Department of Electrical and Computer Engineering
Rutgers, The State University of New Jersey

Computer Architecture and Assembly Language Lab

Fall 2016

Lab 6

GPU Parallelism and performance

Goal

After completing this lab, you will:

- Know the basic concepts of the Graphics Processing Unit (GPU)
- Be able to write a simple program in a simplified GPU Assembly Language

Preparation

Please read Chapter 6 and Appendix C which is an online content in the textbook. This knowledge is required for this lab.

Introduction

Nowadays, with the development of the gaming industry and video streaming, it seems we need something more than CPUs for real-time graphics processing. The GPU is a processor optimized for graphics, video and visual computing and display [1]. Basically, it is different from general-purpose CPUs since:
- The GPU acts like a supplement of a CPU, it does not need to be able to perform all the tasks the CPU does. Therefore, GPUs dedicate all their resources to graphics.
- GPUs rely on hardware multithreading and high parallelism.
- DRAM chips used for GPUs have wider and higher bandwidth than DRAM chips for CPUs.

Department of Electrical and Computer Engineering
Rutgers, The State University of New Jersey

- **The Historical PC (circa 1990)**

Figure C.2.1 shows a high-level block diagram of a legacy PC, circa 1990. The north bridge (see Chapter 6) contains high-bandwidth interfaces, connecting the CPU, memory, and PCI bus. The south bridge contains legacy interfaces and devices: ISA bus (audio, LAN), interrupt controller; DMA controller; time/counter. In this system, the display was driven by a simple frame buffer subsystem known as a VGA (video graphics array) which was attached to the PCI bus. Graphics subsystems with built-in processing elements (GPUs) did not exist in the PC landscape of 1990s. In fact, until graphics cards were introduced in the mid 90s, all graphics computation were done in software by the CPU. Then graphics cards were introduced to offload the heavy graphics computations from the CPU. The first graphics cards were expensive, and had two chips on them, one for performing geometry calculations, the other doing pixel computations. Then nVidia unified these two chips into one, called the GPU.

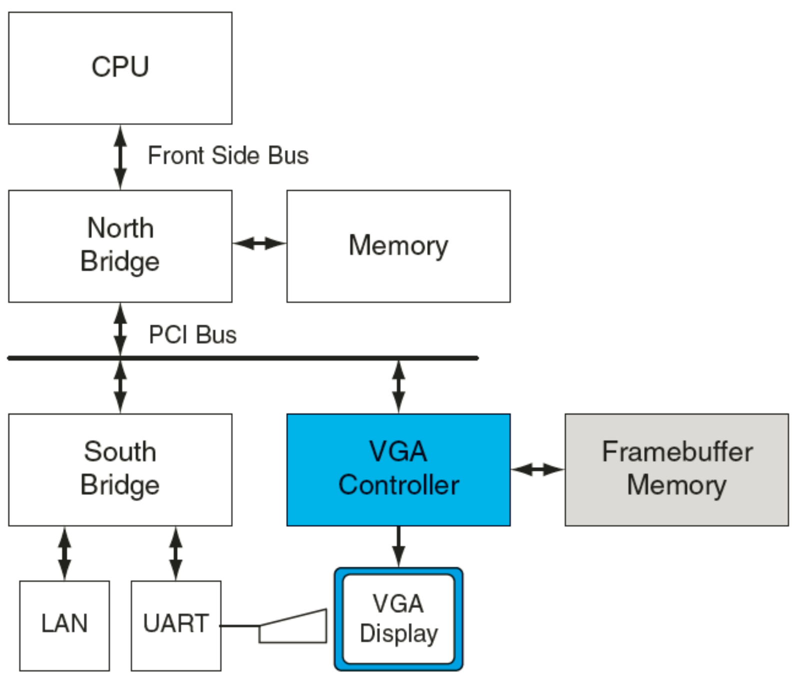

FIGURE C.2.1 Historical PC. VGA controller drives graphics display from framebuffer memory.

- **GPU System Architectures**

In this section, we survey GPU system architectures in common use today. We discuss system configurations, GPU functions and services, standard programming interfaces, and a basic GPU internal architecture.

