Execution Performance Tester User Guide for Speculative Approach to Clipping Line Segments Algorithm

Zsolt Bagoly

January 23, 2018

1 Measurement Environment Overview

The scope of the present execution performance tester (EPT) is an FPGA NIOS II based embedded design for measuring the execution performance of the speculative approach to clipping line segments algorithm. The system complies with the following requirements listed below:

Execution-sensitive parameters: with proper usage of these values it can be guaranteed that the CPU processes the code snippet in 100%.

- CPU instruction and/or data cache - *not implemented*
- Branch prediction - *not implemented*
- Interrupt requests - *disabling all IRQ input lines at measurement*

Cycle counter: that module is similar to the Intel's timestamp cycle counter (TSC) solution; an embedded n-bit counter counts the clock cycles (this module is driven by the same clock source as the CPU). Its value can be read via Avalon or NIOS II custom instruction interface. The counter reset is available only via Avalon bus.

Basic test procedure in pseudo code:

1. Disable all IRQ
2. Calibrate counter: $resetCounter; getCounter(timeStamp0); getCounter(timeStamp1) \rightarrow offset = timeStamp1 - timeStamp0$
3. Measuring: $resetCounter; getCounter(timeStamp0); |...algorithmToBeMeasured...| getCounter(timeStamp1);$
4. Evaluate the result: $elapsed = timeStamp1 - timeStamp0 - offset; timeMicroSec = elapsed/(systemClock/1000000)$
5. Enable all IRQ

Test Procedure Environment:

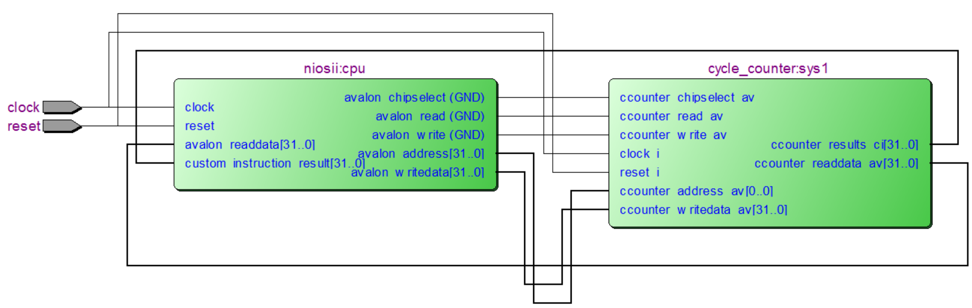
- Run N times the basic procedure and calculate the measurement accuracy $\rightarrow tolerance = max - min; accuracy = \frac{resultAverage - tolerance}{resultAverage} \cdot 100\%$
- The speculative line clipper algorithm contains 11 segments, so a constant test vector array is created based on these parts. The test vectors contain the input arguments for the line clipper functions.
- At performance measurement the basic procedure is executed N times in each test vector. Finally a simple report generator displays the result.

2 Hardware Architecture

It consists of the following parts:

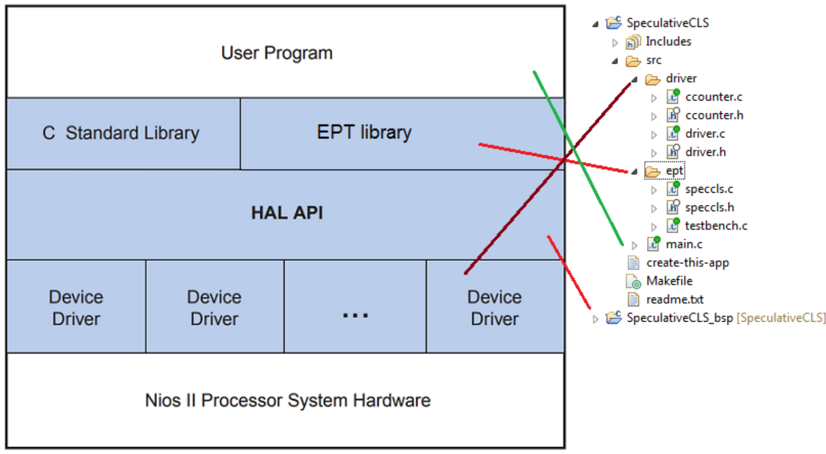
- Clock source: $f_{sys} = 50MHz$
- NIOS II CPU economy core
- Floating point arithmetic and logical custom instruction
- Cycle counter with Avalon and CI interface
- 200 KB ON-chip RAM (minimizing the latency and maximizing the accuracy - SDRAM is not suitable for this)
- JTAG UART interface
- SystemID peripheral

N-bit Cycle counter: The main question is how to determine the value of the N? Should it be implemented as a Byte, a Word, a DWord or may be a QWord counter? Intel TSC uses 64-bit solution but this is overkill for that kind of measurement. Let consider the DWord design: at 50 MHz frequency the maximum execution duration can be $t_{max} = \frac{2^{32}-1}{50 \cdot 10^{-6}} \approx 85.9sec$ without overflow. It should be enough if reset is inserted before each measurement starts.



