

National Institute of Technology Karnataka Surathkal
Department of Information Technology



IT 301 Parallel Computing

Introduction

Dr. Geetha V

Assistant Professor

Dept of Information Technology

NITK Surathkal

Index

- 1: Moore's Law and Need for Parallel Processing
- 2. Parallel Processing in Uniprocessor System
 - Bit Level , Instruction level, Task level and Data level parallelism
- 3: Multiprocessor and Multicores System

1: Introduction

Course Plan: Theory:

Part A: Parallel Computer Architectures

Week 1,2,3: ***Introduction to Parallel Computer Architecture:***

Parallel Computing,

Parallel architecture,

bit level, instruction level , data level and task level parallelism.

Instruction level parallelisms: pipelining(Data and control instructions),

scalar and superscalar processors,

vector processors.

Parallel computers and computation.

1: Moore's Law and Need for Parallel Processing

- Chip performance doubles every 18-24 months
- Power consumption is proportional to frequency of the system
- Limitations of serial Computing
 - Heating issues
 - Limit to transmission speeds
 - Leakage currents
 - Limit to miniaturization
[downsizing of electronic devices]
- Parallel processing in **Uniprocessor systems**
 - Bit level parallelism, Instruction level parallelism (pipelining, Superscalar processors), Data parallelism (vector processors) and task level parallelism (threads)
- Parallel processing in **Multiprocessor** and **Multicore systems**
 - Multi core processor, clusters, grids

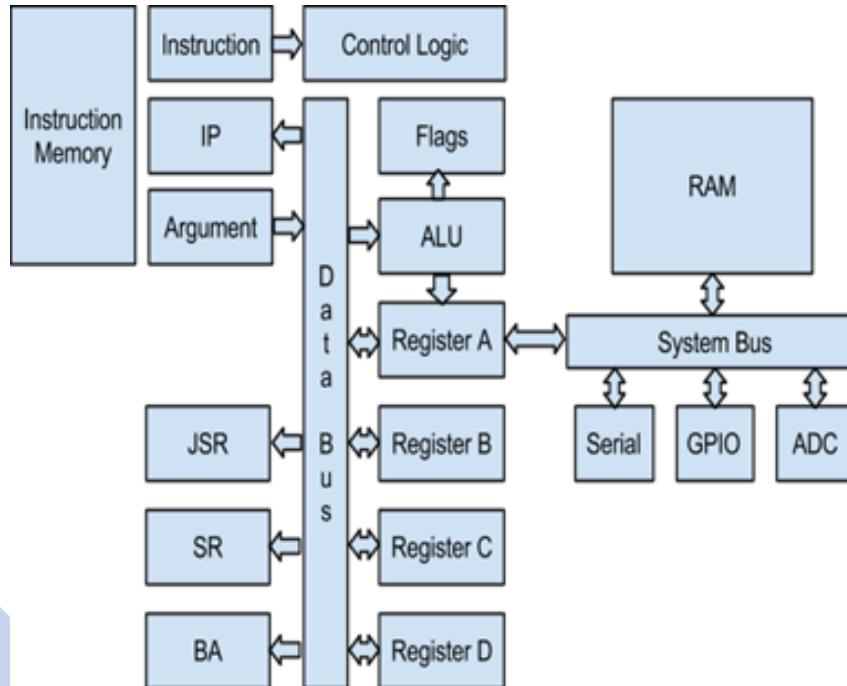
- Smaller transistor => faster processors
- Faster processor => increased power consumption
- Increased power consumption => increased heat
- Increased heat => unreliable processors

2. Parallel Processing in Uniprocessor System

- Bit level parallelism
- Instruction level parallelism (pipelining, Superscalar processors)
- Data parallelism (vector processors)
- Task level parallelism (threads)

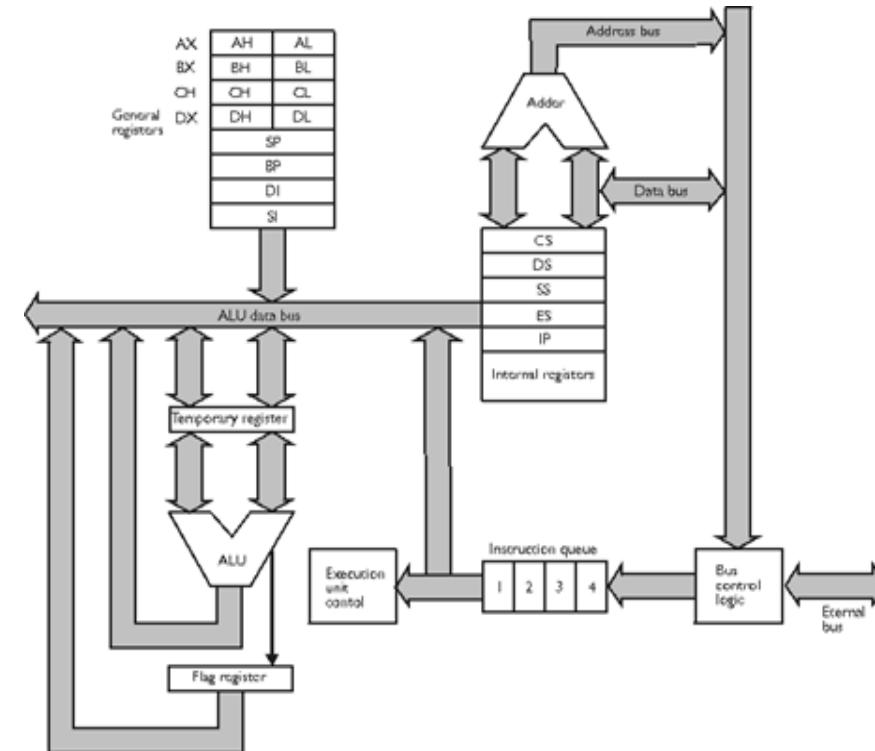
2. Parallel Processing in Uniprocessor System

- Bit level parallelism
- Adding 16-bit number in 8-bit processor



<http://ryanohs.com/2015/09/high-level-architecture/>

- Adding 16-bit number in 16-bit processor



http://www.c-jump.com/CIS77/CPU/VonNeumann/lecture.html#V77_0010_von_neumann

2. Parallel Processing in Uniprocessor System

- Bit Level Parallelism (eg. 1516H +2829H)
 - Adding 16-bit number in 8-bit processor
 - Adding 16-bit number in 16-bit processor

MVI A, #16H

MVI B, #29H

ADD B

MOV C, A

MVI A, #15H

MVI B, #28H

ADC B

MOV D, A

MVI AX, #1516H

MVI BX, #2829H

ADD AX, BX

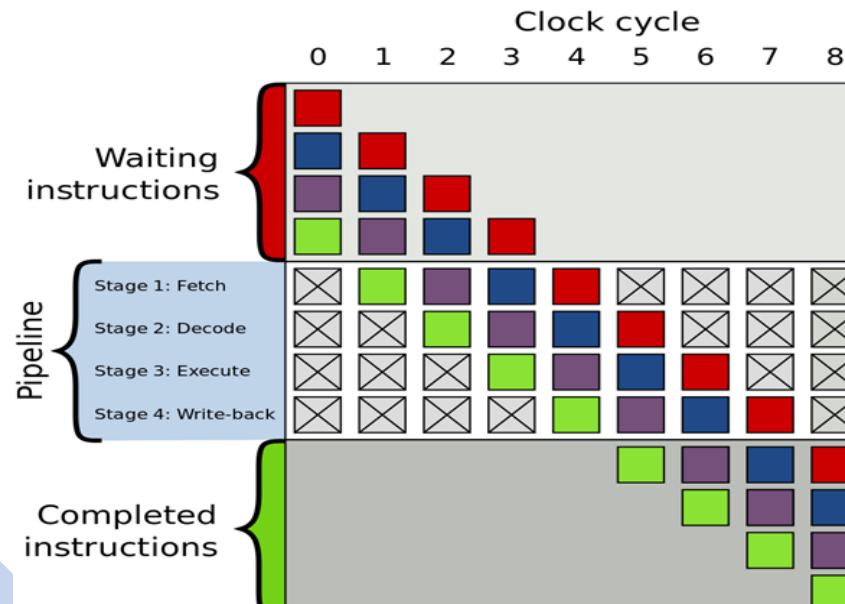
MOV CX, AX

2. Parallel Processing in Uniprocessor System

- Bit Level Parallelism
 - 8-bit processor (1972-1974)
Intel 8008, Intel 8080
 - 16-bit processor (1979 – 1982)
Intel 8086, Intel 286
 - 32-bit processor (1985 -2006)
Intel 386, Intel 486, P5 (Pentium)
Pentium Pro, Pentium II, Pentium III, Pentium M, Intel Core, Dual-core Xeon LV
 - 64-bit processor (2004 onwards)
Itanium , Itanium 2 (IA-64), Pentium 4F, Pentium D, Intel 6: Xeon (NetBurst), Intel Core 2, Intel Pentium Dual Core, Celeron, Celeron M; Nehalem: Intel Pentium, Core i3, i5, i7, Xeon.....etc

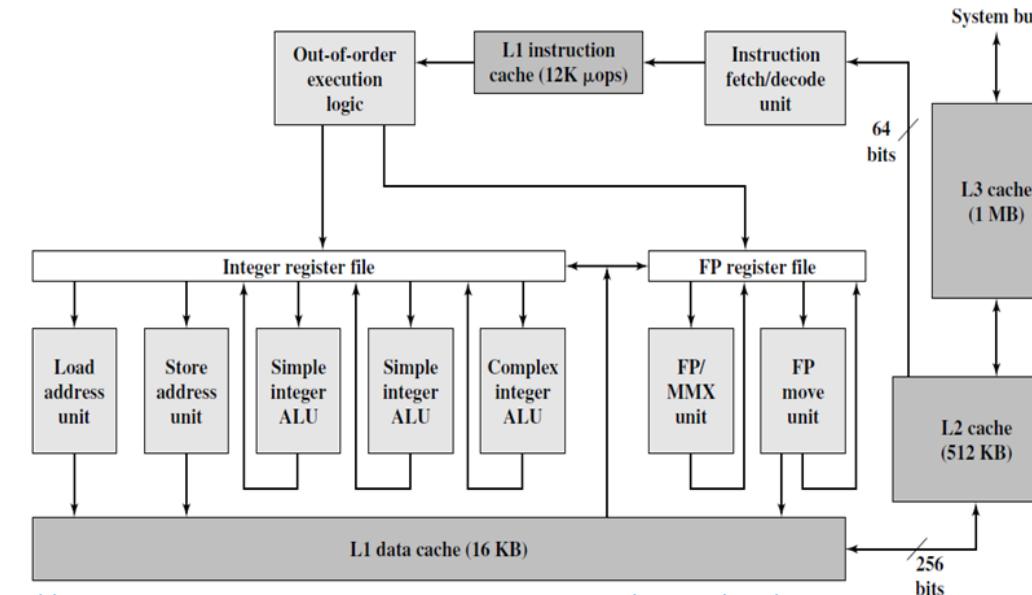
2. Parallel Processing in Uniprocessor System

- Instruction level parallelism (pipelining, Superscalar processors)
- Scalar Processors
 - Goal towards one instruction execution per cycle
- Superscalar Processors
 - Multiple Instruction per cycle



https://en.wikipedia.org/wiki/Instruction_pipelining

- Superscalar Processors
 - Multiple Instruction per cycle



<http://kkucoecomarch2016-2.blogspot.com/2017/02/mapping-function.html>

2. Parallel Processing in Uniprocessor System

- Instruction level parallelism (pipelining, Superscalar processors)
 - Scalar Processors
 - Goal towards one instruction execution per cycle
 - Reduced Instruction Set Computer (RISC) Scalar processors
 - Intel i860, Motorola MC8810, SUN's SPARC CY7C601 etc
 - Complex Instruction Set Computer (CISC) scalar Processors
 - Intel 386, 486; Motorola's 68030, 68040; etc
 - Superscalar Processors
 - Multiple Instruction per cycle
 - Pentium , Pentium Pro, Pentium II, Pentium III Motorola 88110 etc.

2. Parallel Processing in Uniprocessor System

- Task level parallelism

Task parallelism or function level parallelism is a form of parallelization of computer code across multiple processors in parallel computing environment

In Uniprocessor system, Thread level parallelism is used.

- Models

- Task dependency graph model
- Master Slave Model
- Pipeline/Producer Consumer model

- Data level parallelism

Data parallelism is a form of parallelization of computing across multiple processors in parallel computing environment.

Vector processors or coprocessors are added in case of Uniprocessor System

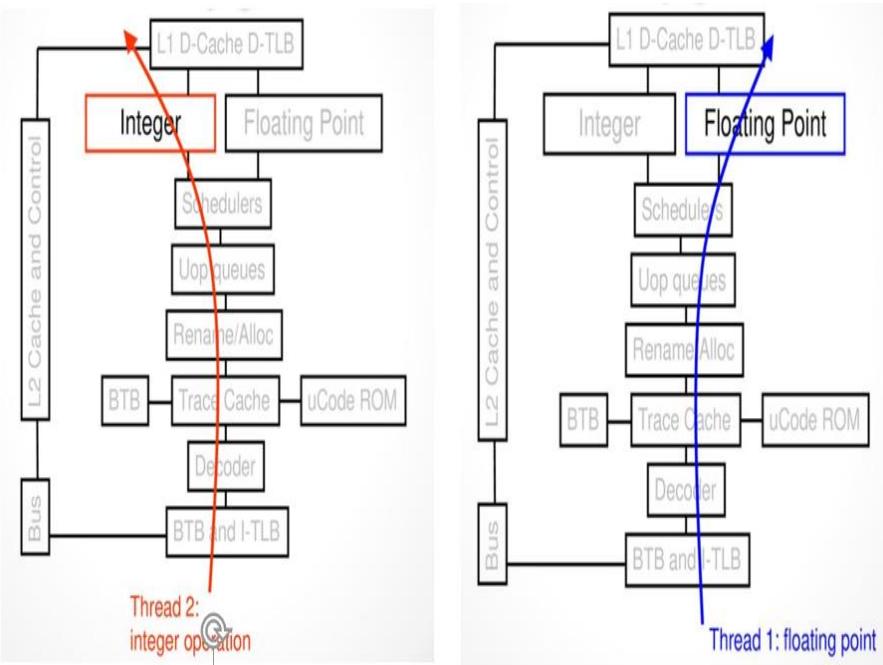
- Architectures

- Vector Processors
- Single Instruction Multiple Data (SIMD)

2. Parallel Processing in Uniprocessor System

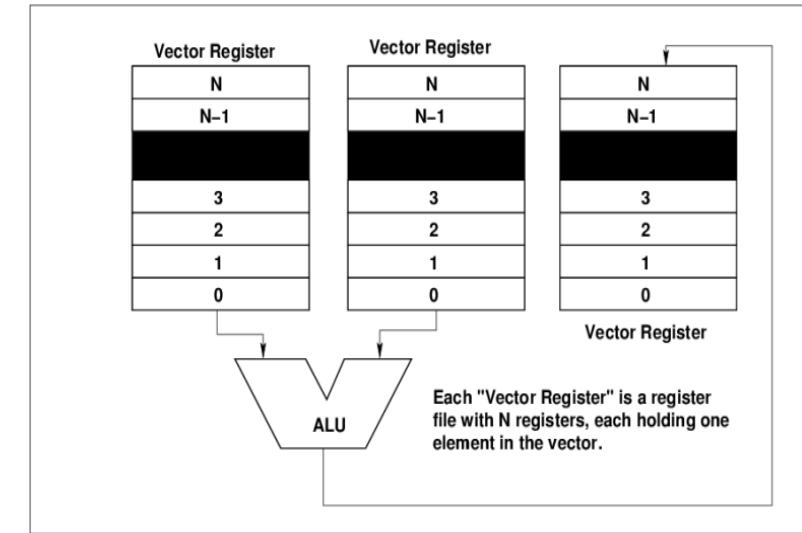
- Task level parallelism

Intel Xeon hyperthreading

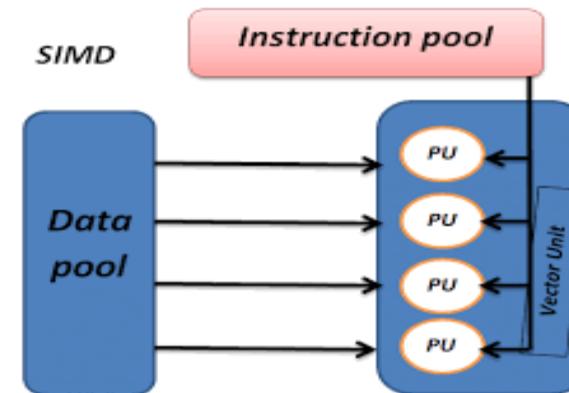


- Data level parallelism

• Vector Processor
Eg. Cray 1

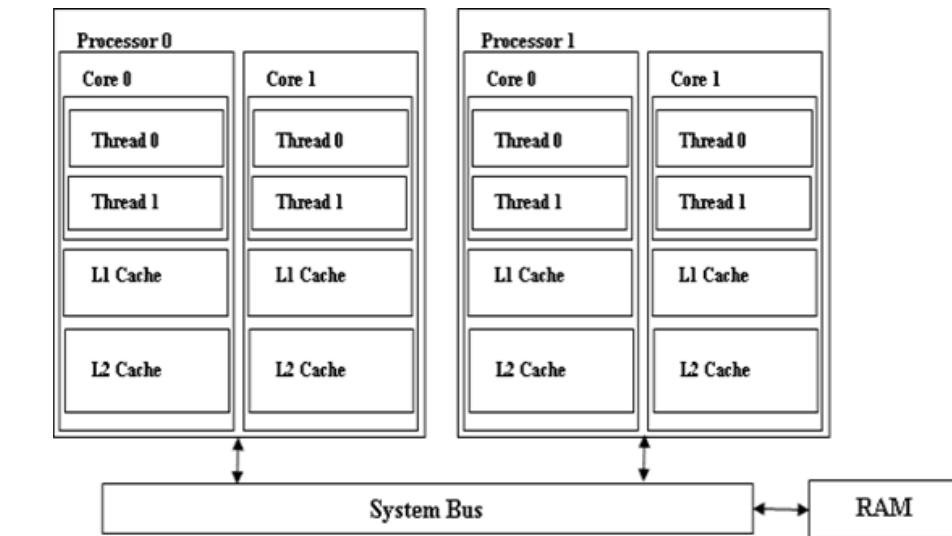
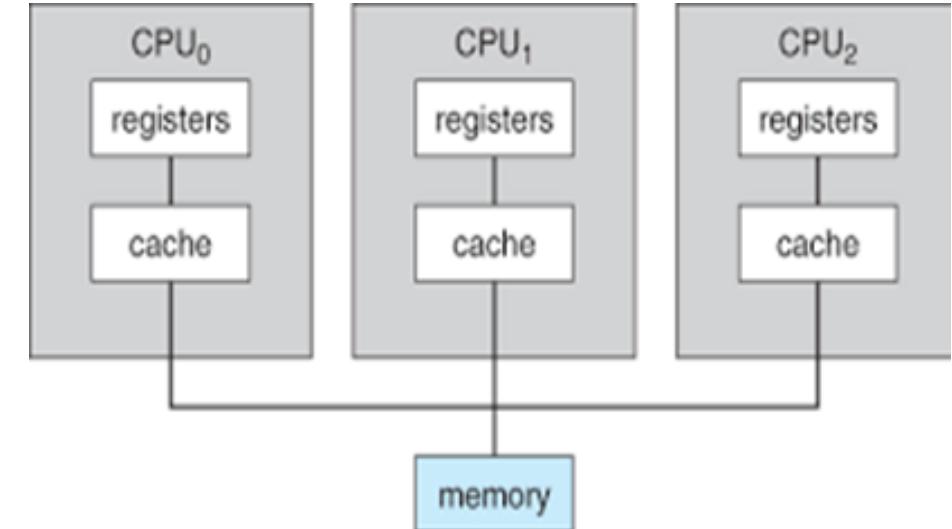


• SIMD
Eg. ILLIAC IV



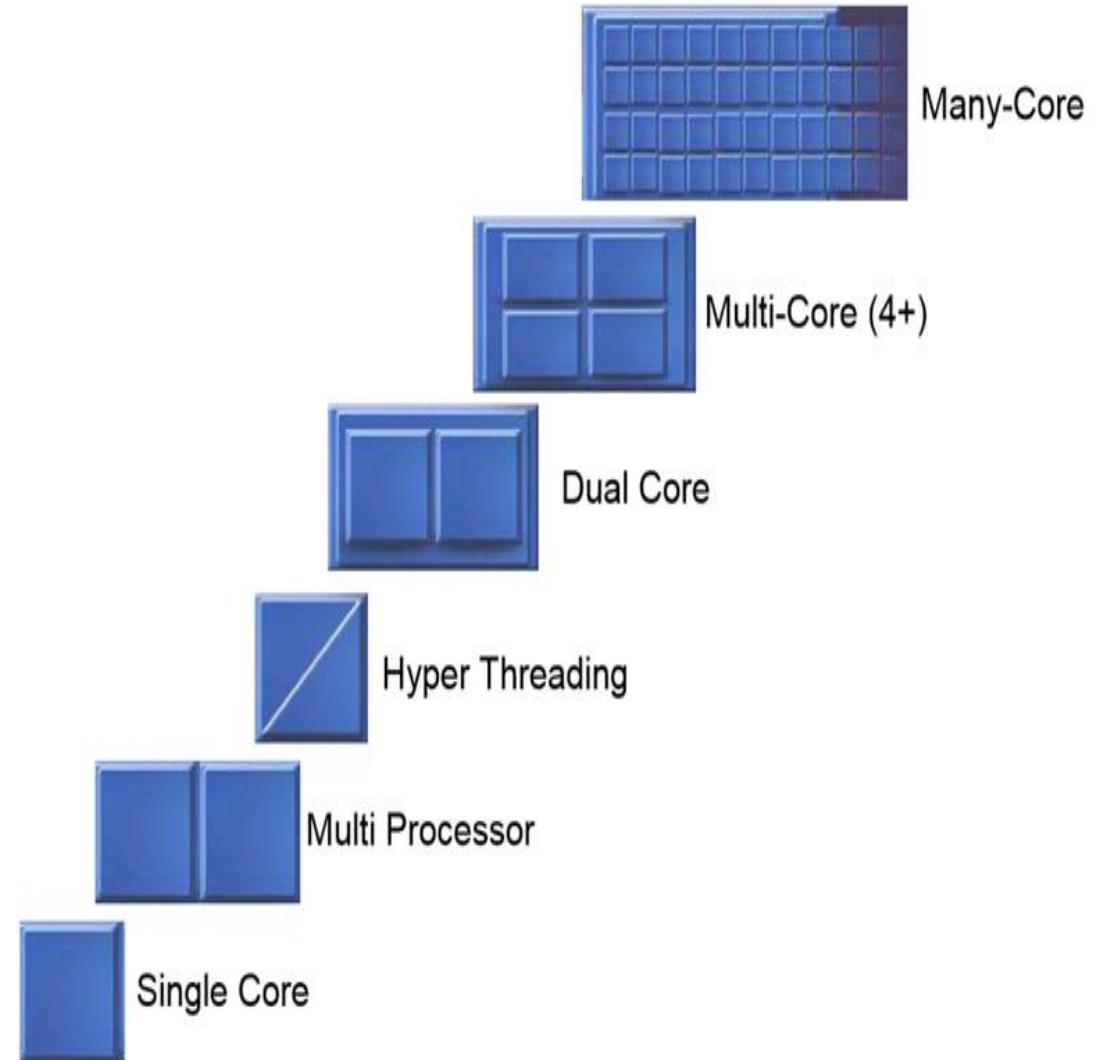
3: Multiprocessor and Multicore system

- **Multiprocessor system:** Two or more CPU within single computer system
- **Multicore processor:** Multiple Execution units (cores) of the same chip.



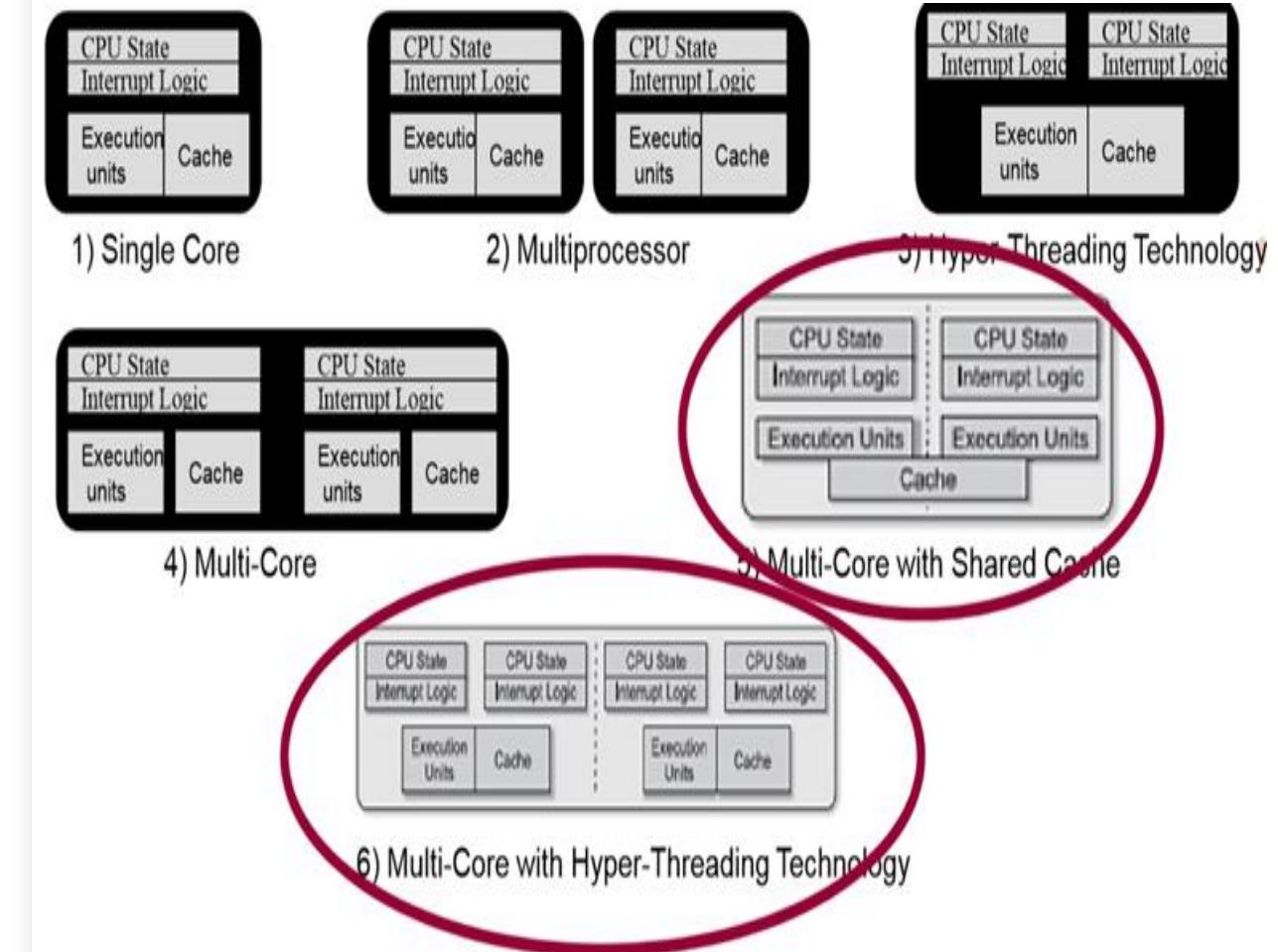
3: Multiprocessor and Multicore system

- **Multiprocessor system:** Two or more CPU within single computer system, sharing common memory and peripherals
- **Multicore processor:** Multiple Execution units (cores) of the same chip.



3: Multiprocessor and Multicore system

- **Single Core:** One execution unit in one chip
- **Multiprocessor:** Two or more CPU within single computer system
- **Hyperthreading:** Task level parallelism
- **Multicore processor:** Multiple Execution units (cores) on the same chip.
- **Multicore with shared cache:** All core shares same cache
- **Multicore with Hyperthreading:** task level parallelism



Hyper Threading and Multiple Threads

H Y P E R T H R E A D I N G V E R S U S M U L T I T H R E A D I N G

HYPER THREADING

.....
A technology that allows a single processor to operate like two separate processors to the operating system and the application programs that use it

.....
A physical processor is divided into two virtual or logical processors

MULTITHREADING

.....
A mechanism that allows multiple threads to exist within the context of a process such that they execute independently but share their process resources

.....
A process is divided into multiple threads

Reference

Textbooks and/or Reference Books:

1. Professional CUDA C Programming – John Cheng, Max Grossman, Ty McKercher, 2014
2. Wilkinson, M. Allen, "Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers", Pearson Education, 1999
3. I. Foster, "Designing and building parallel programs", 2003
4. Parallel Programming in C using OpenMP and MPI – Micheal J Quinn, 2004
5. Introduction to Parallel Programming – Peter S Pacheco, Morgan Kaufmann Publishers, 2011
6. Advanced Computer Architectures: A design approach, Dezso Sima, Terence Fountain, Peter Kacsuk, 2002
7. Parallel Computer Architecture : A hardware/Software Approach, David E Culler, Jaswinder Pal Singh Anoop Gupta, 2011
8. Introduction to Parallel Computing, Ananth Grama, Anshul Gupta, George Karypis, Vipin Kumar, Pearson, 2011

Thank You

National Institute of Technology Karnataka Surathkal
Department of Information Technology



IT 301 Parallel Computing
Shared Memory Programming Technique
OpenMP :*parallel, for*

Dr. Geetha V
Assistant Professor
Dept of Information Technology
NITK Surathkal

Index

- **Introduction**
 - Shared memory and Distributed Memory
 - OpenMP
- **OpenMP**
 - Program Structure
 - Directives : *parallel*, *for*
- **References**

Course Outline

Course Plan: Theory:

Part A: Parallel Computer Architectures

Week 1,2,3: ***Introduction to Parallel Computer Architecture:*** Parallel Computing, Parallel architecture, bit level, instruction level , data level and task level parallelism. Instruction level parallelism: pipelining(Data and control instructions), scalar and superscalar processors, vector processors. Parallel computers and computation.

Week 4,5: Memory Models: UMA, NUMA and COMA. Flynn's classification, Cache coherence,

Week 6,7: Amdahl's Law. Performance evaluation, Designing parallel algorithms : Divide and conquer, Load balancing, Pipelining.

Week 8 - 11: ***Parallel Programming techniques like Task Parallelism using TBB, TL2, Cilk++ etc. and software transactional memory techniques.***

Course Outline

Part B: OpenMP/MPI/CUDA

Week 1,2,3 : **Shared Memory Programming Techniques: Introduction to OpenMP :**

Directives: *parallel, for, sections, task, single, critical, barrier, taskwait, atomic.*

Clauses: *private, shared, firstprivate, lastprivate, reduction, nowait, ordered, schedule, collapse, num_threads, shared, if().*

Week 4,5: **Distributed Memory programming Techniques:** MPI: Blocking, Non-blocking.

Week 6,7 : CUDA : OpenCL, Execution models, GPU memory, GPU libraries.

Week 10,11,: **Introduction to accelerator programming using CUDA/OpenCL and Xeon-phi. Concepts of Heterogeneous programming techniques.**

Practical:

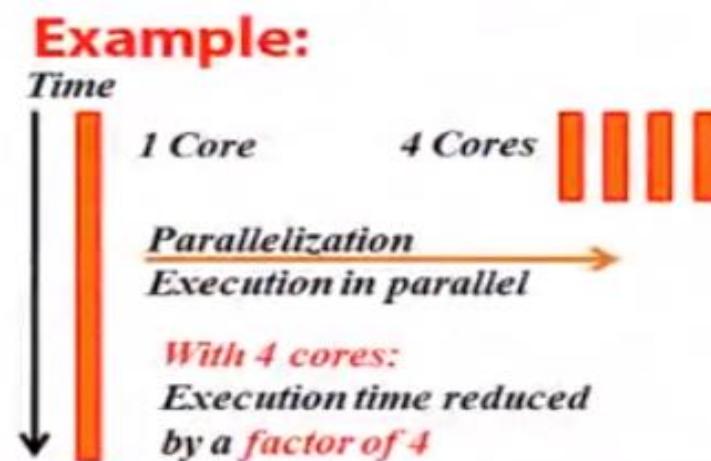
Implementation of parallel programs using OpenMP/MPI/CUDA.

Assignment: Performance evaluation of parallel algorithms (in group of 2 or 3 members)

1. Introduction

- Serial Programming
 - Develop a serial program and Optimize for performance
- Real World scenario:
 - Run multiple programs
 - Large and Complex problems
 - Time consuming
- Solution:
 - Use parallel machines
 - Use Multi-core Machines
- Why Parallel ?
 - Reduce the execution time
 - Run multiple programs

- What is parallel programming?
 - Obtain the same amount of computation with multiple cores or threads at low frequency (Fast)



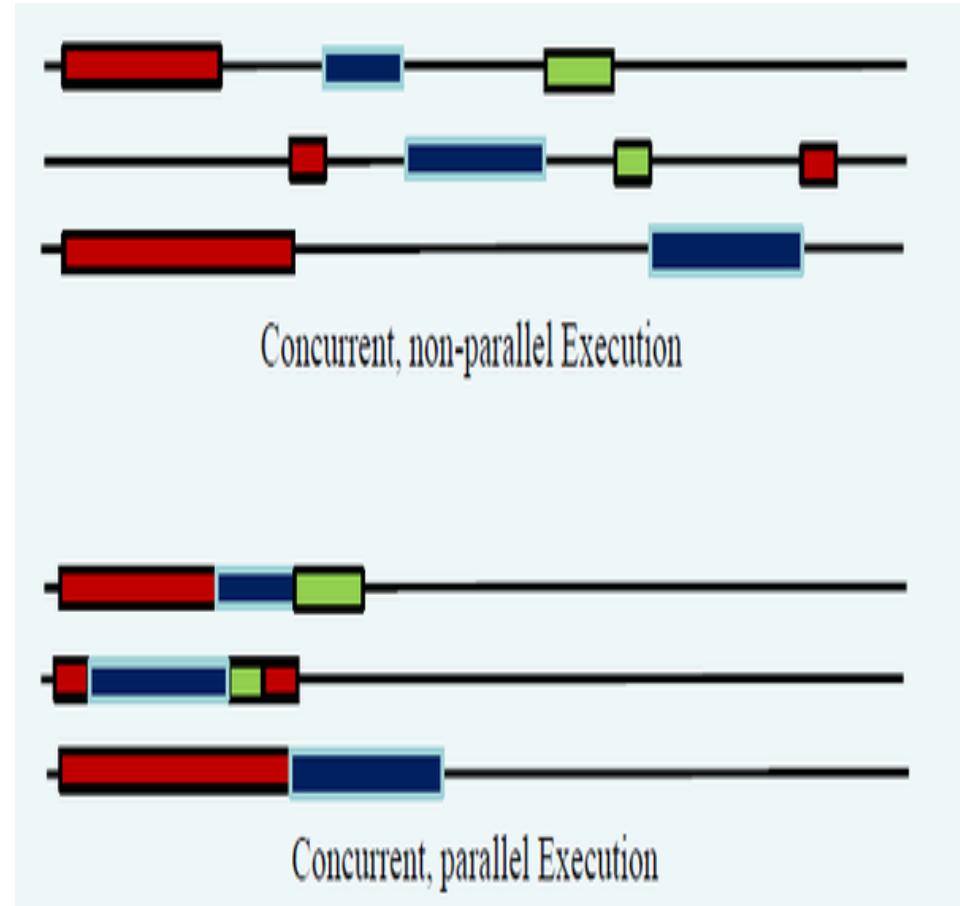
1. Introduction

- **Concurrency**

- Condition of a system in which multiple tasks are logically active at the same timebut they may not necessarily run in parallel

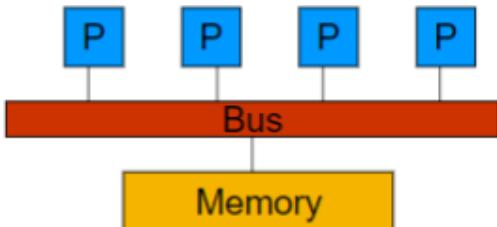
- **Parallelism**

- Subset of concurrency
- Condition of a system in which multiple tasks are active at the same time and run in parallel

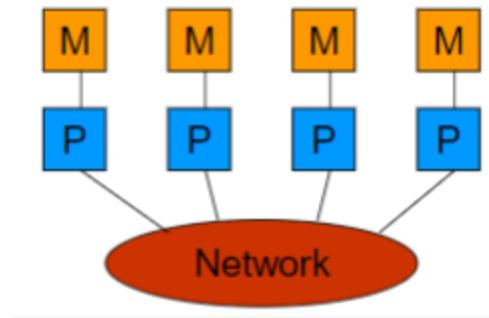


1. Introduction

- **Shared Memory Machines**
 - All processors share the same memory
 - The variables can be shared or private
 - Communication via shared memory
- **Multi-threading**
 - Portable, easy to program and use
 - Not very scalable
- **OpenMP based Programming**



- **Distributed Memory Machines**
 - Each processor has its own memory
 - The variables are Independent
 - Communication by passing messages (network)
- **Multi-Processing**
 - Difficult to program
 - Scalable
- **MPI based Programming**



1. OpenMP : API

Open Specification for Multi-Processing (OpenMP)

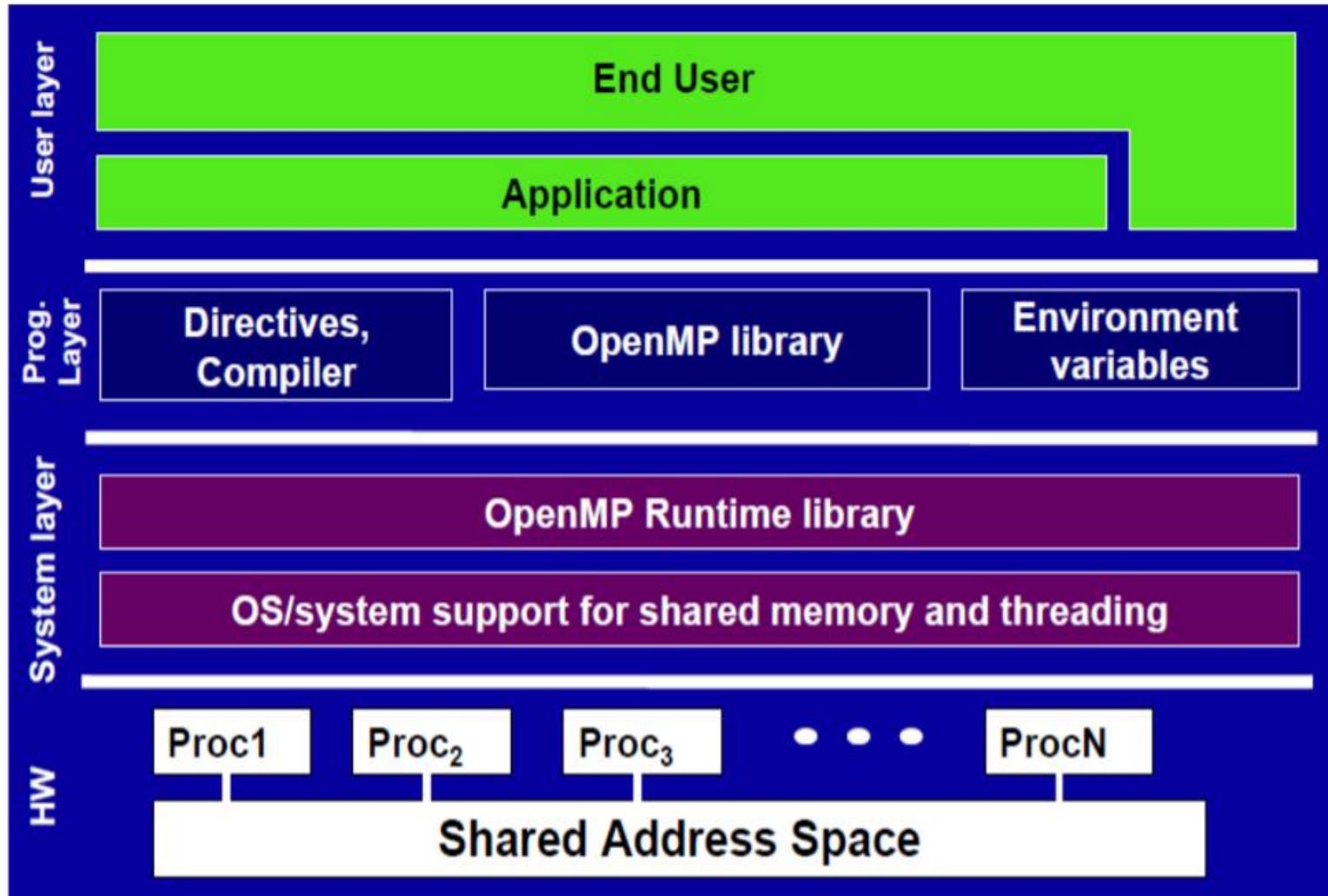
- Library used to divide computational work in a program and add parallelism to serial program (create threads)
- An Application Program Interface (API) that is used explicitly direct multi-threaded, shared memory parallelism
- API Components
 - Compiler Directives
 - Runtime library routines
 - Environment variables
- Standardization
 - Jointly defined and endorsed by major computer hardware and software vendors

1. OpenMP : API

- Open Specification for Multi-Processing (OpenMP)
- History
 - In 1991, Parallel Computing Forum (PCF) group invented a set of directives for specifying loop parallelism in Fortran Programs
 - X3H5, an ANSI subcommittee developed an ANSI standard based on PCF
 - In 1997, the first version of OpenMP for Fortran was defined by OpenMP Architecture Review Board.
 - Binding for C/C++ was introduced later.
 - Version 3.0 is available since 2008.
 - Version 4.0 is available since 2013.
 - Current version is 5.0, released in November 2018

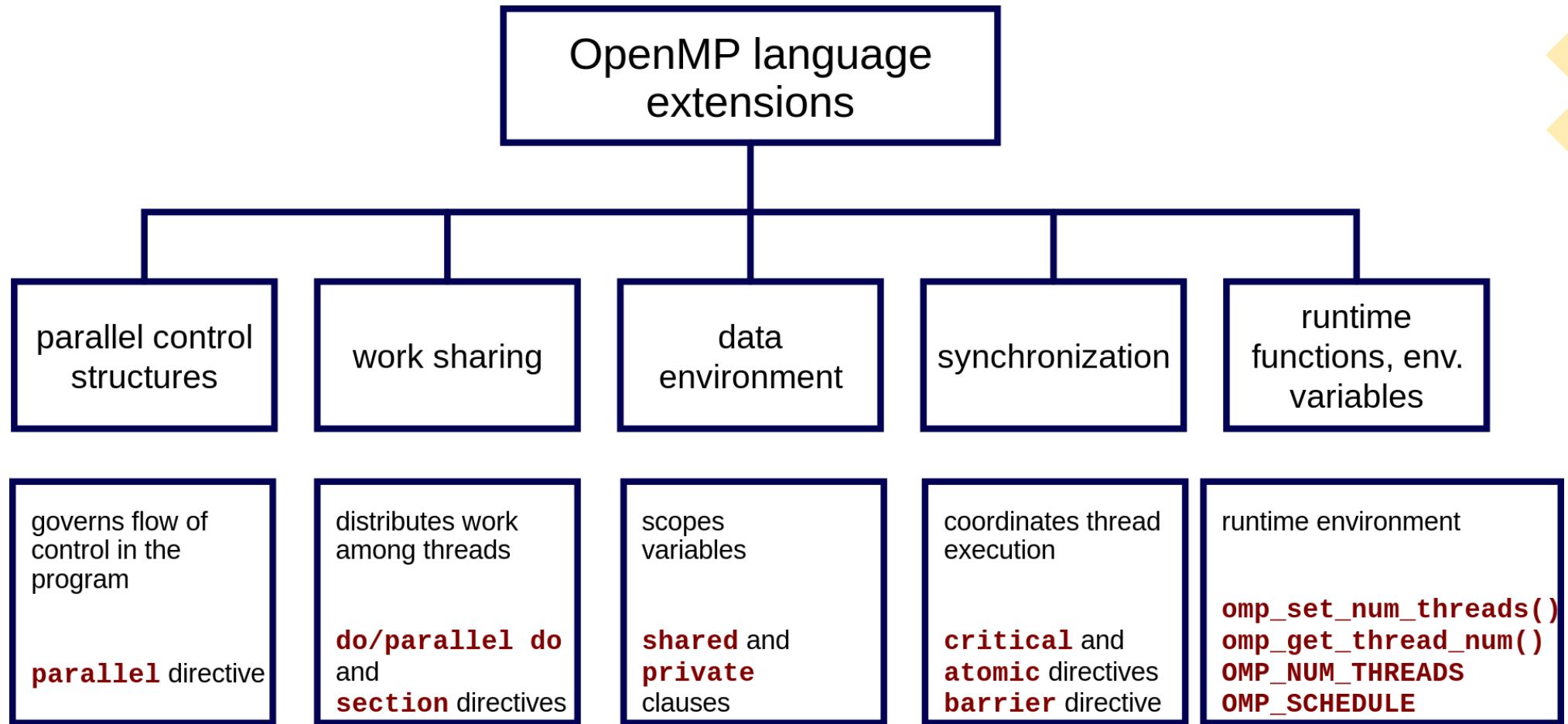
1. OpenMP : API

- Architecture



1. OpenMP : API

• OpenMP Language Extensions



1. OpenMP

Threads and Process

- A **process** is an instance of a computer program that is being executed. It contains the **program code** and its **current activity**.
- A **thread** of execution is the smallest unit of processing that can be scheduled by an operating system.
- **Differences between threads and processes:**
 - A thread is contained inside a process.
 - Multiple threads can exist within the same process and share resources such as memory.
 - The threads of a process share the latter's instructions (code) and its context (values that its variables reference at any given moment).
 - Different processes do not share these resources.