Department of Electrical and Computer Engineering
Rutgers, The State University of New Jersey

- **Heterogeneous CPU–GPU System Architecture**

A heterogeneous computer system architecture using a GPU and a CPU can be described at a high level by two primary characteristics: first, how many functional subsystems and/or chips are used and what are their interconnection technologies and topology; and second, what memory subsystems are available to these functional subsystems. See Chapter 6 for background on the PC I/O systems and chip sets.

Figure C.2.2 illustrates two configurations in common use today. These are characterized by a separate GPU (discrete GPU) and CPU with respective memory subsystems. In Figure C.2.2a, with an Intel CPU, we see the GPU attached via a 16-lane PCI-Express 2.0 link to provide a peak 16 GB/s transfer rate, (peak of 8 GB/s in each direction). Similarly, in Figure C.2.2b, with an AMD CPU, the GPU is attached to the chipset, also via PCI-Express with the same available bandwidth. In both cases, the GPUs and CPUs may access each other's memory, albeit with less available bandwidth than their access to the more directly attached memories. In the case of the AMD system, the north bridge or memory controller is integrated into the same die as the CPU.

A low-cost variation on these systems, a unified memory architecture (UMA) system, uses only CPU system memory, omitting GPU memory from the system. These systems have relatively low performance GPUs, since their achieved performance is limited by the available system memory bandwidth and increased latency of memory access, whereas dedicated GPU memory provides high bandwidth and low latency.

A high performance system variation uses multiple attached GPUs, typically two to four working in parallel, with their displays daisy-chained. An example is the NVIDIA SLI (scalable link interconnect) multi-GPU system, designed for high performance gaming and workstations.

The next system category integrates the GPU with the north bridge (Intel) or chipset (AMD) with and without dedicated graphics memory. Chapter 5 explains how caches maintain coherence in a shared address space. With CPUs and GPUs, there are multiple address spaces. GPUs can access their own physical local memory and the CPU system's physical memory using virtual addresses. Generally, the GPU assumes a different address space from the actual physical address of the CPU's memory, called Virtual Address Space. This Virtual Address Space is translated to actual physical memory addresses by a unit in the GPU called Memory Management Unit or MMU. The operating system kernel manages the GPU's page tables. A system physical page can be accessed using either coherent or non-coherent PCI-Express transactions, determined by an attribute in the

GPU's page table. The CPU can access GPU's local memory through an address range (also called aperture) in the PCI-Express address space.

