

CSCE 313: Embedded Systems

Performance Analysis and Tuning

Instructor: Jason D. Bakos



Performance Considerations

- time per frame = (number of pixels) * (time per pixel)
(fixed)
- time per pixel = (cycles per pixel) * (clock period)
(fixed)
- cycles per pixel = (instructions per pixel) * (cycles per instruction)
Influenced by:
program code,
compiler
Influenced by:
stalls,
cache misses



Easy Performance Tweaks (Do First!)

1. Add floating-point hardware

- Nios2 has no hardware floating point instructions
 - Compiler replaces float multiplies with function call to `__muldf3`
 - Compiler replaces int-to-float typecast to `__floatsisf`
- Add floating point “custom instructions”
 - Works for float only (not double)
 - Floating point instructions performed with the “custom” instruction
- Make sure you don’t have any double types in your code
 - `cos()` should be `cosf()`, `floor()` should be `floorf()`
 - scalar constants: `1.0` should `1.0f`

2. Turn on compiler optimization (in APP directory)

- Command:
`nios2-app-update-makefile --app-dir . --set-optimization -O3`



Floating-Point Unit

Floating Point Hardware 2 - nios_custom_instr_floating_point_2_0

altera_nios_custom_instr_floating_point_2

Block Diagram

Show signals

Description

This component is the 2nd generation set of single-precision floating-point custom instructions. It offers improved performance and lower resource usage than the 1st generation at the expense of full IEEE 754 compliance (limited subnormal support and simplified rounding modes).

This component supports optional functional blocks with the following operations:

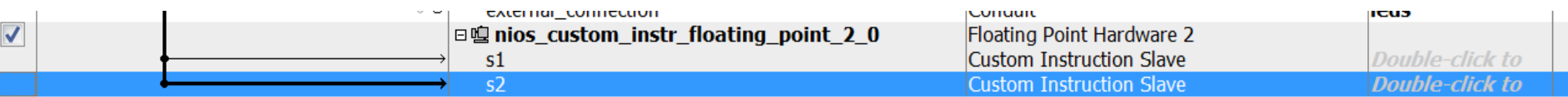
Function	Operation	Cycles to Execute
Arithmetic	add/subtract	5
	multiply	4
	divide	16
	negate/absolute	1
Roots	square root	8
Conversions	integer to float	4
	float to integer	2
Comparisons	min/max	1
	less than/equal	1
	greater than/equal	1
	equal/not equal	1

Both slaves must be connected to the Nios II custom_instruction_master. Include system.h to take advantage of square root and min/max operations if they are enabled.

Function

- Arithmetic
- Roots
- Conversions
- Comparisons

Cancel Finish



Counting Cycles

- Performance counters
 - Hardware counters that increment under certain conditions
 - E.g. count cache misses, branch mispredictions, load/store instructions

- Nios2 only provides cycle counters
 - In Lab 3, we will use them to time sections of code
 - counter = <current counter state>
 - <code to time>
 - cycles elapsed = <current counter state> - counter



Performance Counters in NIOS SOPC

- To use, add in Platform Designer:
 - Performance Counter Unit
 - Name: "performance_counter_0"
 - Number of simultaneous counters = 3 (default)
 - Make sure you put it on **sys_clk**

Performance Counter Unit Intel FPGA IP
altera_avalon_performance_counter

Configuration

Number of simultaneously-measured sections:

Description

This peripheral and associated software macros allow minimally-intrusive, real-time hardware profiling of your software program.

You can simultaneously measure several *sections* of your program. Each measured section uses both a **64-bit** time-counter and a 32-bit occurrence counter. The time counter measures the total time spent in a section of code with single-clock resolution. The occurrence-counter measures how many times a section of code is entered.

Macros declared in this peripheral's header file make it easy to start and stop counters when entering and exiting sections of C-code. C library routines allow you to retrieve and analyze the results.

See the datasheet in the Documentation links for more information.

Performance Counters in NIOS SOPC

- To use:
 - include file:

```
#include <altera_avalon_performance_counter.h>
```
 - declare in outer function:

```
alt_u64 time1,time2,cycles;
```
 - Measure cycles per pixel:

```
time1=perf_get_total_time ((void*)PERFORMANCE_COUNTER_0_BASE);  
PERF_START_MEASURING(PERFORMANCE_COUNTER_0_BASE);  
  
<pixel code>  
  
time2=perf_get_total_time ((void*)PERFORMANCE_COUNTER_0_BASE);  
PERF_START_MEASURING(PERFORMANCE_COUNTER_0_BASE);  
cycles = time2 - time1;
```



Set up Platform Design

1. Delete the interval timer from **Platform Designer** AND **bsp-editor!**

- Ignore Platform Designer warnings about “no matching role found” relating to custom instruction module

