

# GPU Architectures and Programming

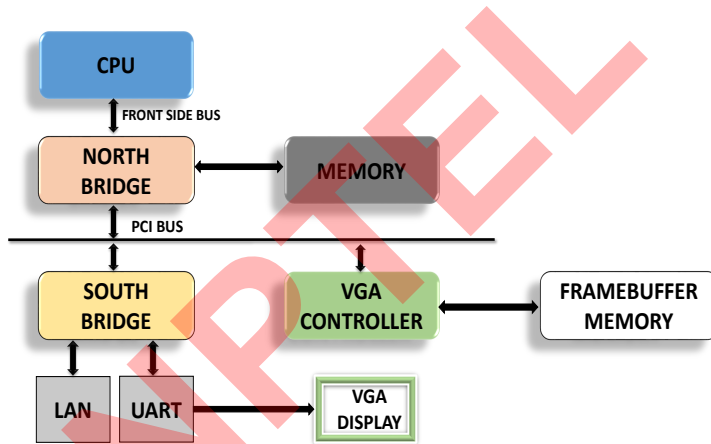
Soumyajit Dey, Assistant Professor,  
CSE, IIT Kharagpur

December 5, 2019



- ▶ Fifteen years ago, Graphics on a PC were performed by a video graphics array (VGA) controller.
- ▶ VGAs evolved to more complex hardware : accelerating graphics functions
- ▶ Early GPUs and their associated drivers implemented the OpenGL and DirectX models (APIs) of graphics processing.
- ▶ With time, HW functionality evolved as programmable SW





**Figure:** Historical PC. - Hennessy and Patterson "Computer Organization and Design" -  
(Figure reproduced)



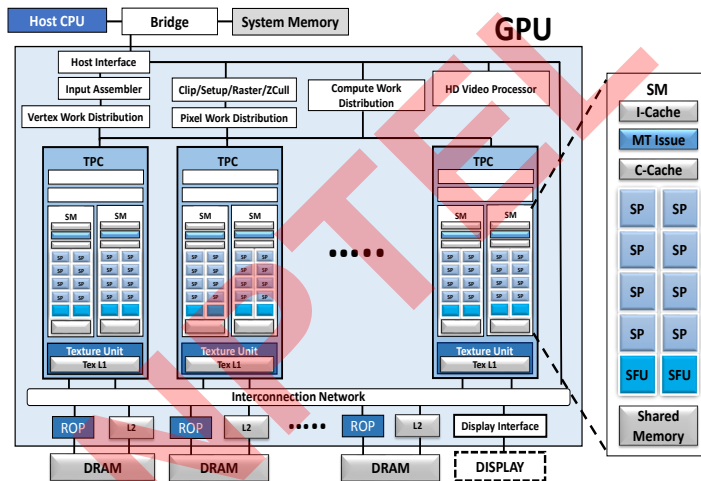


Figure: GPU Architecture - Hennessy Patterson (Figure reproduced)



## Course Organization

Topic	Week	Hours
Review of basic COA w.r.t. performance	1	2
Intro to GPU architectures	2	3
Intro to CUDA programming	3	2
Multi-dimensional data and synchronization	4	2
Warp Scheduling and Divergence	5	2
Memory Access Coalescing	6	2
Optimizing Reduction Kernels	7	3
Kernel Fusion, Thread and Block Coarsening	8	3
OpenCL - runtime system	9	3
OpenCL - heterogeneous computing	10	2
Efficient Neural Network Training/Inferencing	11-12	6



## Section 1

# The classic 5-stage RISC pipeline



## Basic RISC architecture

- ▶ The operation of a processor is characterized by a fetch⇒ decode⇒execute cycle.
- ▶ RISC n CISC ⇒ two different philosophies of computing hardware design
- ▶ RISC/CISC - Reduced/Complex Instruction Set Computing
- ▶ CISC approach - complete a task with as few instructions (instrs) as possible
- ▶ A CISC instruction : MUL *addr<sub>1</sub>* *addr<sub>2</sub>* *addr<sub>3</sub>*
- ▶ Equivalent RISC : LOAD R2 *addr<sub>2</sub>*; LOAD R3 *addr<sub>3</sub>*; MUL R1 R2 R3; STORE *addr<sub>1</sub>* R1



# CISC vs RISC

## CISC features

- ▶ Older ISA
- ▶ Multi-cycle instructions, HW intensive design
- ▶ Efficient RAM usage
- ▶ Instructions - complex and variable length, lots of them
- ▶ Micro-code support
- ▶ Compound addressing modes