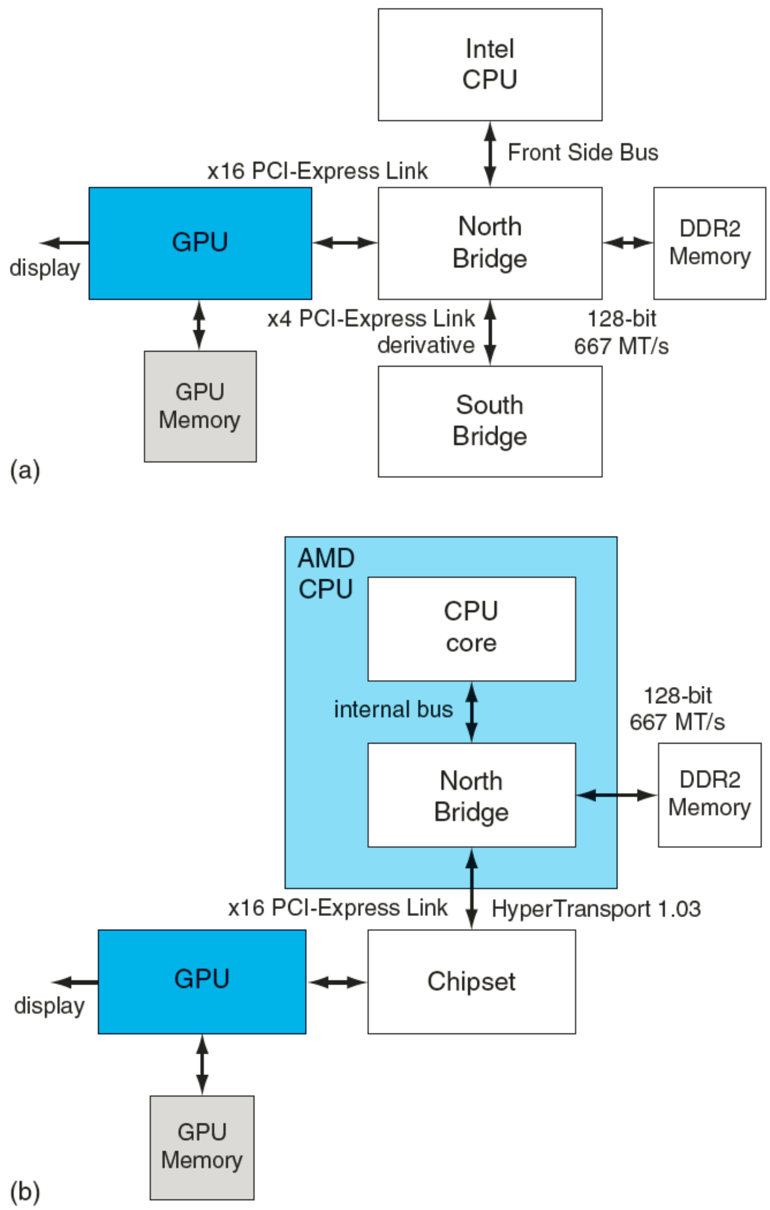

FIGURE C.2.2 Contemporary PCs with Intel and AMD CPUs. See Chapter 6 for an explanation of the components and interconnects in this figure.

- **Graphics Pipeline**

FIGURE C.2.3 Graphics logical pipeline. Programmable graphics shader stages are blue, and fixed-function blocks are white.

Figure C.2.3 illustrates the major processing stages, and highlights the important programmable stages (vertex, geometry, and pixel shader stages).

- **Mapping Graphics Pipeline to Unified GPU Processors**
  Figure C.2.4 shows how the logical pipeline comprising separate independent programmable stages is mapped onto a physical distributed array of processors.

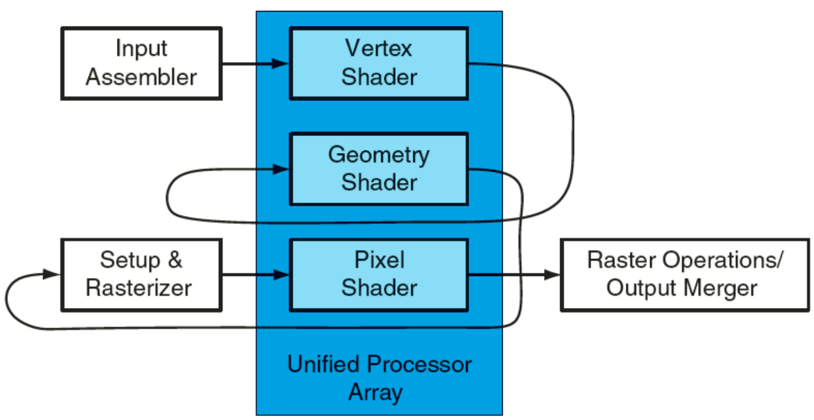

FIGURE C.2.4 Logical pipeline mapped to physical processors. The programmable shader stages execute on the array of unified processors, and the logical graphics pipeline dataflow recirculates through the processors.

- **Basic Unified GPU Architecture**

Unified GPU architectures are based on a parallel array of many programmable processors. They unify vertex, geometry, and pixel shader processing and parallel computing on the same processors, unlike earlier graphics cards which had separate processors dedicated to each processing stage of the graphics pipeline. The programmable processor array is tightly integrated with fixed function processors for texture filtering, rasterization, raster operations, anti-aliasing, compression, decompression, display, video decoding, and high-definition video processing. Although the fixed-function processors significantly outperform more general programmable processors in terms of absolute performance constrained by an area, cost, or power budget, we will focus on the programmable processors here. Compared with multicore CPUs, many core GPUs have a different architectural design point, one focused on executing many parallel threads efficiently on many processor cores so to assure fast (as in "real-time" – or instantaneous) graphics. By using many simpler cores and optimizing for data-parallel behavior among groups of threads, more of the per-chip transistor budget is devoted to computation, and less to on-chip caches and overhead.

- **Processor Array**

A unified GPU processor array contains many processor cores, typically organized into multithreaded multiprocessors. Figure C.2.5 shows a GPU with an array of 112 streaming processor (SP) cores, organized as 14 multithreaded streaming multiprocessors (SMs). Each SP core is highly multithreaded, managing 96 concurrent threads and their state in hardware. The processors connected with four 64-bit-wide DRAM partitions via an interconnection network. Each SM has eight SP cores, two special function units (SFUs), instruction and constant caches, a multithreaded instruction unit, and a shared memory. This is the basic Tesla architecture implemented in the NVIDIA GeForce 8800 graphics card (not to be confused with Tesla the car, or Tesla the late scientist who invented for Edison). It has a unified architecture in which the traditional graphics programs for vertex, geometry, and pixel shading run on the unified SMs and their SP cores, and computing programs run on the same processors.

