Summer School

e-Science with Many-core CPU/GPU Processors

Lecture 0
Beginner's tutorial on many-core processors, multi-core processors, and C programming

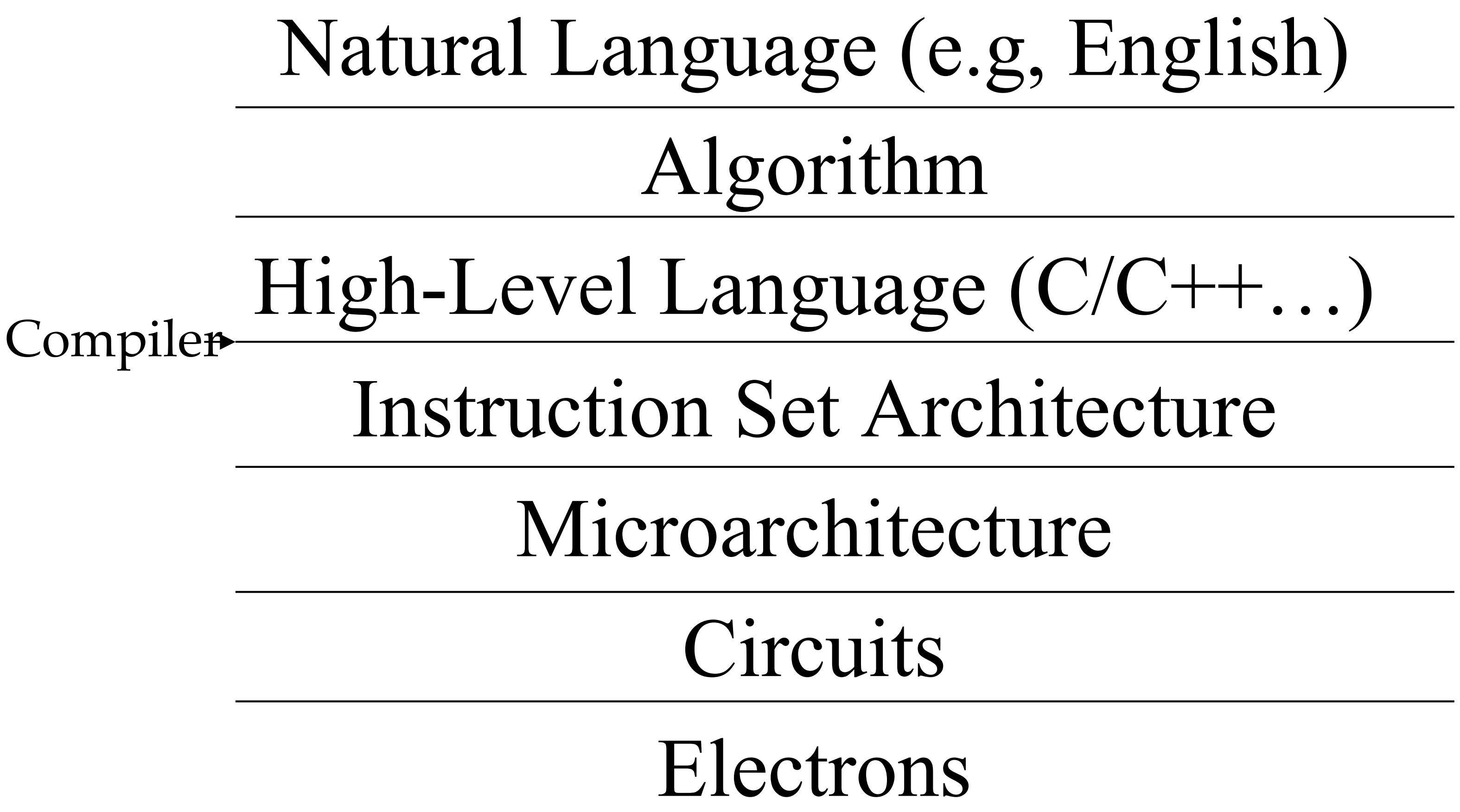
Agenda

- Instruction cycle
- The history of parallelism
- Some C concepts
- Piecing a computer together
- Memory hierarchy

Agenda

- Instruction cycle
- The history of parallelism
- Some C concepts
- Piecing a computer together
- Memory hierarchy

From Natural Language to Electrons



Natural Language (e.g, English)

Algorithm

High-Level Language (C/C++…)