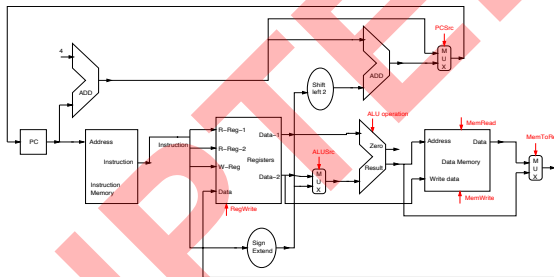
## RISC features

- ▶ Ideas emerged in 1980s
- ▶ Single-cycle instructions, SW intensive design
- ▶ Heavy RAM usage, Large Register file
- ▶ Small no. of simple fixed length instructions
- ▶ Less no. of addressing modes





## Elementary CPU Datapath



- ▶ The datapath 'fetches' instruction, 'decodes' and 'executes' it
- ▶ Control logic generates suitable activation signals
- ▶ Executes different instructions with variable delays



## Single cycle implementation of datapath

- ▶ The choice of clock rate is limited by the instruction with maximum delay
- ▶ Options : choose the clock period more than latency of 'slowest' instruction or,
- ▶ choose variable periods for diff instructions – not practical !
- ▶ Alternate possibility - break the instruction execution cycle into a series of basic steps
- ▶ Basic steps have less delay, choose a fast clock and use it to execute one basic step at a time



## Multi-cycle instructions

A basic stage represents one of the following states in the execution of an instruction

- ▶ Fetch (IF):  $IR \leftarrow \text{Memory}[PC]$ ;  $PC = PC + 4$
- ▶ Decode (ID): Understand instruction semantics
- ▶ Execute (EX): based on instruction type
  - ▶ Arithmetic/logical operation, Mem address / Branch condition computation
- ▶ Memory (MEM): For load/store Instr, read/write data from/to memory
- ▶ Writeback (WB): Update register file



# Pipelining

- ▶ Operate IF→ID→EX→MEM→WB in parallel for a sequence of instructions
- ▶ Every basic stage is always processing some instruction
- ▶ In every clock cycle, one instruction completes - ideal scenario
- ▶ Practical issues - pipeline hazards



## Structural hazard

- ▶ Consider a sequence of 4 lw (load-word) instructions
- ▶ When the first instruction fetches data from memory, the fourth instruction itself is to be fetched from memory
- ▶ This is *structural hazard* as the pipeline needs to stall due to lack of resources, if the hardware cannot support multiple reads in parallel



## Data Hazard : MIPS example

- ▶ `sub $2, $1, $3; and $12, $2, $5` Read after Write (RAW)
- ▶ if 'sub' is in IF stage in  $i + 1$ -th clock cycle, \$2 is updated in  $(i + 5)$ -th cycle
- ▶ 'and' is in EX stage in  $i + 4$ -th cycle, updated value of \$2 is not yet ready
- ▶ Solution : 'sub' computes the value for \$2 in  $(i + 3)$ -th stage,
- ▶ this may be *forwarded* directly to execution of 'and'
- ▶ need suitable logic to detect hazard and forwarding requirement



## Control hazards

- ▶ Branch decisions : the branch condition needs evaluation (beq \$1, \$2, offset)
- ▶ The branch decision is inferred only in MEM stage
- ▶ Optimization : assume branch not taken, operate pipeline *normally*,
- ▶ Execute branch when decision is evaluated as true (taken) and flush intermediate instructions from pipeline
- ▶ Sophisticated schemes : use branch prediction HW (predict a branch decision based on branch history table content)