[http://en.wikipedia.org/wiki/Process_\(computing\)](http://en.wikipedia.org/wiki/Process_(computing))

1. OpenMP

Process

- A process contains all the information needed to execute the program

- Process ID
- Program code
- Data on run time stack
- Global data
- Data on heap

Each process has its own address space.

- In multitasking, processes are given time slices in a round robin fashion.
 - If computer resources are assigned to another process, the status of the present process must be saved, in order that the execution of the suspended process can be resumed later.

1. OpenMP

Threads

- Thread model is an **extension of the process model**.
- Each process consists of multiple independent instruction streams (or threads) that are assigned computer resources by some scheduling procedure.
- Threads of a process **share the address space** of this process.
 - Global variables and all dynamically allocated data objects are accessible by all threads of a process
- Each thread has its own run-time stack, register, program counter.
- Threads can **communicate by reading/writing variables** in the common address space.

1. OpenMP

OpenMP Programming Model

- Shared memory, thread-based parallelism
 - OpenMP is based on the existence of multiple threads in the shared memory programming paradigm.
 - A shared memory process consists of multiple threads.
- Explicit Parallelism
 - Programmer has full control over parallelization. OpenMP is not an automatic parallel programming model.
- Compiler directive based
 - Most OpenMP parallelism is specified using compiler directives which are embedded in the source code.

1. OpenMP

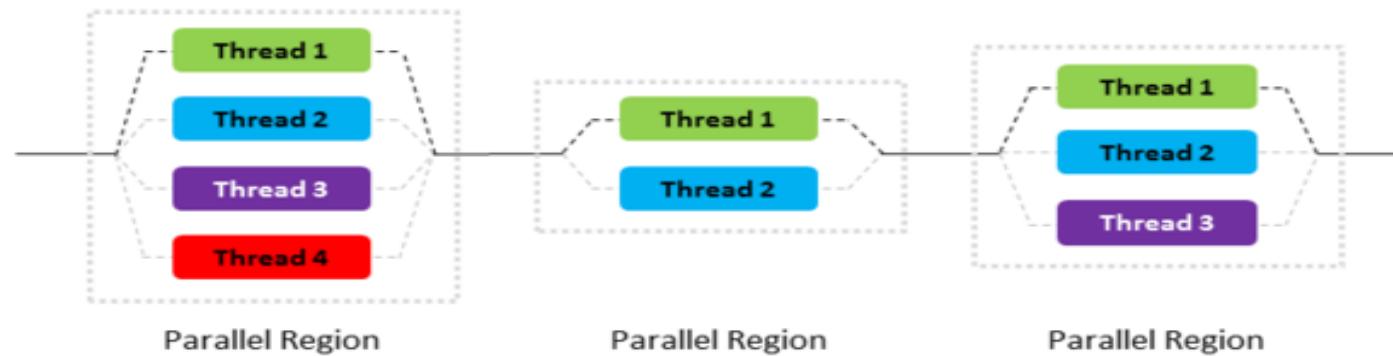
OpenMP is not

- Necessarily implemented identically by all vendors
- Meant for distributed-memory parallel systems (it is designed for shared address spaced machines)
- Guaranteed to make the most efficient use of shared memory
- Required to check for data dependencies, data conflicts, race conditions, or deadlocks
- Required to check for code sequences
- Meant to cover compiler-generated automatic parallelization and directives to the compiler to assist such parallelization
- Designed to guarantee that input or output to the same file is synchronous when executed in parallel.

1. OpenMP

FORK – JOIN Parallelism

- OpenMP program begin as a single process: the master thread. The master thread executes sequentially until the first parallel region construct is encountered.
- When a parallel region is encountered, master thread
 - Create a group of threads by FORK.
 - Becomes the master of this group of threads and is assigned the thread id 0 within the group.
- The statement in the program that are enclosed by the parallel region construct are then executed in parallel among these threads.
- JOIN: When the threads complete executing the statement in the parallel region construct, they synchronize and terminate, leaving only the master thread.



1. OpenMP

- I/O
 - OpenMP does not specify parallel I/O.
 - It is up to the programmer to ensure that I/O is conducted correctly within the context of a multithreaded program.
- Memory Model
 - Threads can “cache” their data and are not required to maintain exact consistency with real memory all the time.
 - When it is critical that all threads view a shared variable identically, the programmer is responsible for ensuring that the variable is updated by all threads as needed. (*flush*)

2. OpenMP Programming

```
%Program hello.c
#include <stdio.h>
#include <omp.h>

int main(void)
{
    #pragma omp parallel
    printf("Hello, world.\n");
    return 0;
}
```

Compiling the program

```
$ gcc -fopenmp hello.c -o hello
```

Output

```
Hello, world.
```

```
Hello, world.
```

2. OpenMP Programming

- **Directives**
 - An OpenMP executable directive applies to the succeeding structured block or an OpenMP Construct. A “structured block” is a single statement or a compound statement with a single entry at the top and a single exit at the bottom.
- **Clauses**
 - Not all the clauses are valid on all directives. The set of clauses that is valid on a particular directive is described with the directive. Most of the clauses accept a comma-separated list of list items. All list items appearing in a clause must be visible.
- **Runtime Library Routines**
 - Execution environment routines affect and monitor threads, processors, and the parallel environment. Lock routines support synchronization with OpenMP locks. Timing routines support a portable wall clock timer. Prototypes for the runtime library routines are defined in the file “omp.h”.

2. OpenMP Programming : *Parallel*

Directives

- Parallel
- For
- Sections
- Single
- Task
- Master
- Critical
- Barrier
- Taskwait
- Atomic
- Flush
- Ordered
- Threadprivate

Clauses

- Default
(shared/none)
- Shared
- Private
- Firstprivate
- Lastprivate
- Reduction
- Copyin
- copyprivate

Run time variables

- `omp_set_num_threads`
- `omp_get_num_threads`
- `omp_get_max_threads`
- `omp_get_thread_num(v oid)`
-etc

2. OpenMP Programming

- **Directives**

- An OpenMP executable directive applies to the succeeding structured block or an OpenMP Construct. A “structured block” is a single statement or a compound statement with a single entry at the top and a single exit at the bottom.
- They are case sensitive
- Starts with **#pragma omp**
- Directives cannot be embedded in embedded within continued statements, and statements cannot be embedded within directives.
- Only one directive-name can be specified per directive

2. OpenMP Programming: Directives

The parallel construct forms a team of threads and starts parallel execution.

```
#pragma omp parallel [clause[,]clause...] new-line  
Structured-block  
Clause: if(scalar-expression)  
        num_threads(integer-expression)  
        default(shared/none)  
        private(list)  
        firstprivate(list)  
        shared(list)  
        copyin(list)  
        reduction(operator:list)
```

Restrictions

- A program which branches into or out of a parallel region is non-conforming.
- A program must not depend on any ordering of the evaluations of the clauses of the parallel directive, or on any side effects of the evaluations of the clauses.
- At most one ***if*** clause can appear on the directive.
- At most one ***num_threads*** clause can appear on the directive. The ***num_threads*** expression must evaluate to a positive integer value.

2. OpenMP Programming: Directives

```
#pragma omp parallel [clause[,]clause...]
new-line
Structured-block
Clause: if (scalar-expression)
        num_threads(integer-expression)
        default(shared/none)
        private(list)
        firstprivate(list)
        shared(list)
        copyin(list)
        reduction(operator:list)
```

- A team of threads is created to execute the parallel region
- A thread which encounters the parallel construct becomes master thread
- The thread id of master is 0
- All threads including master thread executes parallel region
- **omp_get_thread_num()** provides thread id
- There is implied barrier at the end of a parallel region.
- If a thread in a team executing a parallel region encounters another parallel directive, it creates a new team, and that thread is master of new team.

2. OpenMP Programming: Directives

```
#pragma omp parallel [clause[,]clause...]
new-line
Structured-block
Clause: if(scalar-expression)
        num_threads(integer-expression)
        default(shared/none)
        private(list)
        firstprivate(list)
        shared(list)
        copyin(list)
        reduction(operator:list)
```

- If execution of a thread terminates while inside a parallel region, execution of all threads in all teams terminates.
- The order of termination of threads is unspecified
- All the work done by a team prior to any barrier which the team has passed in the program is guaranteed to be complete.
- The amount of work done by each thread after the last barrier that it passed and before it terminates is unspecified

2. OpenMP Programming

```
%Program hello.c  
%%Num_threads() sets nuber of threads  
  
#include <stdio.h>  
#include <omp.h>  
  
int main(void)  
{  
    #pragma omp parallel num_threads(4)  
    printf("Hello, world.\n");  
    return 0;  
}
```

Compiling the program

```
$ gcc -fopenmp hello.c -o  
hello
```

Output

```
Hello, world.  
Hello, world.  
Hello, world.  
Hello, world.
```

2. OpenMP Programming: Directives - Loop

```
#pragma omp parallel  
  
#pragma omp parallel  
{  
    int threadnum = omp_get_thread_num(),  
        numthreads = omp_get_num_threads();  
  
    int low = N*threadnum/numthreads,  
        high = N*(threadnum+1)/numthreads;  
  
    for (i=low; i<high; i++)  
        // do something with i  
}
```

```
# pragma omp parallel for  
  
#pragma omp parallel  
#pragma omp for  
for (i=0; i<N; i++) {  
    // do something with i  
}
```

2. OpenMP Programming: Directives

A loop Construct specifies that the iterations of loops will be distributed among and executed by the encountering team of threads.

```
#pragma omp for [clause[,]clause...] new-line  
for-loops
```

Clause: **private**(*list*)

firstprivate(*list*)

lastprivate(*list*)

reduction(*operator*:*list*)

schedule(*kind* [, *chunk_size*])

collapse(*n*)

ordered

nowait

Example

```
#pragma omp parallel  
#pragma omp for  
for (i=0; i<N; i++) {  
    // do something with i  
}
```

2. OpenMP Programming

- *#pragma omp for* Restrictions
 - The values of the loop control expressions of the loop associated with the loop directive must be the same for all the threads in the team.
 - Only a single **schedule** clause can appear on a loop directive.
 - *chunk_size* must be a loop invariant integer expression with a positive value.
 - The value of the *chunk_size* expression must be the same for all threads in the team.
 - When **schedule(runtime)** is specified, *chunk_size* must not be specified.
 - Only a single **ordered** clause can appear on a loop directive.
 - The **ordered** clause must be present on the loop construct if any **ordered** region ever binds to a loop region arising from the loop construct.
 - The loop iteration variable may not appear in a **threadprivate** directive.
 - The *for-loop* must be a structured block, and in addition, its execution must not be terminated by a **break** statement.
 - The *for-loop* iteration variable var must have a signed integer type.
 - Only a single **nowait** clause can appear on a **for** directive.

Index

- **Introduction**
 - Shared memory and Distributed Memory
 - OpenMP
- **OpenMP**
 - Program Structure
 - Directives : parallel, for
- **References**

Reference

Text Books and/or Reference Books:

1. Professional CUDA C Programming – John Cheng, Max Grossman, Ty McKercher, 2014
2. B.Wilkinson, M.Allen, "Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers", Pearson Education, 1999
3. I.Foster, "Designing and building parallel programs", 2003
4. Parallel Programming in C using OpenMP and MPI – Micheal J Quinn, 2004
5. Introduction to Parallel Programming – Peter S Pacheco, Morgan Kaufmann Publishers, 2011
6. Advanced Computer Architectures: A design approach, Dezso Sima, Terence Fountain, Peter Kacsuk, 2002
7. Parallel Computer Architecture : A hardware/Software Approach, David E Culler, Jaswinder Pal Singh Anoop Gupta, 2011
8. Introduction to Parallel Computing, Ananth Grama, Anshul Gupta, George Karypis, Vipin Kumar, Pearson, 2011

Reference

Acknowledgements

1. Introduction to OpenMP <https://www3.nd.edu/~zxu2/acms60212-40212/Lec-12-OpenMP.pdf>
2. Introduction to parallel programming for shared memory
Machines <https://www.youtube.com/watch?v=LL3TAHpxOig>
3. OpenMP Application Program Interface Version 2.5 May 2005
4. OpenMP Application Program Interface Version 5.0 November 2018

Thank You

National Institute of Technology Karnataka Surathkal
Department of Information Technology



IT 301 Parallel Computing
Shared Memory Programming Technique (2)
OpenMP :*parallel, for* Clauses

Dr. Geetha V
Assistant Professor
Dept of Information Technology
NITK Surathkal

Index

- OpenMP
 - Program Structure
 - Directives : *parallel* , *for*
 - Clauses
 - If
 - Private
 - Shared
 - Default(shared/none)
 - Firstprivate
 - Num_threads
- References

Course Outline

Course Plan: Theory:

Part A: Parallel Computer Architectures

Week 1,2,3: ***Introduction to Parallel Computer Architecture:*** Parallel Computing, Parallel architecture, bit level, instruction level , data level and task level parallelism. Instruction level parallelism: pipelining(Data and control instructions), scalar and superscalar processors, vector processors. Parallel computers and computation.

Week 4,5: Memory Models: UMA, NUMA and COMA. Flynn's classification, Cache coherence,

Week 6,7: Amdahl's Law. Performance evaluation, Designing parallel algorithms : Divide and conquer, Load balancing, Pipelining.

Week 8 - 11: ***Parallel Programming techniques like Task Parallelism using TBB, TL2, Cilk++ etc. and software transactional memory techniques.***

Course Outline

Part B: OpenMP/MPI/CUDA

Week 1,2,3 : **Shared Memory Programming Techniques:** Introduction to OpenMP :
Directives: parallel, for, sections, task, single, critical, barrier, taskwait, atomic.
Clauses: private, shared, firstprivate, lastprivate, reduction, nowait, ordered, schedule, collapse, num_threads, if().

Week 4,5: **Distributed Memory programming Techniques:** MPI: Blocking, Non-blocking.

Week 6,7 : CUDA : OpenCL, Execution models, GPU memory, GPU libraries.

Week 10,11,: **Introduction to accelerator programming using CUDA/OpenCL and Xeon-phi. Concepts of Heterogeneous programming techniques.**

Practical:

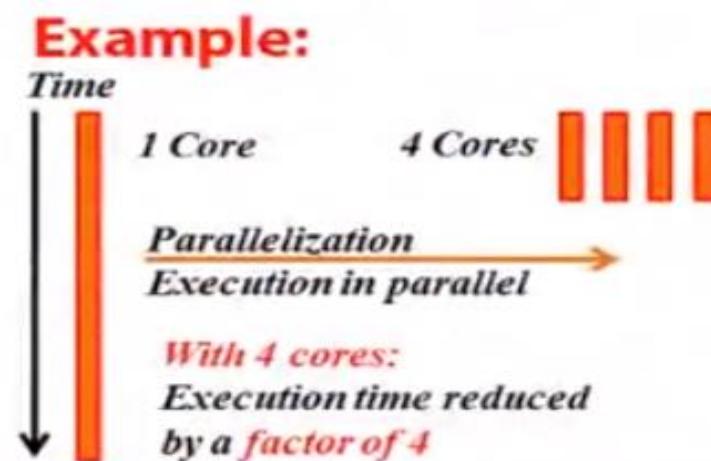
Implementation of parallel programs using OpenMP/MPI/CUDA.

Assignment: Performance evaluation of parallel algorithms (in group of 2 or 3 members)

1. Introduction

- Serial Programming
 - Develop a serial program and Optimize for performance
- Real World scenario:
 - Run multiple programs
 - Large and Complex problems
 - Time consuming
- Solution:
 - Use parallel machines
 - Use Multi-core Machines
- Why Parallel ?
 - Reduce the execution time
 - Run multiple programs

- What is parallel programming?
 - Obtain the same amount of computation with multiple cores or threads at low frequency (Fast)



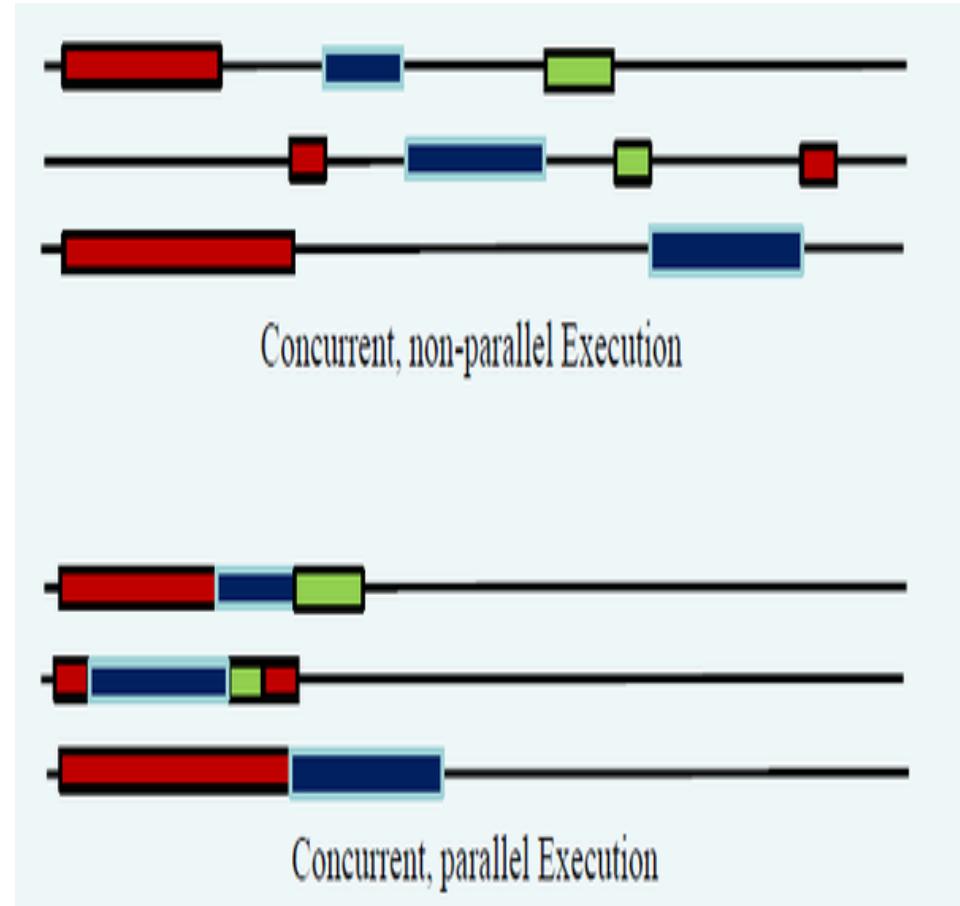
1. Introduction

- **Concurrency**

- Condition of a system in which multiple tasks are logically active at the same timebut they may not necessarily run in parallel

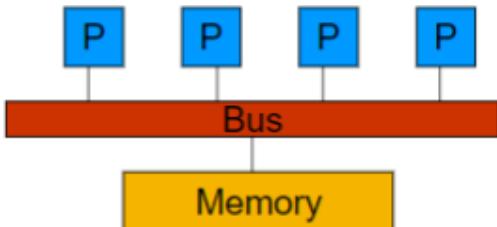
- **Parallelism**

- Subset of concurrency
- Condition of a system in which multiple tasks are active at the same time and run in parallel

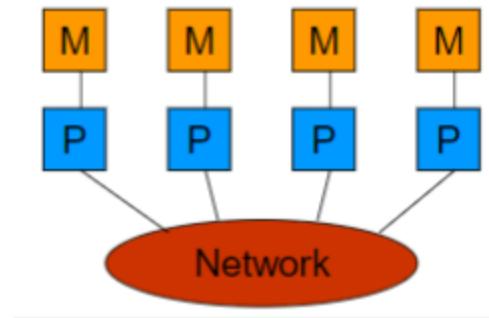


1. Introduction

- **Shared Memory Machines**
 - All processors share the same memory
 - The variables can be shared or private
 - Communication via shared memory
- **Multi-threading**
 - Portable, easy to program and use
 - Not very scalable
- **OpenMP based Programming**



- **Distributed Memory Machines**
 - Each processor has its own memory
 - The variables are Independent
 - Communication by passing messages (network)
- **Multi-Processing**
 - Difficult to program
 - Scalable
- **MPI based Programming**



1. OpenMP : API

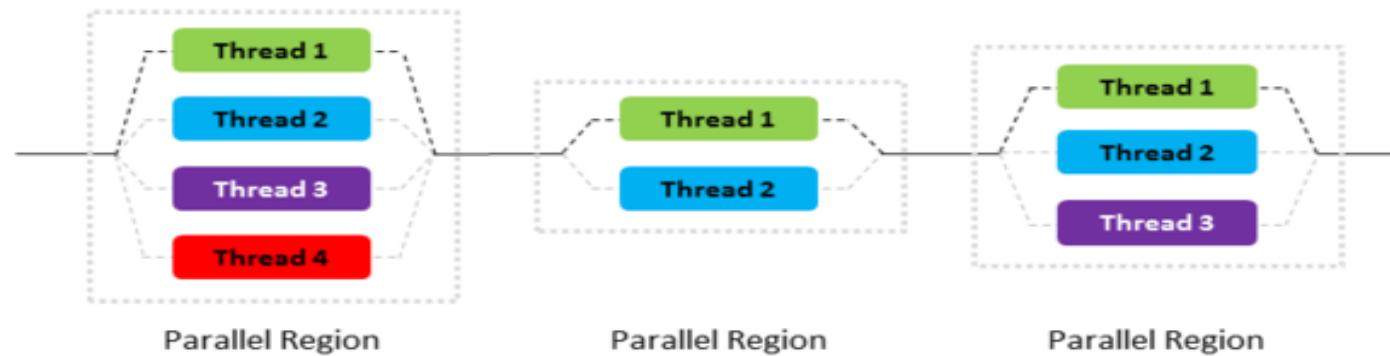
Open Specification for Multi-Processing (OpenMP)

- Library used to divide computational work in a program and add parallelism to serial program (create threads)
- An Application Program Interface (API) that is used explicitly direct multi-threaded, shared memory parallelism
- API Components
 - Compiler Directives
 - Runtime library routines
 - Environment variables
- Standardization
 - Jointly defined and endorsed by major computer hardware and software vendors

1. OpenMP

FORK – JOIN Parallelism

- OpenMP program begin as a single process: the master thread. The master thread executes sequentially until the first parallel region construct is encountered.
- When a parallel region is encountered, master thread
 - Create a group of threads by FORK.
 - Becomes the master of this group of threads and is assigned the thread id 0 within the group.
- The statement in the program that are enclosed by the parallel region construct are then executed in parallel among these threads.
- JOIN: When the threads complete executing the statement in the parallel region construct, they synchronize and terminate, leaving only the master thread.

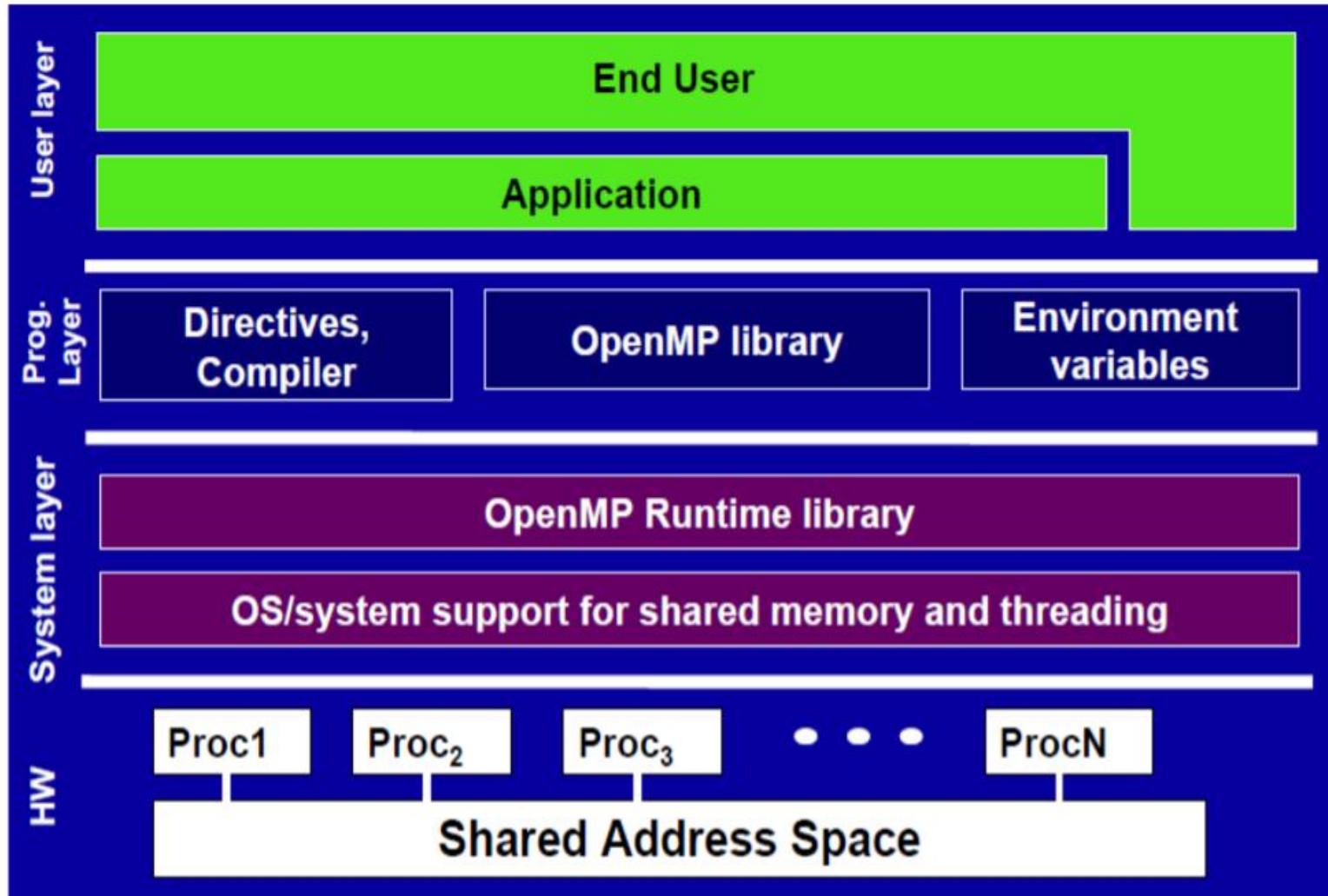


1. OpenMP

- I/O
 - OpenMP does not specify parallel I/O.
 - It is up to the programmer to ensure that I/O is conducted correctly within the context of a multithreaded program.
- Memory Model
 - Threads can “cache” their data and are not required to maintain exact consistency with real memory all the time.
 - When it is critical that all threads view a shared variable identically, the programmer is responsible for ensuring that the variable is updated by all threads as needed. (*flush*)

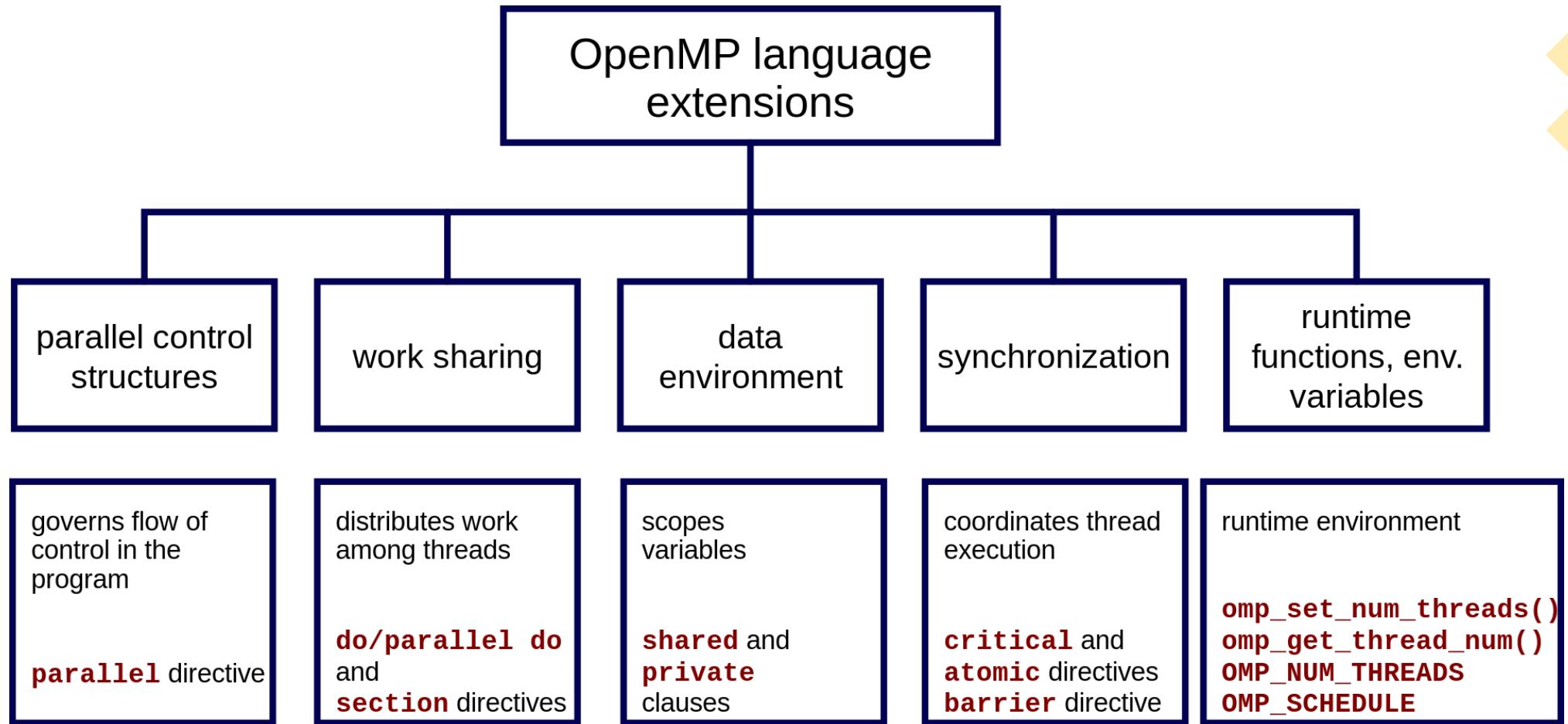
1. OpenMP : API

- Architecture



1. OpenMP : API

• OpenMP Language Extensions



2. OpenMP Programming

```
%Program hello.c
#include <stdio.h>
#include <omp.h>

int main(void)
{
    #pragma omp parallel
    printf("Hello, world.\n");
    return 0;
}
```

Compiling the program

```
$ gcc -fopenmp hello.c -o hello
```

Output

Hello, world.

Hello, world.

2. OpenMP Programming

- **Directives**
 - An OpenMP executable directive applies to the succeeding structured block or an OpenMP Construct. A “structured block” is a single statement or a compound statement with a single entry at the top and a single exit at the bottom.
- **Clauses**
 - Not all the clauses are valid on all directives. The set of clauses that is valid on a particular directive is described with the directive. Most of the clauses accept a comma-separated list of list items. All list items appearing in a clause must be visible.
- **Runtime Library Routines**
 - Execution environment routines affect and monitor threads, processors, and the parallel environment. Lock routines support synchronization with OpenMP locks. Timing routines support a portable wall clock timer. Prototypes for the runtime library routines are defined in the file “omp.h”.

2. OpenMP Programming : *Parallel*

Directives

- Parallel
- For
- Sections
- Single
- Task
- Master
- Critical
- Barrier
- Taskwait
- Atomic
- Flush
- Ordered
- Threadprivate

Clauses

- Default
(shared/none)
- Shared
- Private
- Firstprivate
- Lastprivate
- Reduction
- Copyin
- copyprivate

Run time variables

- `omp_set_num_threads`
- `omp_get_num_threads`
- `omp_get_max_threads`
- `omp_get_thread_num(v oid)`
-etc

2. OpenMP Programming

- **Directives**

- An OpenMP executable directive applies to the succeeding structured block or an OpenMP Construct. A “structured block” is a single statement or a compound statement with a single entry at the top and a single exit at the bottom.
- They are case sensitive
- Starts with `#pragma omp`
- Directives cannot be embedded in embedded within continued statements, and statements cannot be embedded within directives.
- Only one directive-name can be specified per directive

2. OpenMP Programming: Directives

The parallel construct forms a team of threads and starts parallel execution.

```
#pragma omp parallel [clause[,]clause...] new-line  
Structured-block  
Clause: if(scalar-expression)  
        num_threads(integer-expression)  
        default(shared/none)  
        private(list)  
        firstprivate(list)  
        shared(list)  
        copyin(list)  
        reduction(operator:list)
```

Restrictions

- A program which branches into or out of a parallel region is non-conforming.
- A program must not depend on any ordering of the evaluations of the clauses of the parallel directive, or on any side effects of the evaluations of the clauses.
- At most one ***if*** clause can appear on the directive.
- At most one ***num_threads*** clause can appear on the directive. The ***num_threads*** expression must evaluate to a positive integer value.

2. OpenMP Programming: Directives

```
#pragma omp parallel [clause[,]clause...]
new-line
Structured-block
Clause: if (scalar-expression)
        num_threads(integer-expression)
        default(shared/none)
        private(list)
        firstprivate(list)
        shared(list)
        copyin(list)
        reduction(operator:list)
```

- A team of threads is created to execute the parallel region
- A thread which encounters the parallel construct becomes master thread
- The thread id of master is 0
- All threads including master thread executes parallel region
- **omp_get_thread_num()** provides thread id
- There is implied barrier at the end of a parallel region.
- If a thread in a team executing a parallel region encounters another parallel directive, it creates a new team, and that thread is master of new team.

2. OpenMP Programming: Directives

```
#pragma omp parallel [clause[,]clause...]
new-line
Structured-block
Clause: if(scalar-expression)
        num_threads(integer-expression)
        default(shared/none)
        private(list)
        firstprivate(list)
        shared(list)
        copyin(list)
        reduction(operator:list)
```

- If execution of a thread terminates while inside a parallel region, execution of all threads in all teams terminates.
- The order of termination of threads is unspecified
- All the work done by a team prior to any barrier which the team has passed in the program is guaranteed to be complete.
- The amount of work done by each thread after the last barrier that it passed and before it terminates is unspecified

2. OpenMP Programming: Clauses

```
#pragma omp parallel [clause[,]clause...]
new-line
Structured-block
Clause: if(scalar-expression)
        num_threads(integer-expression)
        default(shared/none)
        private(list)
        firstprivate(list)
        shared(list)
        copyin(list)
        reduction(operator:list)
```

If Clause

- A structured block is executed in parallel if the evaluation of **if()** clause is evaluated as true.
- A missing if clause is equivalent to an if clause that evaluates true.
- At most one **if ()** clause can appear on the directive.

2. OpenMP Programming : if clause

```
% %Program hello.c  
% % if () clause
```

```
#include <stdio.h>  
#include <omp.h>  
  
int main(void)  
{  
    int par=0;  
    #pragma omp parallel if(par==1)num_threads(4)  
    printf("Hello,world.\n");  
    return 0;  
}
```

What is the result of the program?

Hello, world.

OR

Hello, world.

Hello, world.

Hello, world.

Hello, world.

2. OpenMP Programming: clauses num_threads

```
#pragma omp parallel [clause[,]clause...] new-line
```

Structured-block

Clause: if(*scalar-expression*)

num_threads(*integer-expression*)

default(*shared/none*)

private(*list*)

firstprivate(*list*)

shared(*list*)

copyin(*list*)

reduction(*operator:list*)

num_threads () Clause

- **num_threads** must evaluate to positive integer.
- It sets number of threads for the execution of parallel region
- Some of the execution environment routines
 - To set number of threads
void omp_set_num_threads(int num_threads);
 - To get number of threads
int omp_get_num_threads(void);
 - To find number of threads which can for a team
int omp_get_max_threads(void);
 - To get thread id
int omp_get_thread_num(void);

2. OpenMP Programming : num_threads clause

```
% %Program hello.c  
% % if () clause
```

```
#include <stdio.h>  
#include <omp.h>  
  
int main(void)  
{  
    int par=0;  
    #pragma omp parallel if(par==1) num_threads(6)  
    printf("Hello, world.\n");  
    return 0;  
}
```

What is the result of the program?

Hello, world.

OR

Hello, world.

Hello, world.

Hello, world.

Hello, world.

Hello, world.

Hello, world.

2. OpenMP Programming: num_threads

```
% %Program hello.c  
% % if () clause
```

```
#include <stdio.h>  
  
#include <omp.h>  
  
int main(void)  
{  
    int par=1;  
  
    #pragma omp parallel if(par==1) num_threads(6)  
    printf("Hello, world.\n");  
  
    return 0;  
}
```

What is the result of the program?

Hello, world.

OR

Hello, world.

Hello, world.

Hello, world.

Hello, world.

Hello, world.

Hello, world.

2. OpenMP Programming: Clauses

```
#pragma omp parallel [clause[,]clause...] new-line
```

Structured-block

Clause: `if(scalar-expression)`

`num_threads(integer-expression)`

`default(shared/none)`

`private(list)`

`firstprivate(list)`

`shared(list)`

`copyin(list)`

`reduction(operator:list)`

Default(shared/none) , shared(list) Clause

- **default (shared)** clause causes all variables referenced in the construct which have implicitly determined sharing attributes to be shared.
- **default(none)** clause requires that each variable which is referenced in the construct, and that does not have a predetermined sharing attribute, must have its sharing attribute explicitly determined by being listed in a data sharing attribute clause
- **shared(list)** : One or more list items must be shared among all the threads in a team.

2. OpenMP Programming: Directives

```
#pragma omp parallel [clause[,]clause...] new-line
```

Structured-block

Clause: `if(scalar-expression)`

`num_threads(integer-expression)`

`default(shared/none)`

`private(list)`

`firstprivate(list)`

`shared(list)`

`copyin(list)`

`reduction(operator:list)`

private (list) Clause

- **private (list)** clause declares one or more list items must be private to a thread.
- A list item that appears in the **reduction** clause of a parallel construct must not appear in a **private** clause on a work-sharing construct.

2. OpenMP Programming: Clauses – default(shared)

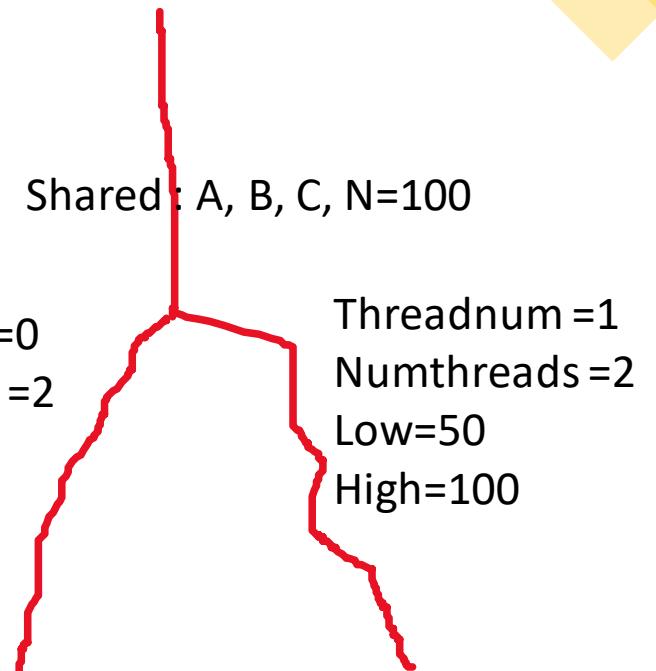
```
// N: total number of iterations
#pragma omp parallel default(shared)
private(threadnum, numthreads, low,high, i)
{
    int threadnum = omp_get_thread_num(),
        numthreads = omp_get_num_threads();

    int low = N*threadnum/numthreads,
        high = N*(threadnum+1)/numthreads;

    for (i=low; i<high; i++)
        a[i]=b[i]+c[i]
}
```

Shared : A, B, C

Other variables are default



2. OpenMP Programming: Directives – Default(none)

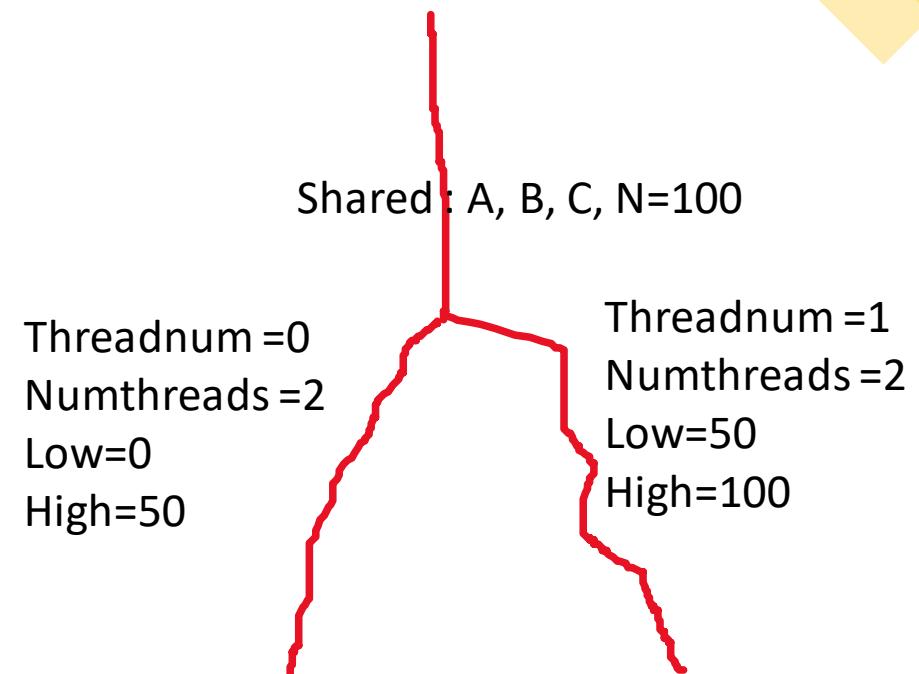
```
// N: total number of iterations
#pragma omp parallel default(none)
    shared(A,B,C,N) private(threadnum, numthreads,
    low,high, i)
{
    int threadnum = omp_get_thread_num(),
        numthreads = omp_get_num_threads();

    int low = N*threadnum/numthreads,
        high = N*(threadnum+1)/numthreads;

    for (i=low; i<high; i++)
        a[i]=b[i]+c[i]
}
```

Shared : A, B, C

Other variables are default



2. OpenMP Programming: Directives - shared

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>

int main (void) {
int x=0;
#pragma omp parallel shared(x)
{
    int tid=omp_get_thread_num();
    x=x+1;
    printf("Thread [%d] value of x is %d \n",tid,x);
}

return 0;
}
```

```
Thread [1] value of x is 1
Thread [0] value of x is 2
Thread [2] value of x is 3
Thread [3] value of x is 4
```

```
Thread [0] value of x is 2
Thread [1] value of x is 1
Thread [2] value of x is 3
Thread [3] value of x is 4
```

Since the x is shared, the change in one thread is visible to all other threads too.

2. OpenMP Programming: Directives - private

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>

int main (void) {
int x=0;
printf("x value ouside parallel:%d\n",x);
#pragma omp parallel private(x)
{
    x=10;
    int tid=omp_get_thread_num();
    x=x+1;
    printf("Thread [%d] value of x is %d \n",tid,x);
}

return 0;
}
```

```
x value ouside parallel:0
Thread [0] value of x is 11
Thread [2] value of x is 11
Thread [1] value of x is 11
Thread [3] value of x is 11
```

The variable x is private here . So the change of value performed in one thread is not visible to other threads.