Compiler

Instruction Set Architecture

Microarchitecture

Circuits

Electrons

©Yale Patt and Sanjay Patel, *From bits and bytes to gates and beyond*

The ISA

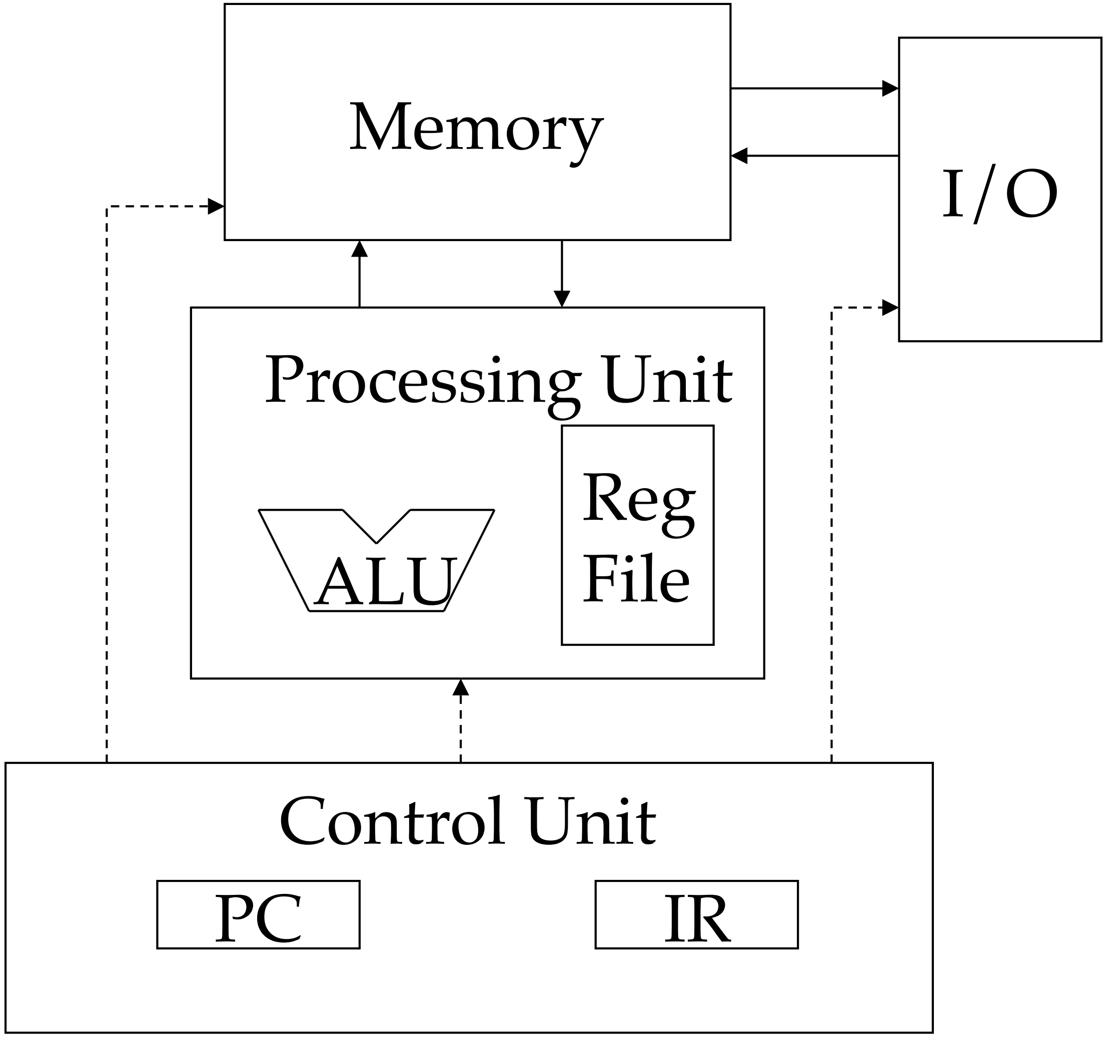
- An Instruction Set Architecture (ISA) is a contract between the hardware and the software.

- As the name suggests, it is a set of instructions that the architecture (hardware) can execute.

A program at the ISA level

- A Program is a set of single instructions stored in memory that can be read, interpreted, and executed by the hardware.
- Program instructions operate on data stored in memory or provided by Input/Output (I/O) device.

The Von-Neumann Model



Going back to the program

- Every instruction needs to be fetched from memory, decoded, then executed.
- Instructions come in three flavors: Operate, Data transfer, and Program Control Flow.
- An example instruction cycle is the following:

Fetch | Decode | Execute | Memory

Operate Instructions

- Example of an operate instruction:

  ADD R1, R2, R3

- Instruction cycle for an operate instruction:

  Fetch | Decode | Execute | Memory

Data Transfer Instructions

- Examples of data transfer instruction:

  LDR R1, R2, #2

  STR R1, R2, #2

- Instruction cycle for an operate instruction:

  Fetch | Decode | Execute | Memory

Control Flow Operations

- Example of control flow instruction:

  BRp #-4

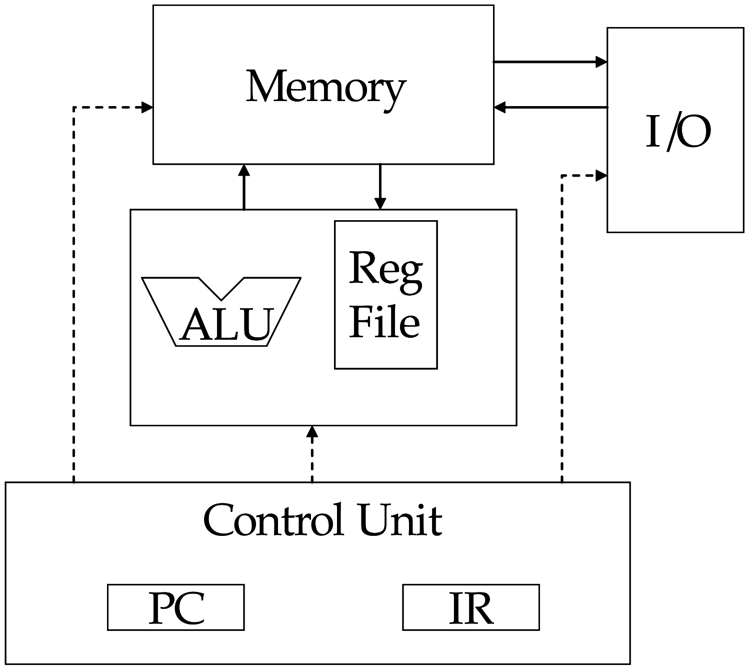
  if the condition is zero, skip the next four instructions

