CE/CZ 3001:
Advanced Computer Architecture

Module 7: Data and Thread Level Parallelism

Asst Prof Liu Weichen
School of Computer Science and Engineering
Nanyang Technological University, Singapore

Three levels of parallelism

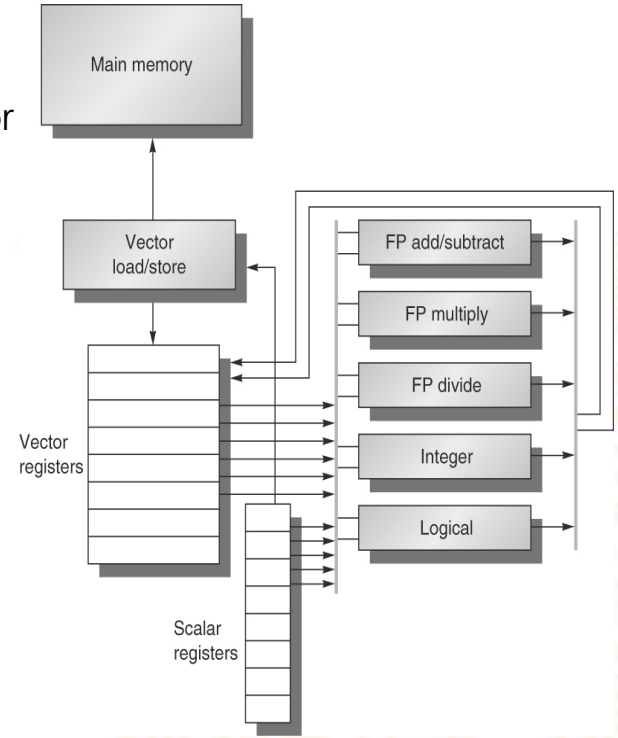
- Instruction-Level Parallelism:
  - Multiple independent instructions are identified and grouped to be executed concurrently in different functional units in a single processor. Can reduce CPI to values less than 1. Examples: Superscalar and VLIW processors.
- Data-Level Parallelism:
  - The same operation is performed on multiple data values concurrently in multiple processing units. Can reduce the Instruction Count to enhance performance. Examples: Vector processors and array processors.
- Thread-Level Parallelism:
  - More than one independent threads/tasks are executed simultaneously. Can reduce the total execution time of multiple tasks. Examples: multi-core and multi-processor systems.

Summary: data-level parallelism

- Single instruction multiple data (SIMD)
  - Vector processor
  - Array processor
  - Multimedia SIMD instruction set extensions

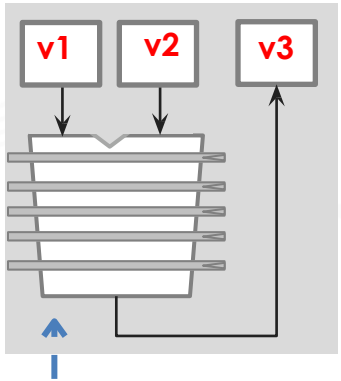
An example architecture: VMIPS (Part 1/2)

- Vector registers
  - Each register holds a 64-element, 64 bits/element vector
  - Register file has 16 read ports and 8 write ports
- Vector functional units
  - Fully pipelined
  - Data and control hazards are detected
- Vector load-store unit
  - Fully pipelined
  - One word per clock cycle after initial latency
- Scalar registers
  - 32 general-purpose registers
  - 32 floating-point registers



An example architecture: VMIPS (Part 2/2)

- Can use deep pipeline to execute element operations.
- *Fast clock* processing of independent vectors allows deep pipelining.