2. OpenMP Programming: Directives - Private

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
int main (void) {
    int x=10, tid;
    printf("x value ouside parallel:%d\n",x);
    #pragma omp parallel num_threads(4) private(x) private(tid)
    {
        int tid=omp_get_thread_num();
        printf("\n 1. Thread [%d] value of x is %d \n",tid,x);
        x=15;
        printf("\n 2. Thread [%d] value of x is %d \n",tid,x);
        x=x+1;
        printf("\n 3. Thread [%d] value of x is %d \n",tid,x);
    }
    return 0;
}
```

```
x value ouside parallel:10
1. Thread [1] value of x is 6683504
2. Thread [1] value of x is 15
3. Thread [1] value of x is 16
1. Thread [3] value of x is 17143736
2. Thread [3] value of x is 15
3. Thread [3] value of x is 16
1. Thread [2] value of x is 6683600
2. Thread [2] value of x is 15
3. Thread [2] value of x is 16
1. Thread [0] value of x is 0
2. Thread [0] value of x is 15
3. Thread [0] value of x is 16
```

The variable x is private here. The value of x is 10 before parallel region. Since x is private, the value 10 is not reflected in threads. When the parallel region is entered, the x contains some garbage value as it is not initialed .

2. OpenMP Programming: Directives - Shared

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
int main (void) {
int x=10, tid;
printf("x value ouside parallel:%d\n",x);
#pragma omp parallel num_threads(4) shared(x) private(tid)
{
    int tid=omp_get_thread_num();
    printf("\n 1. Thread [%d] value of x is %d \n",tid,x);
    x=15;
    printf("\n 2. Thread [%d] value of x is %d \n",tid,x);
    x=x+1;
    printf("\n 3. Thread [%d] value of x is %d \n",tid,x);
}
return 0;
}
```

```
x value ouside parallel:10
1. Thread [2] value of x is 10
2. Thread [2] value of x is 15
3. Thread [2] value of x is 16
1. Thread [3] value of x is 10
2. Thread [3] value of x is 15
3. Thread [3] value of x is 16
1. Thread [1] value of x is 10
2. Thread [1] value of x is 15
3. Thread [1] value of x is 16
1. Thread [0] value of x is 10
2. Thread [0] value of x is 15
3. Thread [0] value of x is 16
```

X is shared. X is assigned value 15 inside parallel region.
Each thread is assigning value as 15. And updating it with $x=x+1$; But update is not getting reflected in other threads.

2. OpenMP Programming: Directives - Shared

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
int main (void) {
    int x=10, tid;
    printf("x value outside parallel:%d\n",x);
    #pragma omp parallel num_threads(4) shared(x) private(tid)
    {
        int tid=omp_get_thread_num();
        printf("\n 1. Thread [%d] value of x is %d \n",tid,x);
        //x=15;

        printf("\n 2. Thread [%d] value of x is %d \n",tid,x);
        x=x+1;

        printf("\n 3. Thread [%d] value of x is %d \n",tid,x);
    }
    return 0;
}
```

```
x value outside parallel:10
1. Thread [1] value of x is 10
2. Thread [1] value of x is 10
3. Thread [1] value of x is 11
1. Thread [3] value of x is 10
2. Thread [3] value of x is 11
3. Thread [3] value of x is 12
1. Thread [0] value of x is 10
2. Thread [0] value of x is 12
3. Thread [0] value of x is 13
1. Thread [2] value of x is 10
2. Thread [2] value of x is 13
3. Thread [2] value of x is 14
```

x is shared. No assignment in the parallel region.
Update is getting reflected in all other threads.
Synchronization is job of programmer.

2. OpenMP Programming: Directives

```
#pragma omp parallel [clause[,]clause...] new-line
```

Structured-block

Clause: `if(scalar-expression)`

`num_threads(integer-expression)`

`default(shared/none)`

`private(list)`

`firstprivate(list)`

`shared(list)`

`copyin(list)`

`reduction(operator:list)`

firstprivate (list) Clause

- **firstprivate (list)** clause declares one or more list items to be private to a thread and initializes each of them with that the corresponding original item has when the construct is encountered.
- For a **firstprivate** clause on a **parallel** construct, the initial value of the new item is the value of the original list item that exists immediately prior to the **parallel** construct for the thread that encounters the construct.

2. OpenMP Programming: Directives - Private

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
int main (void) {
    int x=10, tid;
    printf("x value ouside parallel:%d\n",x);
    #pragma omp parallel num_threads(4) private(x) private(tid)
    {
        int tid=omp_get_thread_num();
        printf("\n 1. Thread [%d] value of x is %d \n",tid,x);
        x=15;
        printf("\n 2. Thread [%d] value of x is %d \n",tid,x);
        x=x+1;
        printf("\n 3. Thread [%d] value of x is %d \n",tid,x);
    }
    return 0;
}
```

```
x value ouside parallel:10
1. Thread [1] value of x is 6683504
2. Thread [1] value of x is 15
3. Thread [1] value of x is 16
1. Thread [3] value of x is 17143736
2. Thread [3] value of x is 15
3. Thread [3] value of x is 16
1. Thread [2] value of x is 6683600
2. Thread [2] value of x is 15
3. Thread [2] value of x is 16
1. Thread [0] value of x is 0
2. Thread [0] value of x is 15
3. Thread [0] value of x is 16
```

X is private. While entering parallel region the x gets assigned with some value.

2. OpenMP Programming: Directives - Private

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
int main (void) {
int x=10, tid;
printf("x value outside parallel:%d\n",x);
#pragma omp parallel num_threads(4) private(x) private(tid)
{
    int tid=omp_get_thread_num();
    printf("\n 1. Thread [%d] value of x is %d \n",tid,x);

    x=x+1;

    printf("\n 3. Thread [%d] value of x is %d \n",tid,x);
}
return 0;
}
```

```
x value outside parallel:10
1. Thread [2] value of x is 6683600
3. Thread [2] value of x is 6683601
1. Thread [0] value of x is 0
3. Thread [0] value of x is 1
1. Thread [1] value of x is 6683504
3. Thread [1] value of x is 6683505
1. Thread [3] value of x is 14195704
3. Thread [3] value of x is 14195705
```

Shared x. x is already with some garbage value. X is private to each thread.

2. OpenMP Programming: Directives - Private

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
int main (void) {
int x=10, tid;
printf("x value ouside parallel:%d\n",x);
#pragma omp parallel num_threads(4) firstprivate(x) private(tid)
{
    int tid=omp_get_thread_num();
    printf("\n 1. Thread [%d] value of x is %d \n",tid,x);
    x=x+1;
    printf("\n 3. Thread [%d] value of x is %d \n",tid,x);
}
return 0;
}
```

```
x value ouside parallel:10
1. Thread [2] value of x is 10
3. Thread [2] value of x is 11
1. Thread [1] value of x is 10
3. Thread [1] value of x is 11
1. Thread [3] value of x is 10
3. Thread [3] value of x is 11
1. Thread [0] value of x is 10
3. Thread [0] value of x is 11
```

First private works as follows.

1. assign the value as in main thread before parallel region.
2. thread is private once it enters to parallel region.

2. OpenMP Programming: Directives

A loop Construct specifies that the iterations of loops will be distributed among and executed by the encountering team of threads.

```
#pragma omp for [clause[,]clause...] new-line  
for-loops
```

Clause: **private(list)**

firstprivate(list)

lastprivate(list)

reduction(operator:list)

schedule(kind[,chunk_size])

collapse(n)

ordered

nowait

Example

```
#pragma omp parallel  
#pragma omp for  
for (i=0; i<N; i++) {  
    // do something with i  
}
```

Index

- OpenMP
 - Program Structure
 - Directives : *parallel* , *for*
 - Clauses
 - If
 - Private
 - Shared
 - Default(shared/none)
 - Firstprivate
 - Num_threads

Reference

Text Books and/or Reference Books:

1. Professional CUDA C Programming – John Cheng, Max Grossman, Ty McKercher, 2014
2. B.Wilkinson, M.Allen, "Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers", Pearson Education, 1999
3. I.Foster, "Designing and building parallel programs", 2003
4. Parallel Programming in C using OpenMP and MPI – Micheal J Quinn, 2004
5. Introduction to Parallel Programming – Peter S Pacheco, Morgan Kaufmann Publishers, 2011
6. Advanced Computer Architectures: A design approach, Dezso Sima, Terence Fountain, Peter Kacsuk, 2002
7. Parallel Computer Architecture : A hardware/Software Approach, David E Culler, Jaswinder Pal Singh Anoop Gupta, 2011
8. Introduction to Parallel Computing, Ananth Grama, Anshul Gupta, George Karypis, Vipin Kumar, Pearson, 2011

Reference

Acknowledgements

1. Introduction to OpenMP <https://www3.nd.edu/~zxu2/acms60212-40212/Lec-12-OpenMP.pdf>
2. Introduction to parallel programming for shared memory
Machines <https://www.youtube.com/watch?v=LL3TAHpxOig>
3. OpenMP Application Program Interface Version 2.5 May 2005
4. OpenMP Application Program Interface Version 5.0 November 2018

Thank You

National Institute of Technology Karnataka Surathkal
Department of Information Technology



IT 301 Parallel Computing
Shared Memory Programming Technique (3)
OpenMP :parallel, for –reduction, master

Dr. Geetha V
Assistant Professor
Dept of Information Technology
NITK Surathkal

Index

- OpenMP
 - Program Structure
 - Directives : master
 - Clauses
 - Reduction
 - lastprivate
- References

Course Outline

Course Plan: Theory:

Part A: Parallel Computer Architectures

Week 1,2,3: ***Introduction to Parallel Computer Architecture:*** Parallel Computing, Parallel architecture, bit level, instruction level , data level and task level parallelism. Instruction level parallelism: pipelining(Data and control instructions), scalar and superscalar processors, vector processors. Parallel computers and computation.

Week 4,5: Memory Models: UMA, NUMA and COMA. Flynn's classification, Cache coherence,

Week 6,7: Amdahl's Law. Performance evaluation, Designing parallel algorithms : Divide and conquer, Load balancing, Pipelining.

Week 8 - 11: ***Parallel Programming techniques like Task Parallelism using TBB, TL2, Cilk++ etc. and software transactional memory techniques.***

Course Outline

Part B: OpenMP/MPI/CUDA

Week 1,2,3 : **Shared Memory Programming Techniques:** Introduction to OpenMP : Directives: *parallel, for, sections, task, master, single, critical, barrier, taskwait, atomic.* Clauses: *private, shared, firstprivate, lastprivate, reduction, nowait, ordered, schedule, collapse, num_threads, if()*.

Week 4,5: **Distributed Memory programming Techniques:** MPI: Blocking, Non-blocking.

Week 6,7 : CUDA : OpenCL, Execution models, GPU memory, GPU libraries.

Week 10,11,: **Introduction to accelerator programming using CUDA/OpenCL and Xeon-phi. Concepts of Heterogeneous programming techniques.**

Practical:

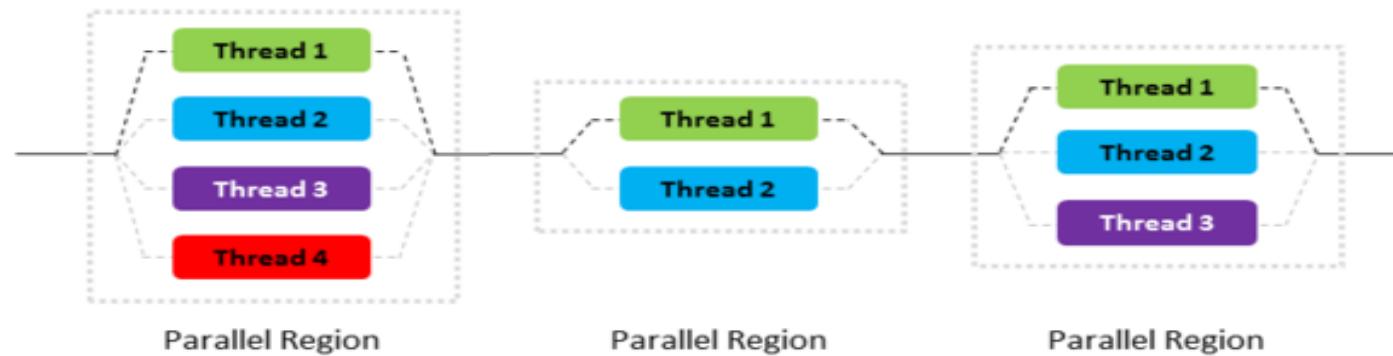
Implementation of parallel programs using OpenMP/MPI/CUDA.

Assignment: Performance evaluation of parallel algorithms (in group of 2 or 3 members)

1. OpenMP

FORK – JOIN Parallelism

- OpenMP program begin as a single process: the master thread. The master thread executes sequentially until the first parallel region construct is encountered.
- When a parallel region is encountered, master thread
 - Create a group of threads by FORK.
 - Becomes the master of this group of threads and is assigned the thread id 0 within the group.
- The statement in the program that are enclosed by the parallel region construct are then executed in parallel among these threads.
- JOIN: When the threads complete executing the statement in the parallel region construct, they synchronize and terminate, leaving only the master thread.



2. OpenMP Programming: Directives

```
#pragma omp parallel [clause[,]clause...] new-line
```

Structured-block

Clause: **if**(*scalar-expression*)

num_threads(*integer-expression*)

default(*shared/none*)

private(*list*)

firstprivate(*list*)

shared(*list*)

copyin(*list*)

reduction(*operator:list*)

2. OpenMP Programming: Clauses

```
#pragma omp parallel [clause[,]clause...]  
new-line
```

Structured-block

Clause: **if(scalar-expression)**

num_threads(integer-expression)

default(shared/none)

private(list)

firstprivate(list)

shared(list)

copyin(list)

reduction(operator:list)

Default(shared/none) , shared(list) Clause

- **if(scalar_expression):** if true execute in parallel
- **num_threads(int):** set number of threads
- **default (shared)** clause causes all variables referenced in the construct which have implicitly determined sharing attributes to be shared.
- **default(None)** clause requires that each variable which is referenced in the construct, and that does not have a predetermined sharing attribute, must have its sharing attribute explicitly determined by being listed in a data sharing attribute clause
- **shared(list) :** One or more list items must be shared among all the threads in a team.
- **private (list)** clause declares one or more list items must be private to a thread.
- **firstprivate (list)** clause declares one or more list items to be private to a thread and initializes each of them with that the corresponding original item has when the construct is encountered.

2. OpenMP Programming: Directives - Shared

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
int main (void) {
int x=10, tid;
printf("x value outside parallel:%d\n",x);
#pragma omp parallel num_threads(4) shared(x) private(tid)
{
    int tid=omp_get_thread_num();
    printf("\n 1. Thread [%d] value of x is %d \n",tid,x);
    x=15;
    printf("\n 2. Thread [%d] value of x is %d \n",tid,x);
    x=x+1;
    printf("\n 3. Thread [%d] value of x is %d \n",tid,x);
}
return 0;
}
```

```
x value outside parallel:10
1. Thread [2] value of x is 10
2. Thread [2] value of x is 15
3. Thread [2] value of x is 16
1. Thread [3] value of x is 10
2. Thread [3] value of x is 15
3. Thread [3] value of x is 16
1. Thread [1] value of x is 10
2. Thread [1] value of x is 15
3. Thread [1] value of x is 16
1. Thread [0] value of x is 10
2. Thread [0] value of x is 15
3. Thread [0] value of x is 16
```

X is shared. X is assigned value 15 inside parallel region.
Each thread is assigning value as 15. And updating it with x=x+1; But update is not getting reflected in other threads.

2. OpenMP Programming: Directives - Shared

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
int main (void) {
int x=10, tid;
printf("x value outside parallel:%d\n",x);
#pragma omp parallel num_threads(4) shared(x) private(tid)
{
    int tid=omp_get_thread_num();
    printf("\n 1. Thread [%d] value of x is %d \n",tid,x);
    //x=15;

    printf("\n 2. Thread [%d] value of x is %d \n",tid,x);
    x=x+1;

    printf("\n 3. Thread [%d] value of x is %d \n",tid,x);
}
return 0;
}
```

```
x value outside parallel:10
1. Thread [1] value of x is 10
2. Thread [1] value of x is 10
3. Thread [1] value of x is 11
1. Thread [3] value of x is 10
2. Thread [3] value of x is 11
3. Thread [3] value of x is 12
1. Thread [0] value of x is 10
2. Thread [0] value of x is 12
3. Thread [0] value of x is 13
1. Thread [2] value of x is 10
2. Thread [2] value of x is 13
3. Thread [2] value of x is 14
```

x is shared. No assignment in the parallel region.
Update is getting reflected in all other threads.
Synchronization is job of programmer.

2. OpenMP Programming: Directives - master

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
int main (void) {
    int x=10, tid;
    printf("x value ouside parallel:%d\n",x);
    #pragma omp parallel num_threads(4) shared(x) private(tid)
    {

        int tid=omp_get_thread_num();
        printf("\n 1. Thread [%d] value of x is %d \n",tid,x);
        #pragma omp master
        {
            x=15;
        }

        printf("\n 2. Thread [%d] value of x is %d \n",tid,x);
        x=x+1;

        printf("\n 3. Thread [%d] value of x is %d \n",tid,x);
    }
    return 0;
}
```

```
x value ouside parallel:10
1. Thread [0] value of x is 10
2. Thread [0] value of x is 15
3. Thread [0] value of x is 16
1. Thread [3] value of x is 10
2. Thread [3] value of x is 16
3. Thread [3] value of x is 17
1. Thread [1] value of x is 10
2. Thread [1] value of x is 17
3. Thread [1] value of x is 18
1. Thread [2] value of x is 10
2. Thread [2] value of x is 18
3. Thread [2] value of x is 19
```

x is shared. Assignment statement is done only by master.

2. OpenMP Programming: Directives - master

The master construct specifies a structured block that is executed by the master thread of the team.

```
#pragma omp master new-line
```

Structured block

- A master region binds to the innermost enclosing parallel region.
- Only the master thread executes the structured block
- There is no implied barrier on entry or exit, for master construct. So other threads need not fork or join.

```
#pragma omp master
{
    Structured block
}
```

Consider a program for adding sum of elements in an array.

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
int main (void) {
int tid,p,a[50],sum[20],dsum,finalsum;
int i, low, high;
int n=20;

//initialise
for(i=0;i<20;i++)
{
    a[i]=i;
    dsum=dsum+i;
}
printf("\n 1. dsum=%d \n",dsum);

#pragma omp parallel num_threads(4) default(shared) private(tid,low,high,i)
{
    p=omp_get_num_threads();
    int tid=omp_get_thread_num();
    //assign the iterations to threads
    low=n*tid/p;
    high=n*(tid+1)/p;
    printf("\n 2. Thread [%d] low=%d high=%d \n",tid,low,high);

    //find partial sum
    sum[tid]=0;
    for(i=low;i<high;i++)
        sum[tid]=sum[tid]+a[i];

    printf("3: partial sum [%d] = %d \n",tid,sum[tid]);
}

} //close parallel
```

```
finalsum=0;
//add all partial sum
for(i=0;i<p;i++)
    finalsum=finalsum+sum[i];

printf("\n 4. finalsum= %d \n",finalsum);

return 0;
}
```

The parallel region assigns iterations to each thread .

Partial sum is calculated in each thread

Partial sum is stored in an array

Master thread computes final sum

Consider a program for adding sum of elements in an array.

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
int main (void) {
int tid,p,a[50],sum[20],dsum,finalsum;
int i, low, high;
int n=20;

//initialise
for(i=0;i<20;i++)
{
    a[i]=i;
    dsum=dsum+i;
}
printf("\n 1. dsum=%d \n",dsum);

#pragma omp parallel num_threads(4) default(shared) private(tid,low,high,i)
{
    p=omp_get_num_threads();
    int tid=omp_get_thread_num();
    //assign the iterations to threads
    low=n*tid/p;
    high=n*(tid+1)/p;
    printf("\n 2. Thread [%d] low=%d high=%d \n",tid,low,high);

    //find partial sum
    sum[tid]=0;
    for(i=low;i<high;i++)
        sum[tid]=sum[tid]+a[i];

    printf("3: partial sum [%d] = %d \n",tid,sum[tid]);
}

//close parallel
```

```
finalsum=0;
//add all partial sum
for(i=0;i<p;i++)
    finalsum=finalsum+sum[i];

printf("\n 4. finalsum= %d \n",finalsum);

return 0;
}
```

```
1. dsum=190

2. Thread [2] low=10 high=15
2. Thread [3] low=15 high=20
3: partial sum [3] = 85
3: partial sum [2] = 60

2. Thread [1] low=5 high=10
3: partial sum [1] = 35

2. Thread [0] low=0 high=5
3: partial sum [0] = 10

4. finalsum= 190
```

```
-----  
Process exited after 0.05116 seconds
```

2. OpenMP Programming: Clauses

```
#pragma omp parallel [clause[,]clause...]
new-line
Structured-block
Clause: if(scalar-expression)
        num_threads(integer-expression)
        default(shared/none)
        private(list)
        firstprivate(list)
        shared(list)
        copyin(list)
        reduction(operator:list)
```

- Reduction(operator: list)
- The **reduction** clause specifies an operator and one or more list items.
- For each list item, a private copy is created on each thread, and is initialized appropriately for the operator.
- After the end of the region, the original list item is updated with the values of the private copies using the specified operator.
- Initialization value depend on data type of the **reduction** variable.

2. OpenMP Programming: Clauses

```
#pragma omp parallel [clause[,]clause...]
```

new-line

Structured-block

Clause: if(*scalar-expression*)

 num_threads(*integer-expression*)

 default(*shared/none*)

 private(*list*)

 firstprivate(*list*)

 shared(*list*)

 copyin(*list*)

reduction(operator:*list*)

- Reduction(operator: list)
- Initialization value depend on data type of the reduction variable.

Operator	Initialization value
+	0
*	1
-	0
&	~0
	0
^	0
&&	1
	0

Consider a program for adding sum of elements in an array: with reduction

```
#include <stdlib.h>
#include <omp.h>
int main (void) {
int tid,p,a[50],sum=0,dsum,finalsum;
int i, low, high;
int n=20;

//initialise
for(i=0;i<20;i++)
{
    a[i]=i;
    dsum=dsum+i;
}
printf("\n 1. dsum=%d \n",dsum);

#pragma omp parallel num_threads(4) default(shared) private(tid,low,high,i) reduction(+:sum)
{
    p=omp_get_num_threads();
    int tid=omp_get_thread_num();
    //assign the iterations to threads
    low=n*tid/p;
    high=n*(tid+1)/p;
    printf("\n 2. Thread [%d] low=%d high=%d \n",tid,low,high);

    //find partial sum
    for(i=low;i<high;i++)
        sum=sum+a[i];
} //close parallel

printf("\n 4. finalsum= %d \n",sum);
return 0;
}
```

1. dsum=190
2. Thread [2] low=10 high=15
2. Thread [3] low=15 high=20
- 3: partial sum [3] = 85
- 3: partial sum [2] = 60
2. Thread [1] low=5 high=10
- 3: partial sum [1] = 35
2. Thread [0] low=0 high=5
- 3: partial sum [0] = 10
4. finalsum= 190

2. OpenMP Programming: Clauses

```
#pragma omp parallel [clause[,]clause...]  
new-line
```

Structured-block

Clause: **if**(*scalar-expression*)

num_threads(*integer-expression*)

default(*shared/none*)

private(*list*)

firstprivate(*list*)

shared(*list*)

copyin(*list*)

reduction(*operator:list*)

- Reduction(operator: list)
- Used for some form of recurrence calculations
- The type of a list item that appears in a reduction clause must be valid for the **reduction** operator.
- Aggregate types(including arrays), pointers types and reference types may not appear in a **reduction** clause
- A variable must appear in a **reduction** clause must not be const-qualified
- The operator specified in a **reduction** clause cannot be overloaded with respect to the variables that appear in that clause.

2. OpenMP Programming: Clauses

#pragma omp for [clause[,]clause...] new-line

for-loops

Clause: **private(list)**

firstprivate(list)

lastprivate(list)

reduction(operator:list)

schedule(kind[,chunk_size])

collapse(n)

ordered

nowait

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
int main (void) {
    int n=20, dsum=0, tid,i,a[20],sum=0;

    for(i=0;i<n;i++)
    {
        a[i]=i;
        dsum=dsum+i;
    }
    printf("dsum=%d\n",dsum);
#pragma omp parallel num_threads(4)
    {
        int tid=omp_get_thread_num();
        #pragma omp for private(i) schedule(static, 5) reduction(+:sum)
        for(i=0;i<n;i++)
            sum=sum+a[i];
    }
    printf("\n sum= %d \n",sum);
return 0;
}
```

2. OpenMP Programming: Clauses

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
int main (void) {
int n=20, dsum=0, tid,i,a[20],sum=0;

for(i=0;i<n;i++)
{
a[i]=i;
dsum=dsum+i;
}
printf("dsum=%d\n",dsum);
#pragma omp parallel num_threads(4)
{
int tid=omp_get_thread_num();
#pragma omp for private(i) schedule(static, 5) reduction(+:sum)
for(i=0;i<n;i++)
sum=sum+a[i];
}
printf("\n sum= %d \n",sum);
return 0;
}
```

```
dsum=190
sum= 190
-----
Process exited after 0.02466 seconds
Press any key to continue
```

2. OpenMP Programming: Clauses

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
int main (void) {
int n=10, dsum=0, tid,i,a[10],sum=0, count=0;

for(i=0;i<n;i++)
{
a[i]=i;
}

#pragma omp parallel num_threads(2) default(shared)
{
int tid=omp_get_thread_num();
count=0;
#pragma omp for schedule(static, 5) private(count)
for(i=0;i<n;i++)
{
a[i]=a[i]+5;
if(a[i]%2==0) count++;
printf("tid[%d] a[%d]=%d count %d\n",tid,i,a[i],count);
}
}
return 0;
}
```

```
tid[0] a[0]=5 count 4199876
tid[0] a[1]=6 count 4199877
tid[0] a[2]=7 count 4199877
tid[0] a[3]=8 count 4199878
tid[0] a[4]=9 count 4199878
tid[1] a[5]=10 count 1790492427
tid[1] a[6]=11 count 1790492427
tid[1] a[7]=12 count 1790492428
tid[1] a[8]=13 count 1790492428
tid[1] a[9]=14 count 1790492429
```

2. OpenMP Programming: Clauses

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
int main (void) {
int n=10, dsum=0, tid,i,a[10],sum=0, count=0;

for(i=0;i<n;i++)
{
a[i]=i;
}
#pragma omp parallel num_threads(2) default(shared)
{
int tid=omp_get_thread_num();
count=0;
#pragma omp for schedule(static, 5) firstprivate(count)
for(i=0;i<n;i++)
{
a[i]=a[i]+5;
if(a[i]%2==0) count++;
printf("tid[%d] a[%d]=%d count %d\n",tid,i,a[i],count);
}
}
return 0;
}
```

```
tid[0] a[0]=5 count 0
tid[0] a[1]=6 count 1
tid[0] a[2]=7 count 1
tid[0] a[3]=8 count 2
tid[0] a[4]=9 count 2
tid[1] a[5]=10 count 1
tid[1] a[6]=11 count 1
tid[1] a[7]=12 count 2
tid[1] a[8]=13 count 2
tid[1] a[9]=14 count 3
```

2. OpenMP Programming: Clauses

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
int main (void) {
int i;
int x;
x=100;
printf("X value before parallel region %d\n",x);
#pragma omp parallel for num_threads(2) private(x)
for(i=0;i<=10;i++){
x=x+i;
printf("Thread number: %d      x: %d\n",omp_get_thread_num(),x);
}
printf("x is %d\n", x);

return 0;
}
```

```
X value before parallel region 100
Thread number: 0      x: 8
Thread number: 0      x: 9
Thread number: 1      x: 2067342
Thread number: 1      x: 2067349
Thread number: 1      x: 2067357
Thread number: 1      x: 2067366
Thread number: 1      x: 2067376
Thread number: 0      x: 11
Thread number: 0      x: 14
Thread number: 0      x: 18
Thread number: 0      x: 23
x is 100
```

2. OpenMP Programming: Clauses

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
int main (void) {
int i;
int x;
x=100;
printf("X value before parallel region %d\n",x);
#pragma omp parallel for num_threads(2) firstprivate(x)
for(i=0;i<=10;i++){
    x=x+i;
    printf("Thread number: %d      x: %d\n",omp_get_thread_num(),x);
}
printf("x is %d\n", x);

return 0;
}
```

```
X value before parallel region 100
Thread number: 0      x: 100
Thread number: 0      x: 101
Thread number: 0      x: 103
Thread number: 0      x: 106
Thread number: 0      x: 110
Thread number: 0      x: 115
Thread number: 1      x: 106
Thread number: 1      x: 113
Thread number: 1      x: 121
Thread number: 1      x: 130
Thread number: 1      x: 140
x is 100
```

2. OpenMP Programming: Clauses

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
int main (void) {
int i;
int x;
x=100;
printf("X value before parallel region %d\n",x);
#pragma omp parallel for num_threads(2) firstprivate(x) lastprivate(x)
for(i=0;i<=10;i++){
    x=x+i;
    printf("Thread number: %d      x: %d\n",omp_get_thread_num(),x);
}
printf("x is %d\n", x);

return 0;
}
```

```
X value before parallel region 100
Thread number: 0      x: 100
Thread number: 1      x: 106
Thread number: 1      x: 113
Thread number: 0      x: 101
Thread number: 1      x: 121
Thread number: 1      x: 130
Thread number: 1      x: 140
Thread number: 0      x: 103
Thread number: 0      x: 106
Thread number: 0      x: 110
Thread number: 0      x: 115
x is 140
```

Index

- OpenMP
 - Program Structure
 - Directives : master
 - Clauses
 - Reduction
 - Lastprivate
- References

Reference

Text Books and/or Reference Books:

1. Professional CUDA C Programming – John Cheng, Max Grossman, Ty McKercher, 2014
2. B.Wilkinson, M.Allen, "Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers", Pearson Education, 1999
3. I.Foster, "Designing and building parallel programs", 2003
4. Parallel Programming in C using OpenMP and MPI – Micheal J Quinn, 2004
5. Introduction to Parallel Programming – Peter S Pacheco, Morgan Kaufmann Publishers, 2011
6. Advanced Computer Architectures: A design approach, Dezso Sima, Terence Fountain, Peter Kacsuk, 2002
7. Parallel Computer Architecture : A hardware/Software Approach, David E Culler, Jaswinder Pal Singh Anoop Gupta, 2011
8. Introduction to Parallel Computing, Ananth Grama, Anshul Gupta, George Karypis, Vipin Kumar, Pearson, 2011

Reference

Acknowledgements

1. Introduction to OpenMP <https://www3.nd.edu/~zxu2/acms60212-40212/Lec-12-OpenMP.pdf>
2. Introduction to parallel programming for shared memory
Machines <https://www.youtube.com/watch?v=LL3TAHpxOig>
3. OpenMP Application Program Interface Version 2.5 May 2005
4. OpenMP Application Program Interface Version 5.0 November 2018

Thank You

National Institute of Technology Karnataka Surathkal
Department of Information Technology



IT 301 Parallel Computing
Shared Memory Programming Technique (4)
OpenMP : *for -schedule*

Dr. Geetha V
Assistant Professor
Dept of Information Technology
NITK Surathkal

Index

- OpenMP
 - Directives : if, for
 - Clauses
 - Schedule
 - Static
 - Dynamic
 - Guided
 - Runtime
- References

Course Outline

Course Plan: Theory:

Part A: Parallel Computer Architectures

Week 1,2,3: ***Introduction to Parallel Computer Architecture:*** Parallel Computing, Parallel architecture, bit level, instruction level , data level and task level parallelism. Instruction level parallelism: pipelining(Data and control instructions), scalar and superscalar processors, vector processors. Parallel computers and computation.

Week 4,5: Memory Models: UMA, NUMA and COMA. Flynn's classification, Cache coherence,

Week 6,7: Amdahl's Law. Performance evaluation, Designing parallel algorithms : Divide and conquer, Load balancing, Pipelining.

Week 8 -11: ***Parallel Programming techniques like Task Parallelism using TBB, TL2, Cilk++ etc. and software transactional memory techniques.***

Course Outline

Part B: OpenMP/MPI/CUDA

Week 1,2,3 : **Shared Memory Programming Techniques:** Introduction to OpenMP : Directives: *parallel, for, sections, task, master, single, critical, barrier, taskwait, atomic.* Clauses: *private, shared, firstprivate, lastprivate, reduction, nowait, ordered, schedule, collapse, num_threads, if()*.

Week 4,5: **Distributed Memory programming Techniques:** MPI: Blocking, Non-blocking.

Week 6,7 : CUDA : OpenCL, Execution models, GPU memory, GPU libraries.

Week 10,11,: **Introduction to accelerator programming using CUDA/OpenCL and Xeon-phi. Concepts of Heterogeneous programming techniques.**

Practical:

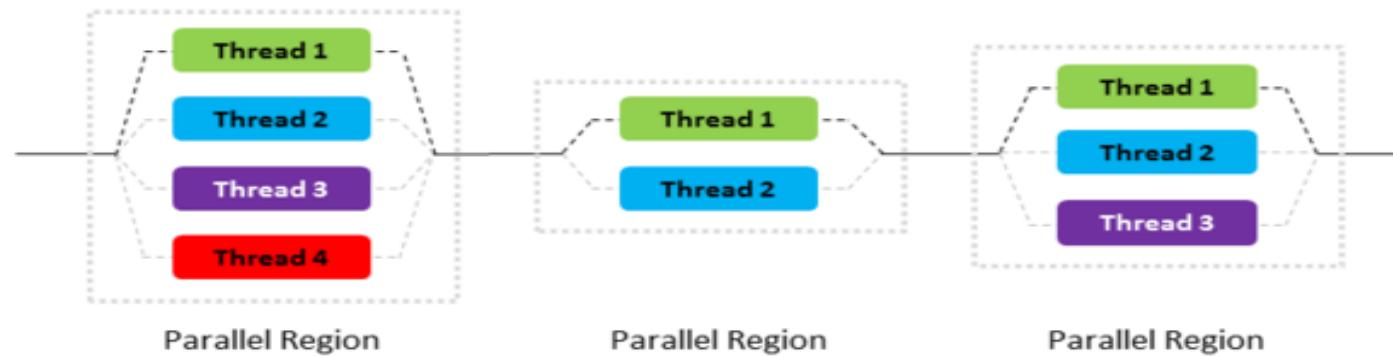
Implementation of parallel programs using OpenMP/MPI/CUDA.

Assignment: Performance evaluation of parallel algorithms (in group of 2 or 3 members)

1. OpenMP

FORK – JOIN Parallelism

- OpenMP program begin as a single process: the master thread. The master thread executes sequentially until the first parallel region construct is encountered.
- When a parallel region is encountered, master thread
 - Create a group of threads by FORK.
 - Becomes the master of this group of threads and is assigned the thread id 0 within the group.
- The statement in the program that are enclosed by the parallel region construct are then executed in parallel among these threads.
- JOIN: When the threads complete executing the statement in the parallel region construct, they synchronize and terminate, leaving only the master thread.



2. OpenMP Programming: Directives : Parallel, For

```
#pragma omp parallel [clause[,]clause...]new-line
Structured-block
Clause: if(scalar-expression)
        num_threads(integer-expression)
        default(shared/none)
        private(list)
        firstprivate(list)
        shared(list)
        copyin(list)
        reduction(operator:list)
```

```
#pragma omp for [clause[,]clause...]new-line
for-loops
Clause: private(list)
        firstprivate(list)
        lastprivate(list)
        reduction(operator:list)
        schedule(kind[/chunk_size])
        collapse(n)
        ordered
        nowait
```

2. OpenMP Programming: Clauses : Schedule

```
#pragma omp for [clause[,]clause...] new-line  
for-loops  
Clause: private(list)  
        firstprivate(list)  
        lastprivate(list)  
        reduction(operator:list)  
        schedule(kind[,chunk_size])  
        collapse(n)  
        ordered  
        nowait
```

Schedule(**kind**[,**chunksizes**]) Clause

- Schedule clause specifies how iteration of the loop are divided into contiguous non-empty subsets, called chunks, and how these chunks are assigned among threads of the team.
- Kind: It has following kind.
 - Static
 - Dynamic
 - Guided
 - runtime

2. OpenMP Programming: *schedule(static, chunk_size)*

```
#pragma omp for [clause[,]clause...] new-line  
for-loops  
Clause: private(list)  
        firstprivate(list)  
        lastprivate(list)  
        reduction(operator:list)  
        schedule(kind[,chunk_size])  
        collapse(n)  
        ordered  
        nowait
```

Schedule(static, chunkszie) Clause

- Iterations are divided into chunk of size chunk_size.
- Chunks are statically assigned to threads in round robin fashion in the order of thread number
- Last chunk to be assigned may have smaller number of iterations.
- When no chunk size is specified, iterations/threads
- Example: 28 iteration, threads= 4
- Schedule(static, 5)

thread0	thread1	thread2	Thread3	thread0	thread1
0-4	5-9	10-14	15-19	20-24	25-27

2. OpenMP Programming: *schedule(dynamic, chunk_size)*

```
#pragma omp for [clause[,]clause...] new-line  
for-loops  
Clause: private(list)  
        firstprivate(list)  
        lastprivate(list)  
        reduction(operator:list)  
        schedule(kind[,chunk_size])  
        collapse(n)  
        ordered  
        nowait
```

Schedule(Dynamic, chunkszie) Clause

- Iterations are assigned to threads in chunkszie as the threads request them.
- Thread executes the chunk of iteration and then requests another chunk, until all iterations are complete.
- Each chunk contains chunkszie except for the last chunk assigned.
- Example: 28 iteration, threads= 4
- Schedule(dynamic, 5)

thread1	thread3	thread0	thread2	thread1	thread2
0-4	5-9	10-14	15-19	20-24	25-27

2. OpenMP Programming: *schedule(guided, chunk_size)*

Schedule(Guided, chunkszie)] Clause

- Iterations are assigned to threads of chunkszie as the threads request them.
- Thread executes the chunk of iteration and then requests another chunk, until all iterations are complete.
- Chunk = remaining iterations / #threads
- Chunk size determines the minimum size of chunk , except 1st chunk.
- Default value of chunk_size =1

- Chunk = remaining iterations / #threads
- Example: 28 iteration, threads= 4
- Schedule(guided,3)
- $28/4 = 7$ [remaining = $28-7=21$]
- $21/4=5.2 \Rightarrow 6$ [remaining = $21-6 = 15$]
- $15/4=3.7\Rightarrow 4$ [remaining = $15-4 = 11$]
- $11/4=2.7 \Rightarrow 3$ [remaining = $11-3=8$]
- $8/4 = 2$ [min is chunk size 3 . So assign 3:]
 - [remaining = $8-3 = 5$]
- $5/4 = 1$ [min = 3: remaining : 2]
- $2<=3$, so last chunk = 2

thread2	thread1	thread0	thread3	thread2	thread2	thread2
0-6 (7)	7-12(6)	13-16(4)	17-19(3)	20-22 (3)	23-25(3)	26-27(2)

2. OpenMP Programming: *schedule(runtime)*

```
#pragma omp for [clause[,]clause...] new-line  
for-loops  
Clause: private(list)  
        firstprivate(list)  
        lastprivate(list)  
        reduction(operator:list)  
        schedule(kind[,chunk_size])  
        collapse(n)  
        ordered  
        nowait
```

Schedule(runtime) Clause

- The decision regarding scheduling is deferred until run time, and the schedule and chunk size are taken from the *run-sched-var* control variable.