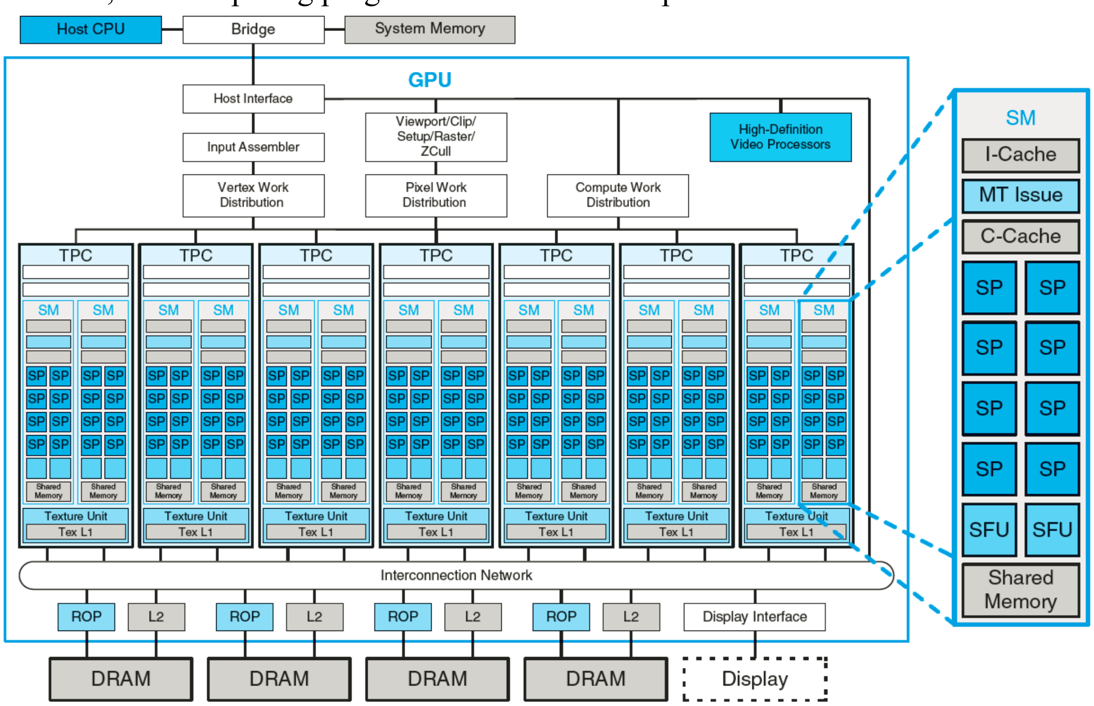

FIGURE C.2.5 Basic unified GPU architecture. Example GPU with 112 streaming processor (SP) cores organized in 14 streaming multiprocessors (SMs); the cores are highly multithreaded. It has the basic Tesla architecture of an NVIDIA GeForce 8800. The processors connect with four 64-bit-wide DRAM partitions via an interconnection network. Each SM has eight SP cores, two special function units (SFUs), instruction and constant caches, a multithreaded instruction unit, and a shared memory.

The processor array architecture is scalable to smaller and larger GPU configurations by scaling the number of multiprocessors and the number of memory partitions. Figure C.2.5 shows seven clusters of two SMs sharing a texture unit and a texture L1 cache. The texture unit delivers filtered results to the SM given a set of coordinates into a texture map. Because filter regions of support often overlap for successive texture requests, a small streaming L1 texture cache is effective to reduce the number of requests to the memory system. The processor array connects with raster operation processors (ROPs), L2 texture caches, external DRAM memories, and system memory via a GPU-wide interconnection network. The number of processors and number of memories can scale to design balanced GPU systems for different performance and market segments.