## Section 2

# The Memory Hierarchy





## Multi-level Arrangement

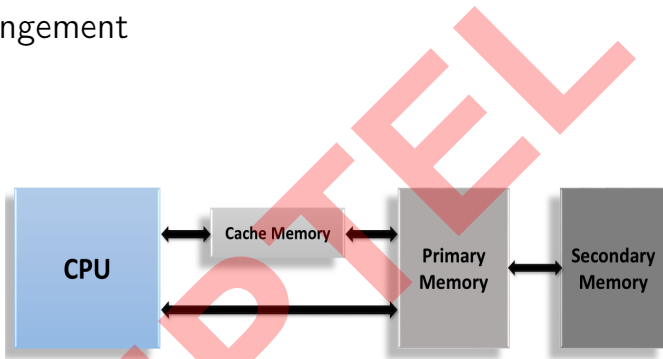


Figure: Near to CPU is faster

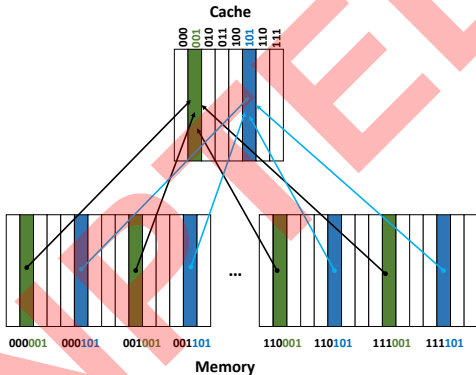


## Principle of locality

- ▶ Temporal locality : If an item is referenced, it will tend to be referenced again soon
- ▶ Spatial locality : If an item is referenced, items at nearby addresses will be referenced soon
- ▶ Hence, computer memory is hierarchically organized
- ▶ Register file provides fastest access,
- ▶ Cache memory uses (fast) SRAM (static random access memory)
- ▶ Main memory uses (slow) DRAM (dynamic random access memory) : is less costly per bit than SRAM



## Cache Mapping



- ▶ Direct mapped : Cache block address = (memory block address) modulo (Number of cache blocks in the cache)
- ▶ Block = minimum unit of information that can be either present or not present



## Cache Blocks

- ▶ With larger blocks we have lower miss rates due to spatial locality, large blocks lead to large miss penalty
- ▶ Nothing is free : with very big block sizes, we have too small no of blocks in cache, eventually the miss rate goes up
- ▶ Handling Cache Miss:
  - ▶ Send the PC value (current PC – 4) to the memory
  - ▶ Read access from main memory, write updated cache entry



## Cache write policy

- ▶ Handling consistency : always write the data into both the memory and the cache (**write-through**)
- ▶ Conservative policy, slows things down
- ▶ Use *write buffer* to perform writes only when buffer is full. Buffer size can be decided by memory speed
- ▶ Alternative policy **write-back** : Writes are updated only in cache. Main memory is update only during cache block replacement
- ▶ Write-back offers better performance in case of frequent writes, is more complex to implement



## Memory System

- ▶ Memory chips are designed to read/write more than one word in parallel (hiding latency)
- ▶ Use a wide bus - allow parallel access to all words in a block
- ▶ OR - keep bus of standard width (= memory word length = register size) and connect bus with multiple memory units in parallel (memory banks)
- ▶ WHY ? bus transmission is fast, memory read/write is slow



## Cache Mapping: alternate schemes

- ▶ Fully associative: a block can be placed in any location in the cache. (Large HW requirement for fast parallel search)
- ▶ Practical only for cache with small number of blocks
- ▶ Optimizing in the middle : set associative cache
- ▶ An  $n$ -way set-associative cache consists of a number of sets, each of which consists of  $n$  blocks.
- ▶ Set number = (Memory Block number) modulo (Number of sets in the cache)
- ▶ Inside a set, all the tags of all the elements must be searched
- ▶ Increasing associativity decreases miss rate up to a point, but increases hit time



## Cache replacement policy

- ▶ In direct mapped cache, a new block can go to exactly one location
- ▶ In fully associative cache, a new block can potentially replace any existing block - how to resolve ?
- ▶ In set associative cache, a new block can potentially replace any existing block inside a matching set - how to resolve ?
- ▶ Least Recently Used (LRU) policy - The block replaced is the one that has been unused for the longest time.





## Section 3

# Instruction Level Parallelism (ILP)



## Actual Pipeline CPI

Pipeline Cycles per instruction (CPI) = Ideal pipeline CPI + Structural stalls + Data hazard stalls + Control stalls

- ▶ Handling hazards require both architectural and compiler techniques
- ▶ Data hazard types while executing instruction  $i$  followed by  $j$  in a pipeline
  - ▶ RAW —  $j$  tries to read a source before  $i$  writes it, so  $j$  incorrectly gets the old value
  - ▶ WAW —  $j$  tries to write an operand before it is written by  $i$ . Will not happen in simple RISC, but in pipelines that write in more than one basic stage or allow an instruction to proceed even when a previous instruction is stalled
  - ▶ WAR -  $j$  tries to write a destination before it is read by  $i$ , can happen in case instructions are reordered
  - ▶ RAR - not a hazard



## Compiler Techniques for ILP

To keep a pipeline full, a compiler can find sequences of unrelated instructions that can be overlapped

```
for (i=100; i>=0; i=i-1)
x[i] = x[i] + s;
```