- Instruction cycle for an operate instruction:

  Fetch | Decode | Execute | Memory

Registers vs Memory

- Registers are “free” to use, however, there are very few of them
- Memory is expensive, but large



Agenda

- Instruction cycle
- The history of parallelism
- Some C concepts
- Piecing a computer together
- Memory hierarchy

History of parallelism

- 1st gen - Instructions are executed sequentially in program order, one at a time.

- Example:

Cycle	1	2	3	4	5	6
Instruction1	Fetch	Decode	Execute	Memory		
Instruction2					Fetch	Decode

History - Cont’d

- 2nd gen - Instructions are executed sequentially, in program order, in an assembly line fashion. (pipeline)
- Example:

Cycle	1	2	3	4	5	6
Instruction1	Fetch	Decode	Execute	Memory		
Instruction2		Fetch	Decode	Execute	Memory	
Instruction3			Fetch	Decode	Execute	Memory

History – Instruction Level Parallelism

- 3rd gen - Instructions are executed in parallel
- Example code 1:

$$\left.\begin{array}{l} c = b + a; \\ d = c + e; \end{array}\right\} \text{Non-parallelizable}$$

- Example code 2:

$$\left.\begin{array}{l} a = b + c; \\ d = e + f; \end{array}\right\} \text{Parallelizable}$$

ILP – Cont'd

- Two forms of ILP:
  - Superscalar: fetch, decode, and execute multiple instructions at a time. Execution may be out of order

Cycle	1	2	3	4	5
Instruction1	Fetch	Decode	Execute	Memory	
Instruction2	Fetch	Decode	Execute	Memory	
Instruction3		Fetch	Decode	Execute	Memory
Instruction4		Fetch	Decode	Execute	Memory

  - VLIW: At compile time, pack multiple, independent instructions in one large instruction and process the large instructions as the atomic units.

History – Cont'd

- 4th gen - SMT: Multiple threads are executed simultaneously on the same processor/core. (will revisit)

- 5th gen - Multi-Core: Multiple threads are executed simultaneously on multiple processors (usually includes SMT as well)

History – Cont'd

- GPUs: Data parallelism - multiple threads executing the same program on multiple processors with different data (Single Program Multiple Data)

- Definition
  - SIMD: Single Instruction Multiple Data

Agenda

- Instruction cycle
- The history of parallelism
- Some C concepts
- Piecing a computer together
- Memory hierarchy

Floating Point

- Floating point is a primitive C/C++ data type
- Used to represent rational numbers
- Floating point is 32 bits, 1 bit for the sign (+/-), 8 bits for the exponent, and 23 bits for the fraction/mantissa
- Numbers are stored in scientific notation, using base two
- Example: $\overset{\text{sign}}{\boxed{-}}1.\overbrace{0101110}^{\text{mantissa}} * 2^{\overset{\text{exponent}}{\boxed{5}}}$

Floating Point – Cont’d

- Floating point can represent very small number and very large number but with limited precision
- The limited precision can lead to inaccurate results

  For example: $10^{24} + 1 = 10^{24}$
- The solution is to increase precision, by using the “double” datatype which is 64 bits

Floating Point – Cont'd

- It is hard to compute with floating point numbers, since they are normalized.

- Floating point calculations require dedicated hardware.

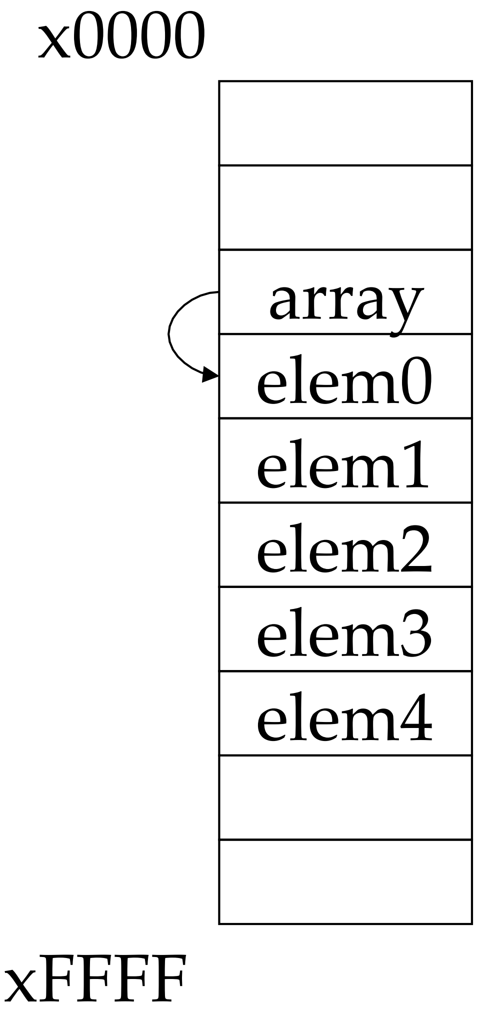
Data Structures

- In software, data is stored in scalar variables or arrays
- Examples:

```
int variable = 2;
int array[5];
```