6-stage **multiply** pipeline

Scalar Implementation of function DAXPY

Example: DAXPY (double precision a * **X** + **Y**), consider the length of the vectors to be 64.

```
        L.D     F0,a        ; load scalar a
        DADDIU  R4,Rx,#512  ; last address to load
Loop:   L.D     F2,0(Rx)    ; load X[i]
        MUL.D   F2,F2,F0    ; a * X[i]
        L.D     F4,0(Ry)    ; load Y[i]
        ADD.D   F4,F2,F2    ; a * X[i] + Y[i]
        S.D     F4,0(Ry)    ; store into Y[i]
        DADDIU  Rx,Rx,#8    ; increment index to X
        DADDIU  Ry,Ry,#8    ; increment index to Y
        SUBBU   R20,R4,Rx   ; compute bound
        BNEZ    R20,Loop    ; check if done
```

- Requires almost 580 MIPS ops

VMIPS Instructions (Part 1/2)

Instruction	Operands	Function
ADDVV.D	V1,V2,V3	Add elements of V2 and V3, then put each result in V1.
ADDVS.D	V1,V2,F0	Add F0 to each element of V2, then put each result in V1.
SUBVV.D	V1,V2,V3	Subtract elements of V3 from V2, then put each result in V1.
SUBVS.D	V1,V2,F0	Subtract F0 from elements of V2, then put each result in V1.
SUBSV.D	V1,F0,V2	Subtract elements of V2 from F0, then put each result in V1.
MULVV.D	V1,V2,V3	Multiply elements of V2 and V3, then put each result in V1.
MULVS.D	V1,V2,F0	Multiply each element of V2 by F0, then put each result in V1.
DIVVV.D	V1,V2,V3	Divide elements of V2 by V3, then put each result in V1.
DIVVS.D	V1,V2,F0	Divide elements of V2 by F0, then put each result in V1.
DIVSV.D	V1,F0,V2	Divide F0 by elements of V2, then put each result in V1.
LV	V1,R1	Load vector register V1 from memory starting at address R1.
SV	R1,V1	Store vector register V1 into memory starting at address R1.

VMIPS Instructions (Part 2/2)

- ADDVV.D: add two vectors
- ADDVS.D: add vector to a scalar
- LV/SV: vector load and vector store from address

Example: DAXPY (double precision a * **X** + **Y**)

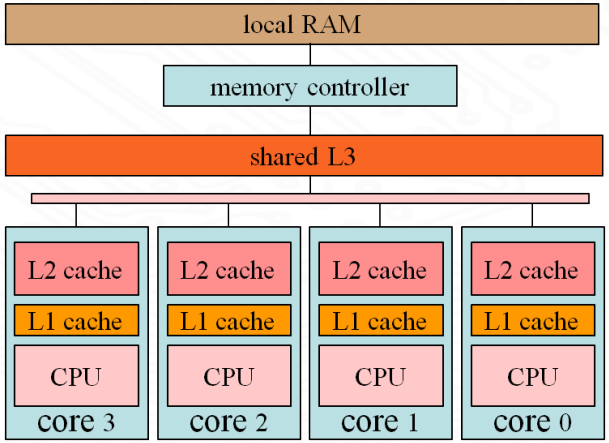
```
    L.D     F0,a      ; load scalar a
    LV      V1,Rx     ; load vector X
    MULVS.D V2,V1,F0  ; vector-scalar multiply
    LV      V3,Ry     ; load vector Y
    ADDVV   V4,V2,V3  ; add
    SV      Ry,V4     ; store the result
```

- Requires 6 instructions

Summary: thread-level parallelism

- Motivation for multicore systems
- Examples for multicore
- Application areas
- Cache coherence problem
- Challenges in efficient multicore system design

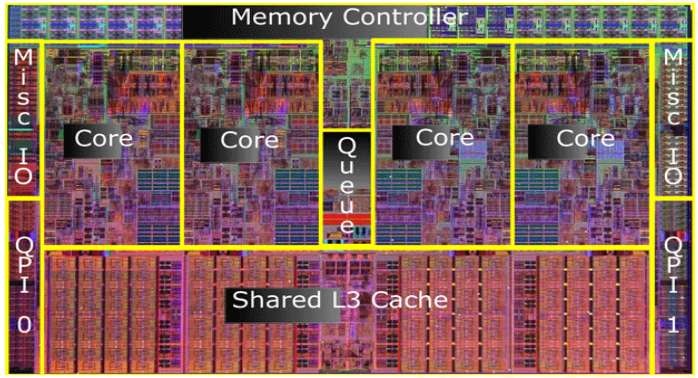
Examples of multicore processors (Part 2/2)