2. OpenMP Programming: *schedule(static, chunk_size)*

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
int main (void) {
int i;
#pragma omp parallel for num_threads(4) schedule(static,5)
for(i=0;i<28;i++){
printf("Thread number: %d : %d\n",omp_get_thread_num(),i);
}
return 0;
}
```

thread0	thread1	thread2	Thread3	thread0	thread1
0-4	5-9	10-14	15-19	20-24	25-27

```
Thread number: 1 : 5
Thread number: 1 : 6
Thread number: 1 : 7
Thread number: 1 : 8
Thread number: 1 : 9
Thread number: 1 : 25
Thread number: 1 : 26
Thread number: 1 : 27
Thread number: 0 : 0
Thread number: 0 : 1
Thread number: 0 : 2
Thread number: 0 : 3
Thread number: 0 : 4
Thread number: 0 : 20
Thread number: 0 : 21
Thread number: 0 : 22
Thread number: 0 : 23
Thread number: 0 : 24
Thread number: 2 : 10
Thread number: 2 : 11
Thread number: 2 : 12
Thread number: 2 : 13
Thread number: 2 : 14
Thread number: 3 : 15
Thread number: 3 : 16
Thread number: 3 : 17
Thread number: 3 : 18
Thread number: 3 : 19
```

2. OpenMP Programming: *schedule(dynamic, chunk_size)*

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
int main (void) {
int i;
#pragma omp parallel for num_threads(4) schedule(dynamic,5)
for(i=0;i<28;i++){
printf("Thread number: %d : %d\n",omp_get_thread_num(),i);
}
return 0;
}
```

thread2	thread1	thread0	thread3	thread1	thread3
0-4	5-9	10-14	15-19	20-24	25-27

```
Thread number: 2 : 0
Thread number: 1 : 5
Thread number: 1 : 6
Thread number: 1 : 7
Thread number: 1 : 8
Thread number: 1 : 9
Thread number: 3 : 15
Thread number: 3 : 16
Thread number: 3 : 17
Thread number: 3 : 18
Thread number: 3 : 19
Thread number: 3 : 25
Thread number: 3 : 26
Thread number: 3 : 27
Thread number: 0 : 10
Thread number: 0 : 11
Thread number: 0 : 12
Thread number: 0 : 13
Thread number: 0 : 14
Thread number: 1 : 20
Thread number: 1 : 21
Thread number: 1 : 22
Thread number: 1 : 23
Thread number: 1 : 24
Thread number: 2 : 1
Thread number: 2 : 2
Thread number: 2 : 3
Thread number: 2 : 4
```

2. OpenMP Programming: *schedule(guided, chunk_size)*

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
int main (void) {
int i;
#pragma omp parallel for num_threads(4) schedule(guided,3)
for(i=0;i<28;i++){
printf("Thread number: %d : %d\n",omp_get_thread_num(),i);
}
return 0;
}
```

thread2	thread1	thread0	thread3	thread2	thread2	thread2
0-6 (7)	7-12(6)	13-16(4)	17-19(3)	20-22 (3)	23-25(3)	26-27(2)

```
Thread number: 2 : 0
Thread number: 2 : 1
Thread number: 2 : 2
Thread number: 2 : 3
Thread number: 2 : 4
Thread number: 2 : 5
Thread number: 2 : 6
Thread number: 2 : 20
Thread number: 2 : 21
Thread number: 2 : 22
Thread number: 2 : 23
Thread number: 2 : 24
Thread number: 2 : 25
Thread number: 2 : 26
Thread number: 2 : 27
Thread number: 3 : 17
Thread number: 3 : 18
Thread number: 3 : 19
Thread number: 1 : 7
Thread number: 1 : 8
Thread number: 1 : 9
Thread number: 1 : 10
Thread number: 1 : 11
Thread number: 1 : 12
Thread number: 0 : 13
Thread number: 0 : 14
Thread number: 0 : 15
Thread number: 0 : 16
```

2. OpenMP Programming: *schedule(runtime)*

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
int main (void) {
int i;
#pragma omp parallel for num_threads(4) schedule(runtime)
for(i=0;i<28;i++){
printf("Thread number: %d : %d\n",omp_get_thread_num(),i);
}
return 0;
}
```

thread1	thread0	thread3	Thread2	thread0
0	2	3	4-6	7-27

```
Thread number: 2 : 1
Thread number: 2 : 4
Thread number: 2 : 5
Thread number: 0 : 2
Thread number: 0 : 7
Thread number: 0 : 8
Thread number: 0 : 9
Thread number: 0 : 10
Thread number: 0 : 11
Thread number: 0 : 12
Thread number: 0 : 13
Thread number: 0 : 14
Thread number: 0 : 15
Thread number: 0 : 16
Thread number: 0 : 17
Thread number: 0 : 18
Thread number: 0 : 19
Thread number: 0 : 20
Thread number: 0 : 21
Thread number: 0 : 22
Thread number: 0 : 23
Thread number: 0 : 24
Thread number: 0 : 25
Thread number: 0 : 26
Thread number: 2 : 6
Thread number: 3 : 3
Thread number: 1 : 0
Thread number: 0 : 27
```

2. OpenMP Programming: *schedule(runtime)*

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
int main (void) {
int i;
#pragma omp parallel for num_threads(4) schedule(runtime)
for(i=0;i<28;i++){
printf("Thread number: %d : %d\n",omp_get_thread_num(),i);
}
return 0;
}
```

Thread	number:	1	:	0
Thread	number:	1	:	4
Thread	number:	1	:	5
Thread	number:	1	:	6
Thread	number:	1	:	7
Thread	number:	1	:	8
Thread	number:	1	:	9
Thread	number:	1	:	10
Thread	number:	1	:	11
Thread	number:	1	:	12
Thread	number:	1	:	13
Thread	number:	1	:	14
Thread	number:	1	:	15
Thread	number:	2	:	1
Thread	number:	1	:	16
Thread	number:	0	:	2
Thread	number:	0	:	19
Thread	number:	0	:	20
Thread	number:	0	:	21
Thread	number:	0	:	22
Thread	number:	0	:	23
Thread	number:	2	:	17
Thread	number:	2	:	25
Thread	number:	2	:	26
Thread	number:	1	:	18
Thread	number:	0	:	24
Thread	number:	2	:	27
Thread	number:	3	:	3

Index

- OpenMP
 - Directives : if, for
 - Clauses
 - Schedule
 - Static
 - Dynamic
 - Guided
 - Runtime
- References

Reference

Text Books and/or Reference Books:

1. Professional CUDA C Programming – John Cheng, Max Grossman, Ty McKercher, 2014
2. B.Wilkinson, M.Allen, "Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers", Pearson Education, 1999
3. I.Foster, "Designing and building parallel programs", 2003
4. Parallel Programming in C using OpenMP and MPI – Micheal J Quinn, 2004
5. Introduction to Parallel Programming – Peter S Pacheco, Morgan Kaufmann Publishers, 2011
6. Advanced Computer Architectures: A design approach, Dezso Sima, Terence Fountain, Peter Kacsuk, 2002
7. Parallel Computer Architecture : A hardware/Software Approach, David E Culler, Jaswinder Pal Singh Anoop Gupta, 2011
8. Introduction to Parallel Computing, Ananth Grama, Anshul Gupta, George Karypis, Vipin Kumar, Pearson, 2011

Reference

Acknowledgements

1. Introduction to OpenMP <https://www3.nd.edu/~zxu2/acms60212-40212/Lec-12-OpenMP.pdf>
2. Introduction to parallel programming for shared memory
Machines <https://www.youtube.com/watch?v=LL3TAHpxOig>
3. OpenMP Application Program Interface Version 2.5 May 2005
4. OpenMP Application Program Interface Version 5.0 November 2018

Thank You

National Institute of Technology Karnataka Surathkal

Department of Information Technology



IT 301 Parallel Computing

Shared Memory Programming Technique (5)
OpenMP : *Synchronization and other clauses*

Dr. Geetha V

Assistant Professor

Dept of Information Technology

NITK Surathkal

Index

- OpenMP
 - Directives : if, for
 - Clauses
 - Master
 - Single
 - Barrier
 - Atomic
 - critical
 - Nowait
 - Ordered
- References

Course Outline

Course Plan: Theory:

Part A: Parallel Computer Architectures

Week 1,2,3: ***Introduction to Parallel Computer Architecture:*** Parallel Computing, Parallel architecture, bit level, instruction level , data level and task level parallelism. Instruction level parallelism: pipelining(Data and control instructions), scalar and superscalar processors, vector processors. Parallel computers and computation.

Week 4,5: Memory Models: UMA, NUMA and COMA. Flynn's classification, Cache coherence,

Week 6,7: Amdahl's Law. Performance evaluation, Designing parallel algorithms : Divide and conquer, Load balancing, Pipelining.

Week 8 -11: ***Parallel Programming techniques like Task Parallelism using TBB, TL2, Cilk++ etc. and software transactional memory techniques.***

Course Outline

Part B: OpenMP/MPI/CUDA

Week 1,2,3 : **Shared Memory Programming Techniques:** Introduction to OpenMP : Directives: *parallel, for, sections, task, master, single, critical, barrier, taskwait, atomic.* Clauses: *private, shared, firstprivate, lastprivate, reduction, nowait, ordered, schedule, collapse, num_threads, if(), threadprivate, copyin, copyprivate*

Week 4,5: **Distributed Memory programming Techniques:** MPI: Blocking, Non-blocking.

Week 6,7 : CUDA : OpenCL, Execution models, GPU memory, GPU libraries.

Week 10,11,: **Introduction to accelerator programming using CUDA/OpenCL and Xeon-phi. Concepts of Heterogeneous programming techniques.**

Practical:

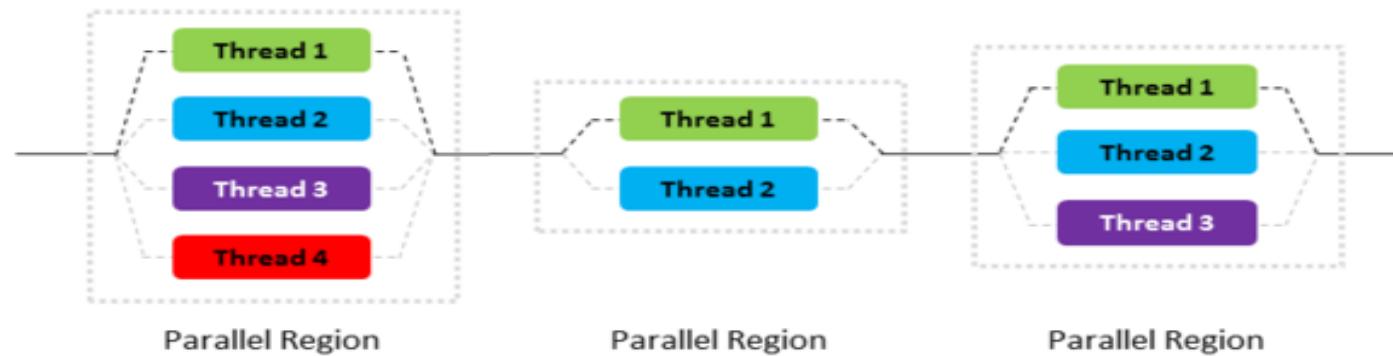
Implementation of parallel programs using OpenMP/MPI/CUDA.

Assignment: Performance evaluation of parallel algorithms (in group of 2 or 3 members)

1. OpenMP

FORK – JOIN Parallelism

- OpenMP program begin as a single process: the master thread. The master thread executes sequentially until the first parallel region construct is encountered.
- When a parallel region is encountered, master thread
 - Create a group of threads by FORK.
 - Becomes the master of this group of threads and is assigned the thread id 0 within the group.
- The statement in the program that are enclosed by the parallel region construct are then executed in parallel among these threads.
- JOIN: When the threads complete executing the statement in the parallel region construct, they synchronize and terminate, leaving only the master thread.



2. OpenMP Programming: Directives : Parallel, For

```
#pragma omp parallel [clause[,]clause...]new-line
```

Structured-block

Clause: **if**(*scalar-expression*)

num_threads(*integer-expression*)

default(*shared/none*)

private(*list*)

firstprivate(*list*)

shared(*list*)

copyin(*list*)

reduction(*operator:list*)

```
#pragma omp for [clause[,]clause...]new-line
```

for-loops

Clause: **private**(*list*)

firstprivate(*list*)

lastprivate(*list*)

reduction(*operator:list*)

schedule(*kind[,chunk_size]*)

collapse(*n*)

ordered

nowait

2. OpenMP Programming: Clauses : master

```
#pragma omp master  
    Structured block
```

Master :

- It specifies a structured block that is executed by the **master thread** of the team
- **Other threads** in the team **do not execute** the associated structured block.
- There is no implied barrier either on entry to, or exit from, the master construct.

2. OpenMP Programming: Clauses : single

```
#pragma omp single [clause[,] clause...]
```

Structured block

Clause:

Private(list)

Firstprivate(list)

Copyprivate(list)

nowait

Single :

- It specifies that the associated structured block is executed by only one thread in the team (not necessarily the master thread).
- The other threads in the team do not execute the block, and wait at an implicit barrier at the end of single construct, unless nowait clause is specified
- *nowait* : The other threads need not wait for synchronization point.
- *Copyprivate* clause must not be used with the *nowait* clause

2. OpenMP Programming: Clauses : critical

critical

- The critical construct restricts execution of the associated structured block to a **single thread at a time.**
- An **optional name** may be used to identify the critical construct. All critical constructs without a name are considered to have the **same unspecified name.**
- A thread waits at the beginning of a critical region until no other thread is executing a critical region with the same name.
- The critical construct enforces **exclusive access** with respect to all critical constructs with the same name in all threads, not just in the current team.

```
#pragma omp critical [(name)] new-line  
Structured block
```

2. OpenMP Programming: Construct: barrier

barrier

- The barrier construct specifies an **explicit barrier** at the point at which the construct appears.
- The barrier directive may only be placed in the program at a position where ignoring or deleting the directive would result in a program with correct syntax.
- All of the threads of the team executing the binding parallel region must **execute the barrier region** before any are allowed to continue execution beyond the barrier.
- Each barrier region must be encountered by all threads in a team or by none at all.
- The sequence of work-sharing regions and barrier regions encountered must be the same for every thread in a team.

```
#pragma omp barrier new-line
```

2. OpenMP Programming: Construct: atomic **atomic**

- The atomic construct ensures that a specific storage location is updated atomically, rather than exposing it to the possibility of multiple, simultaneous writing threads.
- Expression-stmt is an expression statement with one of the following forms:
 - $x \text{ binop}=\text{expr}$
 - $x++$
 - $++x$
 - $x--$
 - $--x$
- Binop is not an overloaded operator and is one of $+, *, -, /, \&, ^, |, <<$, or $>>$.
- Only the load and store of the object designated by x are atomic; the evaluation of expr is not atomic.

```
#pragma omp atomic new-line  
Expression-stmt
```

2. OpenMP Programming: Construct: flush

flush

```
#pragma omp flush [(list)] new-line
```

- The flush construct executes the OpenMP flush operation.
- This operation makes a thread's temporary view of memory consistent with memory and enforces an order on the memory operations of the variables explicitly specified or implied.
- The flush construct with a list applies the flush operation to the items in the list, and does not return until the operation is complete for all specified list items.
- A flush construct without a list, executed on a given thread, operates as if the whole thread-visible data state of the program, as defined by the base language, is flushed.
- If a pointer is present in the list, the pointer itself is flushed, not the object to which the pointer refers.

2. OpenMP Programming: Construct: flush

flush

A flush region without a list is implied at the following locations:

- During a barrier region
- At entry to and exit from parallel, critical and ordered regions.
- At exit from work-sharing regions, unless a *nowait* is present.
- At entry to and exit from combined parallel work-sharing regions
- During set and unset lock, test, nest lock etc.

A flush region with a list is implied at the following locations:

At entry and exit from atomic regions, where the list contains only the object updated in the atomic construct.

```
#pragma omp flush [(list)] new-line
```

2. OpenMP Programming: Construct: ordered

Ordered

The ordered construct specifies a structured block in a loop region which will be executed in **the order of the loop iterations.**

```
#pragma omp ordered new-line  
Structured block
```

This sequentializes and orders the code within an ordered region while allowing code outside the region to run in parallel.

When a thread executing other than first iteration, encounters an ordered region, it waits at the beginning of that ordered region until each of the previous iterations that contains an ordered region has completed the ordered region.

2. OpenMP Programming: Construct: ordered

Ordered

Restrictions to the ordered construct:

- The loop region to which an ordered region binds must have an ordered clause specified on the corresponding loop (or parallel loop) construct.
- During execution of an iteration of a loop within a loop region, the executing thread must not execute more than one ordered region which binds to the same loop region.

```
#pragma omp ordered new-line  
Structured block
```

2. OpenMP Programming: Clause: nowait

nowait

nowait

If there are multiple independent loops within a parallel region, then *nowait* can be used to avoid the implied barrier at the end of the loop construct.

3. OpenMP Programming: Examples

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
int main (void) {
int x=10, tid;
printf("x value ouside parallel:%d\n",x);
#pragma omp parallel num_threads(4) shared(x) private(tid)
{
    int tid=omp_get_thread_num();
    #pragma omp master
    {
        x=15;
        printf("\n 1. Thread [%d] value of x is %d \n",tid,x);
    }
    x=x+1;
    printf("\n 2. Thread [%d] value of x is %d \n",tid,x);
}
return 0;
}
```

```
x value ouside parallel:10
2. Thread [2] value of x is 11
2. Thread [3] value of x is 16
1. Thread [0] value of x is 15
2. Thread [0] value of x is 17
2. Thread [1] value of x is 12
```

3. OpenMP Programming: Examples

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
int main (void) {
int x=10, tid;
printf("x value ouside parallel:%d\n",x);
#pragma omp parallel num_threads(4) shared(x) private(tid)
{
    int tid=omp_get_thread_num();
#pragma omp single
{
    x=15;
    printf("\n 1. Thread [%d] value of x is %d \n",tid,x);
}
x=x+1;

    printf("\n 2. Thread [%d] value of x is %d \n",tid,x);
}
return 0;
}
```

```
x value ouside parallel:10
1. Thread [1] value of x is 15
2. Thread [0] value of x is 16
2. Thread [2] value of x is 17
2. Thread [3] value of x is 18
2. Thread [1] value of x is 19
```

```
x value ouside parallel:10
1. Thread [2] value of x is 15
2. Thread [1] value of x is 16
2. Thread [0] value of x is 17
2. Thread [3] value of x is 18
2. Thread [2] value of x is 19
```

3. OpenMP Programming: Examples

```
int main (void) {
int a[5], i;
#pragma omp parallel
{
    #pragma omp for
    for (i = 0; i < 5; i++)
        a[i] = i * i;

    #pragma omp master
    {
        printf("tid of master %d\n",omp_get_thread_num());
        for (i = 0; i < 5; i++)
            printf("a[%d] = %d\n", i, a[i]);
    }

    // Wait.
    //#pragma omp barrier
    printf("tid %d \n",omp_get_thread_num());
    // Continue with the computation.
    #pragma omp for
    for (i=0;i<5;i++)
        a[i]+=i;

    #pragma omp single
    {
        printf("tid of single %d\n",omp_get_thread_num());
        for (i = 0; i < 5; i++)
            printf("a[%d] = %d\n", i, a[i]);
    }
}

return 0;
}
```

```
tid 2
tid 1
tid of master 0
a[0] = 0
a[1] = 1
a[2] = 6
a[3] = 12
a[4] = 16
tid 0
tid 3
tid of single 2
a[0] = 0
a[1] = 2
a[2] = 6
a[3] = 12
a[4] = 20
```

3. OpenMP Programming: Examples

```
int main (void) {
int a[5], i;
#pragma omp parallel
{
    #pragma omp for
    for (i = 0; i < 5; i++)
        a[i] = i * i;

    #pragma omp master
    {
        printf("tid of master %d\n",omp_get_thread_num());
        for (i = 0; i < 5; i++)
            printf("a[%d] = %d\n", i, a[i]);
    }

    // Wait.
    #pragma omp barrier
    printf("tid %d \n",omp_get_thread_num());
    // Continue with the computation.
    #pragma omp for
    for (i=0;i<5;i++)
        a[i]+=i;

    #pragma omp single
    {
        printf("tid of single %d\n",omp_get_thread_num());
        for (i = 0; i < 5; i++)
            printf("a[%d] = %d\n", i, a[i]);
    }
}

return 0;
}
```

```
tid of master 0
a[0] = 0
a[1] = 1
a[2] = 4
a[3] = 9
a[4] = 16
tid 1
tid 3
tid 2
tid 0
tid of single 1
a[0] = 0
a[1] = 2
a[2] = 6
a[3] = 12
a[4] = 20
```

3. OpenMP Programming: Examples

```
#include<stdio.h>
#include<omp.h>
int main(void)
{
    int i,n,a[50],b[50],sum;
    double t1,t2;
    printf("Enter the value of n");
    scanf("%d",&n);
    t1=omp_get_wtime();
    #pragma omp parallel num_threads(4)
    {
        int id=omp_get_thread_num();
        #pragma omp for ordered reduction(+:sum)
        for(i=0;i<n;i++)
        {
            printf("Thread %d: value of i : %d\n",id,i);
            sum=sum+i;
            #pragma omp ordered
            {
                b[i]=i+1;
                printf("b[%d] value is %d in ORDER\n",i,b[i]);
            }
        }
    }
    t2=omp_get_wtime();
    printf("Time taken is %f",t2-t1);
    return 0;
}
```

```
Enter the value of n5
Thread 3: value of i : 4
Thread 1: value of i : 2
Thread 2: value of i : 3
Thread 0: value of i : 0
b[0] value is 1 in ORDER
Thread 0: value of i : 1
b[1] value is 2 in ORDER
b[2] value is 3 in ORDER
b[3] value is 4 in ORDER
b[4] value is 5 in ORDER
Time taken is 0.001000
```

Index

- OpenMP
 - Directives : if, for
 - Clauses
 - Master
 - Single
 - Barrier
 - Atomic
 - critical
 - Nowait
 - Ordered
- References

Reference

Text Books and/or Reference Books:

1. Professional CUDA C Programming – John Cheng, Max Grossman, Ty McKercher, 2014
2. B.Wilkinson, M.Allen, "Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers", Pearson Education, 1999
3. I.Foster, "Designing and building parallel programs", 2003
4. Parallel Programming in C using OpenMP and MPI – Micheal J Quinn, 2004
5. Introduction to Parallel Programming – Peter S Pacheco, Morgan Kaufmann Publishers, 2011
6. Advanced Computer Architectures: A design approach, Dezso Sima, Terence Fountain, Peter Kacsuk, 2002
7. Parallel Computer Architecture : A hardware/Software Approach, David E Culler, Jaswinder Pal Singh Anoop Gupta, 2011
8. Introduction to Parallel Computing, Ananth Grama, Anshul Gupta, George Karypis, Vipin Kumar, Pearson, 2011

Reference

Acknowledgements

1. Introduction to OpenMP <https://www3.nd.edu/~zxu2/acms60212-40212/Lec-12-OpenMP.pdf>
2. Introduction to parallel programming for shared memory
Machines <https://www.youtube.com/watch?v=LL3TAHpxOig>
3. OpenMP Application Program Interface Version 2.5 May 2005
4. OpenMP Application Program Interface Version 5.0 November 2018

Thank You

National Institute of Technology Karnataka Surathkal

Department of Information Technology



IT 301 Parallel Computing

Shared Memory Programming Technique (6)

OpenMP : *sections, threadprivate, collapse*

Dr. Geetha V

Assistant Professor

Dept of Information Technology

NITK Surathkal

Index

- OpenMP
 - Directives : if, for, master, single, Barrier, atomic, critical,
 - Directives
 - Sections
 - Clauses
 - Threadprivate
 - Collapse
 - Threadwait
 - Copyin, copyprivate
- References

Course Outline

Course Plan: Theory:

Part A: Parallel Computer Architectures

Week 1,2,3: ***Introduction to Parallel Computer Architecture:*** Parallel Computing, Parallel architecture, bit level, instruction level , data level and task level parallelism. Instruction level parallelism: pipelining(Data and control instructions), scalar and superscalar processors, vector processors. Parallel computers and computation.

Week 4,5: Memory Models: UMA, NUMA and COMA. Flynn's classification, Cache coherence,

Week 6,7: Amdahl's Law. Performance evaluation, Designing parallel algorithms : Divide and conquer, Load balancing, Pipelining.

Week 8 -11: ***Parallel Programming techniques like Task Parallelism using TBB, TL2, Cilk++ etc. and software transactional memory techniques.***

Course Outline

Part B: OpenMP/MPI/CUDA

Week 1,2,3 : **Shared Memory Programming Techniques:** Introduction to OpenMP : Directives: *parallel, for, sections, task, master, single, critical, barrier, taskwait, atomic.* Clauses: *private, shared, firstprivate, lastprivate, reduction, nowait, ordered, schedule, collapse, num_threads, if(), threadprivate, copyin, copyprivate*

Week 4,5: **Distributed Memory programming Techniques:** MPI: Blocking, Non-blocking.

Week 6,7 : CUDA : OpenCL, Execution models, GPU memory, GPU libraries.

Week 10,11,: **Introduction to accelerator programming using CUDA/OpenCL and Xeon-phi. Concepts of Heterogeneous programming techniques.**

Practical:

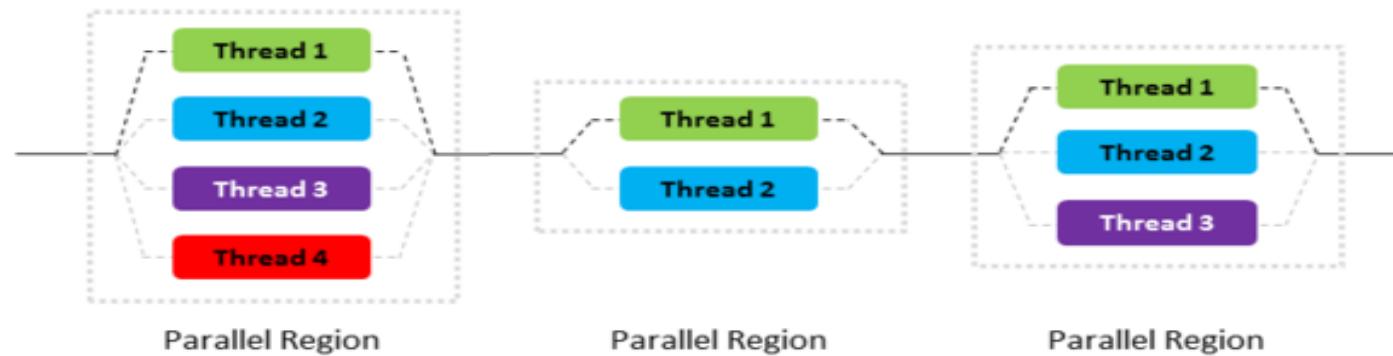
Implementation of parallel programs using OpenMP/MPI/CUDA.

Assignment: Performance evaluation of parallel algorithms (in group of 2 or 3 members)

1. OpenMP

FORK – JOIN Parallelism

- OpenMP program begin as a single process: the master thread. The master thread executes sequentially until the first parallel region construct is encountered.
- When a parallel region is encountered, master thread
 - Create a group of threads by FORK.
 - Becomes the master of this group of threads and is assigned the thread id 0 within the group.
- The statement in the program that are enclosed by the parallel region construct are then executed in parallel among these threads.
- JOIN: When the threads complete executing the statement in the parallel region construct, they synchronize and terminate, leaving only the master thread.



2. OpenMP Programming: Worksharing

Work sharing constructs

- Loop constructs
- Section construct
- Single construct

2. OpenMP Programming: Section

```
#pragma omp sections [clause,...]
{
    [#pragma omp section new-line
        Structured block]
    [#pragma omp section new-line
        Structured-block]
```

Clauses

Private(list)
Firstprivate(list)
Lastprivate(list)
Reduction(operator:list)
nowait

Section :

- It is a non-iterative work-sharing construct that contains a set of structured blocks that are to be divided among, and executed by, the threads in a team.
- Each structured block is executed by one of the threads in the team.
- There is an implicit barrier at the end of sections construct, unless a *nowait* clause is specified.
- Only a single *nowait* clause can appear on a sections directive.

2. OpenMP Programming: Collapse

```
#pragma omp for schedule(static,n)
collapse(2)

for(i=0; i<imax; i++) {
    for(j=0; j<jmax;j++)
        a[i][j] = b[i][j] + c[i][j]
}
```

Collapse :

- It increases the total number of iterations that will be partitioned across the available number of OMP threads by reducing the granularity of work to be done by each thread.
- If the amount of work to be done by each thread is non-trivial (after collapsing is applied), this may improve the parallel scalability of the OMP applications.

2. OpenMP Programming: **threadprivate**

Threadprivate

- Each thread is allowed to have its own temporary view of the shared memory.
- Each thread also has access to another type of memory that must not be accessed by other threads , called threadprivate memory
- **Shared variable:** each thread refers to the original variable.
- **Private variable:** Current thread's private version of the original variable.
- **Threadprivate:** variable appearing in threadprivae directives are threadprivate.

```
#pragma omp threadprivate(list)
```

2. OpenMP Programming: **threadprivate**

Threadprivate

- It specifies that named global-lifetime objects are replicated, with each thread having its own copy.
- Each copy of the *threadprivate* object is initialized once.
- A thread may not reference another thread's copy of a *threadprivate* object.
- A *threadprivate* object must not appear in any clause except the *copyin*, *copyprivate*, *schedule*, *num_threads*, and *if* clauses.
- The list is a comma-separated list of file-scope, names-scope, or static block-scope variables that do not have incomplete types.

```
#pragma omp threadprivate(list)
```

2. OpenMP Programming: **threadprivate**

Private vs Threadprivate

- Private variable scope is defined for only specific parallel region. *Threadprivate* variable scope is declared across the parallel regions.
- *Firstprivate()* is used to copy the values from original variable. *Copyin()* is used to copy the values while entering into parallel region first time.
- Private variables are stored on stack most of the time. *Threadprivate* variables are stored in heap or thread local storage.

```
#pragma omp threadprivate(list)
```

2. OpenMP Programming: *threadprivate*

Data Copying clauses

- These clauses support the copying of data values from private or *threadprivate* variables on one implicit task or thread to the corresponding variables on other implicit tasks or threads in the team.
- **Copyin(list)**: Copies the value of the master thread's *threadprivate* variable to the *threadprivate* variable of each other member of the team executing the parallel region.
- **Copyprivate (list)** : Broadcasts a value from the data environment of one implicit task to the data environments of the other implicit tasks belonging to the parallel region.

```
#pragma omp threadprivate(list)  
  
Copyin(list)  
Copyprivate(list)
```

Index

- OpenMP
 - Directives : if, for, master, single, Barrier, atomic, critical,
 - Directives
 - Sections
 - Clauses
 - Threadprivate
 - Collapse
 - Threadwait
 - Copyin, copyprivate
- References

Reference

Text Books and/or Reference Books:

1. Professional CUDA C Programming – John Cheng, Max Grossman, Ty McKercher, 2014
2. B.Wilkinson, M.Allen, "Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers", Pearson Education, 1999
3. I.Foster, "Designing and building parallel programs", 2003
4. Parallel Programming in C using OpenMP and MPI – Micheal J Quinn, 2004
5. Introduction to Parallel Programming – Peter S Pacheco, Morgan Kaufmann Publishers, 2011
6. Advanced Computer Architectures: A design approach, Dezso Sima, Terence Fountain, Peter Kacsuk, 2002
7. Parallel Computer Architecture : A hardware/Software Approach, David E Culler, Jaswinder Pal Singh Anoop Gupta, 2011
8. Introduction to Parallel Computing, Ananth Grama, Anshul Gupta, George Karypis, Vipin Kumar, Pearson, 2011

Reference

Acknowledgements

1. Introduction to OpenMP <https://www3.nd.edu/~zxu2/acms60212-40212/Lec-12-OpenMP.pdf>
2. Introduction to parallel programming for shared memory
Machines <https://www.youtube.com/watch?v=LL3TAHpxOig>
3. OpenMP Application Program Interface Version 2.5 May 2005
4. OpenMP Application Program Interface Version 5.0 November 2018

Thank You

National Institute of Technology Karnataka Surathkal
Department of Information Technology



IT 301 Parallel Computing
Shared Memory Programming Technique (7)
OpenMP : *Task, taskwait*

Dr. Geetha V
Assistant Professor
Dept of Information Technology
NITK Surathkal

Index

- OpenMP
 - Directives : Task
- References

Course Outline

Course Plan: Theory:

Part A: Parallel Computer Architectures

Week 1,2,3: ***Introduction to Parallel Computer Architecture:*** Parallel Computing, Parallel architecture, bit level, instruction level , data level and task level parallelism. Instruction level parallelism: pipelining(Data and control instructions), scalar and superscalar processors, vector processors. Parallel computers and computation.

Week 4,5: Memory Models: UMA, NUMA and COMA. Flynn's classification, Cache coherence,

Week 6,7: Amdahl's Law. Performance evaluation, Designing parallel algorithms : Divide and conquer, Load balancing, Pipelining.

Week 8 -11: ***Parallel Programming techniques like Task Parallelism using TBB, TL2, Cilk++ etc. and software transactional memory techniques.***

Course Outline

Part B: OpenMP/MPI/CUDA

Week 1,2,3 : **Shared Memory Programming Techniques:** Introduction to OpenMP : Directives: *parallel, for, sections, task, master, single, critical, barrier, taskwait,* atomic. Clauses: *private, shared, firstprivate, lastprivate, reduction, nowait, ordered, schedule, collapse, num_threads, if(), threadprivate, copyin, copyprivate*

Week 4,5: **Distributed Memory programming Techniques:** MPI: Blocking, Non-blocking.

Week 6,7 : CUDA : OpenCL, Execution models, GPU memory, GPU libraries.

Week 10,11,: **Introduction to accelerator programming using CUDA/OpenCL and Xeon-phi. Concepts of Heterogeneous programming techniques.**

Practical:

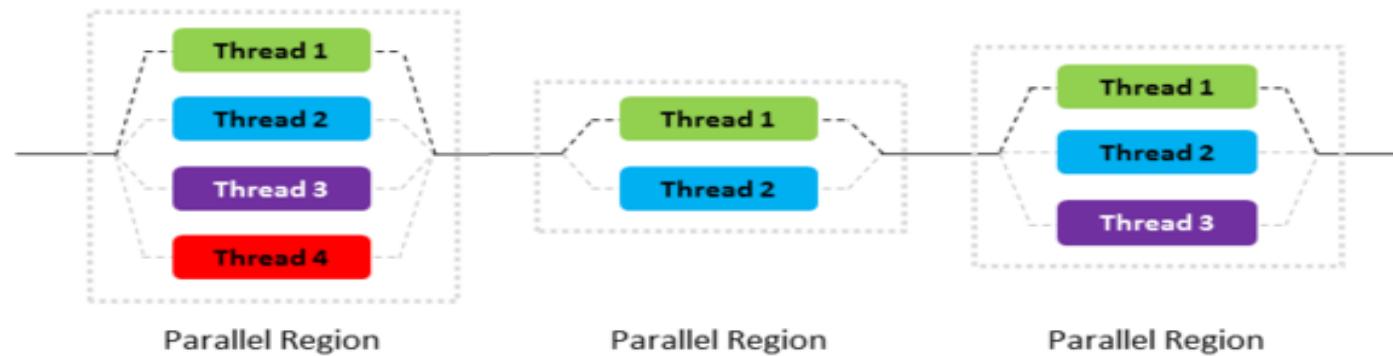
Implementation of parallel programs using OpenMP/MPI/CUDA.

Assignment: Performance evaluation of parallel algorithms (in group of 2 or 3 members)

1. OpenMP

FORK – JOIN Parallelism

- OpenMP program begin as a single process: the master thread. The master thread executes sequentially until the first parallel region construct is encountered.
- When a parallel region is encountered, master thread
 - Create a group of threads by FORK.
 - Becomes the master of this group of threads and is assigned the thread id 0 within the group.
- The statement in the program that are enclosed by the parallel region construct are then executed in parallel among these threads.
- JOIN: When the threads complete executing the statement in the parallel region construct, they synchronize and terminate, leaving only the master thread.



2. OpenMP Programming: Task

- Tasks are available starting with OpenMP 3.0
- A task is composed of
 - Code to be executed
 - Data environment (inputs to be used and outputs to be generated)
 - A Location where the task will be executed (thread)
- The tasks were initially implicit in OpenMP
 - A parallel construct constructs implicit tasks, one per thread
 - Teams of threads are created (or declared)
 - Threads in teams are assigned to each task
 - They synchronize with the master thread using barrier once all tasks completed
- Allowing the application to explicitly create tasks provide support for different execution models
 - More flexible
 - Requires scheduling of task
 - Moves away from the original fork/join model of OpenMP construct

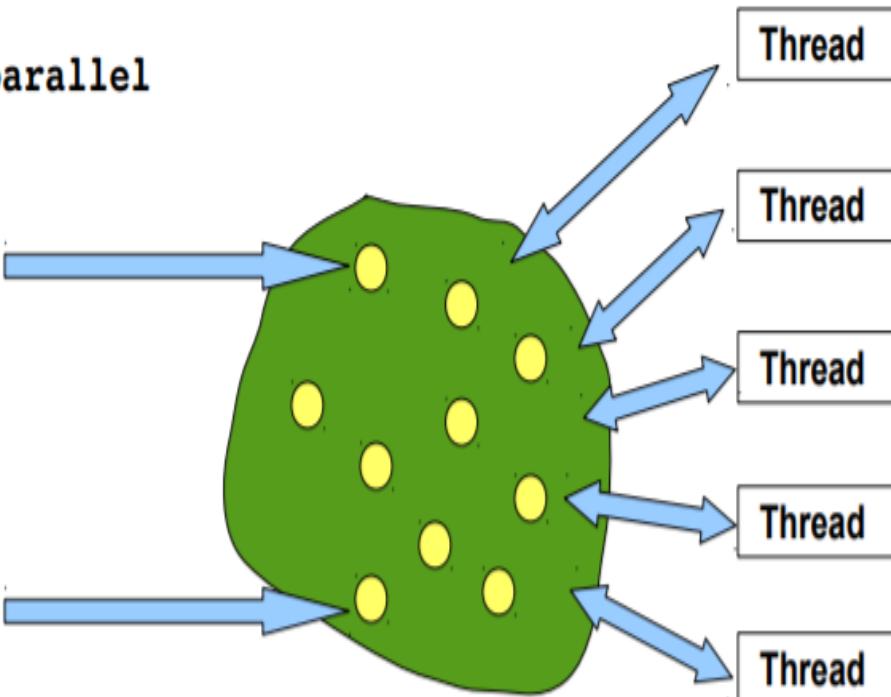
2. OpenMP Programming: Task

- Assumption here is that tasks are independent. Ex : tree traversal

```
#pragma omp parallel  
...  
{  
    [code]  
}  
...
```

```
{  
    [code]  
}
```

```
...
```



```
void preorder(node *p) {  
    process(p->data);  
    if (p->left)  
        #pragma omp task  
        preorder(p->left);  
    if (p->right)  
        #pragma omp task  
        preorder(p->right);  
}
```

2. OpenMP programming - Tasks

Task – Technical Notion

- When a thread encounters a task construct, it may choose to execute the task immediately or defer its execution until a later time. If deferred, the task is placed in a conceptual pool of tasks associated with the current parallel region. All team threads will take tasks out of the pool and execute them until the pool is empty. A thread that executes a task might be different from the thread that originally encountered it.
- The code associated with a task construct will be executed only once. A task is **tied** if the code is executed by the same thread from beginning to end. Otherwise, the task is **untied** (the code can be executed by more than one thread).

2. OpenMP programming - Tasks

Types of Task

- **Undeferred:** the execution is not deferred with respect to its generating task region, and the generating task region is suspended until execution of the **undeferred** task is completed (such as the tasks created with the if clause)
- **Included:** execution is sequentially included in the generating task region (such as a result from a final clause)
- **Subtle difference:** for **undeferred** task, the generating task region is suspended until execution of the **undeferred** task is completed, even if the **undeferred** task is not executed immediately.
 - The **undeferred** task may be placed in the conceptual pool and executed at a later time by the encountering thread or by some other thread; in the meantime, the generating task is suspended. Once the execution of the **undeferred** task is completed, the generating task can resume.
- A merged task is a task whose data environment is the same as that of its generating task region.

2. OpenMP Programming: Task

```
#pragma omp task[clause,...]
```

Structured-block

Clauses

depend(list)

if(expression)

final(expression)

untied

mergeable

default(shared | firstprivate | none)

private(list)

firstprivate(list)

shared(list)

priority(value)

2. OpenMP Programming: Task

```
#pragma omp task[clause,...]
```

Structured-block

Clauses

depend(list)

if(expression)

final(expression)

untied

mergeable

default(shared | firstprivate | none)

private(list)

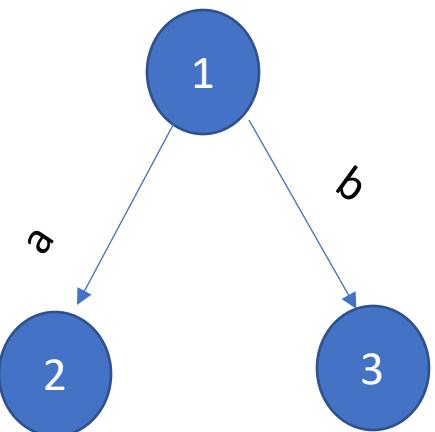
firstprivate(list)

shared(list)

priority(value)

Depend :

- It enforces additional constraints between tasks
- The list in depend contains storage locations (memory addresses) on which the dependency will be tracked
- The dependency is mentioned with **in, out,inout**



```
#pragma omp task shared(a,b) depend(out a,b)
task1(.....) //T1
#pragma omp shared(a) task depend(in a)
task2(.....) //T2
#pragma omp task shared(b) depend(in b)
task3(.....) //T3
T1 must be executed first. T2 and T3 can be
executed in parallel
```

2. OpenMP Programming: Task

```
#pragma omp task[clause,...]
```

Structured-block

Clauses

depend(list)

if(expression)

final(expression)

untied

mergeable

default(shared|firstprivate|none)

private(list)

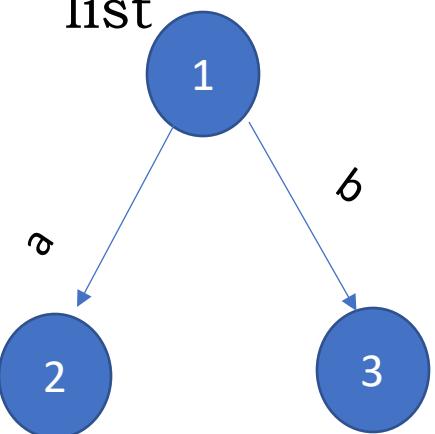
firstprivate(list)

shared(list)

priority(value)

Depend :

- In: The generated task will be dependent of all previously generated sibling tasks that reference at least one of the list items in an **out** or **inout** dependence type.
- Out and **inout**: The generated task will be dependent of all previously generated sibling tasks that reference at least one of the list items in an **in,out** or **inout** dependence type list



```
#pragma omp task shared(a,b) depend(out a,b)
task1(.....) //T1
#pragma omp shared(a) task depend(in a)
task2(.....) //T2
#pragma omp task shared(b) depend(in b)
task3(.....) //T3
T1 must be executed first. T2 and T3 can be
executed in parallel
```

2. OpenMP Programming: Task

```
#pragma omp task[clause,...]
```

Structured-block

Clauses

depend(list)

if(expression)

final(expression)

untied

mergeable

default(shared | firstprivate | none)

private(list)

firstprivate(list)

shared(list)

priority(value)

If (**expression**):

- When the if expression argument evaluate to false, an undefined task is generated, and the encountering thread must suspend the current task region, for which execution cannot be resumed until execution of the strucured block that is associated with the generated task is completed.
- The use of variable in an if clause expression of a task construct causes an implicit reference to the variable in all enclosing constructs

2. OpenMP Programming: Task

```
#pragma omp task[clause,...]
```

Structured-block

Clauses

depend(list)

if(expression)

final(expression)

untied

mergeable

default(shared|firstprivate|none)

private(list)

firstprivate(list)

shared(list)

priority(value)

If (expression):

```
#pragma omp task if(0) // This is undefined
{
    #pragma omp task // This is a regular task
        for( i = 0; i < 3; i++ ) {
            #pragma omp task // This is a regular task
                bar();
        }
}
```

2. OpenMP Programming: Task