- **Instruction Set Architecture (ISA)**

The Instruction Set Architecture described here is a simplified version of the Tesla architecture PTX ISA, a register-based load/store scalar instruction set comprising floating-point, integer, logical, conversion, special functions, flow control, memory access, and texture operations of a single thread. The instruction format is:

```
opcode.type d, a, b, c;
```

where `d` is the destination operand, `a`, `b`, `c` are source operands, and `.type` is either untyped bits, unsigned integer, signed integer or a floating point number. Each type may have 8, 16, 32 or 64 bits. The different supported types are shown in the next table. The basic set of the opcodes is found in figure C.4.3.

Type	.type Specifer
Untyped bits 8, 16, 32, and 64 bits	.b8, .b16, .b32, .b64
Unsigned integer 8, 16, 32, and 64 bits	.u8, .u16, .u32, .u64
Signed integer 8, 16, 32, and 64 bits	.s8, .s16, .s32, .s64
Floating-point 16, 32, and 64 bits	.f16, .f32, .f64

Department of Electrical and Computer Engineering
Rutgers, The State University of New Jersey

Basic PTX GPU Thread Instructions

Group	Instruction	Example	Meaning	Comments
Arithmetic	arithmetic *.type* = .s32, .u32, .f32, .s64, .u64, .f64			
	add.*type*	add.f32 d, a, b	d = a + b;	
	sub.*type*	sub.f32 d, a, b	d = a - b;	
	mul.*type*	mul.f32 d, a, b	d = a * b;	
	mad.*type*	mad.f32 d, a, b, c	d = a * b + c;	multiply-add
	div.*type*	div.f32 d, a, b	d = a / b;	multiple microinstructions
	rem.*type*	rem.u32 d, a, b	d = a % b;	integer remainder
	abs.*type*	abs.f32 d, a	d = |a|;	
	neg.*type*	neg.f32 d, a	d = 0 - a;	
	min.*type*	min.f32 d, a, b	d = (a < b)? a:b;	floating selects non-NaN
	max.*type*	max.f32 d, a, b	d = (a > b)? a:b;	floating selects non-NaN
	setp.*cmp.type*	setp.lt.f32 p, a, b	p = (a < b);	compare and set predicate
	numeric *.cmp* = eq, ne, lt, le, gt, ge; unordered *cmp* = equ, neu, ltu, leu, gtu, geu, num, nan			
	mov.*type*	mov.b32 d, a	d = a;	move
	selp.*type*	selp.f32 d, a, b, p	d = p? a: b;	select with predicate
	cvt.dtype.atype	cvt.f32.s32 d, a	d = convert(a);	convert atype to dtype
Special Function	special *.type* = .f32 (some .f64)			
	rcp.*type*	rcp.f32 d, a	d = 1/a;	reciprocal
	sqrt.*type*	sqrt.f32 d, a	d = sqrt(a);	square root
	rsqrt.*type*	rsqrt.f32 d, a	d = 1/sqrt(a);	reciprocal square root
	sin.*type*	sin.f32 d, a	d = sin(a);	sine
	cos.*type*	cos.f32 d, a	d = cos(a);	cosine
	lg2.*type*	lg2.f32 d, a	d = log(a)/log(2)	binary logarithm
	ex2.*type*	ex2.f32 d, a	d = 2 ** a;	binary exponential
Logical	logic. *type* = .pred, .b32, .b64			
	and.*type*	and.b32 d, a, b	d = a & b;	
	or.*type*	or.b32 d, a, b	d = a | b;	
	xor.*type*	xor.b32 d, a, b	d = a ^ b;	
	not.*type*	not.b32 d, a, b	d = ~a;	one's complement
	cnot.*type*	cnot.b32 d, a, b	d = (a==0)? 1:0;	C logical not
	shl.*type*	shl.b32 d, a, b	d = a << b;	shift left
	shr.*type*	shr.s32 d, a, b	d = a >> b;	shift right
Memory Access	memory *.space* = .global, .shared, .local, .const; *.type* = .b8, .u8, .s8, .b16, .b32, .b64			
	ld.*space.type*	ld.global.b32 d, [a+off]	d = *(a+off);	load from memory *space*
	st.*space.type*	st.shared.b32 [d+off], a	*(d+off) = a;	store to memory *space*
	tex.*nd.dtyp.btype*	tex.2d.v4.f32.f32 d, a, b	d = tex2d(a, b);	texture lookup
	atom.*spc.op.type*	atom.global.add.u32 d,[a], b atom.global.cas.b32 d,[a], b, c	atomic { d = *a; *a = op(*a, b); }	atomic read-modify-write operation
	atom *.op* = and, or, xor, add, min, max, exch, cas; *.spc* = .global; *.type* = .b32			
Control Flow	branch	@p bra target	if (p) goto target;	conditional branch
	call	call (ret), func, (params)	ret = func(params);	call function
	ret	ret	return;	return from function call
	bar.sync	bar.sync d	wait for threads	barrier synchronization
	exit	exit	exit;	terminate thread execution

FIGURE C.4.3 Basic PTX GPU thread instructions.