Arrays

- int array[5];
- The variable “array” is a pointer to a block of memory which holds 5 integers
- In other words, “array” is of type “int *”
- “array” is in itself a variable whose value is the address of the first memory location of the block of 5 integers



Pointers

- To access the value pointed to by a pointer, you have to “dereference” the pointer.
- Syntax:

```
int variable2 = *array;
```

- What if I want to access the second integer in the block?

```
int variable3 = *(array+1); //pointer
```

arithmetic

Pointer and Array Duality

- Another way to dereference a array pointer is to use array syntax
- Syntax:

```
    int variable4 = array[0];
```

- And to access the second element:

```
    int variable5 = array[1];
```

2D arrays

- What about:

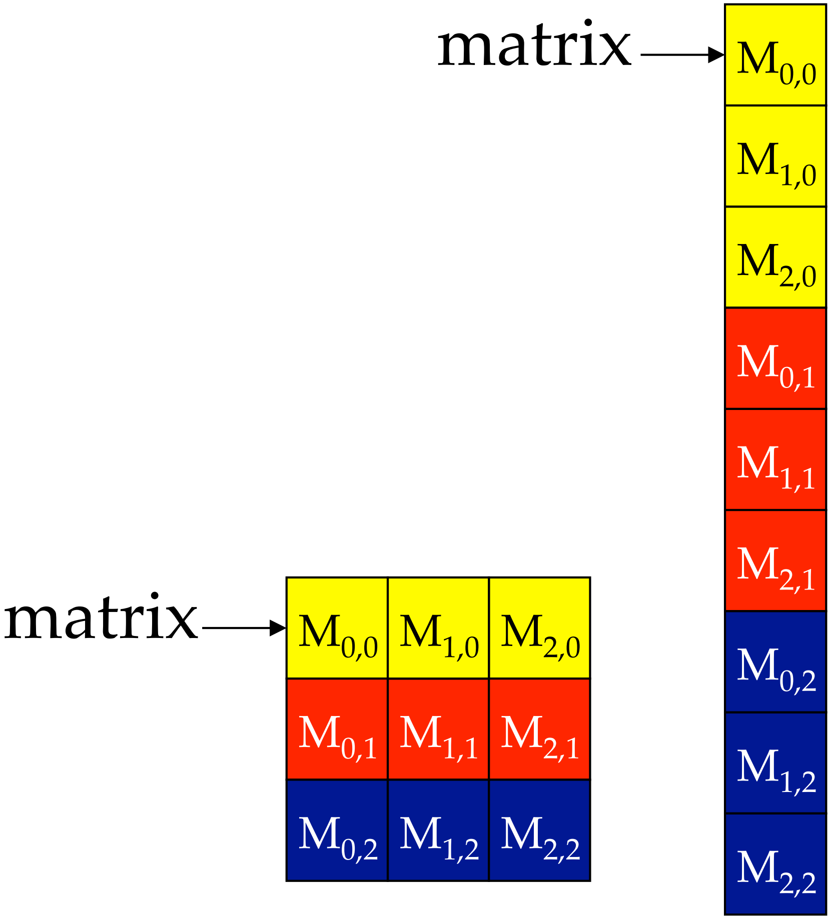
  int matrix[3][3]; // What’s that?

- Answer: it’s a 2-dimensional array

- Memory is 1-dimensional though, so how do we store a 2D array in it?
- Answer: Major ordering

Row Major Ordering

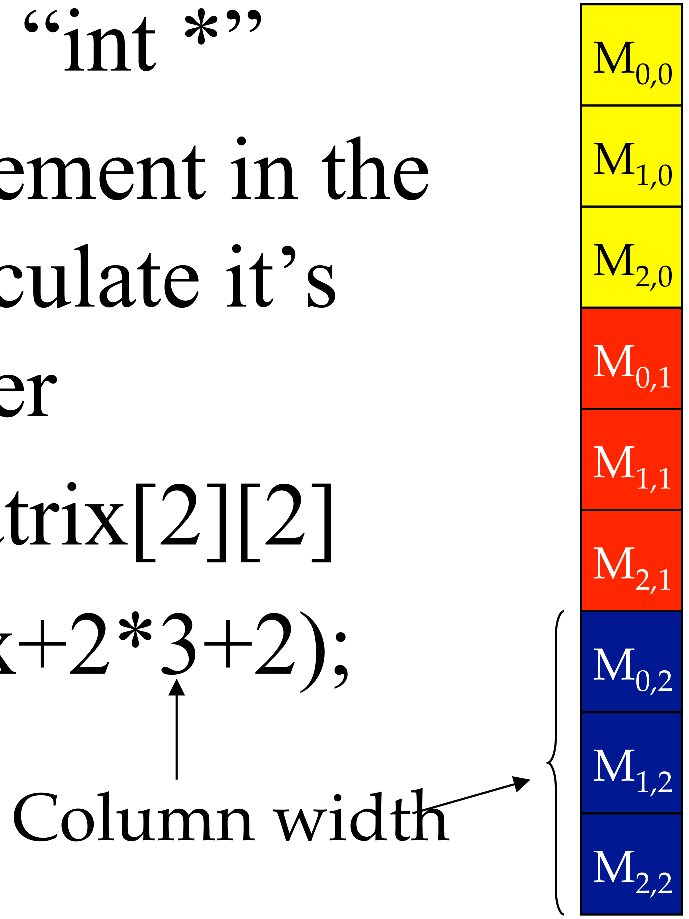
- Row major: starting from the “top-left” corner of the array, store elements in consecutive columns first, then in consecutive rows
- Note: Fortran is column major