```
#pragma omp task[clause,...]
```

Structured-block

Clauses

depend(list)

if(expression)

final(expression)

untied

mergeable

default(shared|firstprivate|none)

private(list)

firstprivate(list)

shared(list)

priority(value)

final (expression):

- When the final clause argument is true
 - The generated task will be a final task
 - All tasks encountered during execution of a final task will generate included tasks
 - An included task is a task for which execution is sequentially included in the generating task region; that is undefined and executed immediately by the encountering threads
- It is another user directed optimization
- Omp_in_final() returns true if the enclosing task region is final. Otherwise, it returns false.

2. OpenMP Programming: Task

```
#pragma omp task[clause,...]
```

Structured-block

Clauses

depend(list)

if(expression)

final(expression)

untied

mergeable

default(shared|firstprivate|none)

private(list)

firstprivate(list)

shared(list)

priority(value)

final (expression):

```
#pragma omp task final(1) // This is a regular task
{
    #pragma omp task // This is included
    for(i=0;i<3;i++) {
        #pragma omp task // This is also included
        bar();
    }
}
```

2. OpenMP Programming: Task

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
int main(void)
{
    printf("A ");
    printf("Race ");
    printf("Car ");
    printf("\n");
    return 0;
}
```

2. OpenMP Programming: Task

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
int main(void)
{
    printf("A ");
    printf("Race ");
    printf("Car ");
    printf("\n");
    return 0;
}
```

- The output of the program is A Race Car

2. OpenMP Programming: Task

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
int main(void)
{
#pragma omp parallel {
    printf("A ");
    printf("Race ");
    printf("Car ");
}
    printf("\n");
return 0;
}
```

- The output of the program is
A Race Car A Race Car A Race
Car
(Assume three threads)

2. OpenMP Programming: Collapse

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
int main(void)
{
#pragma omp parallel {
#pragma omp single{
    printf("A ");
    printf("Race ");
    printf("Car ");
} }
    printf("\n");
return 0;
}
```

- The output of the program is
A Race Car
(Assume three threads)

2. OpenMP Programming: Task

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
int main(void)
{
#pragma omp parallel {
#pragma omp single{
    printf("A ");
#pragma omp task
    { printf("Race "); }
#pragma omp task
    { printf("Car "); }
}
    printf("\n");
return 0;
}
```

- The output of the program is
A Race Car
Or
A Car Race
(Assume three threads)

2. OpenMP Programming: Task

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
int main(void)
{
#pragma omp parallel {
#pragma omp single{
    printf("A ");
#pragma omp task
    { printf("Race "); }
#pragma omp task
    { printf("Car "); }
printf("is fun to watch");
}
    printf("\n");
return 0;
```

- The output of the program is

A is fun to watch Race Car

Or

A is fun to watch Car Race

(Assume two threads)

2. OpenMP Programming: Task

```
int main(void)
{
#pragma omp parallel {
#pragma omp single{
    printf("A ");
#pragma omp task
    { printf("Race "); }
#pragma omp task
    { printf("Car "); }
#pragma omp taskwait
    printf("is fun to watch");
}
    printf("\n");
return 0;
}
```

- The output of the program is

A Race Car is fun to watch

Or

A Car Race is fun to watch

(Assume two threads)

2. OpenMP Programming: Task

```
#pragma omp task[clause,...]
```

Structured-block

Clauses

depend(list)

if(expression)

final(expression)

untied

mergeable

default(shared|firstprivate|none)

private(list)

firstprivate(list)

shared(list)

priority(value)

untied

- Different parts of the task can be executed by different threads. Implies the tasks will yield, allowing the executing thread to switch context and execute another task instead.
- If the task is tied, it is guaranteed that the same thread will execute all the parts of the task, even if the task execution has been temporarily suspended
- An **untied** task generator can be moved from thread to thread allowing the tasks to be generated by different entities

2. OpenMP Programming: Task

```
#pragma omp task[clause,...]
```

Structured-block

Clauses

depend(list)

if(expression)

final(expression)

untied

mergeable

default(shared|firstprivate|none)

private(list)

firstprivate(list)

shared(list)

priority(value)

meargeable

- A merged task is a task whose data environment is the same as that of its generating task region.
- When a mergeable clause is present on a task construct, then the implementation may choose to generate a merged task instead.
- If a merged task is generated, then the behavior is as though there was no task directive at all

2. OpenMP Programming: Task

```
#pragma omp task[clause,...]
```

Structured-block

Clauses

depend(list)

if(expression)

final(expression)

untied

mergeable

default(shared|firstprivate|none)

private(list)

firstprivate(list)

shared(list)

priority(value)

Default, private, firstprivate

- Default defines the data-sharing attributes of variables that are referenced
- **firstprivate:** each construct has a copy of the data item, and it is initialized from the upper construct before the call
- **shared:** All references to a list item within a task refer to the storage area of the original variable
- **private:** each task receive a new item

2. OpenMP Programming: Task