Source operands are scalar 32-bit or 64-bit values in registers, an immediate value, or a constant; predicate operands are 1-bit Boolean values. Destinations are registers, except for store to memory.

Instructions are predicated by prefixing them with `@p` or `@!p`, where `p` is a predicate register. Memory and texture instructions transfer scalars or vectors of two to four components, up to 128 bits in total. PTX instructions specify the behavior of one thread.

The PTX arithmetic instructions operate on 32-bit and 64-bit floating-point, signed integer, and unsigned integer types. Recent GPUs support 64-bit double precision floating-point; see Section C.6. On current GPUs, PTX 64-bit integer and logical instructions are translated to two or more binary microinstructions that perform 32-bit operations. The GPU special function instructions are limited to 32-bit floating-point. The thread control flow instructions are conditional `branch`, function `call` and `return`, thread `exit`, and `bar.sync` (barrier synchronization). The conditional branch instruction `@p bra target` uses a predicate register `p` (or `!p`) previously set by a compare and set predicate `setp` instruction to determine whether the thread takes the branch or not. Other instructions can also be predicated on a predicate register being "true" or "false."

Load/Store instructions can access the three different memory spaces. Those memory spaces are:

- Local memory for private addressable temporary data.
- Shared memory for low-latency access to data shared by cooperating threads in the same thread blocks.
- Global memory shared by all threads of an application. This part of the memory is implemented in the external DRAM.

The load instructions are `ld.local`, `ld.shared`, `ld.global` for the three memory spaces and are followed by a destination register and a source register plus an offset in order to computer the memory address. For example, a load instruction will be

```
ld.local.b32 r5,[r1+8];
```

The instruction above will load a word from local memory at address given by the base in register `r1` and offset `8`. The word will be loaded in register `r5`. The store instructions are almost the same. The differences are that the `ld` opcode of the instruction is changed to `st` and the destination address is before the source register. For example:

```
st.local.b32 [r1+16], r5;
```

All the fundamental arithmetic and logical operations, such as addition, multiplication, `AND`, `XOR`, are supported for all the types and can be executed between two registers. The following examples give an idea how those instructions are used:

```
1) add.s32 d, a, b; // d = a + b
2) mul.f32 d, a, b; // d = a * b
3) xor.b32 d, a, b; // d = a ^ b
```

As can be understood from the above examples, two slashes (//) indicate the beginning of a comment.

Apart from those instructions, some special instructions and key transcendental functions are supported. Those instructions are reciprocal, square root and its reciprocal, binary exponential and logarithm and the trigonometric functions for sine and cosine. These instructions can only be used with floating point (single or double precision) numbers.

The predicate for conditional branches can be set with the set predicate instruction (`setp`) and be stored in a register that will be used as the predicate. The target of a branch is given as a simple label. Take for example the following sequence of code that branches to label L1.

```
setp.gt.s64 r5, r1, r7; // r5 = (r1 > r7)
@r5 bra L1;
```

Assignment 1

Write a program in the ISA described above that utilizes the following pseudocode. Suppose that the data you need are in the local memory starting from the address that is stored in register **`r1`**. (To refer to a register use the notation **`r`** and the number of the register). There are 64 32-bit general purpose registers numbered from 0 to 63. For a 64 bit data use a pair of two consecutive registers with the even register as the reference. For example, if A is a 64 bit number and it is stored in registers r0 and r1, then, in order to reference this number, you just need to use r0.

```
void main()

   {
      int n=10;
      int a,b,c,d;

     int e[10],f[10];

      for (i=0;i<n;i++){

      e[i] =(a*b)-(c*d);

      f[i] =(c*b)+(a*d);

         a=a+b;

         c=c+d;

      }

   }
```

Assignment 2

Department of Electrical and Computer Engineering
Rutgers, The State University of New Jersey