Using pointer syntax on 2D arrays

- “matrix” is also of type “int *”
- In order to access an element in the matrix, you have to calculate it’s index is row-major order
- Example: accessing matrix[2][2]

int variable6 = *(matrix+2*3+2);

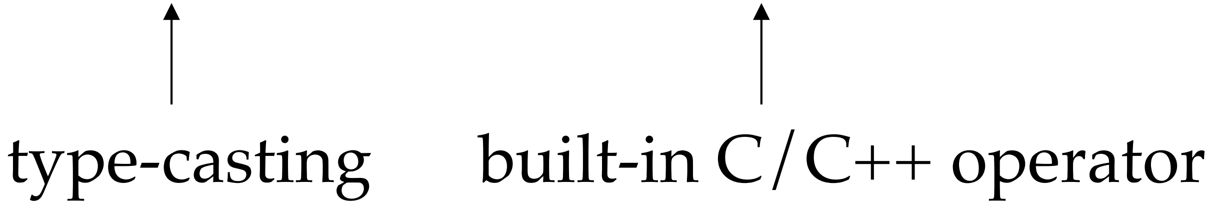


© Wen-mei W. Hwu and Nady Obeid 2009
VSCSE, University of Illinois, Urbana-Champaign

Dynamic Allocation

- What if we don’t know the size of the array before runtime?
- Allocate space at runtime
- Syntax:

```
int *array2 = (int*) malloc (n*sizeof(int));
```



type-casting

built-in C/C++ operator

Dynamic Allocation - Cont'd

- Dynamically allocated arrays have to be explicitly allocated and de-allocated
- Allocate function:

  void* malloc (int size);
- De-Allocate function:

  void free (void * ptr);
- Dynamic arrays can be accessed using array syntax or pointer syntax the same way static arrays can (example goes here)

Structs

- Data in C has to have a type
- Datatypes that are primitive to the language are: int, float, double, char, etc.
- Structs are user-defined datatypes made of primitive datatypes and/or other structs
- Example:

```
struct complex{
        float real;
        float imag;
}
```

Structs – Cont'd

- Declaring a complex variable:

  complex variable7;
- Accessing an element of variable7:

  variable7.real = 2.5;

- A struct variable is stored in consecutive memory locations, and its size is the sum of the size of its components
- You can define an array of type complex the same way you would any primitive data type

Atomic Operations

```
thread1:Reg ← Mem[x]          thread2:Reg ← Mem[x]
        Reg ← Reg + 1                 Reg ← Reg + 1
        Mem[x] ← Reg                  Mem[x] ← Reg
```

If x was initially 0, what would the value of x be after threads 1 and 2 have completed?

The answer may vary due to data races. To avoid data races, you should use atomic operations