```
#pragma omp task[clause,...]
```

Structured-block

Clauses

depend(list)

if(expression)

final(expression)

untied

mergeable

default(shared | firstprivate | none)

private(list)

firstprivate(list)

shared(list)

priority(value)

Priority(n)

- Priority is a hint for the scheduler. A non-negative numerical value, that recommend a task with a high priority to be executed before a task with lower priority

Index

- OpenMP
 - Directives : Task
- References

Reference

Text Books and/or Reference Books:

1. Professional CUDA C Programming – John Cheng, Max Grossman, Ty McKercher, 2014
2. B.Wilkinson, M.Allen, "Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers", Pearson Education, 1999
3. I.Foster, "Designing and building parallel programs", 2003
4. Parallel Programming in C using OpenMP and MPI – Micheal J Quinn, 2004
5. Introduction to Parallel Programming – Peter S Pacheco, Morgan Kaufmann Publishers, 2011
6. Advanced Computer Architectures: A design approach, Dezso Sima, Terence Fountain, Peter Kacsuk, 2002
7. Parallel Computer Architecture : A hardware/Software Approach, David E Culler, Jaswinder Pal Singh Anoop Gupta, 2011
8. Introduction to Parallel Computing, Ananth Grama, Anshul Gupta, George Karypis, Vipin Kumar, Pearson, 2011

Reference

Acknowledgements

1. Introduction to OpenMP <https://www3.nd.edu/~zxu2/acms60212-40212/Lec-12-OpenMP.pdf>
2. Introduction to parallel programming for shared memory
Machines <https://www.youtube.com/watch?v=LL3TAHpxOig>
3. OpenMP Application Program Interface Version 2.5 May 2005
4. OpenMP Application Program Interface Version 5.0 November 2018
5. Introduction to openmp tasks
<http://www.icl.utk.edu/~luszek/teaching/courses/fall2016/cosc462/pdf/W45L1%20-%20OpenMP%20Tasks.pdf>

Thank You

National Institute of Technology Karnataka Surathkal
Department of Information Technology



IT 301 Parallel Computing
Memory Models, Flynn's Classification

Dr. Geetha V

Assistant Professor

Dept of Information Technology

NITK Surathkal

Course Outline

Course Plan: Theory:

Part A: Parallel Computer Architectures

Week 1,2,3: ***Introduction to Parallel Computer Architecture:*** Parallel Computing, Parallel architecture, bit level, instruction level , data level and task level parallelism. Instruction level parallelism: pipelining(Data and control instructions), scalar and superscalar processors, vector processors. Parallel computers and computation.

Week 4,5: Memory Models: UMA, NUMA and COMA. Flynn's classification, Cache coherence,

Week 6,7: Amdahl's Law. Performance evaluation, Designing parallel algorithms : Divide and conquer, Load balancing, Pipelining.

Week 8 - 11: ***Parallel Programming techniques like Task Parallelism using TBB, TL2, Cilk++ etc. and software transactional memory techniques.***

Course Outline

Part B: OpenMP/MPI/CUDA

Week 1,2,3 : ***Shared Memory Programming Techniques:*** Introduction to OpenMP : Directives: parallel, for, sections, task, single, critical, barrier, taskwait, atomic. Clauses: private, shared, firstprivate, lastprivate, reduction, nowait, ordered, schedule, collapse, num_threads, shared, if().

Week 4,5: ***Distributed Memory programming Techniques:*** MPI: Blocking, Non-blocking.

Week 6,7 : CUDA : OpenCL, Execution models, GPU memory, GPU libraries.

Week 10,11,: ***Introduction to accelerator programming using CUDA/OpenCL and Xeon-phi. Concepts of Heterogeneous programming techniques.***

Practical:

Implementation of parallel programs using OpenMP/MPI/CUDA.

Assignment: Performance evaluation of parallel algorithms (in group of 2 or 3 members)

Index

1. Flynn's classification
2. Memory models
3. Perspective on parallel programming

1. Flynn's Classification

- In 1966, M.J. Flynn proposed a classification for the organization of a computer system by the number of instructions and data items that are manipulated simultaneously.
- Flynn uses the stream concept for describing a machine's structure.
- The sequence of instructions read from memory constitutes an **instruction stream**.
- The operations performed on the data in the processor constitute a **data stream**.

Flynn's Classification

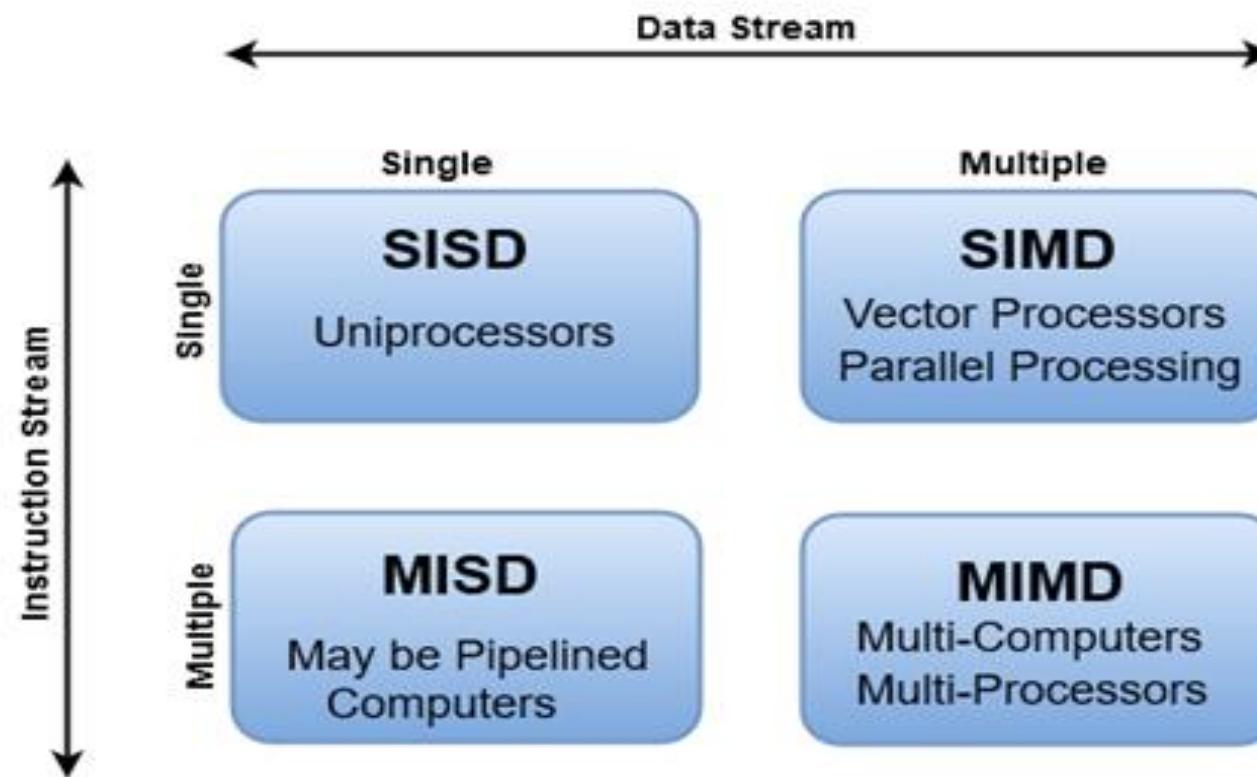
The classification of computer architectures based on the number of instruction streams and data streams (Flynn's Taxonomy)

Flynn's Classification divides computers into four major groups .

- Single instruction stream, single data stream (**SISD**)
- Single instruction stream, multiple data stream (**SIMD**)
- Multiple instruction stream, single data stream (**MISD**)
- Multiple instruction stream, multiple data stream (**MIMD**)

Flynn's Classification

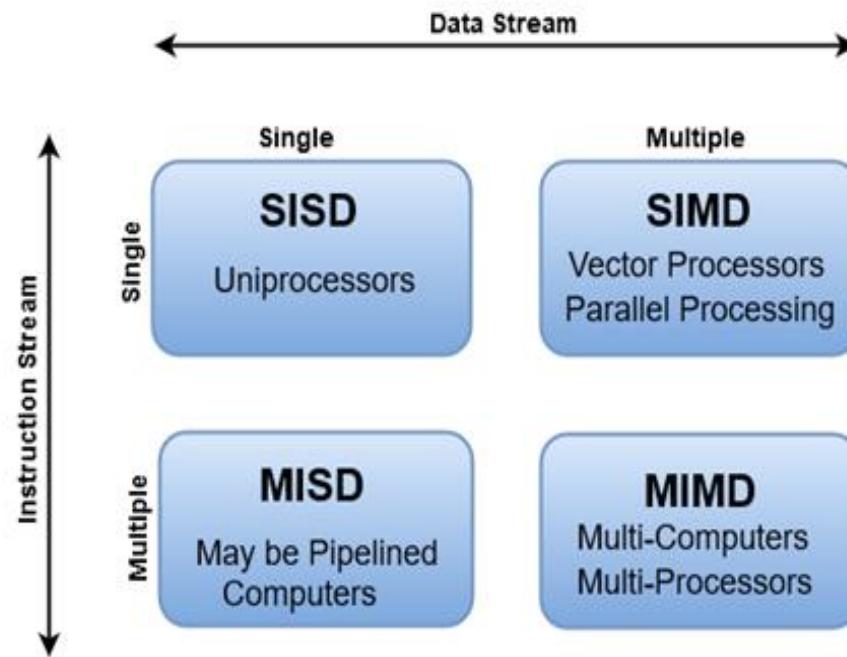
Flynn's Classification divides computers into four major groups .



Ref:<https://www.javatpoint.com/flynn-classification-of-computers>

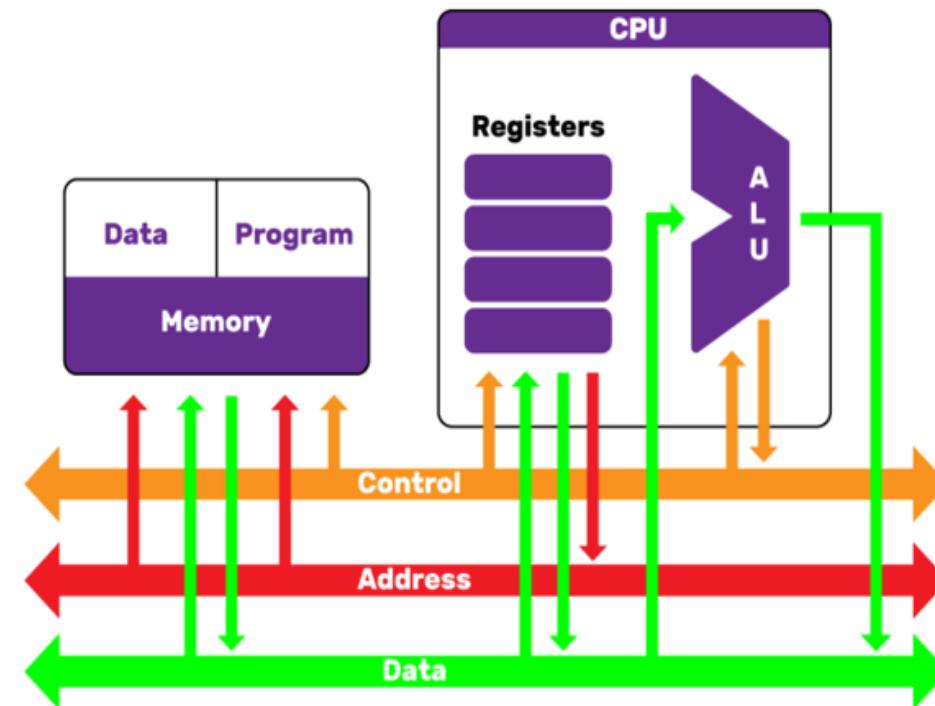
Flynn's Classification: SISD

Flynn's Classification



Ref:<https://www.javatpoint.com/flynn-classification-of-computers>

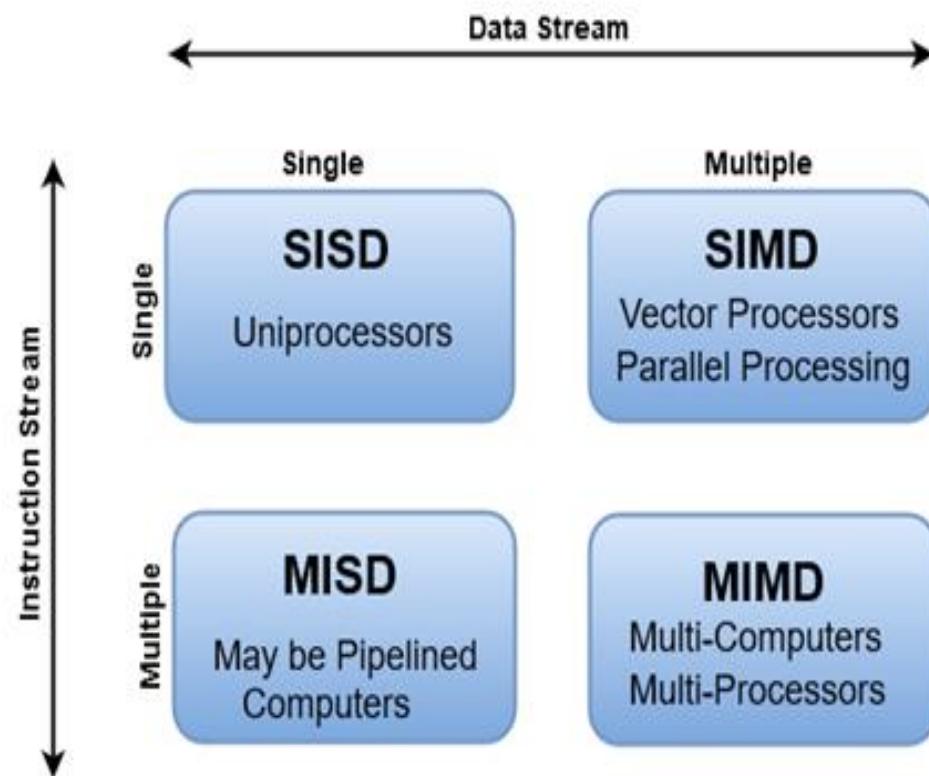
Example: Traditional Computers



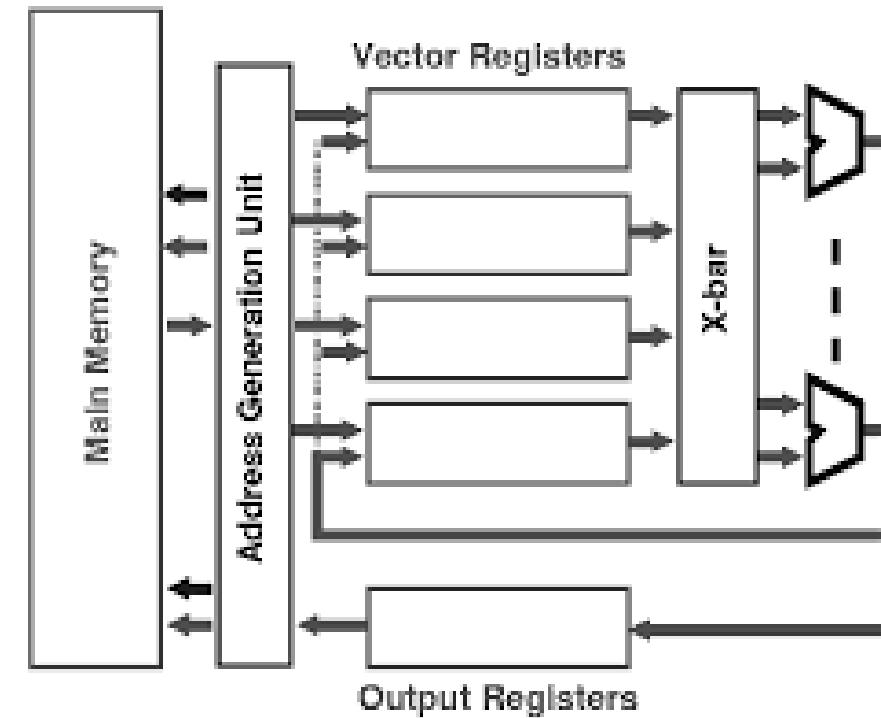
Ref:<https://www.futurelearn.com/courses/how-computers-work/0/steps/49283>

Flynn's Classification: SIMD

Flynn's Classification

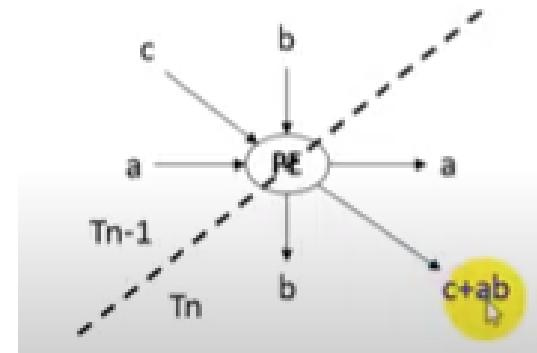
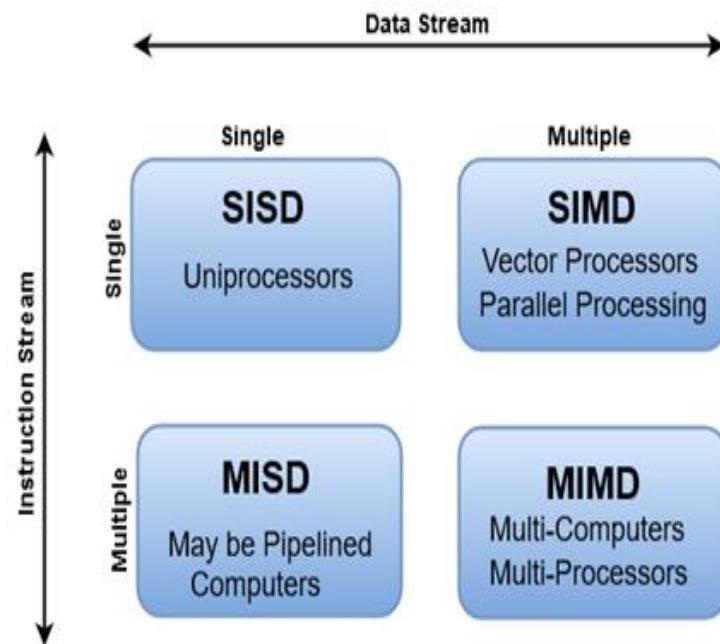


Example: Vector processor CRAY -I



Flynn's Classification: MISD

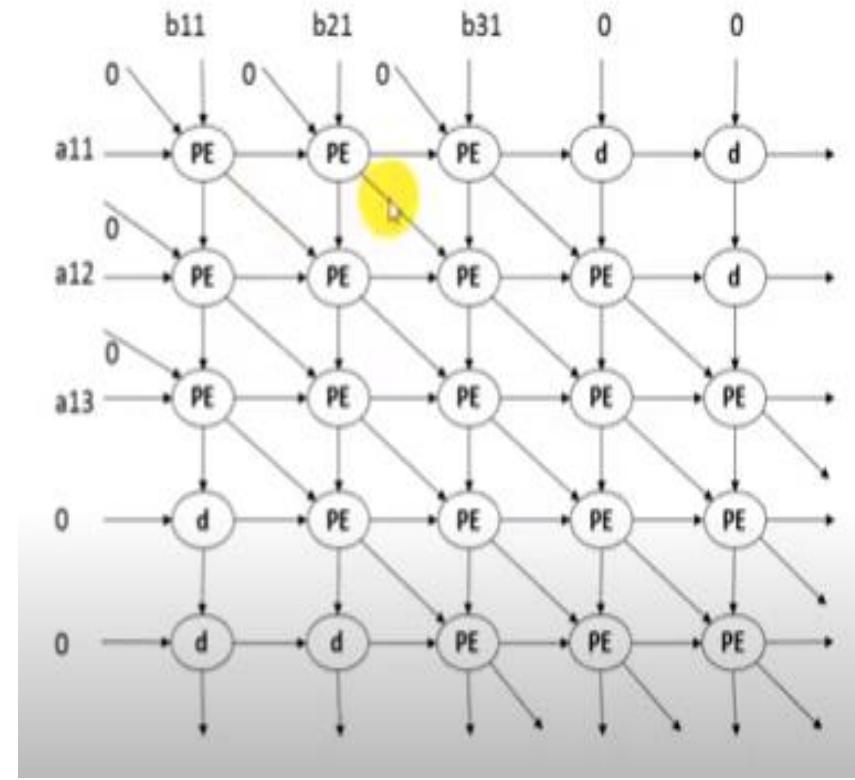
Flynn's Classification



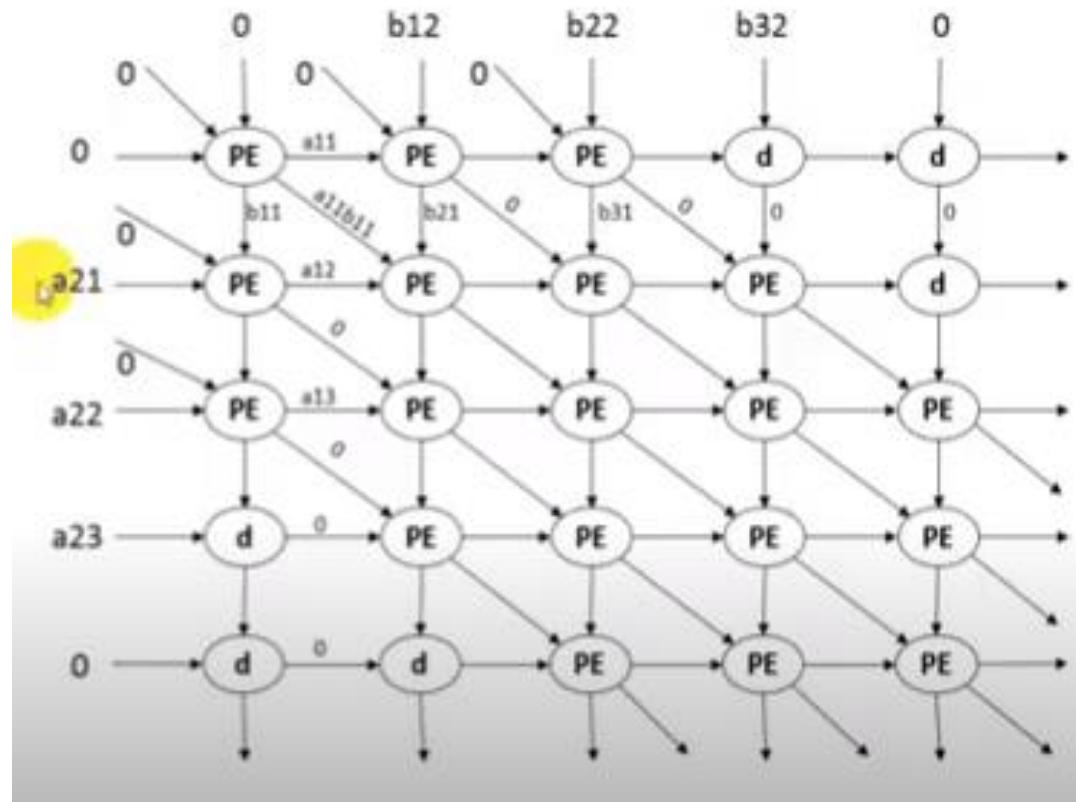
$$\begin{aligned}c_{11} &= c_{11} + a_{11}b_{11} + a_{12}b_{21} + a_{13}b_{31} \\c_{12} &= c_{12} + a_{11}b_{12} + a_{12}b_{22} + a_{13}b_{32} \\c_{13} &= c_{13} + a_{11}b_{13} + a_{12}b_{23} + a_{13}b_{33} \\c_{21} &= c_{21} + a_{21}b_{11} + a_{22}b_{21} + a_{23}b_{31} \\c_{22} &= c_{22} + a_{21}b_{12} + a_{22}b_{22} + a_{23}b_{32} \\c_{23} &= c_{23} + a_{21}b_{13} + a_{22}b_{23} + a_{23}b_{33} \\c_{31} &= c_{31} + a_{31}b_{11} + a_{32}b_{21} + a_{33}b_{31} \\c_{32} &= c_{31} + a_{31}b_{12} + a_{32}b_{22} + a_{33}b_{32} \\c_{33} &= c_{33} + a_{31}b_{13} + a_{32}b_{23} + a_{33}b_{33}\end{aligned}$$

Example: Systolic arrays : eg: 3 x 3 matrix multiplication

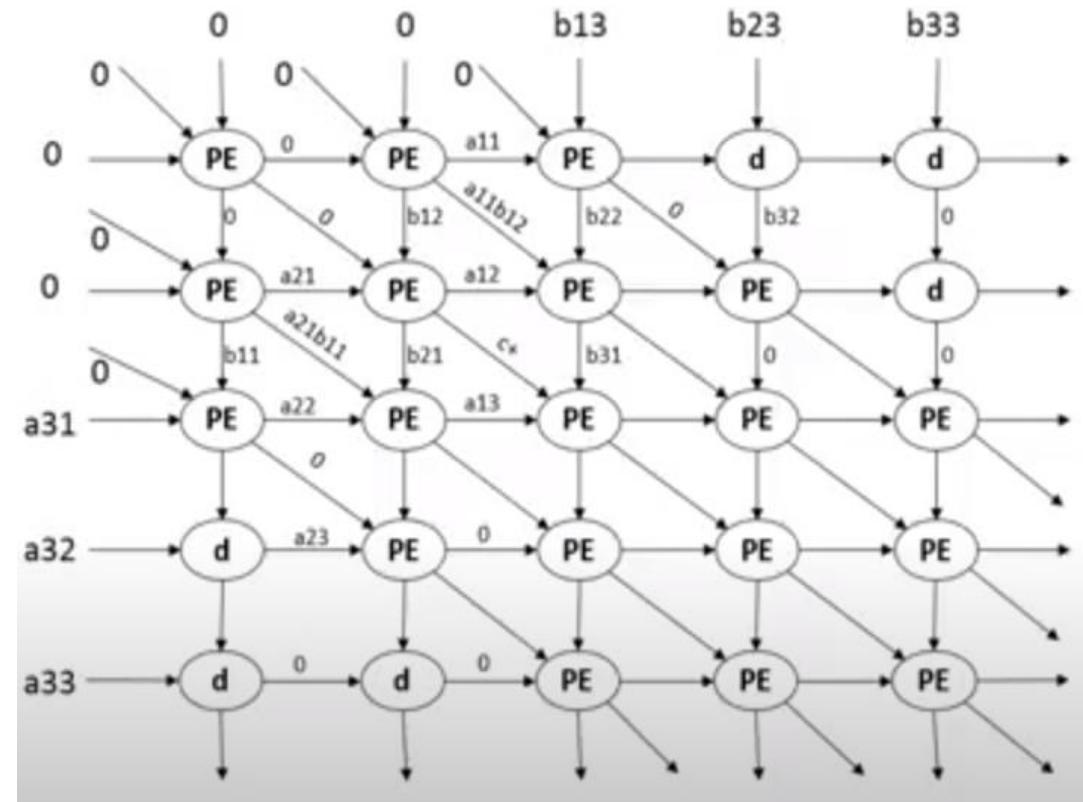
T1 Cycle



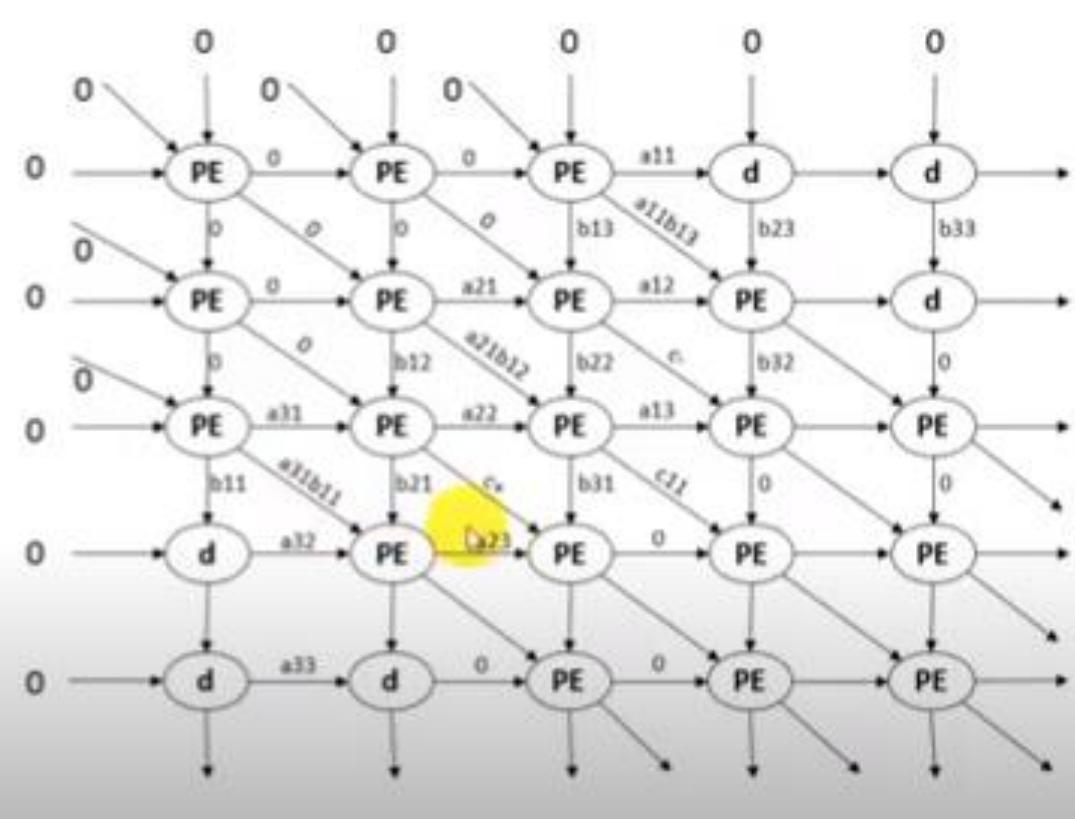
T2 Cycle



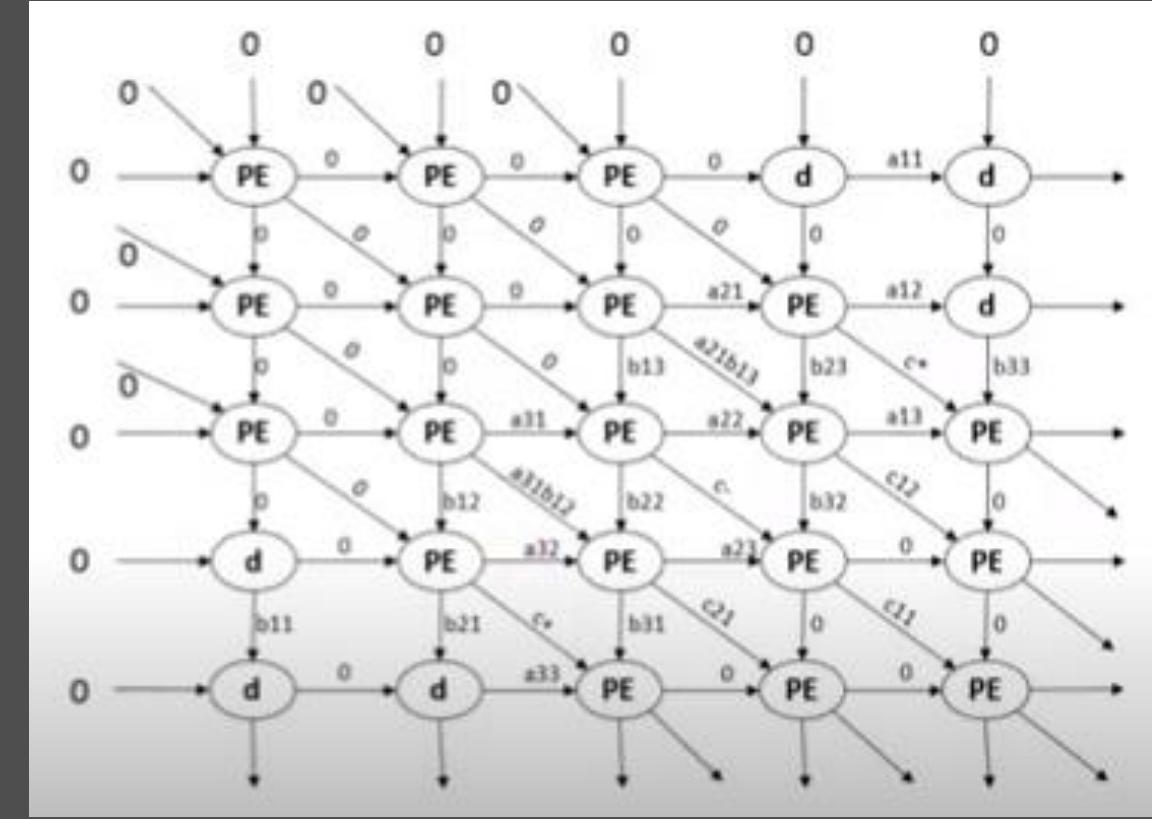
T3 Cycle



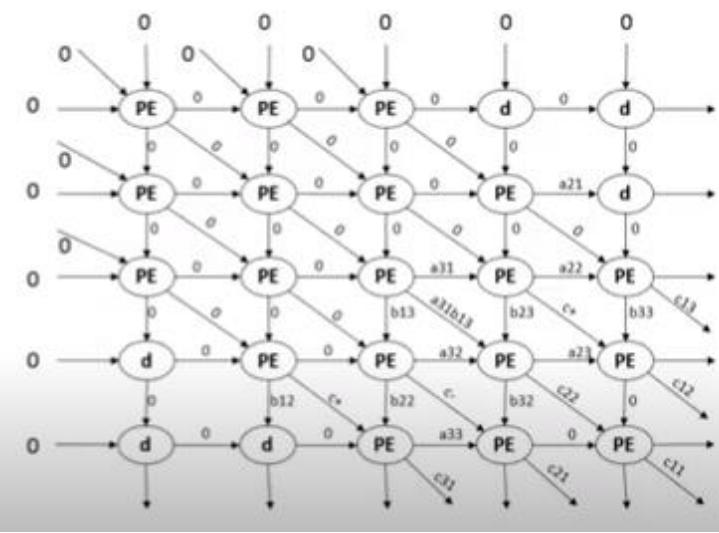
T4 Cycle



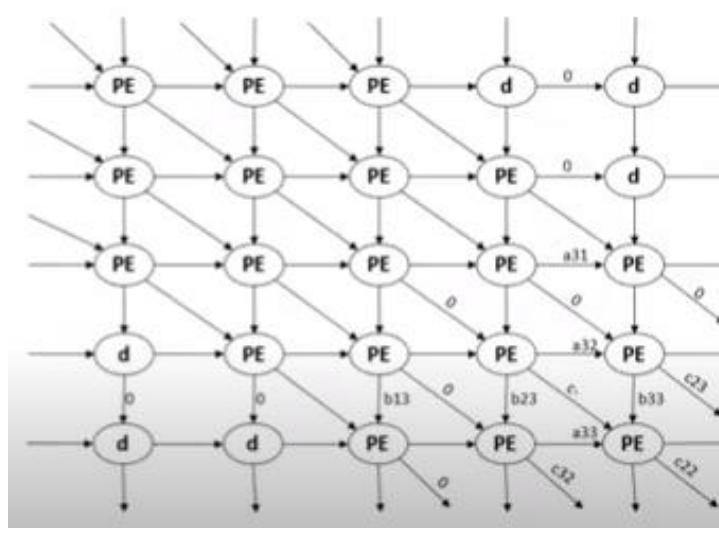
T5 Cycle



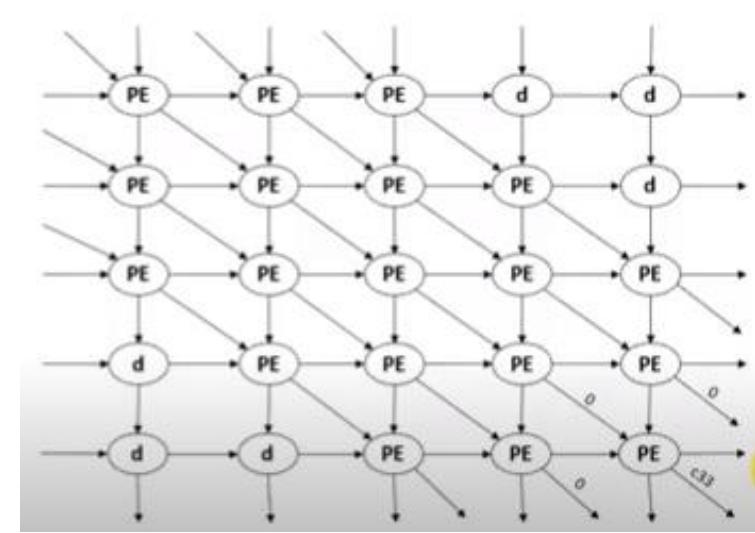
T6 Cycle



T7 Cycle

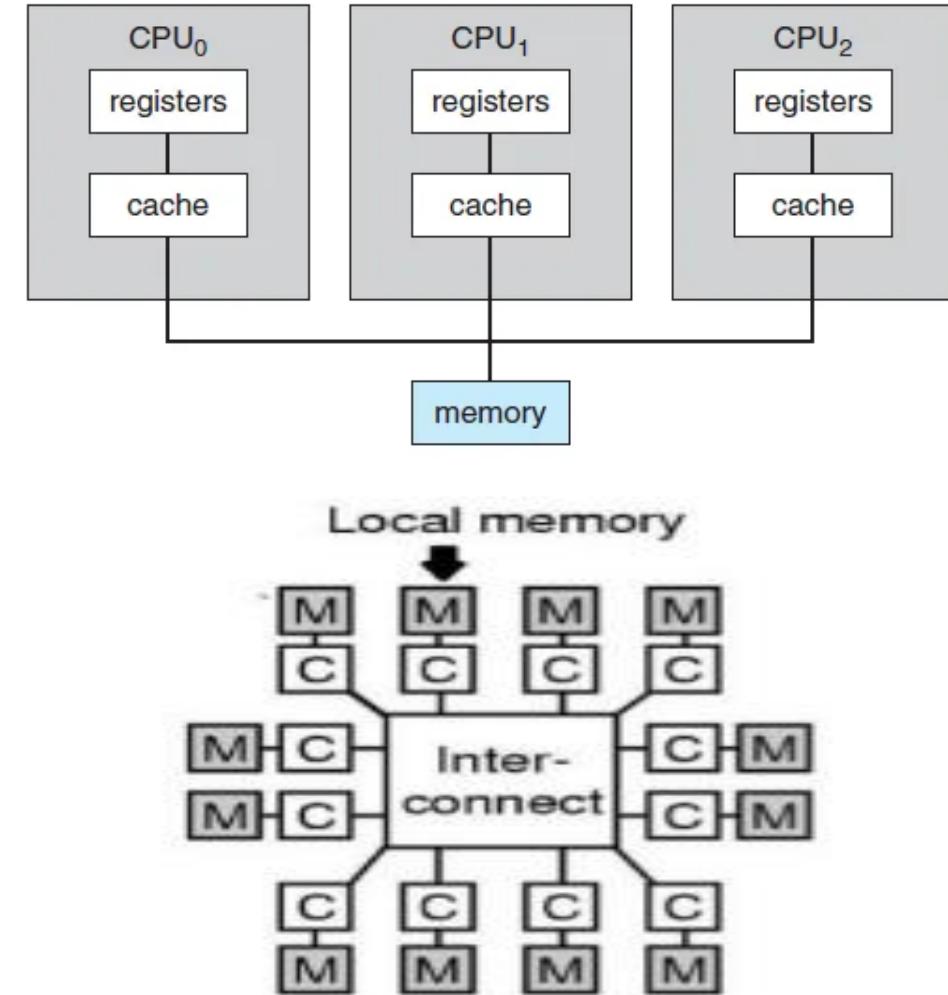
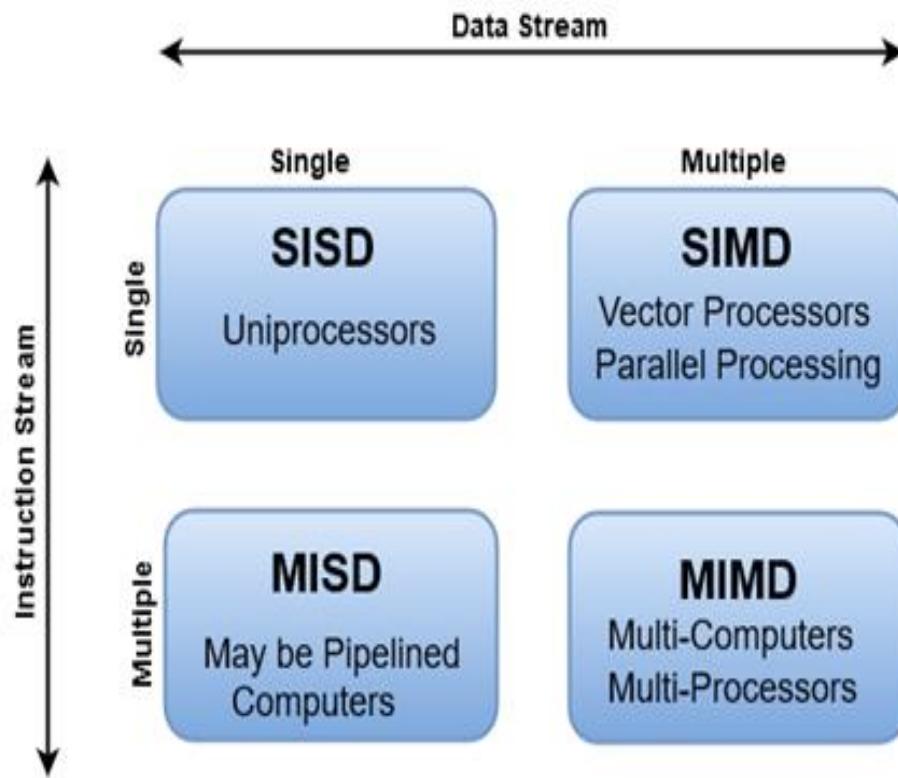


T8 Cycle



Flynn's Classification MIMD

Flynn's Classification



2. Memory Models

Parallel Computers Architectural Model/ Physical Model

Distinguished by having-

1. Shared Common Memory:

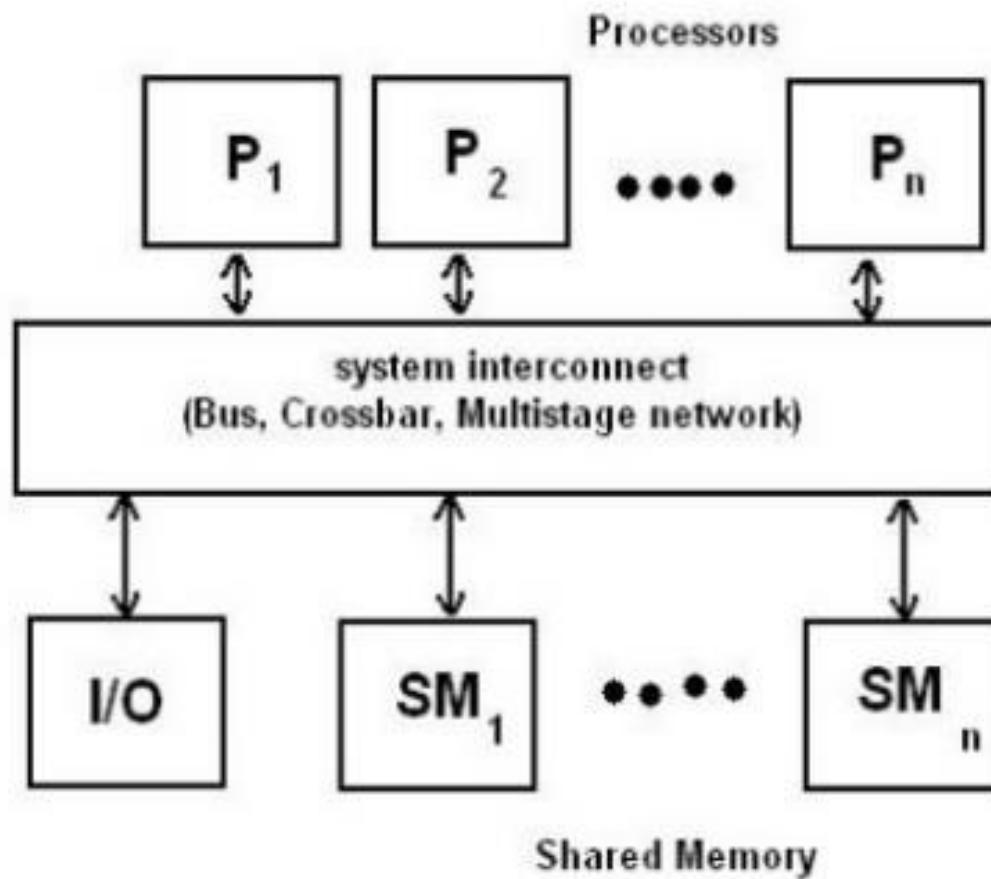
Three Shared-Memory Multiprocessor Models are:

- i. UMA (Uniform-Memory Access)
- ii. NUMA (Non-Uniform-Memory Access)
- iii. COMA (Cache-Only Memory Architecture)

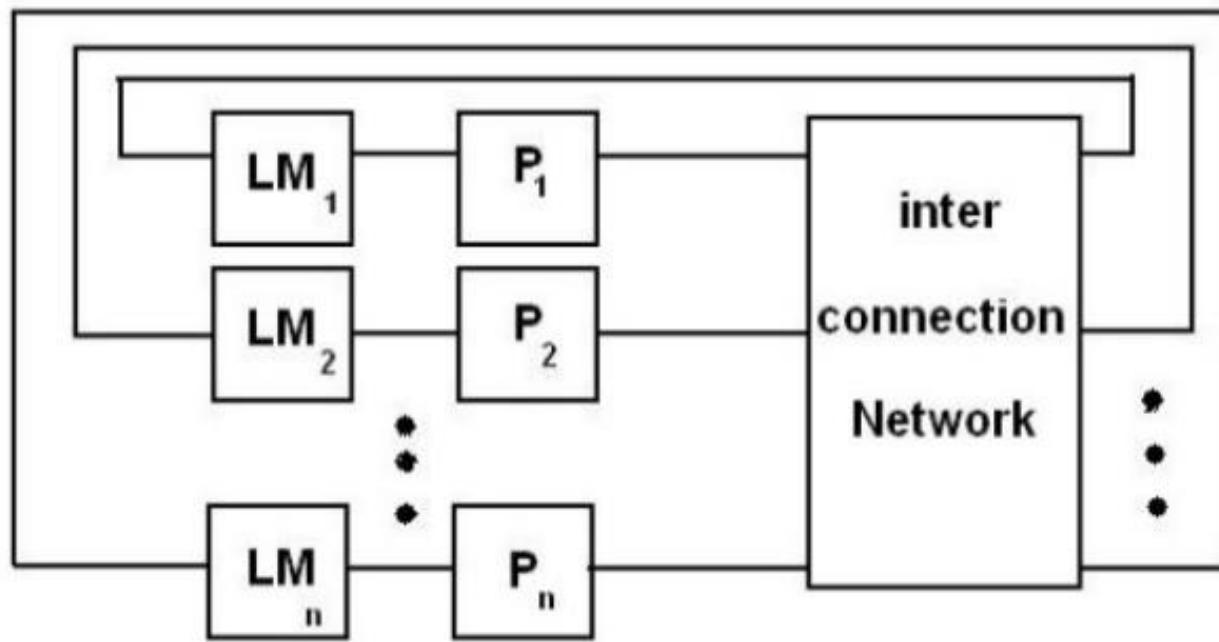
2. Unshared Distributed Memory

- i. CC-NUMA (Cache-Coherent -NUMA)

UMA Multiprocessor Model



NUMA - Memory Models

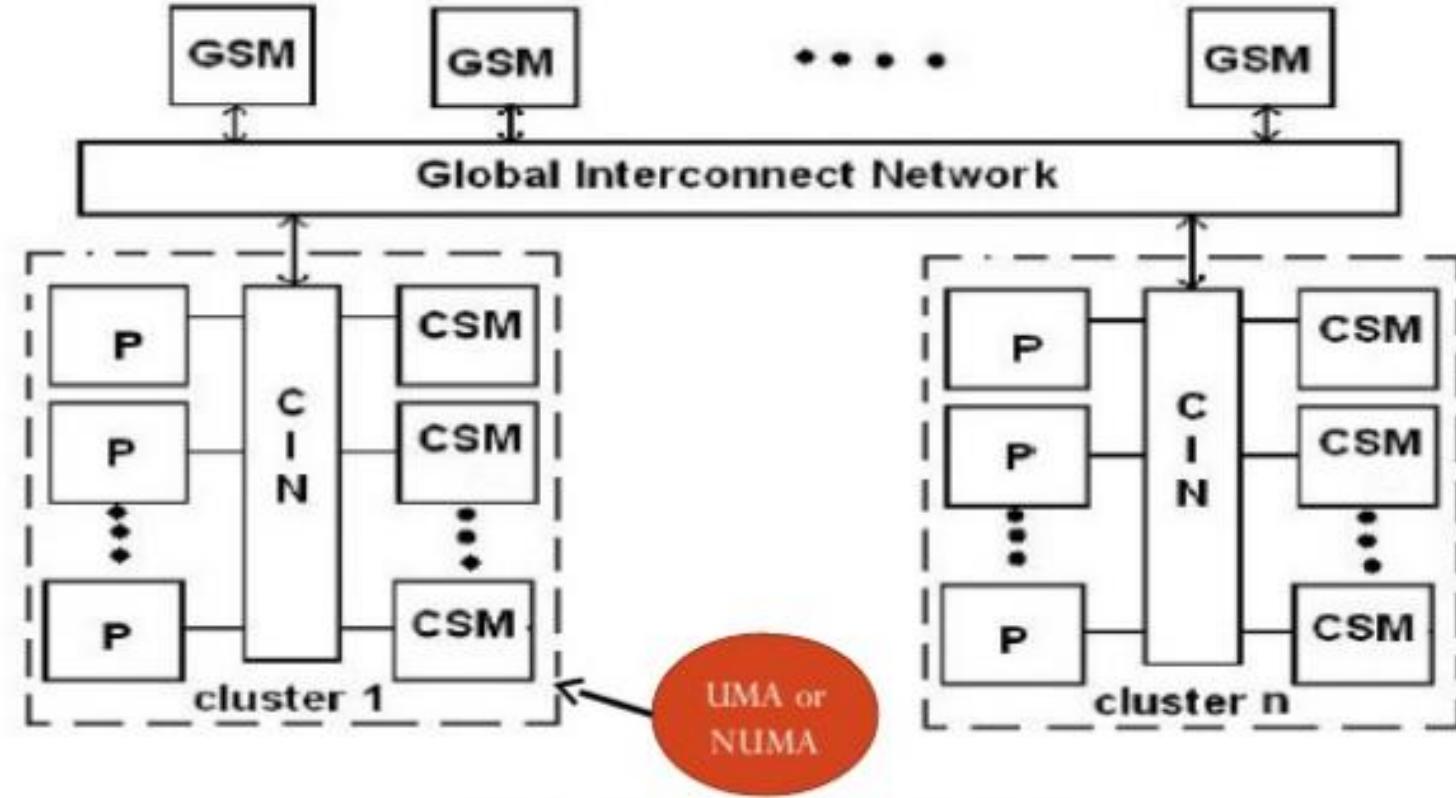


(a) Shared local memories

LM – Local Memory

P - Local Processor

NUMA - Memory Models



(b) A hierarchical cluster model

(Access of Remote Memory)

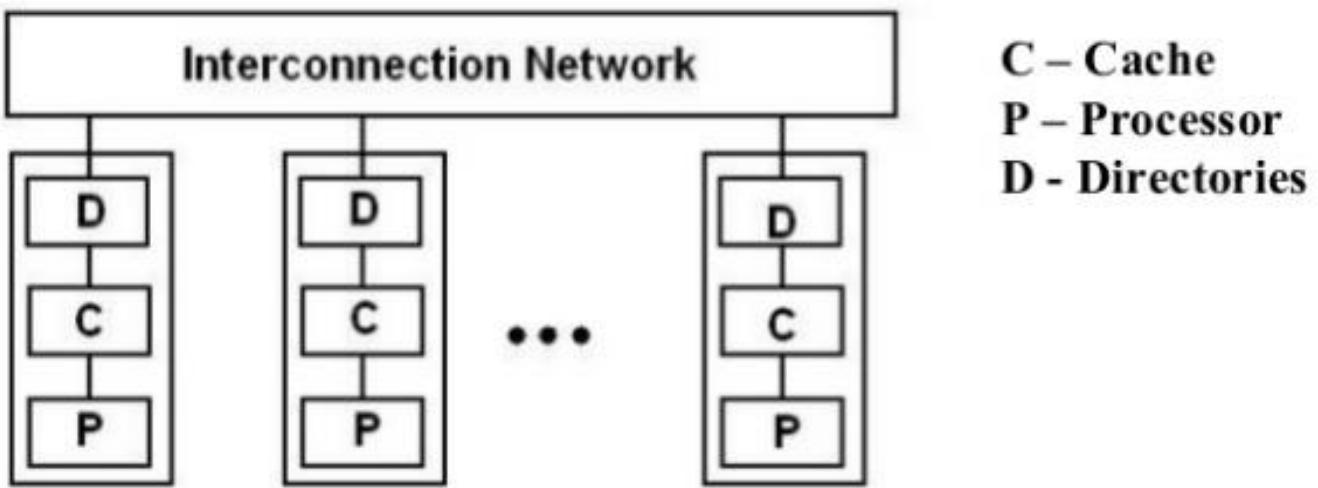
P – Processor

CSM – Cluster Shared Memory

CIN – Cluster Interconnection Network

GSM – Global Shared Memory

COMA Multiprocessor Model



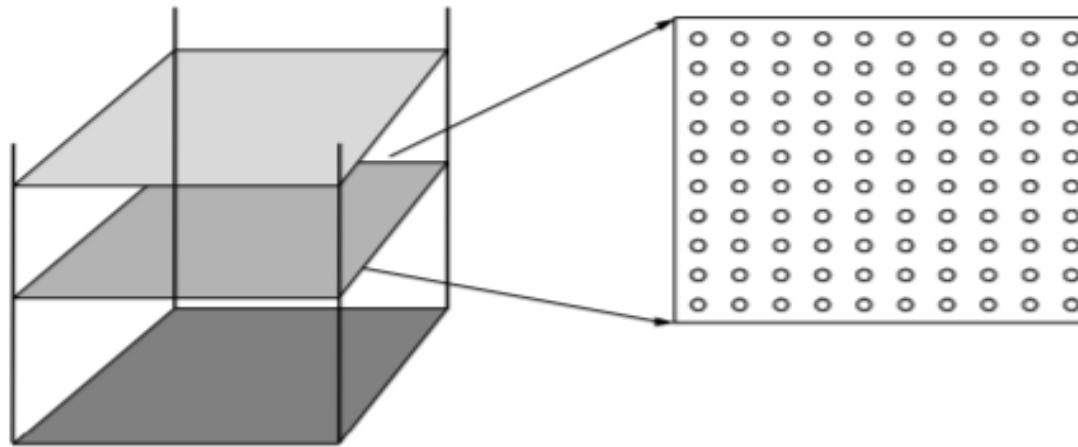
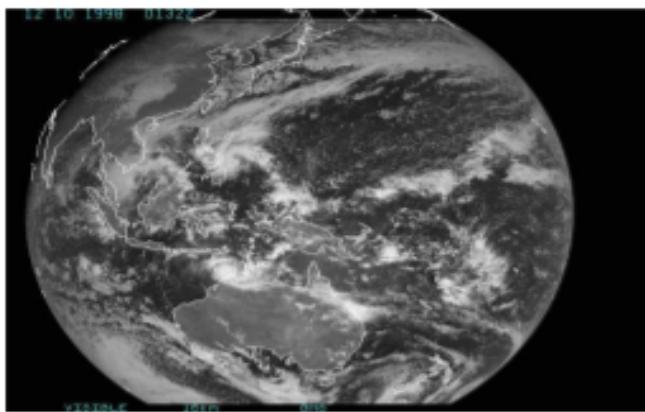
- **Distributed Main Memory converted to Cache**
- **Cache form Global Address Space**
- **Remote Cache access by – Distributed cache Directories**

3. Perspective on parallel programming

- Motivating Problems
- Process of creating a parallel program

3. Perspective on parallel programming

- Motivating Problems : Simulating Ocean Currents
 - Model as two dimensional grid :
 - Discretize in space and time
 - Finer and temporal resolution => greater accuracy
 - Many different computations per time step



(a) Cross sections

(b) Spatial discretization of a cross section



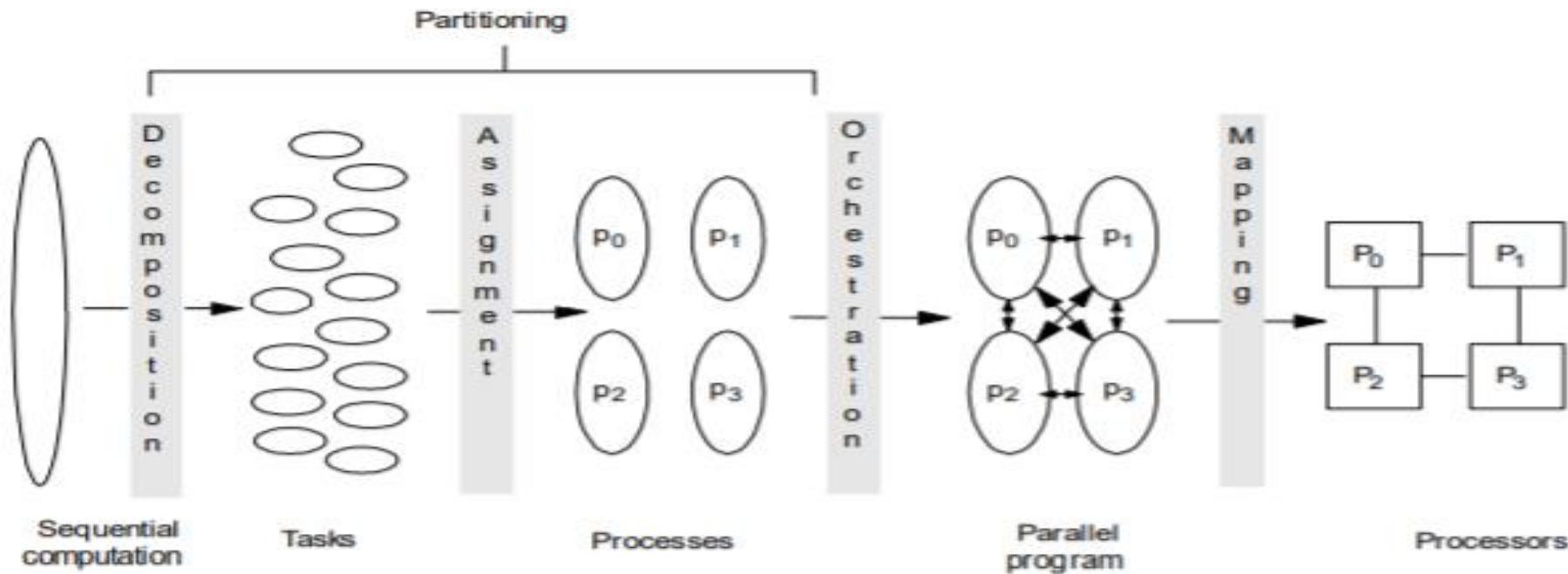
3. Perspective on parallel programming

- Motivating Problems : Simulating interactions of many stars evolving over time.
 - Computing forces is expensive
Eg. Stars on which forces of other elements need to be computed
 - Many time steps, plenty of concurrency across stars within each step.

3. Perspective on parallel programming

- Creating a parallel program
 - Identify the work that can be done in parallel : computation, data access and I/O
 - Partition of work and perhaps data among processes
