus compiling, VGA, and finally, hardware testing. The reason the final result worked well was that each piece of work went through tests by themselves.

15 Lessons Learned

Firstly, hardware/software integration is time-consuming. While writing a SystemVerilog file is one thing, the module should be also connected in Platform Designer, exported properly, compiled in Quartus, configured into the FPGA, accessed through Linux, and tested on the hardware.

Secondly, small test code is extremely helpful when doing such tasks. While using a full test case to test everything might be helpful, writing specific test cases to test color mapping, slow visualization, deadline miss response, and logger cleaning made things simpler.

Thirdly, signals generated by hardware might be too fast for human eyes to observe. Although the activity of the task is really there, the VGA monitor cannot demonstrate a microsecond-level signal. Therefore, waiting for the next `TASK_START` to hold the task id makes things much easier.

Fourthly, it is important to differentiate between the files produced by the design tools and those written by hand. Quartus and Platform Designer produce many files automatically. In the final submission, it is important to explicitly list what was produced by the team versus what was produced by the tool.

Fifthly, deadline checking is more straightforward in hardware. The FPGA can compare the actual runtime of the task with its deadline in just one clock cycle. It is also possible to check the comparison of two timestamps on the software side, but this requires CPU time and may be delayed by Linux.

16 Recommendations for Future Projects

Future teams should integrate their modules early. While they may work well in isolation, problems arise during the connection with Qsys, HPS software, and the board's pins.

Future teams should maintain a tidy project directory. The source files, auto-generated files, documentation, and test files should be kept separate. This makes the final packaging process far easier.

Future teams should maintain backups. Overwriting a source file accidentally can cost significant amounts of time.

Future teams should write one small test for each feature. For example:

```
test register read/write
test VGA color
test deadline miss
test real workload
test buffer readout
```

Future work on the team can include creating demonstrations for humans, rather than only for the hardware. Real hardware events will happen too quickly to see. Visualizing things slowly will make the demo simpler to present.

Future work should also include documenting the construction process throughout building. It is difficult to remember how to correctly use Quartus, Platform Designer, the serial port, and the board at the end of the project.

17 Future Improvements

The first potential improvement includes task-specific deadlines. Currently, there is only one deadline. In an improved future system, there would be individual deadlines for individual tasks.

The second improvement would include multiple outstanding tasks. Current systems rely on one variable to keep track of a `last_start_ts`. Future systems may want to have one `last_start_ts` per task identifier.

The third improvement would include DMA-based event dumping. Currently, events are read out through the register. In a more advanced system, events could be moved into DDR through DMA.

The fourth improvement would be increased event buffer size. Current buffers are sized at 256 events. Future event buffers will be much larger, perhaps having hundreds or even thousands of entries.

The sixth improvement is the implementation of enhanced VGA output. For example, the monitor may display the total number of events, the number of utilized buffer spaces, the current deadline, the total number of deadlines missed, or even a miniature timeline of recently processed events.

18 Complete Listing of Files Written for the Project

In the final report, all the files developed during the project should be included. However, it must be emphasized that files generated by Quartus cannot be presented as handwritten codes.

18.1 Hardware Source Files

```
event_logger.sv
task_visualizer.sv
soc_system_top.sv
event_logger_hw.tcl
```

18.1.1 `event_logger.sv`

This is the main hardware event logger. It implements:

```
- Avalon-MM slave interface
- 32-bit cycle counter
- event timestamp capture
- 256-entry buffer
- register map
- deadline monitor
- sticky deadline flag
- current_task_id output
- deadline_missed output
```

18.1.2 `task_visualizer.sv`

This is the VGA display module. It implements:

```
- VGA timing
- background color based on task ID
- centered white digit
- red flashing deadline miss override
- no frame buffer
```

18.1.3 `soc_system_top.sv`

This is the top-level hardware file. It connects:

```
- Qsys system
- board pins
- VGA outputs
- event logger conduit signals
- task_visualizer instance
```

18.1.4 `event_logger_hw.tcl`

This describes the custom IP to Platform Designer. It defines:

```
- clock interface
- reset interface
- Avalon-MM slave interface
- visualization conduit interface
```

18.2 Quartus and Qsys Files

```
soc_system.qpf
soc_system.qsf
soc_system.qsys
soc_system.sdc
Makefile
```

18.3 C Source Files

```
test_logger.c
test_color.c
test_slow.c
test_miss.c
clear.c
demo.c
```

18.4 Analysis and Documentation Files

```
analyze_events.py
README.md
README_UPDATE.md
DESIGN_DOCUMENT.md
DESIGN_DOCUMENT.docx
```

```
PROGRESS_REPORT.md
PROGRESS_REPORT.docx
TECHNICAL_GUIDE.md
TECHNICAL_GUIDE.docx
Demo_Presentation_Script.pdf
CSEE4840_Final_Presentation.pdf
```

18.5 Files Not Written Manually

These files should not be listed as handwritten source code:

```
db/
incremental_db/
output_files/
soc_system/synthesis/ generated files
*.rpt
*.sof
*.done
*.summary
*.pin
*.jdi
*.qws
```

These files might be produced using Quartus or Platform Designer. They could prove helpful when reconstructing the design or debugging it; however, they do not constitute source files authored by the team.

19 Final `.tar.gz` Package

Lastly, you must provide a `.tar.gz` package containing only the files required to compile the project.

A good `myfiles` list could be:

```
event_logger.sv
task_visualizer.sv
soc_system_top.sv
event_logger_hw.tcl
soc_system.qpf
soc_system.qsf
soc_system.qsys
soc_system.sdc
Makefile
test_logger.c
test_color.c
```

```
test_slow.c
test_miss.c
clear.c
demo.c
analyze_events.py
README.md
README_UPDATE.md
DESIGN_DOCUMENT.md
```

Then create the archive with:

```
tar zcf project.tar.gz $(cat myfiles)
```

This comes from the project instructions which require the archive to contain the necessary files used in compiling the project, just like for lab submission.

The `.tar.gz` file should not contain any unnecessary large generated files unless specifically required by the instructor. The aim of creating this archive file is to submit the clean source and project files necessary to reconstruct the project.

20 Conclusion

This project was successful in implementing a FPGA-based real-time task profiler and deadline monitor on the DE1-SOC platform. It enables the software on Linux to trigger task start and task end events through write operations on some memory mapped registers. The FPGA records the events based on a 32-bit counter clocked at 50 MHz which yields 20 ns resolution.

The most critical part of this implementation is the comparison of deadlines in hardware. On receiving `TASK_START`, the hardware stores the start timestamp. On receiving `TASK_END`, the hardware stores the current timestamp as end time. Then:

```
actual task runtime = TASK_END timestamp - TASK_START timestamp
```

Then it compares:

```
actual task runtime > deadline_cycles
```

In case the measured runtime exceeds the deadline, the hardware generates a sticky bit `deadline_missed`. The value can be read from `STATUS[18]` by the software and from the conduit signal by the VGA visualizer.

The second feature implemented is live VGA visualization. It displays the running task with colored status and red flashing in case of a missed deadline. Thus, the functionality of the hardware becomes easily observable during the demonstration.

The final design was connected to the Qsys environment, synthesized and assembled in Quartus. After programming it into the FPGA, all five designed tests have been successfully conducted:

sanity test, VGA color mapping, workload trace processing, deliberate deadline miss, and slower visualization.

In summary, it can be stated that this project proves the ability of hardware timestamping to deliver more precise timing data compared to software profiling of embedded applications.