Agenda

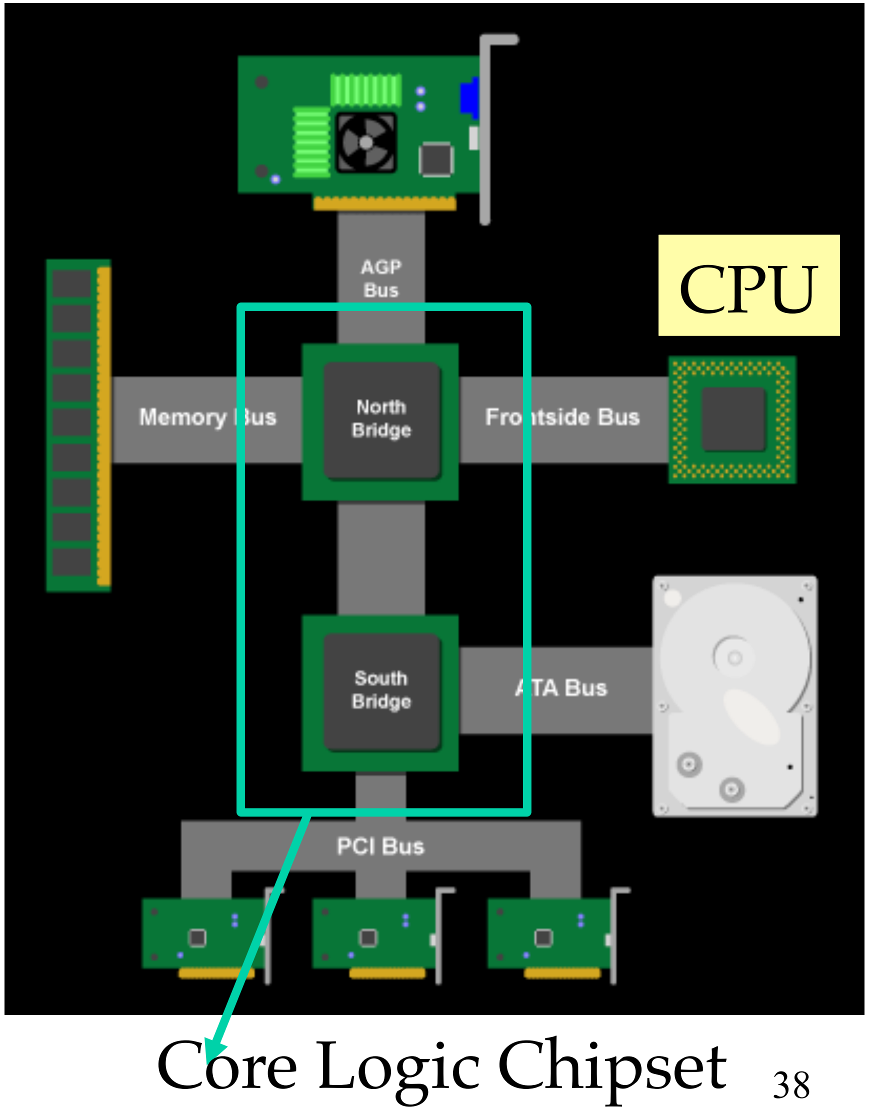
- Instruction cycle
- The history of parallelism
- Some C concepts
- Piecing a computer together
- Memory hierarchy

Piecing a computer together

- A GPU is an accelerator. It is there to aid the main processor(s) compute faster
- In this section, we will discover how the GPU, CPU, memory, etc. all communicate inside a computer.

Classic PC architecture

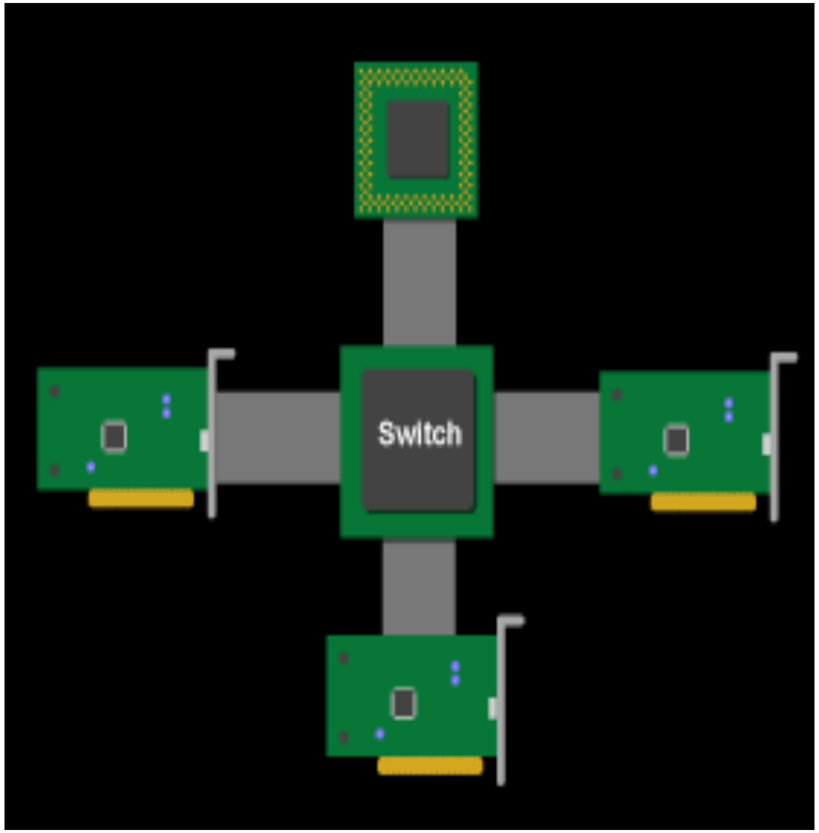
- Northbridge connects 3 components that must communicate at high speed
  - CPU, DRAM, video
  - Video also needs to have 1st-class access to DRAM
  - Previous NVIDIA cards are connected to AGP, up to 2 GB/s transfers
- Southbridge serves as a concentrator for slower I/O devices