2. In BSP directory:

nios2-bsp-bsp-editor

- Generate the BSP

sys_clk_timer:

none ▼

timestamp_timer:

none ▼

3. Re-generate BSP (in BSP directory)

- `nios2-bsp-generate-files --settings settings.bsp --bsp-dir .`



Dyanamic Instruction Counting with GDB

- Copy contents of the pixel loop into a separate function
- Add **__attribute__((noinline))** between return type and function name
 1. “make” in app directory:
 2. cd software/lights
 3. make



Dynamic Instruction Counting with GDB

- In gdb:

```
target remote localhost:8000
load
b iteration
c
stepi
set pagination off
set $count=0
while $pc != iteration
stepi
set $count=$count+1
end
print $count
```



Data Types

- Colors are only 8 bits
- Likewise, weights require 8 bits of precision
 - Weight value of 2^{-8} corresponds to one increment
- Row and col require 10-bit range
 - $\pm 2^9 = [-512, 511]$ will encompass column value for 320×240
- Floating point is overkill for this application
- Use fixed point as 32-bit int with 8 fractional bits



Fixed-Point

- Need a way to represent fractional numbers in binary
- (N,M) Fixed-point representation
 - Assume a decimal point at some location in a value:
 - Example (6,4)-fixed format:

- 6-bit (unsigned) value

1	0.	1	1	0	1
---	----	---	---	---	---
- $= 1 \times 2^1 + 0 \times 2^0 + 1 \times 2^{-1} + 1 \times 2^{-2} + 0 \times 2^{-3} + 1 \times 2^{-4}$
- Range = $[0, 2^{N-M} - 2^{-M}] = [0, 4 - 2^{-4}] = [0, 3.9375]$
- For signed, use two's complement
 - Range = $[-2^{N-M-1}, 2^{N-M-1} - 2^{-M}]$



Lab 3 Fixed-Point

- C doesn't offer a dedicated fixed-point type
- We don't have fixed-point sin and cos
- Use floating-point for this; convert to (32,9) fixed point by:
 - Multiply result by 512.0
 - Typecast result to int
- Multiply:
 - cos/sin, which is (32,9) fixed-point value with
 - row/col, which is a (32,0) fixed-point number,
 - gives a (64,9) value, but if you assign drops upper 32 bits
- Adding two (32,9) values gives a (32,9) value



Lab 3

- Split into two parts:
 - Change CPU to NIOS II/f
 - Enable hardware multiply

Main Vectors Caches and Memory Interfaces Arithmetic Instructions MMU and MPU

Select an Implementation

Nios II Core: Nios II/e Nios II/f

	Nios II/e	Nios II/f
Summary	Resource-optimized 32-bit RISC	Performance-optimized 32-bit RISC
Features	JTAG Debug ECC RAM Protection	JTAG Debug Hardware Multiply/Divide Instruction/Data Caches Tightly-Coupled Masters ECC RAM Protection External Interrupt Controller Shadow Register Sets MPU MMU
RAM Usage	2 + Options	2 + Options

Main Vectors Caches and Memory Interfaces Arithmetic Instructions MMU and MPU Settings

Multiply/Shift/Rotate Hardware: Auto Selection

Divide Hardware: None

Arithmetic Implementation

Multiply Implementation: 3 16-bit multipliers

Multiply Extended Implementation: None

Shift/Rotate Implementation: Logic elements (pipelined)

Summary

Operation	Performance	Resources	Instructions
Multiply	1 cycle	3 16-bit multipliers	MUL, MULI
Multiply Extended	Low	None (Software Implementation)	MULXSS, MULXSU, MULXUU
Shift/rotate	1 cycle	Logic elements (pipelined)	ROL, ROLI, ROR, SLL, SLLI, SRA, SRAI, SRL, SRLI
Divide	Low	None (Software Implementation)	DIV, DIVU



Lab 3

- Use performance counters to find average time, in cycles, to transform each individual pixel when using different cache sizes:
 - 4K instruction cache, 4K data cache
 - 4K instruction cache, 16K data cache

The screenshot shows a configuration window with tabs: Main, Vectors, Caches and Memory Interfaces, Arithmetic Instructions, and MMU. The 'Caches and Memory Interfaces' tab is active. It contains several sections:

- Instruction Cache**: Size: 4 Kbytes, Add burstcount signal to instruction_master: Disable.
- Flash Accelerator**: Line Size: None, Number of Cache Lines: 2.
- Data Cache**: Size: 4 Kbytes (circled in yellow), Victim buffer implementation: RAM, Add burstcount signal to data_master: Disable, Use most-significant address bit in processor to bypass data cache.
- Tightly-coupled Memories**: Number of tightly coupled instruction master ports: None, Number of tightly coupled data master ports: None.
- Peripheral Region**: Size: None, Base Address: 0x00000000.



Lab 3

- For floating- and fixed-point, calculate:
 - Instructions per pixel
 - Make sure transformation doesn't map to an out-of-bounds pixel!
- And, for data cache sizes of 4KB and 16KB:
 - Cycles per pixel
 - Cycles per instruction (CPI)
- Calculate speedup of fixed relative to floating for each cache
- Calculate speedup of 16KB cache over 4KB cache for each datatype