 - Manage data access, communication and synchronization

3. Perspective on parallel programming



- **Decomposition of computation in tasks**
- **Assignment of tasks to processes**
- **Orchestration of data access, comm, synch.**
- **Mapping processes to processors**

Reference

Text Books and/or Reference Books:

1. Professional CUDA C Programming – John Cheng, Max Grossman, Ty McKercher, 2014
2. B.Wilkinson, M.Allen, "Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers", Pearson Education, 1999
3. I.Foster, "Designing and building parallel programs", 2003
4. Parallel Programming in C using OpenMP and MPI – Micheal J Quinn, 2004
5. Introduction to Parallel Programming – Peter S Pacheco, Morgan Kaufmann Publishers, 2011
6. Advanced Computer Architectures: A design approach, Dezso Sima, Terence Fountain, Peter Kacsuk, 2002
7. **Parallel Computer Architecture : A hardware/Software Approach, David E Culler, Jaswinder Pal Singh Anoop Gupta, 2011**
8. **Introduction to Parallel Computing, Ananth Grama, Anshul Gupta, George Karypis, Vipin Kumar, Pearson, 2011**

Thank You

National Institute of Technology Karnataka Surathkal
Department of Information Technology



IT 301 Parallel Computing
Scalar Processors – Instruction Pipeline

Dr. Geetha V

Assistant Professor

Dept of Information Technology

NITK Surathkal

Index

- 1: Instruction Pipeline - Introduction
- 2. Five Stage pipeline design
- 3: Issues of pipeline and Design Solutions
- 4: Performance evaluation of pipelining

1: Introduction

Course Plan: Theory:

Part A: Parallel Computer Architectures

Week 1,2,3: ***Introduction to Parallel Computer Architecture:***

Parallel Computing,

Parallel architecture,

bit level, instruction level , data level and task level parallelism.

Instruction level parallelisms: pipelining (Data and control instructions),

scalar processors and superscalar processors,

vector processors.

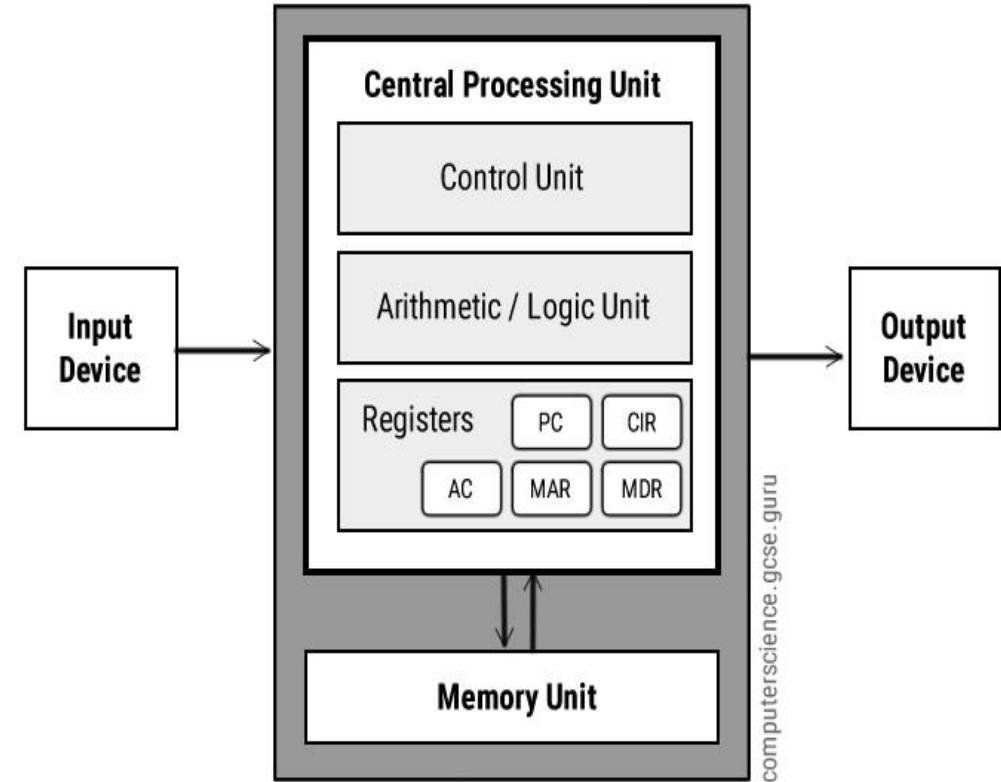
Parallel computers and computation.

1: Instruction Pipeline - Introduction

Features

- Established by John von Neumann in 1945.
- Stored Program Concept
- **PC** : Program Counter
- **CIR**: Current Instruction Register
- **AC**: Accumulator,
- **MAR**: Memory Address Register,
- **MDR**: Memory Data Register
- **Buses**: Address, Data and Control
- **Execution** : One Instruction at a Time

Von Neumann Architecture

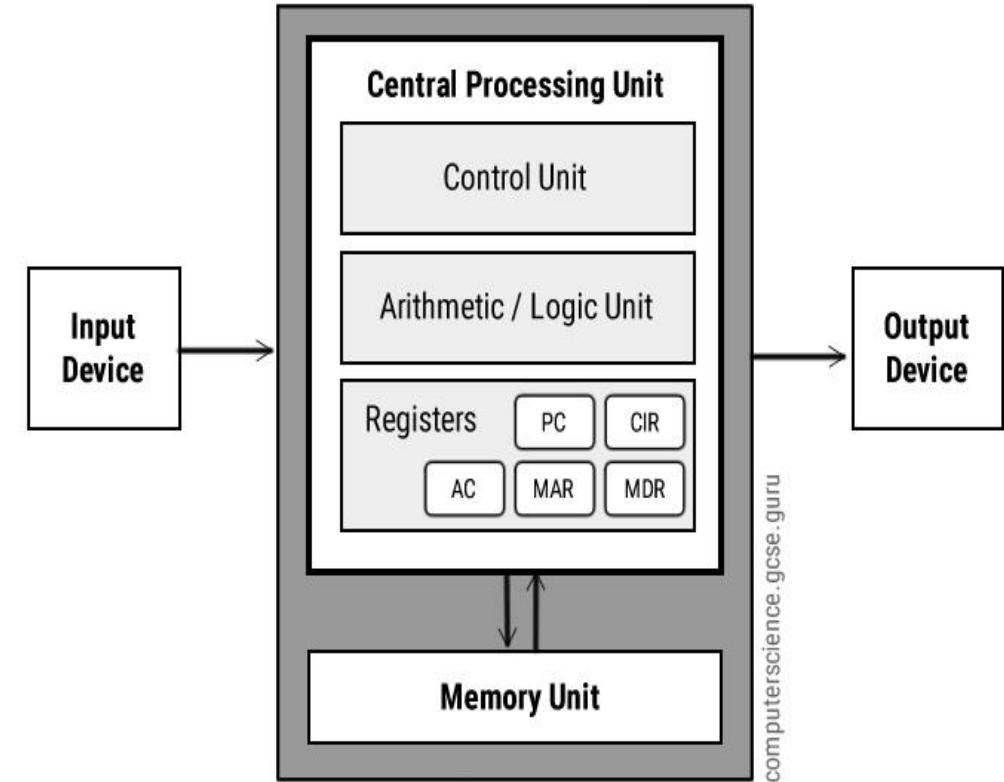


1: Instruction Pipeline - Introduction

Features of RISC

- **RISC:** Reduced Instruction Set Computer
- Fixed **instruction size**
- **Load/Store** Memory Operation
- Few **Addressing Modes**
- **Register to Register** operation
- These features leads towards **pipelined execution**

Von Neumann Architecture



2. Five Stage Pipeline Design

- **Fetch Stage:** The address in program counter is sent to Instruction register and instruction is fetched from memory. PC is incremented.
- **Decode Stage:** The operands are fetched from register file; Sign extension done if necessary.
- **Execution Stage:** The instruction is executed.
- **Memory Stage:** The Load and Store instructions execution.
- **Write Back Stage:** The results are written back to register file.



2. Five Stage Pipeline Design

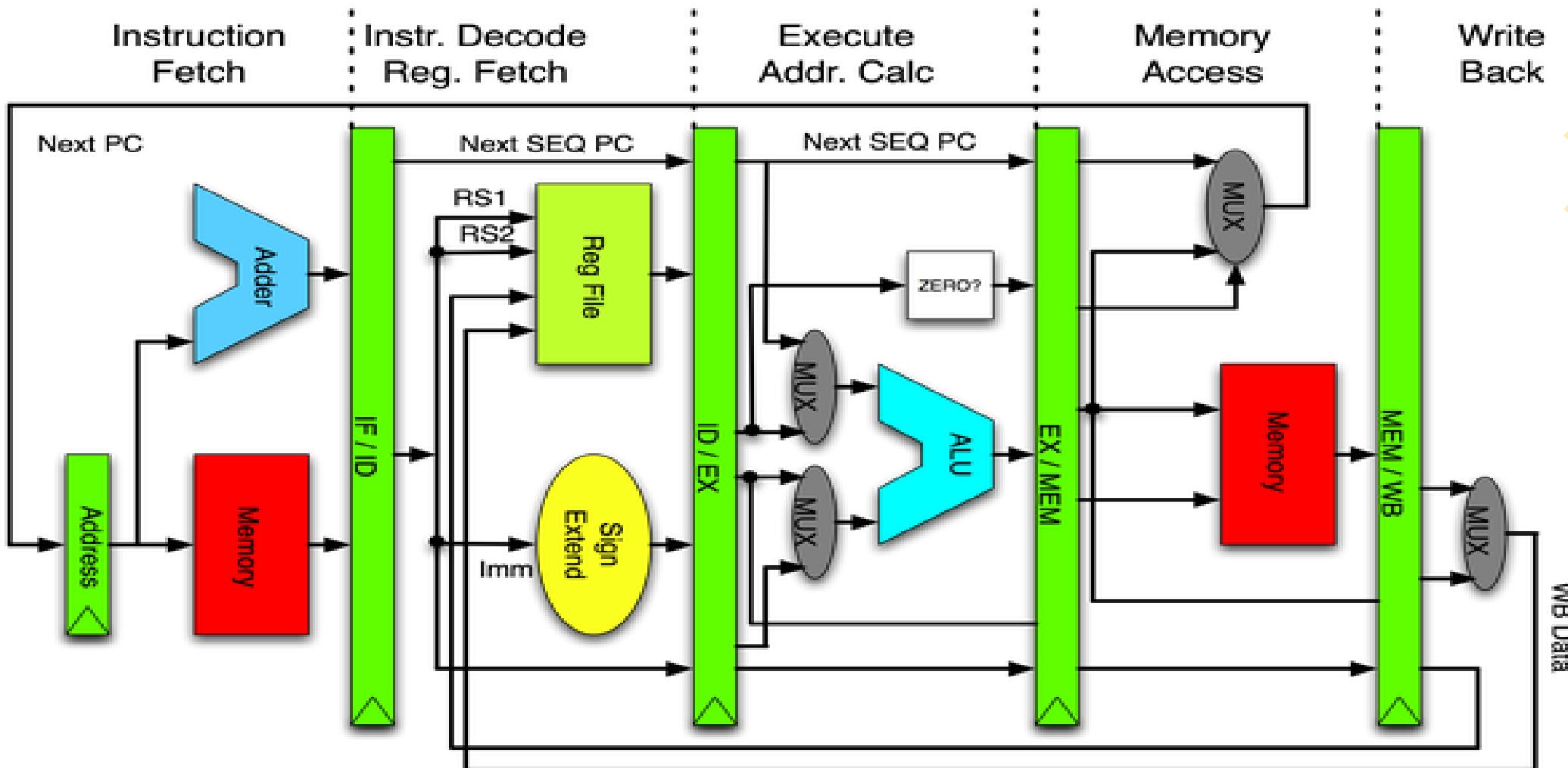


Image from Wikimedia

2. Five Stage Pipeline Design

	Instruction	(F)	(D)	(E)	(M)	(W)	
I ₁	Move R5, #2000h	I ₁					
I ₂	Move R1, #15	I ₂	I ₁				
I ₃	Move R2, #16	I ₃	I ₂	I ₁			
I ₄	Move R3, R1	I ₄	I ₃	I ₂	I ₁		
I ₅	Add R1, R3	I ₅	I ₄	I ₃	I ₂	I ₁	I1: Complete
I ₆	Move R4, R2	I ₆	I ₅	I ₄	I ₃	I ₂	I2: Complete
I ₇	Add R2, R4	I ₇	I ₆	I ₅	I ₄	I ₃	I3: Complete
I ₈	Add R1, R2	I ₈	I ₇	I ₆	I ₅	I ₄	I4: Complete
I ₉	Store [R5], R1	I ₉	I ₈	I ₇	I ₆	I ₅	I5: Complete
I ₁₀			I ₉	I ₈	I ₇	I ₆	I6: Complete
I ₁₁				I ₉	I ₈	I ₇	I7: Complete
I ₁₂					I ₉	I ₈	I8: Complete
I ₁₃						I ₉	I9: Complete

- Adding $(2x+2y)$
 - $x=15$ $y=16$
 - If no pipeline, then this program takes
9 x 5 = 45 cycles
 - If 5 stage pipeline is used, then it takes
5 + 8 = 13 cycles
- (first instruction takes 'n' stages/cycle) + (n-1) instructions

3: Issues of pipeline and Design Solutions

	Instruction	(F)	(D)	(E)	(M)	(W)	
I ₁	Move R5, #2000h	I ₁					
I ₂	Move R1, #15	I ₂	I ₁				
I ₃	Move R2, #16	I ₃	I ₂	I ₁			
I ₄	Move R3, R1	I ₄	I ₃	I ₂	I ₁		
I ₅	Add R1, R3	I ₅	I ₄	I ₃	I ₂	I ₁	I1: Complete
I ₆	Move R4, R2	I ₆	I ₅	I ₄	I ₃	I ₂	I2: Complete
I ₇	Add R2, R4	I ₇	I ₆	I ₅	I ₄	I ₃	I3: Complete
I ₈	Add R1, R2	I ₈	I ₇	I ₆	I ₅	I ₄	I4: Complete
I ₉	Store [R5], R1	I ₉	I ₈	I ₇	I ₆	I ₅	I5: Complete
I ₁₀			I ₉	I ₈	I ₇	I ₆	I6: Complete
I ₁₁				I ₉	I ₈	I ₇	I7: Complete
I ₁₂					I ₉	I ₈	I8: Complete
I ₁₃						I ₉	I9: Complete

- Data Dependency**

Dependency of instructions due to operand values unavailability

- Control Dependency**

Dependency due to unresolved decision on control instructions.

3: Issues of pipeline and Design Solutions

Cycle 1: I1 in fetch stage

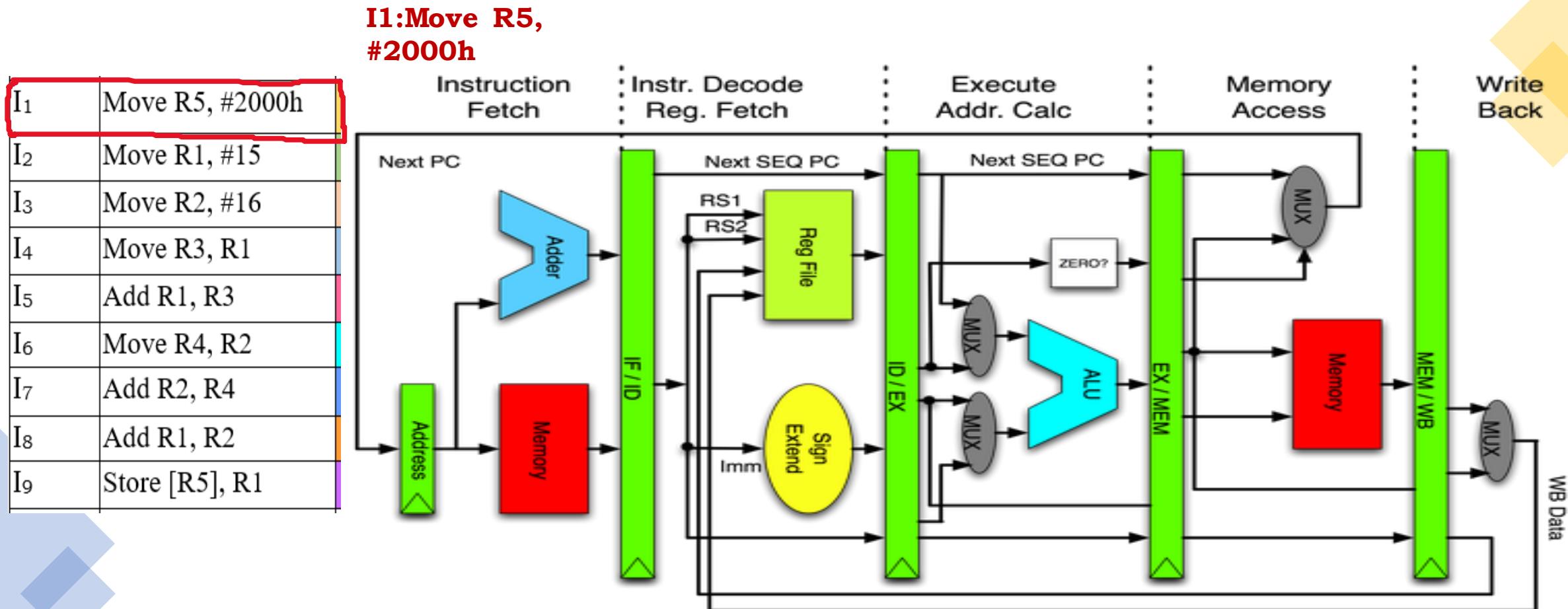


Image from Wikimedia

3: Issues of pipeline and Design Solutions

Cycle 2: I1 in Decode stage, I2 in fetch stage

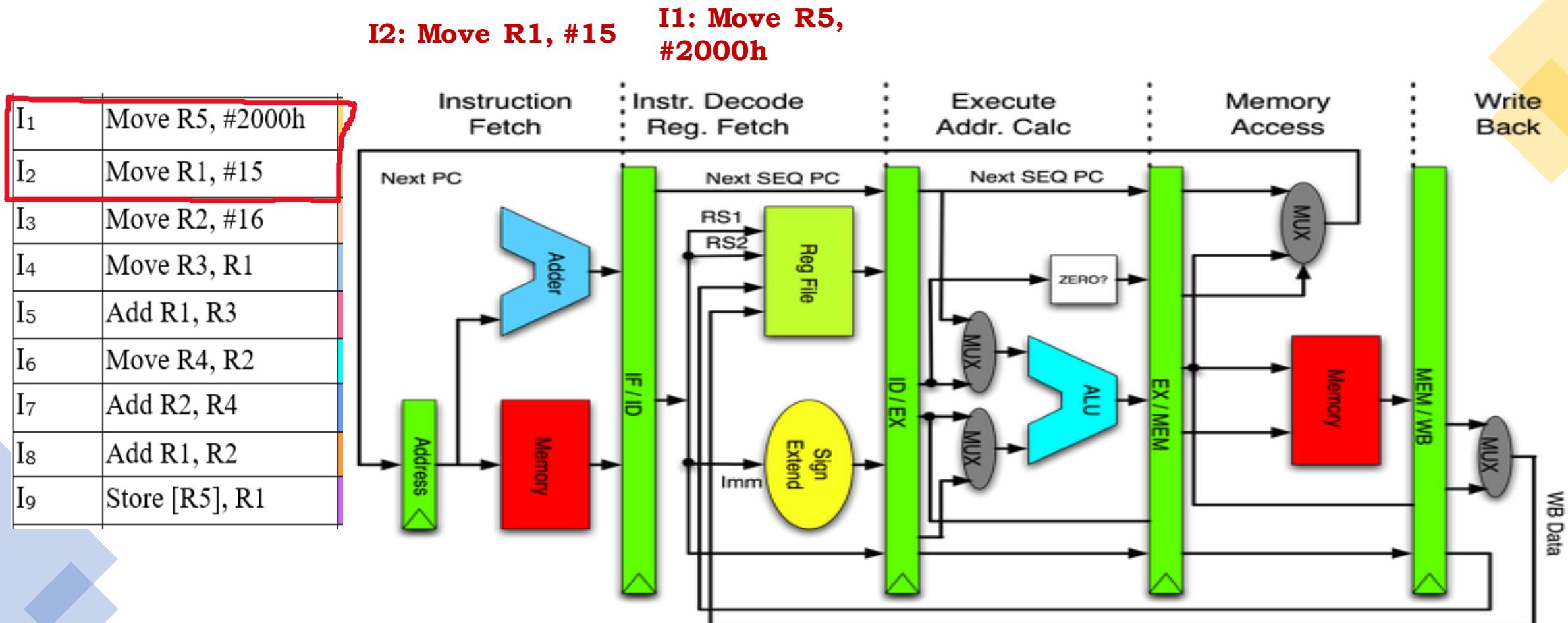


Image from Wikimedia

3: Issues of pipeline and Design Solutions

Cycle 3: I1 in Execute stage, I2 in Decode stage, I3 in fetch stage

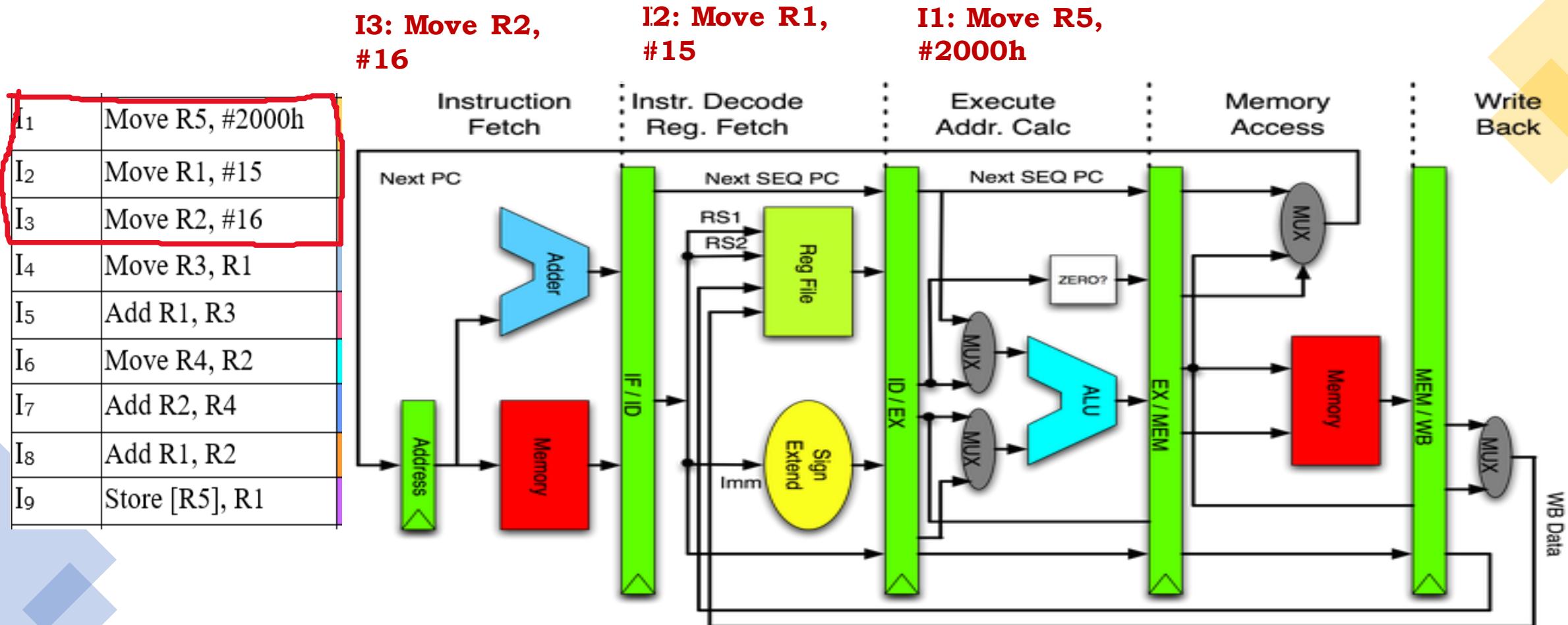


Image from Wikimedia

3: Issues of pipeline and Design Solutions

Cycle 4: I1 in memory access stage, I2 in execute , I3 in Decode and I4 in fetch stage

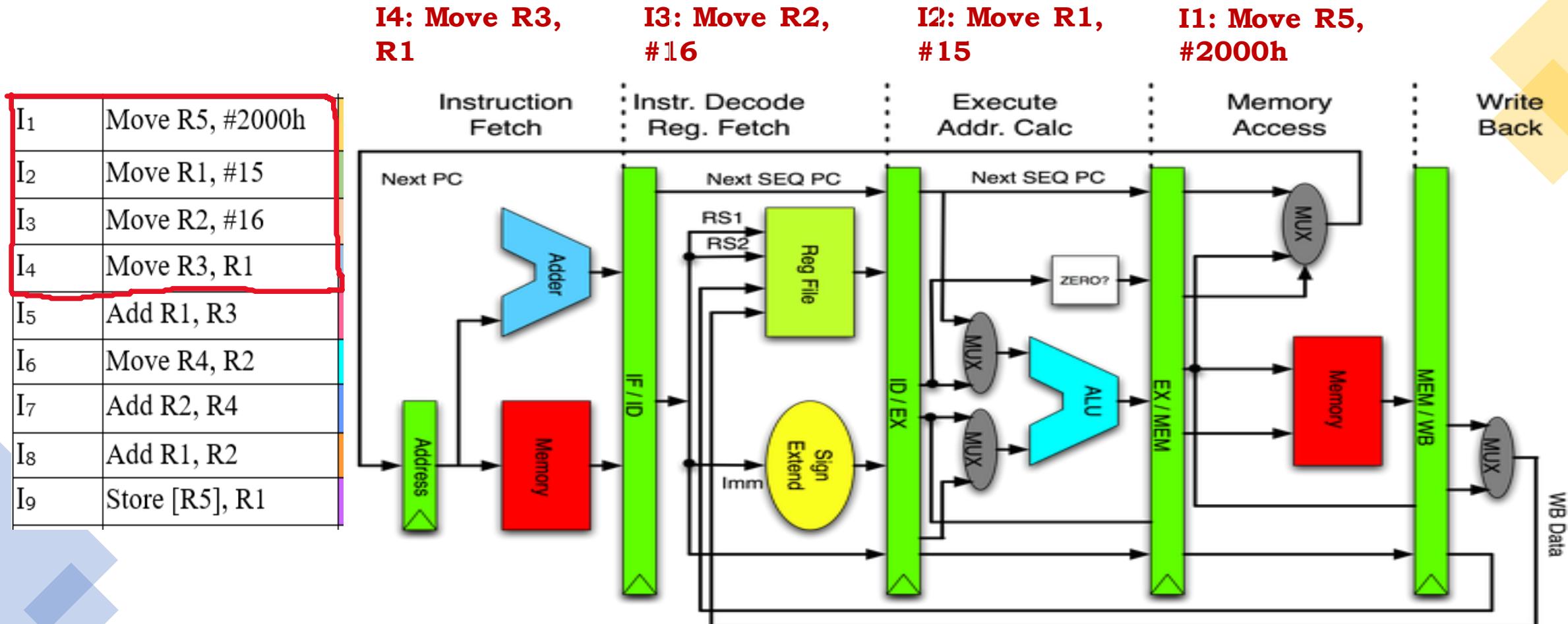


Image from Wikimedia

3: Issues of pipeline and Design Solutions

Cycle 5: R1 data must be Written from I1 to Register file ; R1 Data must be read by Instruction 4.

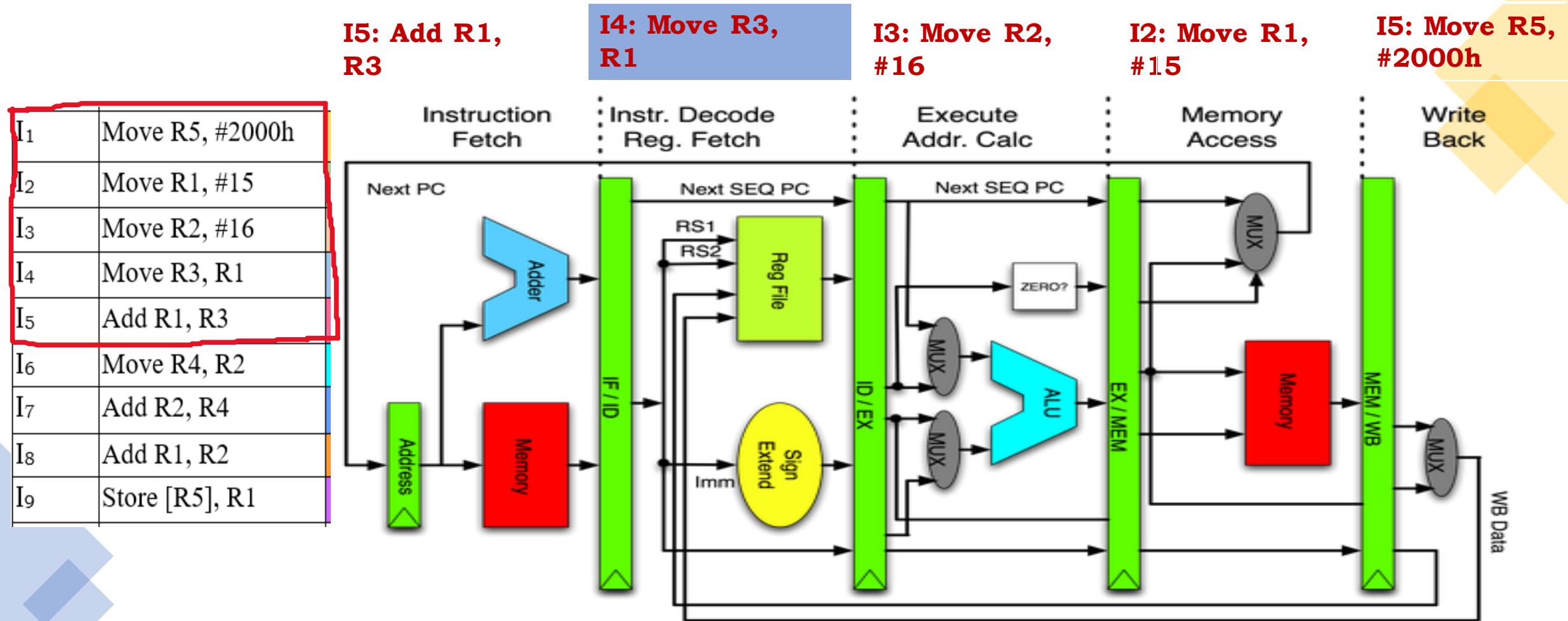


Image from Wikimedia

3: Issues of pipeline and Design Solutions

Cycle 5: R1 data must be Written from I1 to Register file ; R1 Data must be read by Instruction 4.

Data Dependency Issue at Decode Stage: Solve by Read register content after write operation.

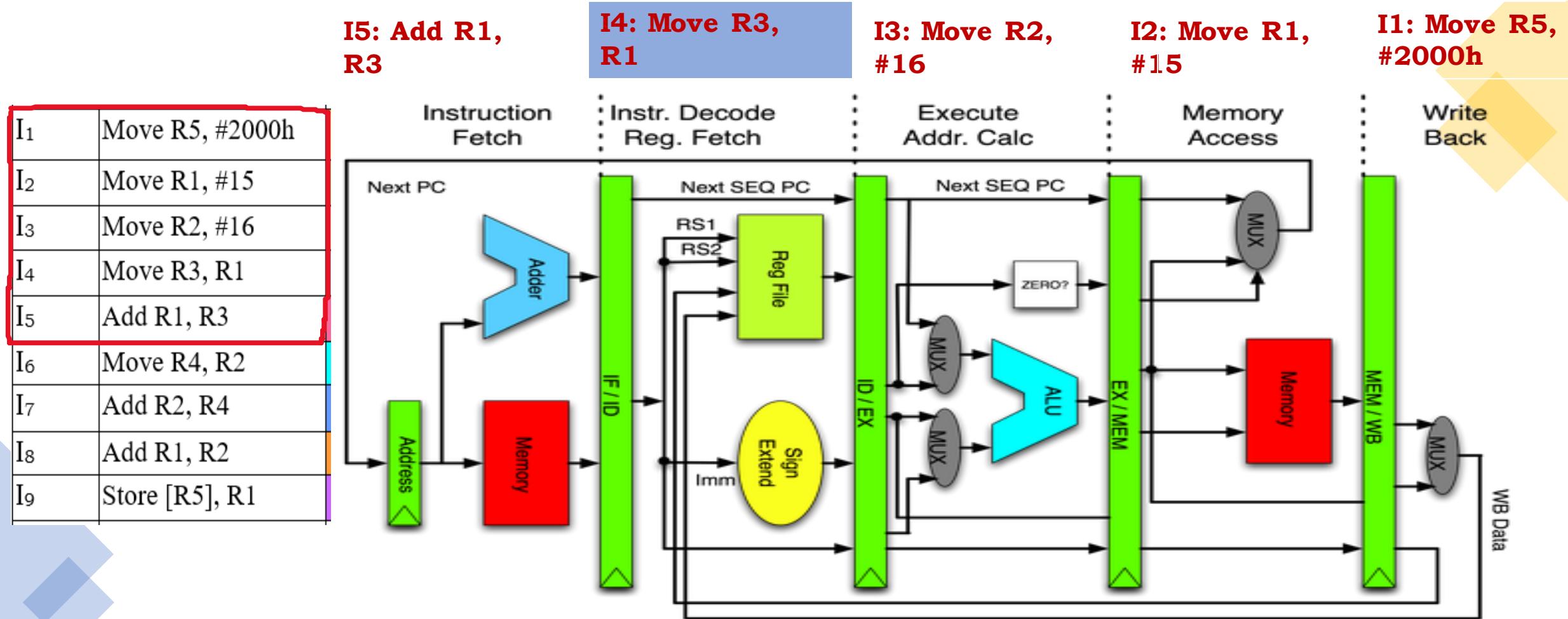


Image from Wikimedia

3: Issues of pipeline and Design Solutions

Cycle 6: R3 data must be Written from I4 to Register file ; R3 Data must be read by Instruction 5.

Data Dependency Issue at Decode Stage: Since Data is not ready, R3 old value is read

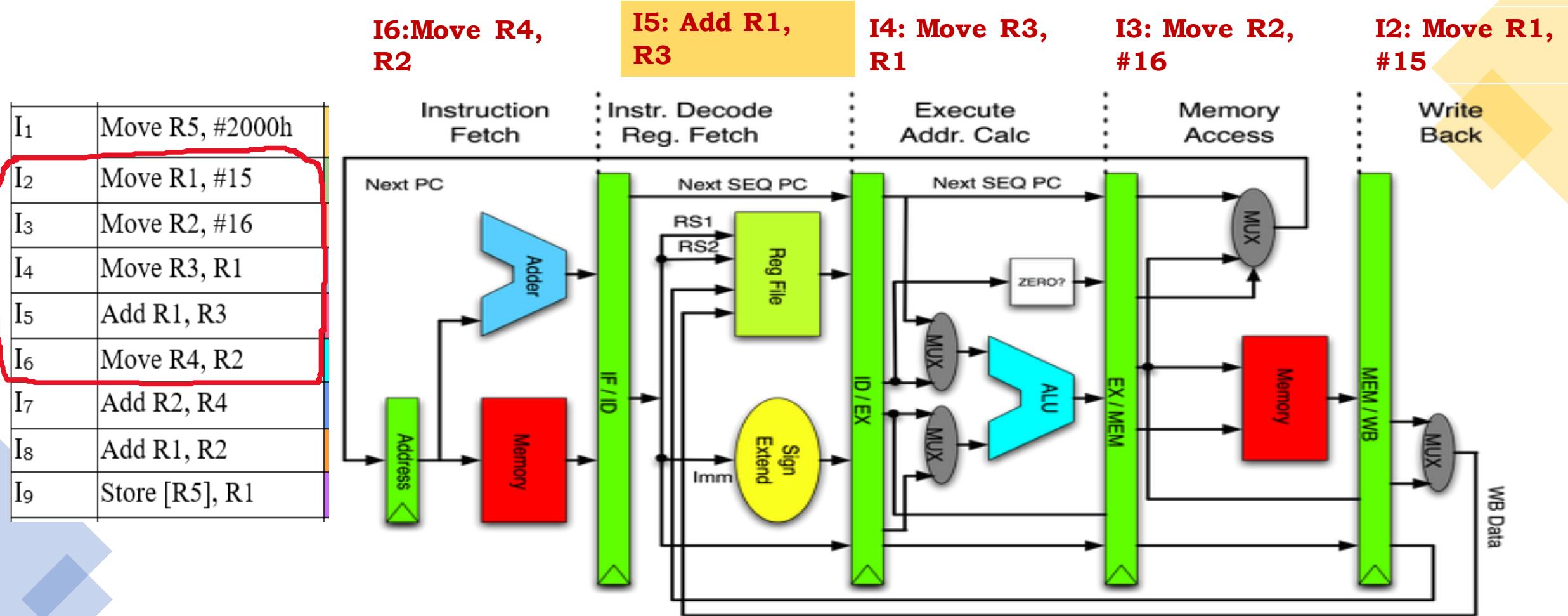


Image from Wikimedia

3: Issues of pipeline and Design Solutions

Cycle 7: R3 data not ready due to instruction dependency

Data Dependency Issue at Execution stage : Stall the pipeline

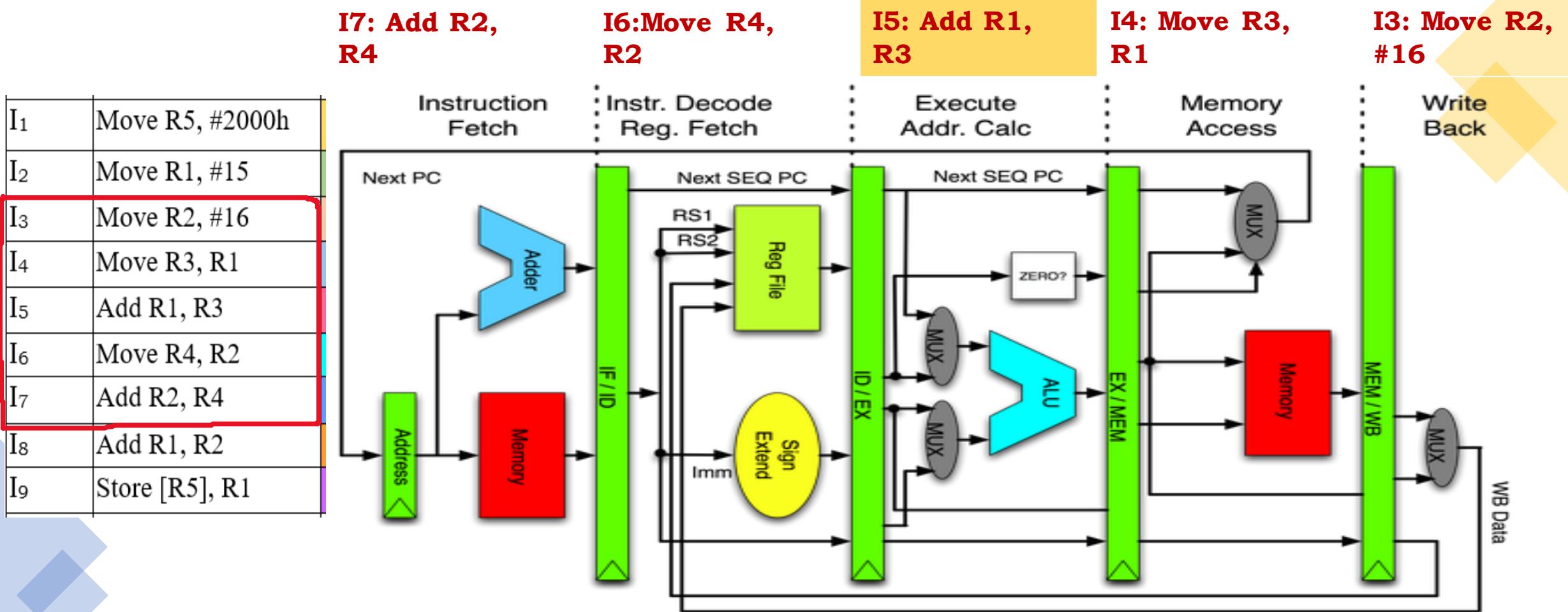


Image from Wikimedia

3: Issues of pipeline and Design Solutions

Cycle 8: Pipeline stall: no new fetch instruction. Only write back happens

Data Dependency Issue at Execution stage : Stall the pipeline

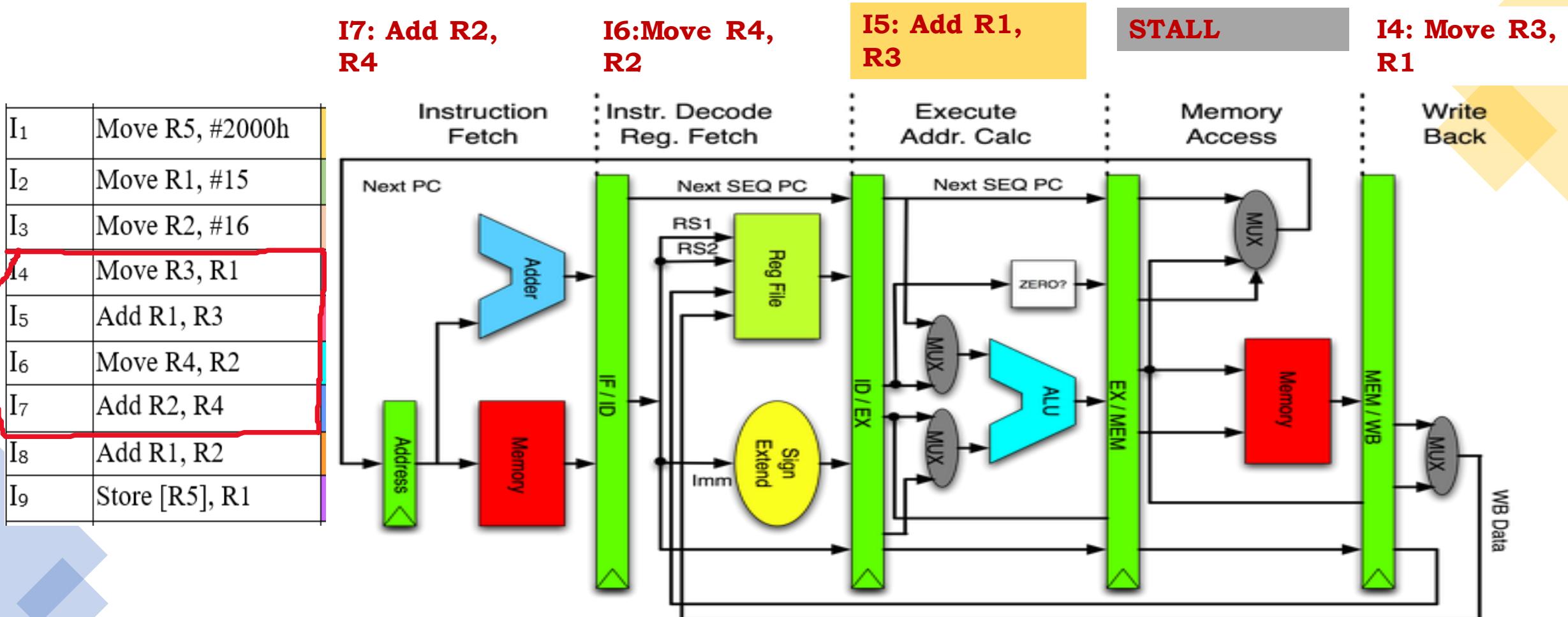


Image from Wikimedia

3: Issues of pipeline and Design Solutions

Cycle 8: We need data of R3 at execution stage , Bu I4 is performing write back to register now.

Use the concept of Data Forwarding

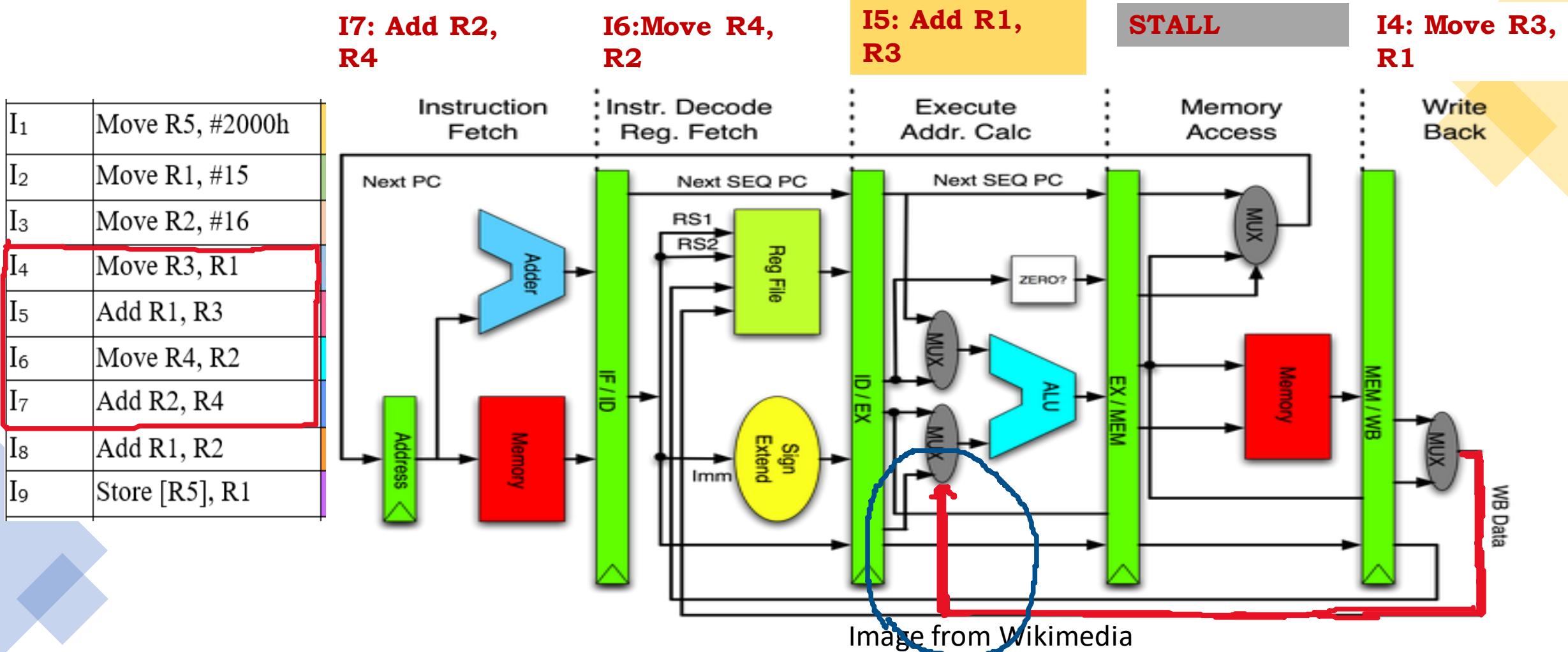
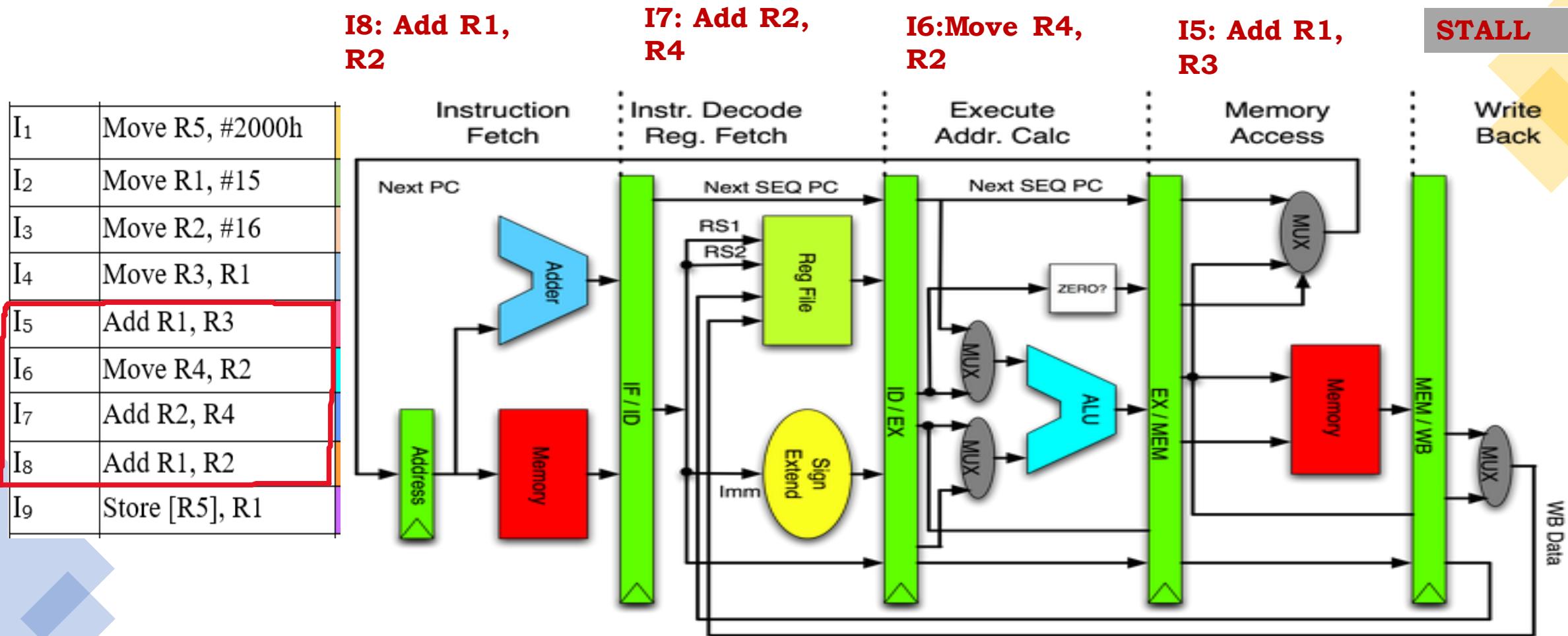


Image from Wikimedia

3: Issues of pipeline and Design Solutions

Cycle 9: Nothing in Writeback stage



3: Issues of pipeline and Design Solutions

Cycle 10: Data dependency at Execution stage; Wait

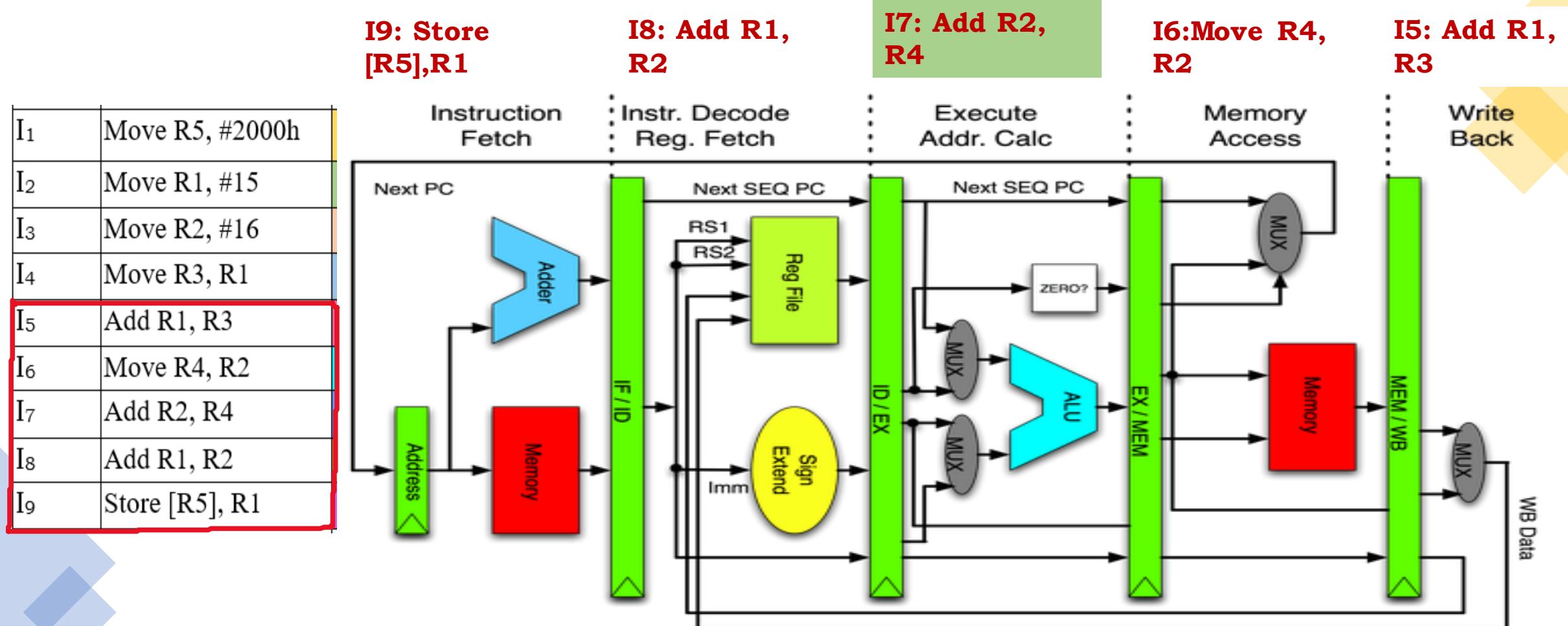
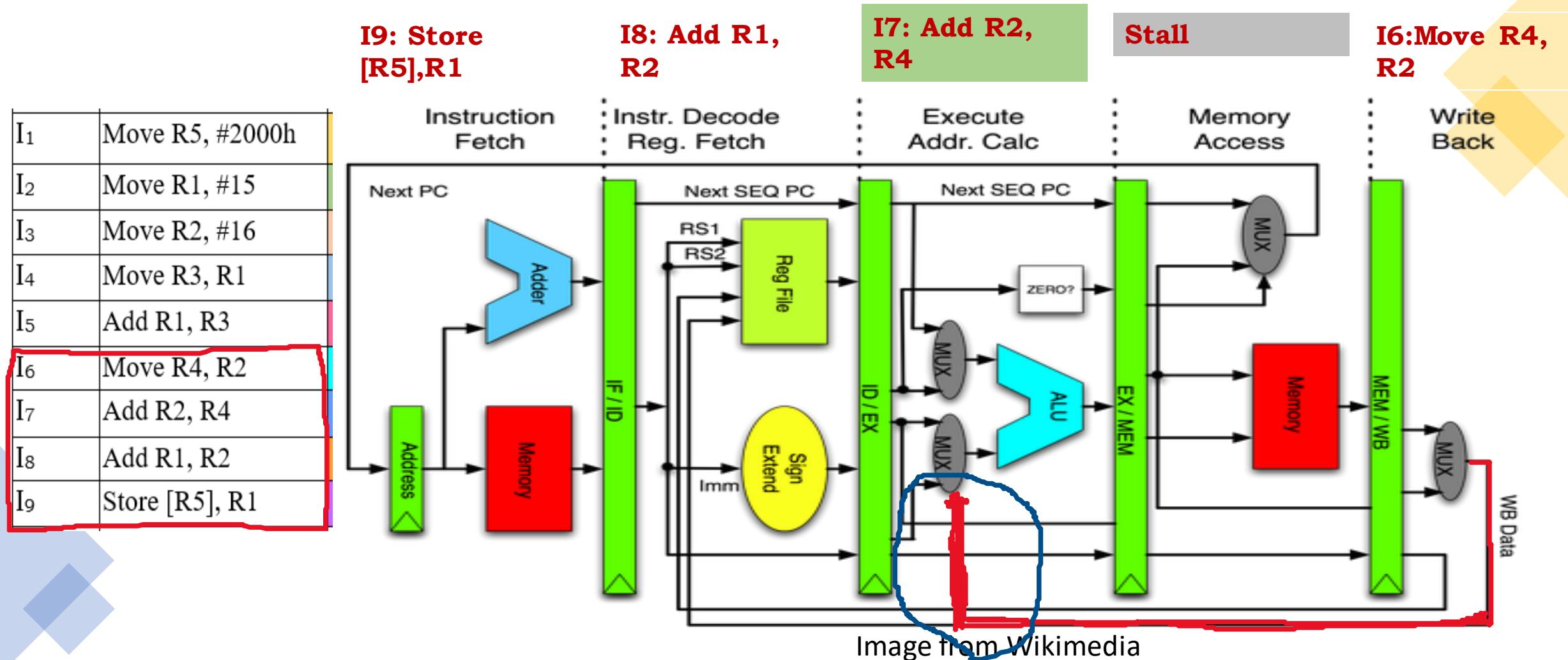


Image from Wikimedia

3: Issues of pipeline and Design Solutions

Cycle 11: Data dependency at Execution stage; Data forwarding from writeback stage



3: Issues of pipeline and Design Solutions

Cycle 12: Data dependency at Execution stage; Wait

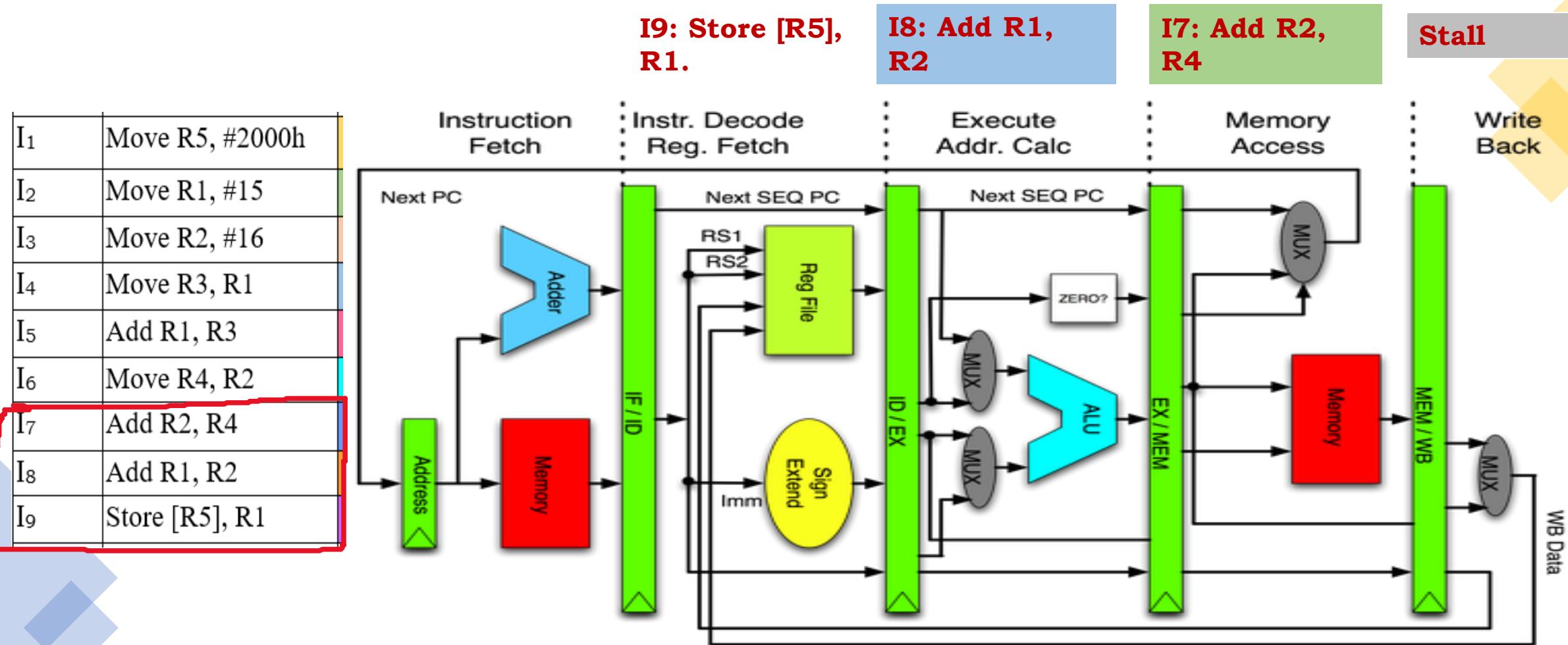
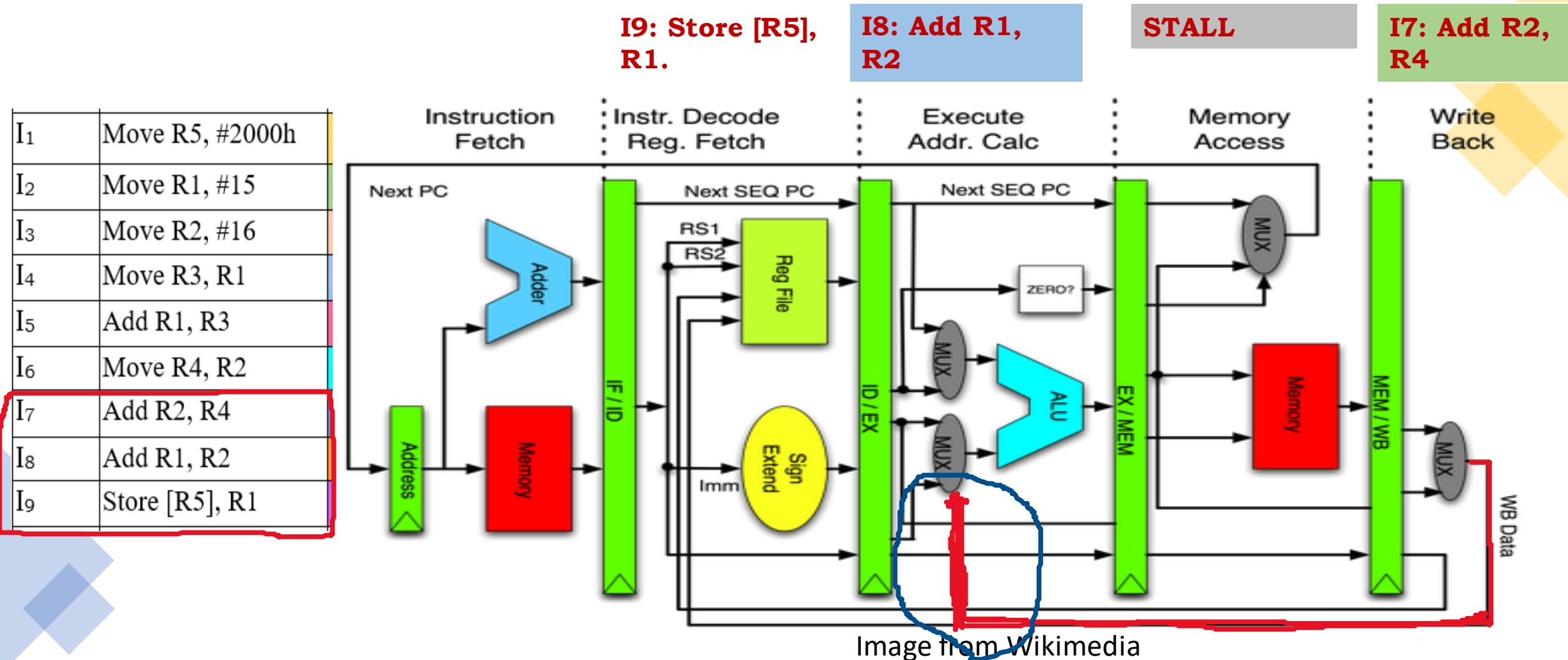


Image from Wikimedia

3: Issues of pipeline and Design Solutions

Cycle 13: Data dependency at Execution stage; Data Forwarding



3: Issues of pipeline and Design Solutions

Cycle 14: Data dependency at Execution stage; Stall Pipeline

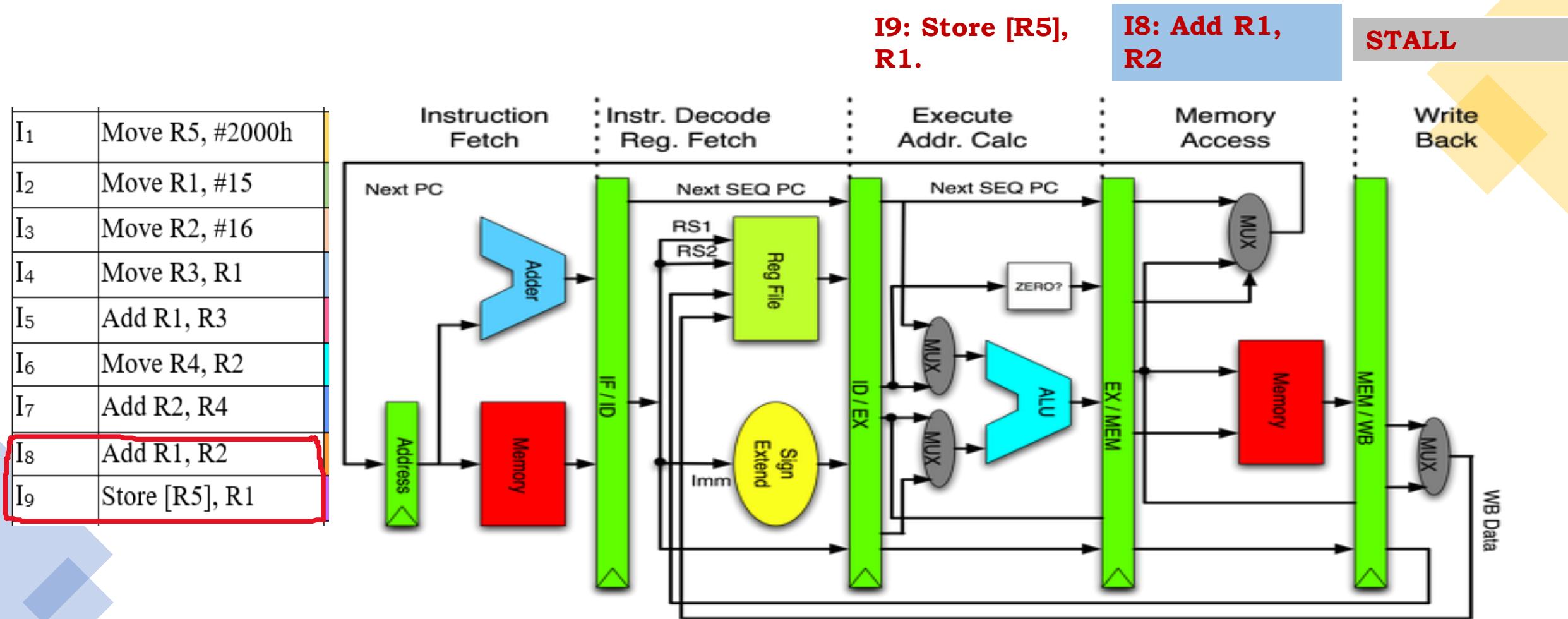
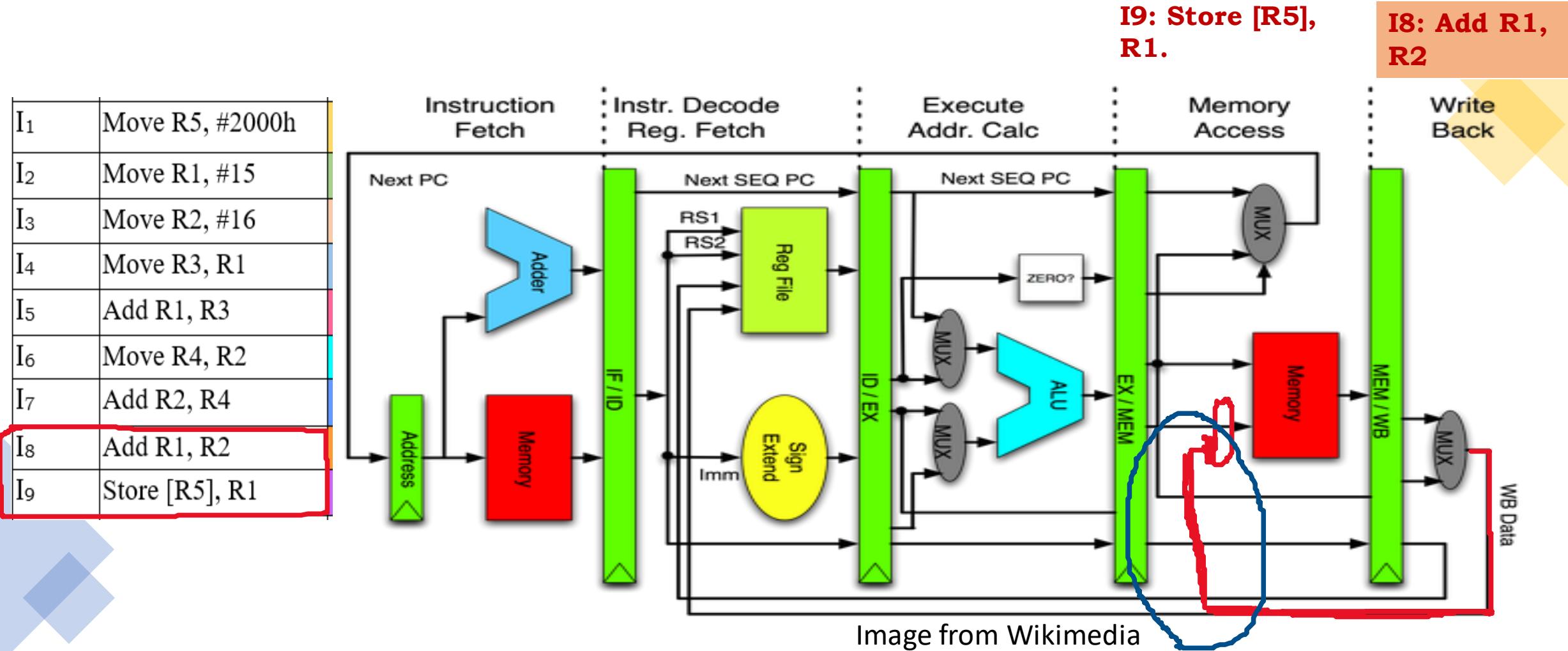


Image from Wikimedia

3: Issues of pipeline and Design Solutions

Cycle 15: Data Dependency at Memory Write Stage : Use Data Forwarding .



3: Issues of pipeline and Design Solutions

Cycle 16: Execution of Program Complete

Total Number of cycles taken = 16

I9: Store [R5], R1.

I1	Move R5, #2000h
I2	Move R1, #15
I3	Move R2, #16
I4	Move R3, R1
I5	Add R1, R3
I6	Move R4, R2
I7	Add R2, R4
I8	Add R1, R2
I9	Store [R5], R1

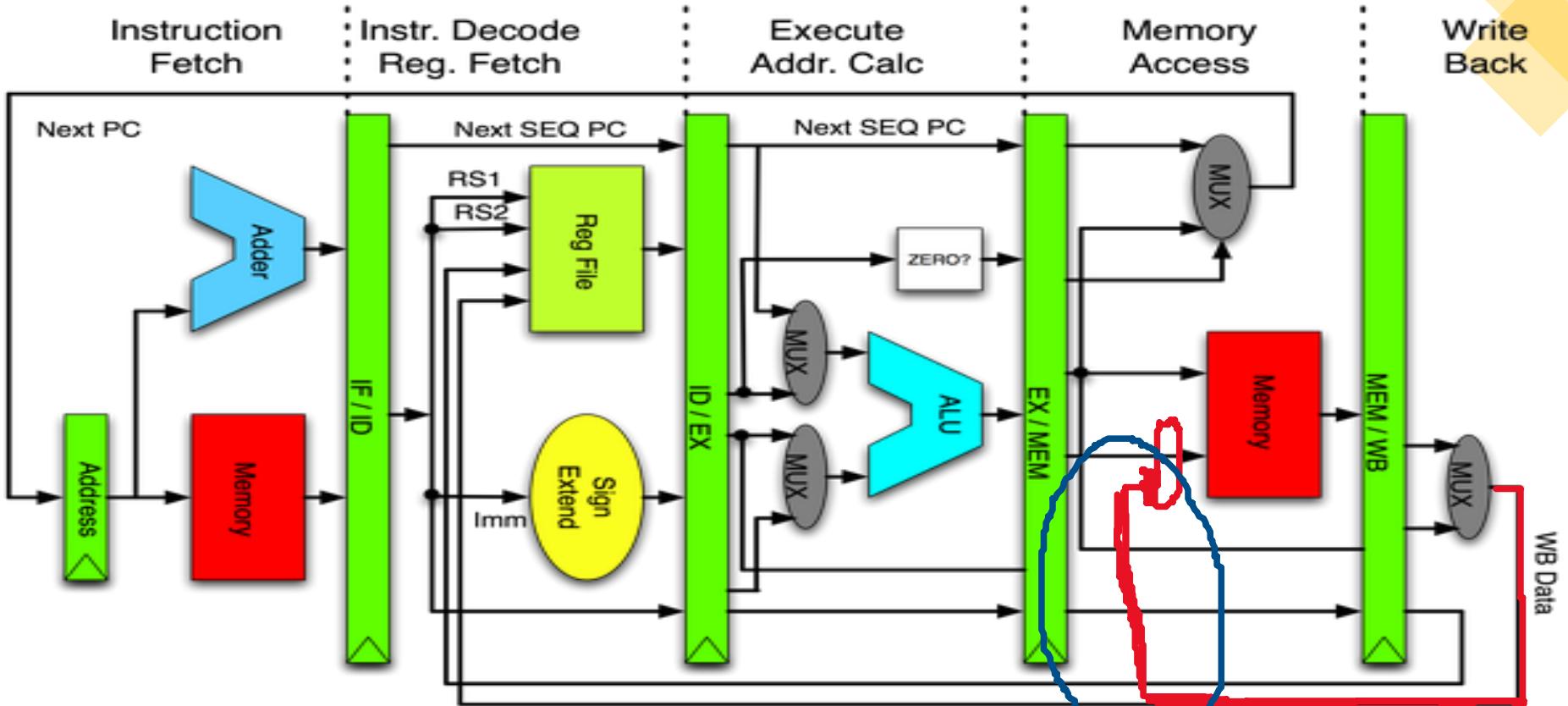


Image from Wikimedia

2. Five Stage Pipeline Design

	Instruction	(F)	(D)	(E)	(M)	(W)	
I ₁	Move R5, #2000h	I ₁					
I ₂	Move R1, #15	I ₂	I ₁				
I ₃	Move R2, #16	I ₃	I ₂	I ₁			
I ₄	Move R3, R1	I ₄	I ₃	I ₂	I ₁		
I ₅	Add R1, R3	I ₅	I ₄	I ₃	I ₂	I ₁	I1: Complete