Write a program in the ISA described above that utilizes the following pseudocode. Suppose that the data you need are in the local memory starting from the address that is stored in register **r1**. For 64 bit data use a pair of registers with the even register as the reference. For example, if A is a 64 bit number and it is stored in registers r0 and r1, then, in order to reference this number, you just need to use r0. Assume that each memory address can store 2 bytes.

```
void main()

  {

      int x[10], y[10],z[10];
      int pos=0,neg=0,zero=0;

      for (i=1;i<=10;i++){
           if (x[i] < 0){

                y[neg]=x[i];
                neg=neg+1;

           }
           else if (x[i] > 0){

                z[pos]=x[i];
                pos=pos+1;
           }

           else
                zero=zero+1;
      }
  }
```

Assignment 3

a) Create a program in MIPS and in simplified PTX ISA that calculates the approximate value of mathematical constant $\boldsymbol{e}$ by the following function:

$$\mathrm{e} \approx \sum_{k=0}^{n} \frac{1}{k!}$$

The MIPS program should ask the user for the value **n**, which is an unsigned integer, then printout the result. The PTX program should assume that input **n** is in register **r1**.

b) Compare the number of instructions for both programs. Supposing that each instruction in both programs require 1 cycle to run record how many cycles each program needs to calculate a polynomial of degree 10. Explain your findings.

Assignment 4

Write a program in MIPS and in simplified PTX ISA that returns the determinant of the matrix of a 5 by 5 matrix using Laplace expansion. For this assignment consider that all values are stored in the shared memory and the following matrix A. Compare the number of instructions for both programs.

$$A = \begin{bmatrix} 2 & 4 & 6 & 8 & 10 \\ 12 & 14 & 16 & 18 & 20 \\ 22 & 24 & 26 & 28 & 30 \\ 32 & 34 & 36 & 38 & 40 \\ 42 & 44 & 46 & 48 & 50 \end{bmatrix}$$

Assignment 5

The hyperbolic functions may be calculated from the following polynomial approximations for $0 \leq x \leq \pi/2$: (**assume the first four terms of the sequence**)

$$\sinh^{-1} x = \sum_{n=0}^{\infty} \frac{(-1)^n (2n)!}{4^n (n!)\hat{}2(2n+1)} x^{2n+1} = x - \frac{x^3}{6} + \frac{3x^5}{40} + \cdots$$

$$\tanh^{-1} x = \sum_{n=0}^{\infty} \frac{1}{2n+1} x^{2n+1} = x + \frac{x^3}{3} + \frac{x^5}{5} + \cdots$$

Department of Electrical and Computer Engineering
Rutgers, The State University of New Jersey

Write a program in MIPS and in simplified PTX ISA that calculates the values of $sinh^{-1}(x)$ and $\tanh^{-1}(x)$ for a single precision floating point number x in the [0, π/2]. Discuss the differences between the two programs you wrote. How much faster is the PTX ISA program?

For your MIPS program use registers **f11** and **f12** to store the values of $sinh^{-1}(x)$ and $\tanh^{-1}(x)$. For your PTX ISA program use registers r63 and r64 to store these two values.

Experiment report

Write a proper report including your codes, results (snapshot of output) and the conclusion of assignments and convert it to **pdf** format. Please also attach the **code** files (*.s,*.asm) to **Sakai** together with the report. Each lab report is **due** before the start of the next lab. Please include your name and Student ID in both report and the code.

References

1. Patterson and Hennessy, "Computer Organization and Design: The Hardware / Software interface", 5th Edition.
2. Daniel J. Ellard, "MIPS Assembly Language Programming: CS50 Discussion and Project Book", September 1994.
3. NVIDIA, "NVIDIA Compute PTX: Parallel Thread Execution, ISA Version 1.4", 2009-03-31, http://www.nvidia.com/content/CUDA-ptx_isa_1.4.pdf
4. Memory Management Unit, http://en.wikipedia.org/wiki/Memory_management_unit
5. Matrix Determinant, http://en.wikipedia.org/wiki/Determinant
6. Euler's Formula, http://en.wikipedia.org/wiki/Euler's_formula
7. Milton Abramowitz & Irene A. Stegun, " Handbook of Mathematical Functions with Formulas, Graphs and Mathematical Tables", Dover Publications Inc., New York, 1965