EX: AMD Phenom II X4,
Intel Core i5 2500T,
Intel Nehalem

Quad core processor

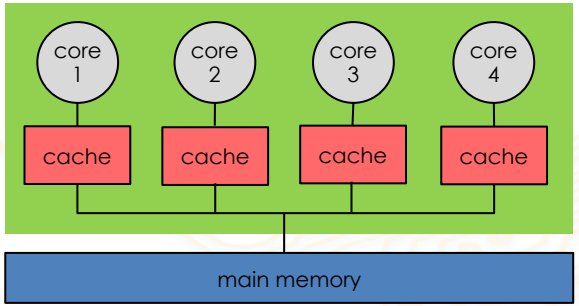
HTC One X,
Samsung galaxy note 2



Intel Quad Core Nehalem

Cache Coherence

- There could be many copies of an instruction operand:
  - One copy in the main memory and one in each cache.
  - If one copy of an operand is changed, then other copies must be changed as well.
  - Ideally the changes in other copies should happen instantaneously.
- Cache coherence ensures that for any change in the value of a shared operand there should be necessary changes in other copies in time to eliminate chances of processing the old data by any of the cores.

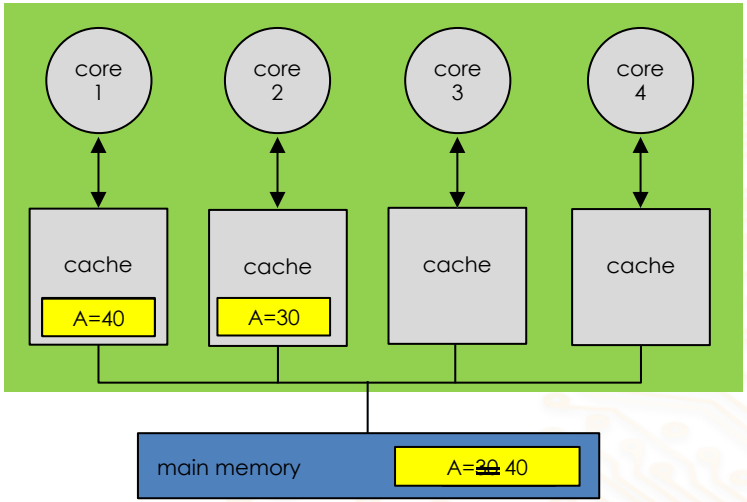


Shared memory system with a separate cache for each processor/core

Illustration of the coherence problem

- Let core-1 reads a variable A from the main memory and caches that.
- Let core-2 also reads the same value 30 of the variable A from the main memory and caches that.
- According to the program suppose core-1 modifies (30) of A to 40.
- Write-back policy – Then main memory value of A=30 will remain till the cache line has to be evicted.
- Write-through policy – then memory value of A=40.

Core-2 assumes a different and incorrect value of A, while for core-3 and core-4 the value of A depends on cache update policy.



Solutions for cache coherence (Part 1/2)

- Software-based approach: shared data are not to be cached
  - makes the access of shared data too slow
- Invalidation approach: one core will have ownership of the data and all others will invalidate on occurrence of write
  - When a write operation is observed to a location that a cache has a copy of, the cache controller invalidates its own copy of the snooped memory location, which forces a read from main memory of the new value on its next access.
  - will lead to lot of cache misses: loss of temporal locality
  - will demand higher memory bandwidth

Solutions for cache coherence (Part 2/2)

- Update approach: writes are broadcast to all cores if they have a copy in the cache (i.e., if they share the data) and write-through policy should be used to update the main memory
  - bus traffic will increase with the number of cores which share the data.

- The memory bandwidth requirement increases.

- Hardware complexity of the implementation of cache coherence also will increase with the number of cores.


THANK YOU!