Core Logic Chipset

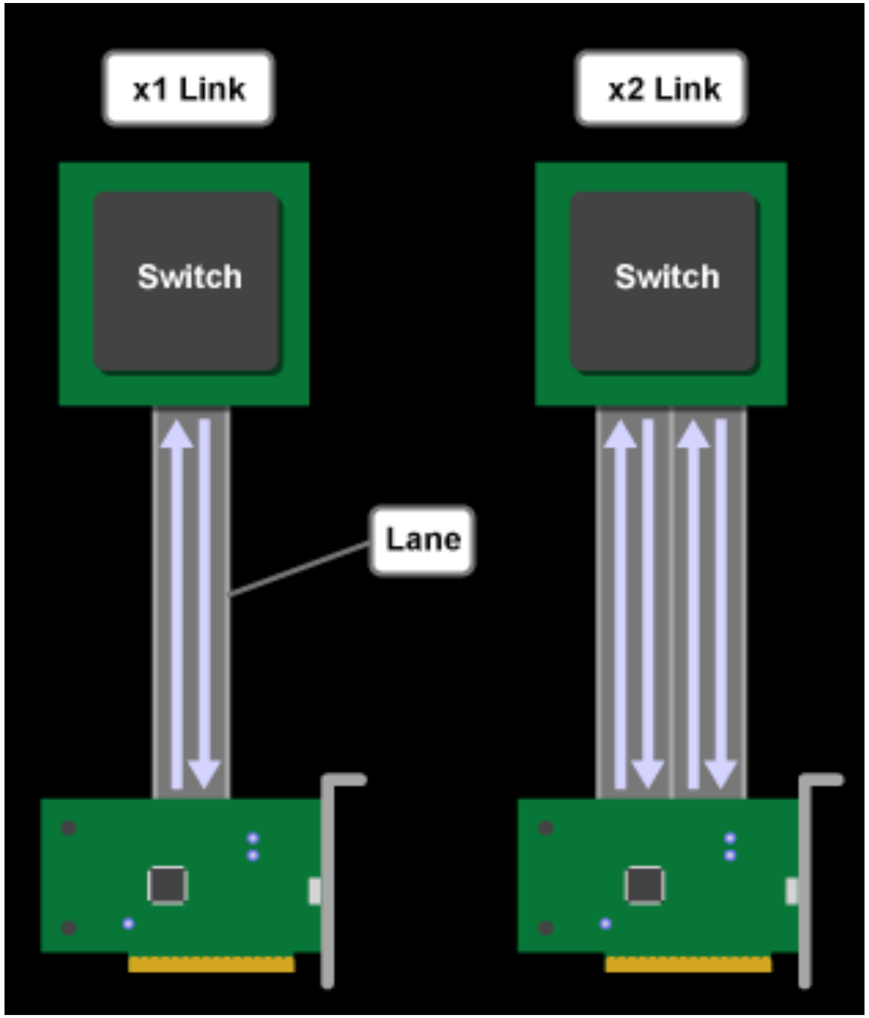
PCI Express (PCIe)

- Switched, point-to-point connection
  - Each card has a dedicated “link” to the central switch, no bus arbitration.
  - Packet switches messages form virtual channel
  - Prioritized packets for QoS
    - E.g., real-time video streaming