Cycle counter's implementation

3 Software Architecture



The software architecture

User Program: In this case just the main function.

```c
1  int main()
2  {
3    specClsResult_t resultVector[getTestVectorSize()];
4    alt_irq_context context;
5
6    //---Measurement Environment---
7    context = alt_irq_disable_all();
8
9    // Run the measurement based on the Test Vector constants
10   runVectorMeasurement(resultVector);
11
12   alt_irq_enable_all(context);
13   //---Measurement Environment---
14
15   // Report and validation
16   generateReport(resultVector);
17
18   return 0;
19 }
```

1. *Line 7.:* Disable all IRQ

2. *Line 10.:* Execute the vector measurement which wraps the speculative line clipper algorithm. A single point test is being executed 10 times for calculating the measurement accuracy.

3. *Line 12.:* Enable all IRQ

4. *Line 16.:* Generate report from the result vector.

The Test Vector Constant: Contains the input arguments for the clipper.

```c
// Speculative CLS data type
typedef struct specClsData
{
  float x1;
  float y1;
  float x2;
  float y2;
  float xL;
  float yB;
  float xR;
  float yT;
  char segment[SEGMENT_CHAR_LENGTH];
} specClsData_t;
// Specify vectors for testbench
//  @ modify the test based on data requirement
const specClsData_t testVector[] =
{
//  x1   y1   x2   y2   xL   yB   xR   yT  Segment
  { 50, 250,  60, 260, 100, 100, 200, 200, "Top-left corner"},
  { 50, 150,  60, 160, 100, 100, 200, 200, "Left-edge region"},
  { 50,  50,  60,  60, 100, 100, 200, 200, "Bottom-left corner"},
  {150, 250, 160, 260, 100, 100, 200, 200, "Top-edge region"},
  {150, 150, 160, 160, 100, 100, 200, 200, "Window region"},
  {150,  50, 160,  60, 100, 100, 200, 200, "Bottom-edge region"},
  {250, 250, 260, 260, 100, 100, 200, 200, "Top-right corner"},
  {250, 150, 260, 160, 100, 100, 200, 200, "Right-edge region"},
  {250,  50, 260,  60, 100, 100, 200, 200, "Bottom-right corner"},
  { 50, 150, 110, 160, 100, 100, 200, 200, "Left-edge region, k=1, C&T wins"},
  { 90, 190, 110, 240, 100, 100, 200, 200, "Left-edge region, k=2, C&T loses"}
};
```

Cycle Measurement: The clipper algorithm itself contains the assembly macros for getting the counter values.

```
// Cycle counter data type
typedef struct ccounter
{
  alt_u32 timeStamp0;
  alt_u32 timeStamp1;
  alt_u32 offset;
  alt_u32 elapsed;
} ccounter_t;
//Speculative Algorithm to Clipping Line Segments
ccounter_t clip(float x1, float y1, float x2, float y2, float xL, float yB, float xR, float yT)
{
  ccounter_t ccounter;
  float x;

  // Calibrating the cycle counter
  ccounter = ccounterCalibration();

  // Start the measurement
  CCOUNTER_reset; // Reset the cycle counter
  ccounter.timeStamp0 = CCOUNTER_getValueCI; // Get cycle counter's value via custom instruction macro
  // The speculative algorithm
  if (x1 < xL)
  {
    if (x2 < xL)
    {
      // Stop and evaluate the measurement
      ccounter.timeStamp1 = CCOUNTER_getValueCI; // Get cycle counter's value via custom instruction macro
      cycleElapsed(&ccounter); // Calculating the elapsed time
      return ccounter; // Return with the measured data
    }
  }
  // ... other "else if / else" branches with same cycle counter handling
}
```

4 Measurement Results

The report generator prints the measurement results to the NIOS II console including all the 11 segments. The accuracy - based on 10 times test repeating in each segment - values are 100% because of the skipped system related non-deterministic parameters (cache, branch prediction and disabled IRQ's).

```

--- Measurement Report ----

Vector | CycleAVG[1] | CycleAVG[%] | TimeAVG[us] | ACC[%] | Segment
1.          68           4.878          1.360    100.0   Top-left corner
2.          68           4.878          1.360    100.0   Left-edge region
3.          68           4.878          1.360    100.0   Bottom-left corner
4.         136           9.756          2.720    100.0   Top-edge region
5.         408          29.268          8.160    100.0   Window region
6.         102           7.317          2.040    100.0   Bottom-edge region
7.         136           9.756          2.720    100.0   Top-right corner
8.         170          12.195          3.400    100.0   Right-edge region
9.         102           7.317          2.040    100.0   Bottom-right corner
10.         68           4.878          1.360    100.0   Left-edge region, k
    =1, C&T wins
11.         68           4.878          1.360    100.0   Left-edge region, k
    =2, C&T loses
-----------
Total:      1394     100.000      27.880
Average:     126                   2.535
Repeated test in each single vector point: 10
System clock: 50 MHz
```

Report summary of measurement results