### Unoptimized MIPS

Loop:

```
L.D F0,0(R1) ;F0=array element
```

```
ADD.D F4,F0,F2 ;add scalar in F2
```

```
S.D F4,0(R1) ;store result
```

```
DADDUI R1,R1,#-8 ;decrement pointer //loop overhead
```

```
;8 bytes (per DW)
```

```
BNE R1,R2,Loop ;branch R1!=R2 //branch decision
```



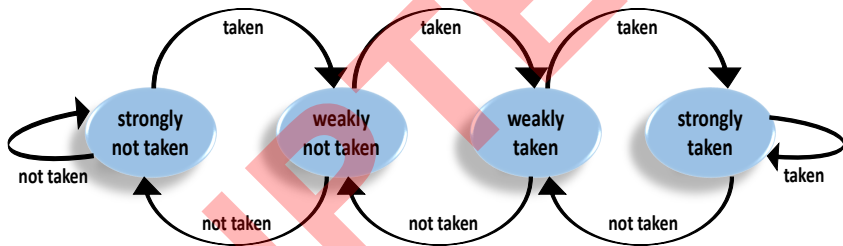
Unrolling: eliminated three branches and decrements of R1 (Hen Pat etl. al.)

```
Loop: L.D F0,0(R1)
      ADD.D F4,F0,F2
      S.D F4,0(R1)
      L.D F6,-8(R1)
      ADD.D F8,F6,F2
      S.D F8,-8(R1) //Code size increase - more instr cache miss
      L.D F10,-16(R1) //more no. of live values - increased register pressure
      ADD.D F12,F10,F2
      S.D F12,-16(R1)
      L.D F14,-24(R1)
      ADD.D F16,F14,F2
      S.D F16,-24(R1)
      DADDUI R1,R1,#-32
      BNE R1,R2,Loop
```



## Branch Prediction assisted ILP

General single level predictor with 2-bit saturating counter



- ▶ conditional jump has to deviate twice from past before the prediction changes.
- ▶ Consider a sequence of altering decisions in a loop and calculate performance improvement over 1-bit saturating counter !!!!



## Hierarchical Prediction

How about generalizing the idea of prediction with larger branch histories.

- ▶ store  $m$  length history of a branch -  $2^m$  possibilities
- ▶ for each possibility use an  $n$ -bit predictor :  $(m, n)$  prediction scheme
- ▶ a two-level predictor with  $m$ -bit history can predict any repetitive sequence with any period if all  $m$ -bit sub-sequences are different.



## Dynamic Scheduling for ILP

- ▶ Simple pipelines execute instructions in-order
  - DIV.D F0,F2,F4
  - ADD.D F10,F0,F8
  - SUB.D F12,F8,F14
- ▶ SUB.D suffers as ADD.D stalls due to dependence
- ▶ different ordering will avoid stall in this case
- ▶ Out of order execution brings in the possibility of WAR and WAW hazards

Robert Tomasulo: developed algorithm to minimize WAW and WAR hazards while allowing out of order execution (tracks when operands for instructions are available to minimize RAW hazards and uses *register renaming* to minimize WAW and WAR).



## Register Renaming

```
DIV F0,F2,F4
ADD F6,F0,F8 //(RAW for DIV : F0)
S    F6,0(R1) //(RAW for ADD : F6)
SUB F8,F10,F14 //(WAR for ADD : F8)
MUL F6,F10,F8 // (WAR for S, WAW for ADD)
//(RAW for SUB : F8)
```

- ▶ RAW is due to data dependency, stalls in-order pipeline
- ▶ WAR/WAW constrains out-of-order execution

⇒

```
DIV F0,F2,F4
ADD S,F0,F8
S    S,0(R1)
SUB T,F10,F14
MUL F6,F10,T
```

- ▶ S removes WAR of MUL,
- ▶ S removes WAW of MUL,
- ▶ T removes WAR of SUB,





## ILP Using Multiple Issue and Static Scheduling

Multiple-issue processors - allow multiple instructions to be issued in a clock cycle

- ▶ VLIW (very long instruction word) - Parallel instructions statically scheduled by compiler; issue a fixed number of instructions formatted as one large instruction
- ▶ Statically scheduled superscalar - issue a varying rather than a fixed number of instructions (compiler decided) per clock, in-order execution
- ▶ Dynamically scheduled superscalar - issue a varying rather than a fixed number of instructions (hardware decided) per clock, out-of-order execution

For large issue width VLIW (with multiple independent FUs) is preferred w.r.t. statically scheduled superscalar