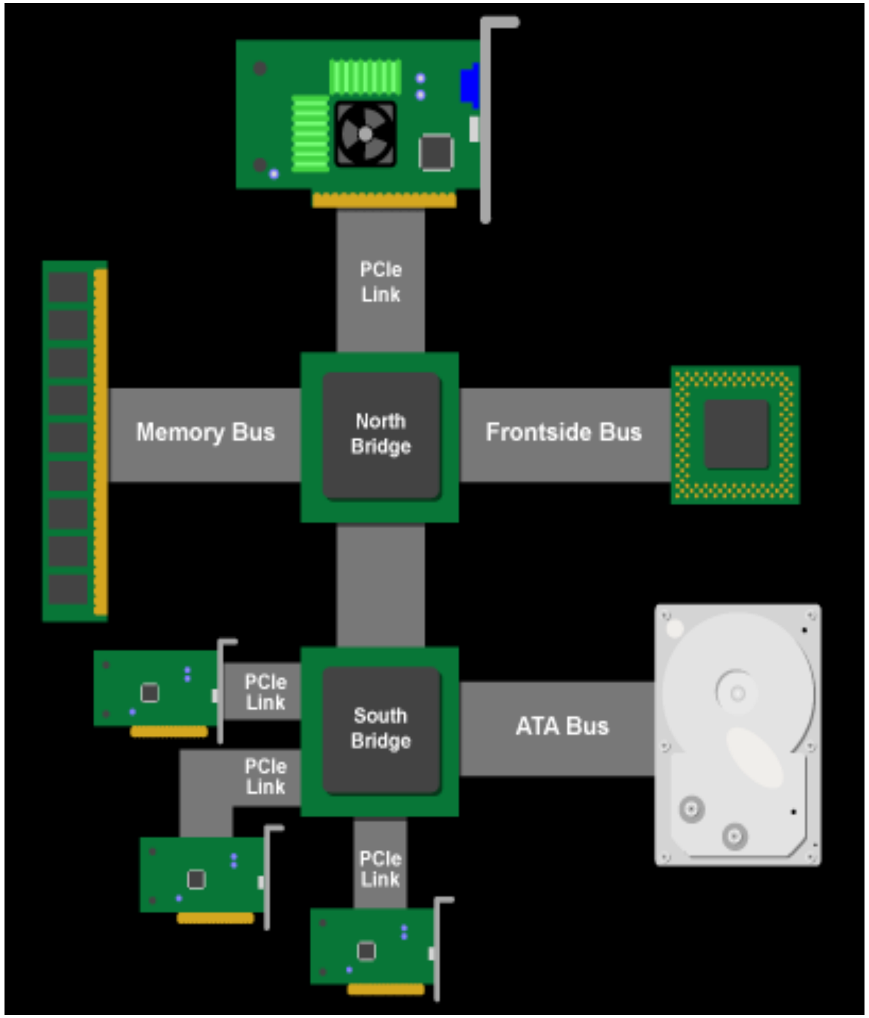
PCIe Links and Lanes

- Each link consists of one more lanes
  - Each lane is 1-bit wide (4 wires, each 2-wire pair can transmit 2.5Gb/s in one direction)
    - Upstream and downstream now simultaneous and symmetric
  - Each Link can combine 1, 2, 4, 8, 12, 16 lanes- x1, x2, etc.
  - Each byte data is **8b/10b** encoded into 10 bits with equal number of 1's and 0's; net data rate 2 Gb/s per lane each way.
  - Thus, the net data rates are 250 MB/s (x1) 500 MB/s (x2), 1GB/s (x4), 2 GB/s (x8), 4 GB/s (x16), each way



PCIe PC Architecture

- PCIe forms the interconnect backbone
  - Northbridge/Southbridge are both PCIe switches
  - Some Southbridge designs have built-in PCI-PCIe bridge to allow old PCI cards
  - Some PCIe cards are PCI cards with a PCI-PCIe bridge
- Source: Jon Stokes, PCI Express: An Overview
  - http://arstechnica.com/articles/paedia/hardware/pcie.ars



Agenda

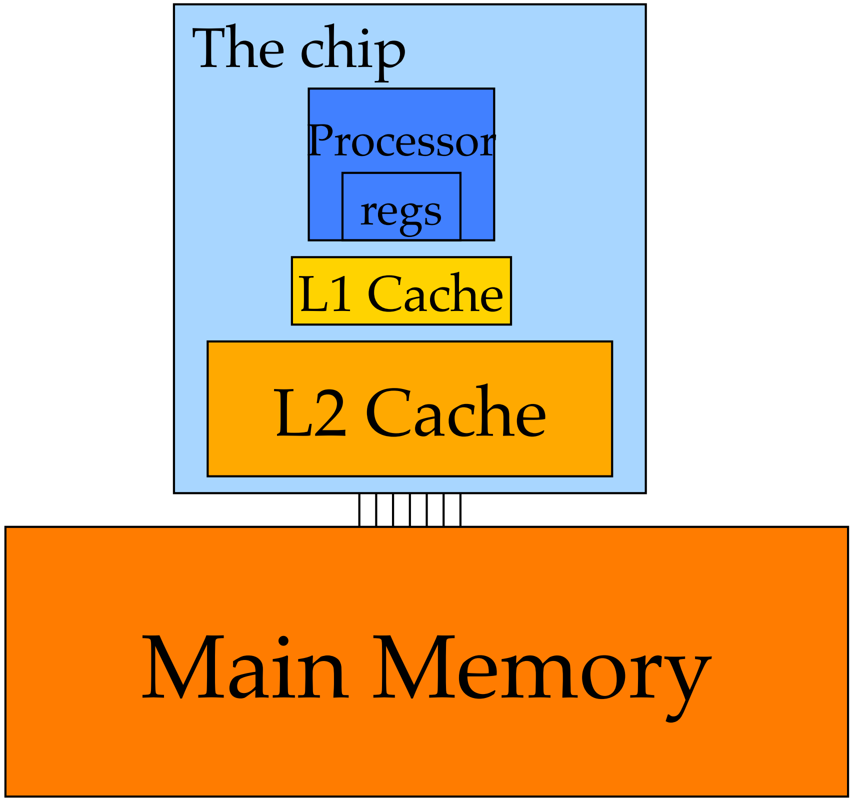
- Instruction cycle
- The history of parallelism
- Some C concepts
- Piecing a computer together
- Memory hierarchy

Memory Hierarchies

- If every time we needed a piece of data, we had to go to main memory to get it, computers would take a lot longer to do anything
- On today's processors, main memory accesses take hundreds of cycles
- One solution: Caches

Cache - Cont’d

- In order to keep cache fast, it needs to be small, so we cannot fit the entire data set in it



Cache - Cont’d

- Cache is unit of volatile memory storage
- A cache is an “array” of cache lines
- Cache line can usually hold data from several consecutive memory address
- When data is requested from memory, an entire cache line is loaded into the cache, in an attempt to reduce main memory requests

Caches - Cont'd

Some definitions:

- Spacial locality: is when the data elements stored in consecutive memory locations are access consecutively
- Temporal locality: is when the same data element is access multiple times in short period of time

- Both spacial locality and temporal locality improve the performance of caches

Scratchpad vs. Cache

- Scratchpad is another type of temporary storage used to relieve main memory contention.
- In terms of distance from the processor, scratchpad is similar to L1 cache.
- Unlike cache, scratchpad does not necessarily hold a copy of data that is in main memory
- It requires explicit data transfer instructions, whereas cache doesn't

An alternative to caches

- We use caches to reduce the latency of going to memory for data. But what if we could hide that latency instead?
- Multithreading: Run multiple threads simultaneously. Once a thread stalls for a memory request, start executing from the next one.
- The challenge of multithreading is having enough work to do while a memory request is being serviced.