I ₆	Move R4, R2	I ₆	I ₅	I ₄	I ₃	I ₂	I2: Complete
I ₇	Add R2, R4	I ₇	I ₆	I ₅	I ₄	I ₃	I3: Complete
I ₈	Add R1, R2	I ₈	I ₇	I ₆	I ₅	I ₄	I4: Complete
I ₉	Store [R5], R1	I ₉	I ₈	I ₇	I ₆	I ₅	I5: Complete
I ₁₀			I ₉	I ₈	I ₇	I ₆	I6: Complete
I ₁₁				I ₉	I ₈	I ₇	I7: Complete
I ₁₂					I ₉	I ₈	I8: Complete
I ₁₃						I ₉	I9: Complete

- If no pipeline, then this program takes **9 x 5 = 45 cycles**
- If 5 stage pipeline is used, then it takes **5 + 8 = 13 cycles**
(first instruction takes 'n' stages/cycle) + (n-1) instructions
- Actual Cycles taken due to **data dependency** = **16 Cycles.**

Reference

Textbooks and/or Reference Books:

1. Professional CUDA C Programming – John Cheng, Max Grossman, Ty McKercher, 2014
2. Wilkinson, M. Allen, "Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers", Pearson Education, 1999
3. I. Foster, "Designing and building parallel programs", 2003
4. Parallel Programming in C using OpenMP and MPI – Micheal J Quinn, 2004
5. Introduction to Parallel Programming – Peter S Pacheco, Morgan Kaufmann Publishers, 2011
6. Advanced Computer Architectures: A design approach, Dezso Sima, Terence Fountain, Peter Kacsuk, 2002
7. Parallel Computer Architecture : A hardware/Software Approach, David E Culler, Jaswinder Pal Singh Anoop Gupta, 2011
8. Introduction to Parallel Computing, Ananth Grama, Anshul Gupta, George Karypis, Vipin Kumar, Pearson, 2011

Thank You

National Institute of Technology Karnataka Surathkal
Department of Information Technology



IT 301 Parallel Computing
Scalar Processors – Control Instructions

Dr. Geetha V

Assistant Professor

Dept of Information Technology

NITK Surathkal

Index

- 1. Five Stage pipeline design
- 2: Control Hazards

1: Introduction

Course Plan: Theory:

Part A: Parallel Computer Architectures

Week 1,2,3: ***Introduction to Parallel Computer Architecture:***

Parallel Computing,

Parallel architecture,

bit level, instruction level , data level and task level parallelism.

Instruction level parallelisms: pipelining (Data and control instructions),

scalar processors and superscalar processors,

vector processors.

Parallel computers and computation.

1. Five Stage Pipeline Design

- **Fetch Stage:** The address in program counter is sent to Instruction register and instruction is fetched from memory. PC is incremented.
- **Decode Stage:** The operands are fetched from register file; Sign extension done if necessary.
- **Execution Stage:** The instruction is executed.
- **Memory Stage:** The Load and Store instructions execution.
- **Write Back Stage:** The results are written back to register file.



1. Five Stage Pipeline Design

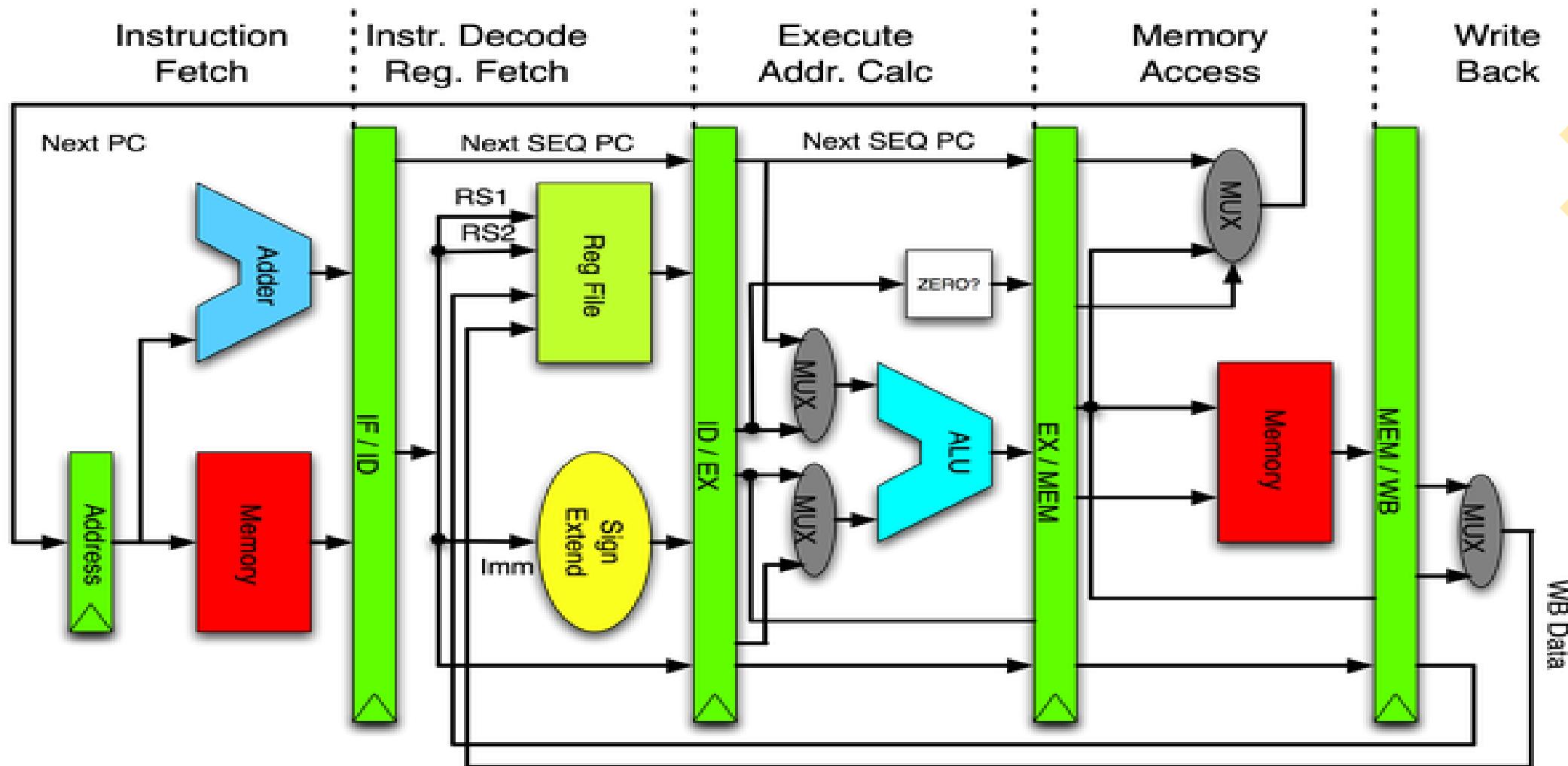


Image from Wikimedia

1: Five stage pipeline design

- **Data Dependency (Data Hazard)**

Dependency of instructions due to operand values unavailability

Solution :

- (a) Data forwarding

- (b) Stall the pipeline until the data is available.

- **Control Dependency (Control Hazard)**

Dependency due to unresolved decision on control instructions.

2. Control hazards

- The branches – conditional branches – cause pipeline hazard
- The outcome of a conditional branch is not known until the end of the EX stage, but is required at IF to load another instruction and keep the pipeline full.

JGE Next

Add CX, 02

Dec BL

JMP NEXT

NEXT:



2. Control Hazards

- **Data Dependency (Data Hazard)**

Dependency of instructions due to operand values unavailability

Solution :

- (a) Data forwarding
- (b) Stall the pipeline until the data is available.

- **Control Dependency (Control Hazard)**

Dependency due to unresolved decision on control instructions.

- (a) Insert NOP (No Operation)
- (b) NOP after execution of instruction
- (c) Branch Prediction

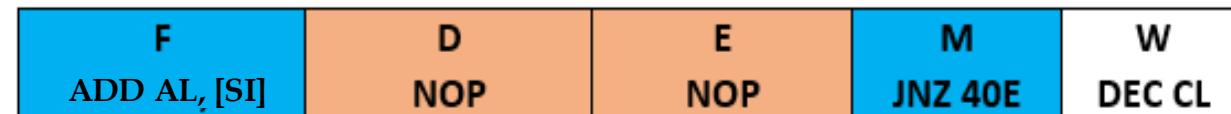
2. Control hazards

Memory	Instruction	Operation
400	MOV SI, 500	SI <- 500
403	MOV DI, 600	DI <- 600
406	MOV AX, 0000	AX = 0000
409	MOV CL, [SI]	CL <- [SI]
40B	MOV BL, CL	BL <- CL
40D	INC SI	SI = SI + 1
40E	NXT: ADD AL, [SI]	AL = AL + [SI]
410	ADC AH, 00	AH = AH + 00 + cy
412	INC SI	SI = SI + 1
413	DEC CL	CL = CL - 1
415	JNZ NXT (40E)	JUMP if ZF = 0
417	DIV BL	AX = AX / BL
419	MOV [DI], AX	[DI] <- AX
41B	HLT	Stop

- Program to find average of array elements

(a) Inserting NOP

1. Insert NOP (No operation) when Branch instruction



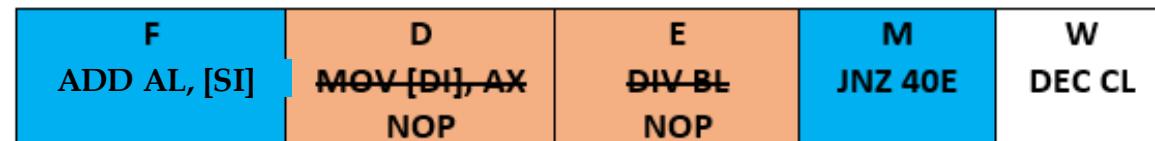
2. Control hazards

Memory	Instruction	Operation
400	MOV SI, 500	SI <- 500
403	MOV DI, 600	DI <- 600
406	MOV AX, 0000	AX = 0000
409	MOV CL, [SI]	CL <- [SI]
40B	MOV BL, CL	BL <- CL
40D	INC SI	SI = SI + 1
40E	NXT: ADD AL, [SI]	AL = AL + [SI]
410	ADC AH, 00	AH = AH + 00 + cy
412	INC SI	SI = SI + 1
413	DEC CL	CL = CL - 1
415	JNZ NXT (40E)	JUMP if ZF = 0
417	DIV BL	AX = AX / BL
419	MOV [DI], AX	[DI] <- AX
41B	HLT	Stop

- Program to find average of array elements

(b) NOP after execution of instruction

2. Insert NOP (No operation) when Branch is taken (after execution of branch)



2. Control hazards

Memory	Instruction	Operation
400	MOV SI, 500	SI <- 500
403	MOV DI, 600	DI <- 600
406	MOV AX, 0000	AX = 0000
409	MOV CL, [SI]	CL <- [SI]
40B	MOV BL, CL	BL <- CL
40D	INC SI	SI = SI + 1
40E	NXT: ADD AL, [SI]	AL = AL + [SI]
410	ADC AH, 00	AH = AH + 00 + cy
412	INC SI	SI = SI + 1
413	DEC CL	CL = CL - 1
415	JNZ NXT (40E)	JUMP if ZF = 0
417	DIV BL	AX = AX / BL
419	MOV [DI], AX	[DI] <- AX
41B	HLT	Stop

- Program to find average of array elements

(c) Branch Prediction : Branch Taken

3. Branch Prediction: Branch Taken

F ADC AH, 00	D ADD AL, [SI]	E JNZ 40E	M DEC CL	W INC SI
-----------------	-------------------	--------------	-------------	-------------

Branch not taken

F ADC AH, 00 NOP	D ADD AL, [SI] NOP	E JNZ 40E	M DEC CL	W INC SI
------------------------	--------------------------	--------------	-------------	-------------

F DIV BL	D ADC AH, 00 NOP	E ADD AL, [SI] NOP	M JNZ 40E	W DEC CL
-------------	------------------------	--------------------------	--------------	-------------

Branch Taken

F INC SI	D ADC AH, 00	E ADD AL, [SI]	M JNZ 40E	W DEC CL
-------------	-----------------	-------------------	--------------	-------------

Thank You

Reference

Textbooks and/or Reference Books:

1. Professional CUDA C Programming – John Cheng, Max Grossman, Ty McKercher, 2014
2. Wilkinson, M. Allen, "Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers", Pearson Education, 1999
3. I. Foster, "Designing and building parallel programs", 2003
4. Parallel Programming in C using OpenMP and MPI – Micheal J Quinn, 2004
5. Introduction to Parallel Programming – Peter S Pacheco, Morgan Kaufmann Publishers, 2011
6. Advanced Computer Architectures: A design approach, Dezso Sima, Terence Fountain, Peter Kacsuk, 2002
7. Parallel Computer Architecture : A hardware/Software Approach, David E Culler, Jaswinder Pal Singh Anoop Gupta, 2011
8. Introduction to Parallel Computing, Ananth Grama, Anshul Gupta, George Karypis, Vipin Kumar, Pearson, 2011

National Institute of Technology Karnataka Surathkal
Department of Information Technology



IT 301 Parallel Computing
Superscalar Processors 1

Dr. Geetha V
Assistant Professor
Dept of Information Technology
NITK Surathkal

Index

- 1. Instruction Level Parallelism
- 2: Data Hazards and Instruction Issue
- 3. Inorder Vs Out of order Execution

1: Introduction

Course Plan: Theory:

Part A: Parallel Computer Architectures

Week 1,2,3: ***Introduction to Parallel Computer Architecture:***

Parallel Computing,

Parallel architecture,

bit level, instruction level , data level and task level parallelism.

Instruction level parallelisms: pipelining (Data and control instructions),

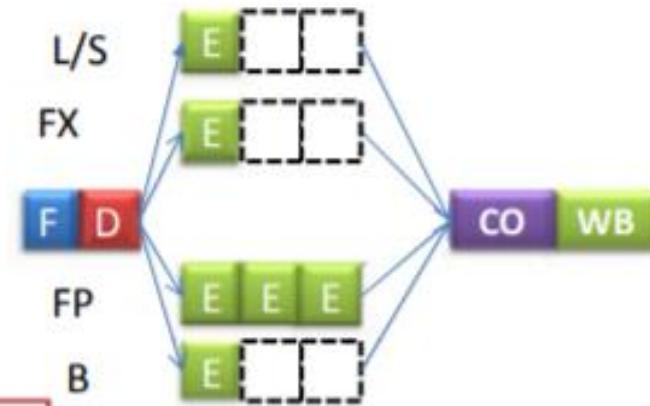
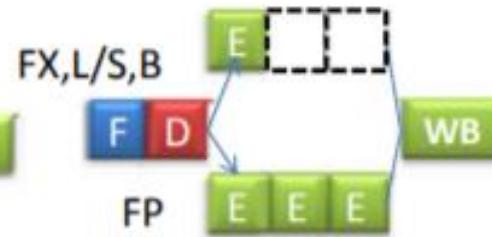
scalar processors and **superscalar processors**,

vector processors.

Parallel computers and computation.

1. Instruction level parallelism

Scheduling and Preserving Sequential Consistency



**Early RISC Processors
Without an FP unit**

**RISC I, II (1983)
MIPS (1981)
IBM 801 (1978)**

**Scalar Processors
With an FP unit**

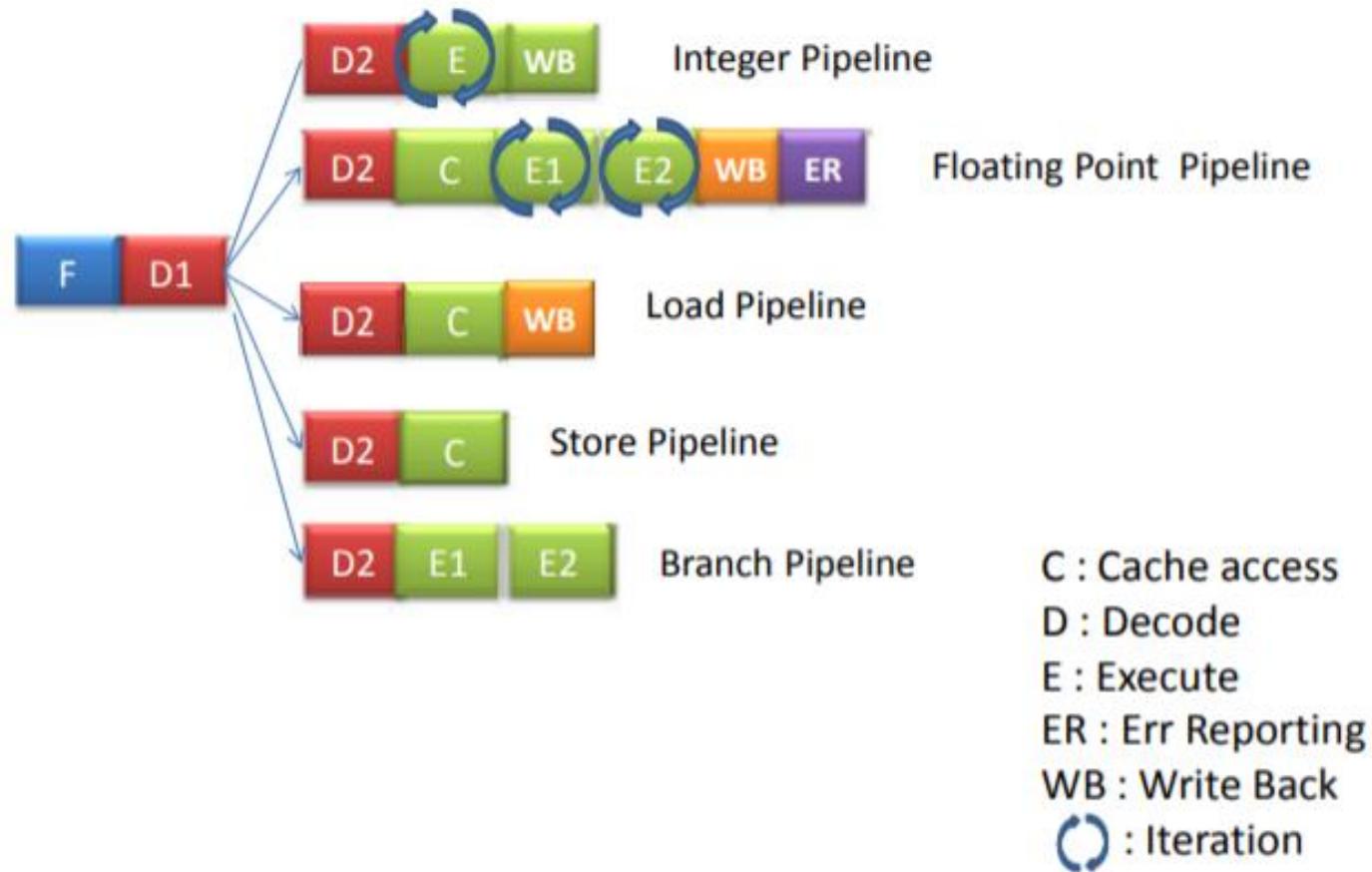
**Pentium (1993)
I486 (1988)
MIPS R 2/3000 (1988)**

Superscalar Processors

**Pentium Pro(1993)
PowerPC 603 (1993)**

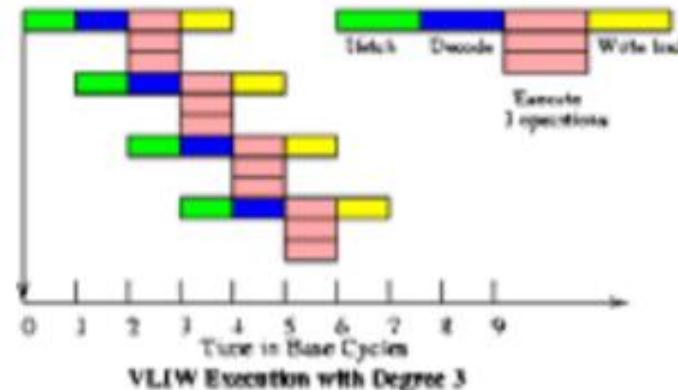
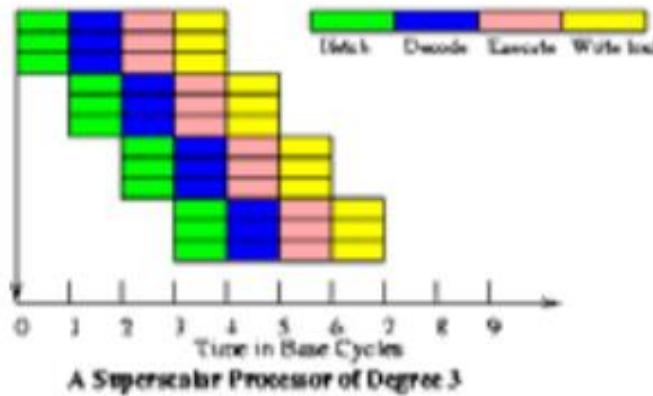
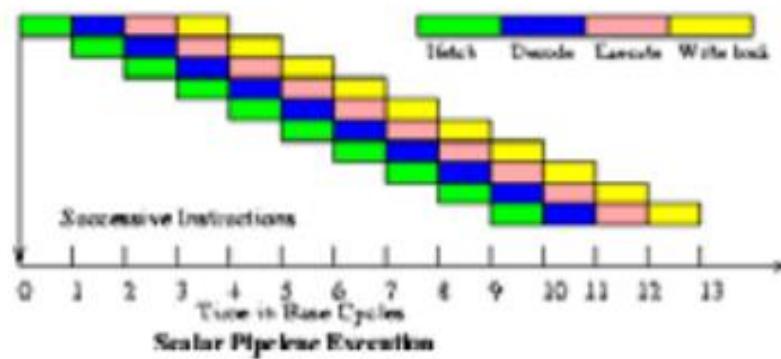
1. Instruction level parallelism

Case Study : Logical Layout of Pentium's Pipeline



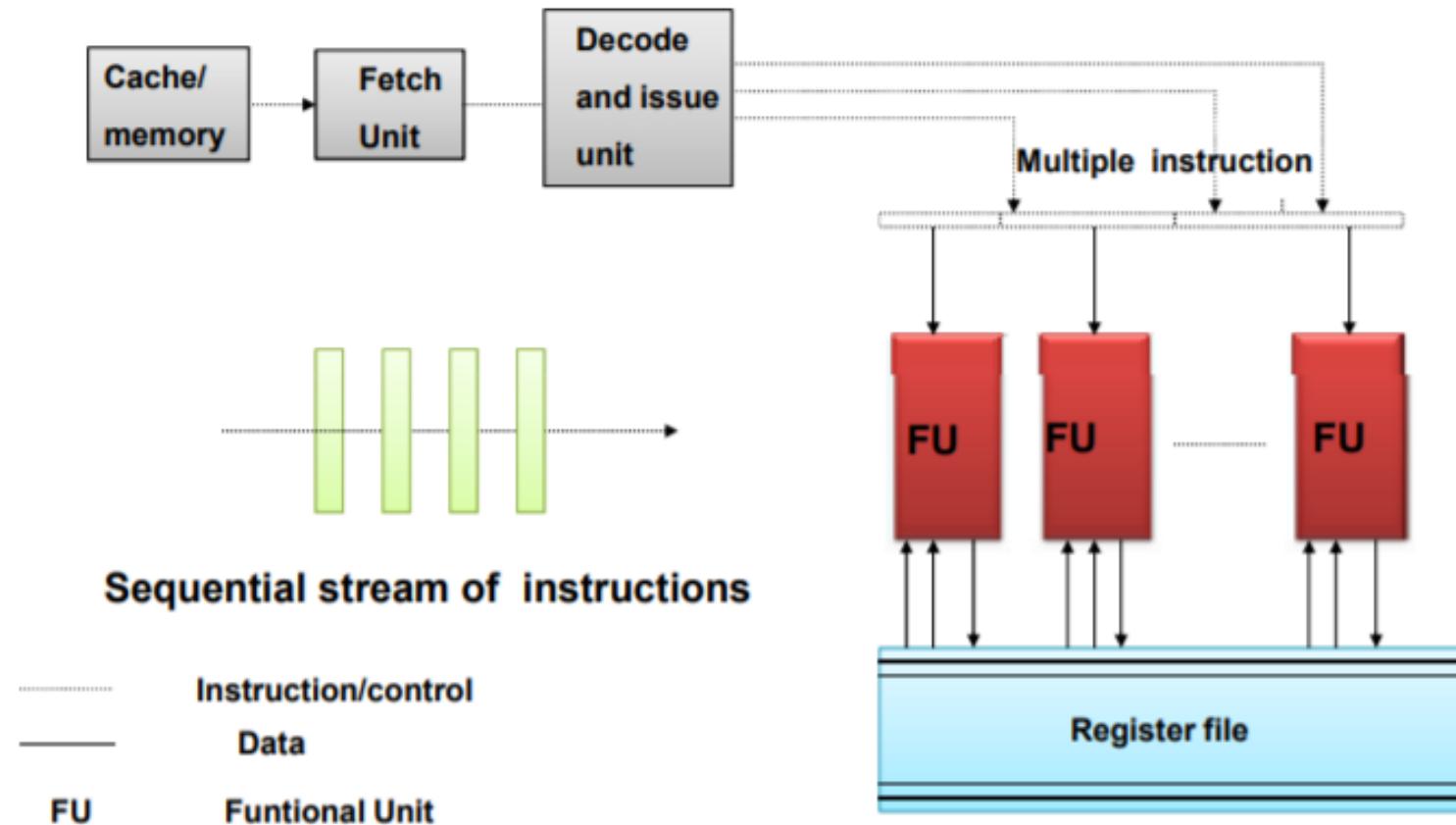
1. Instruction level parallelism

Superscalar vs VLIW



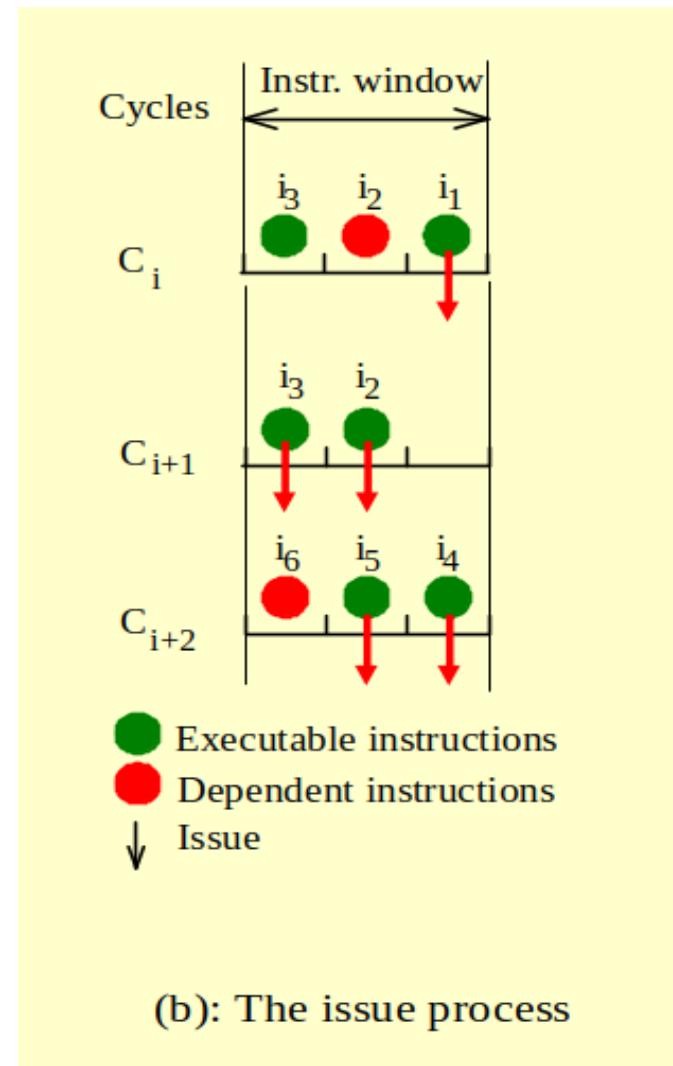
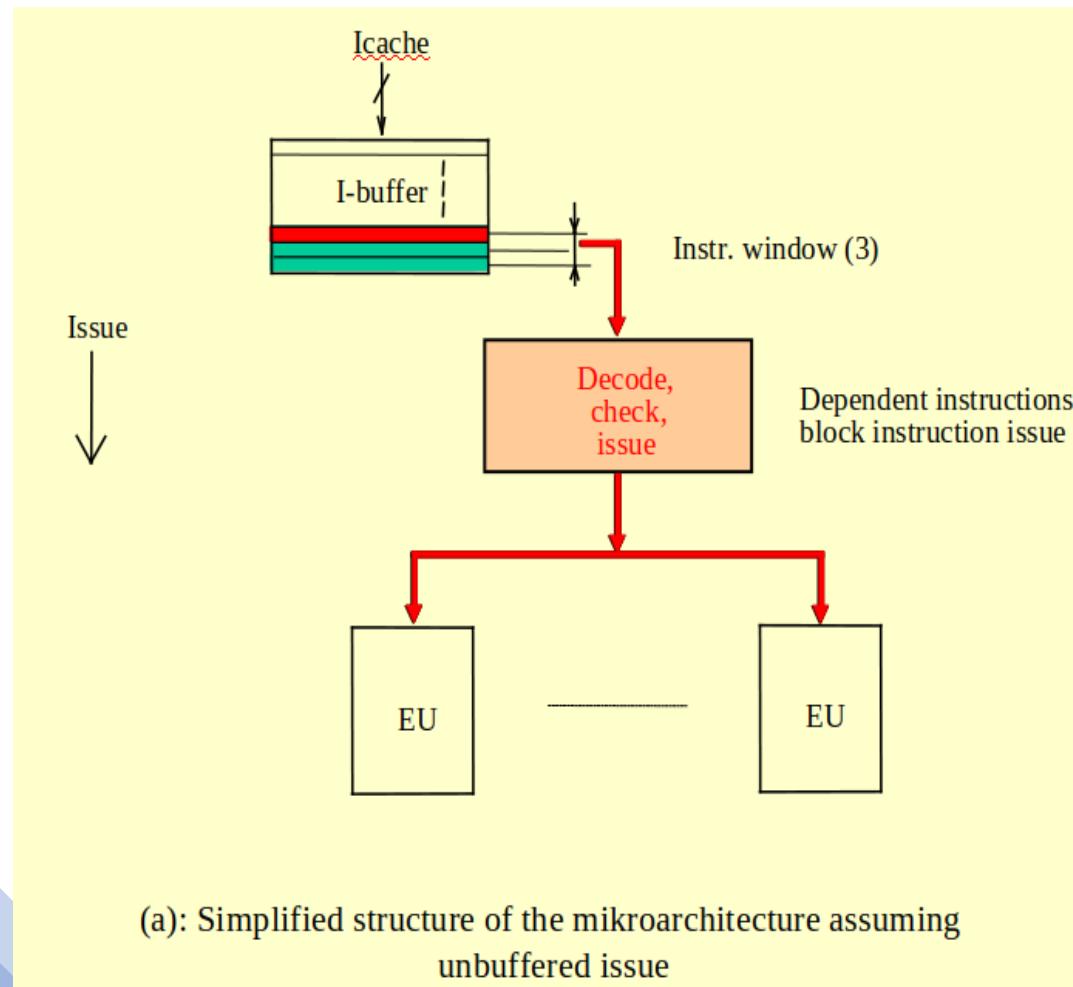
1. Instruction level parallelism

ILP in Superscalar processor



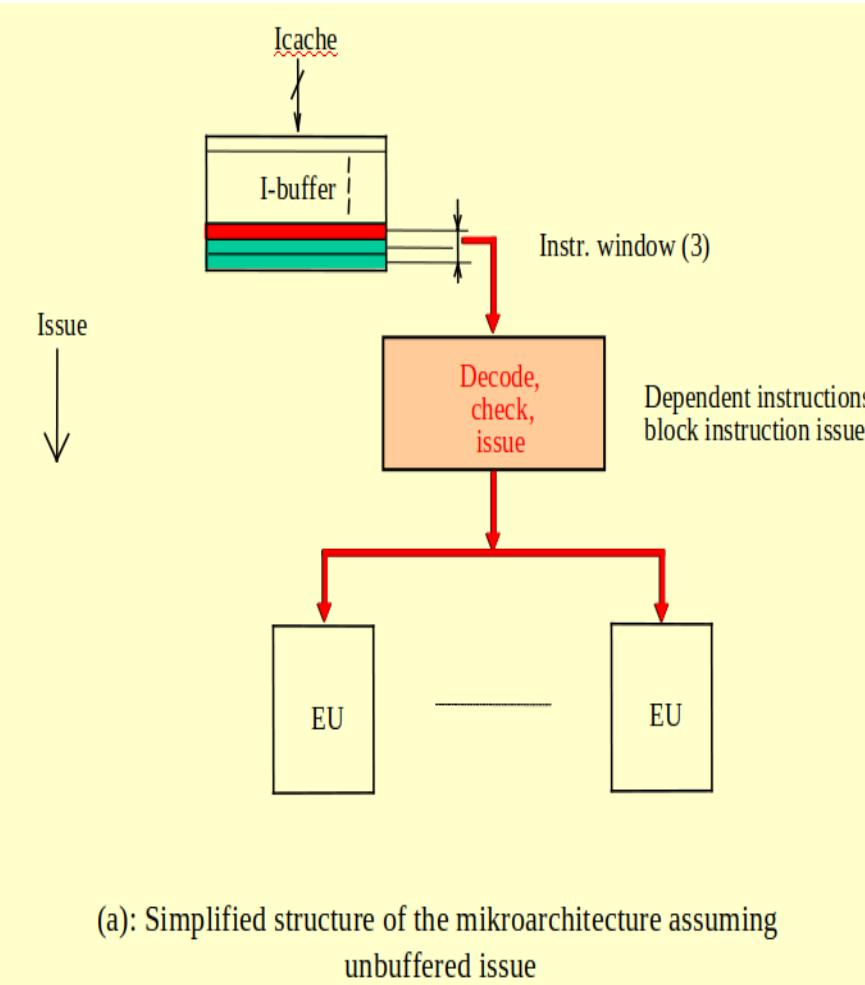
2: Data Hazard and Instruction Issue

The issue bottleneck

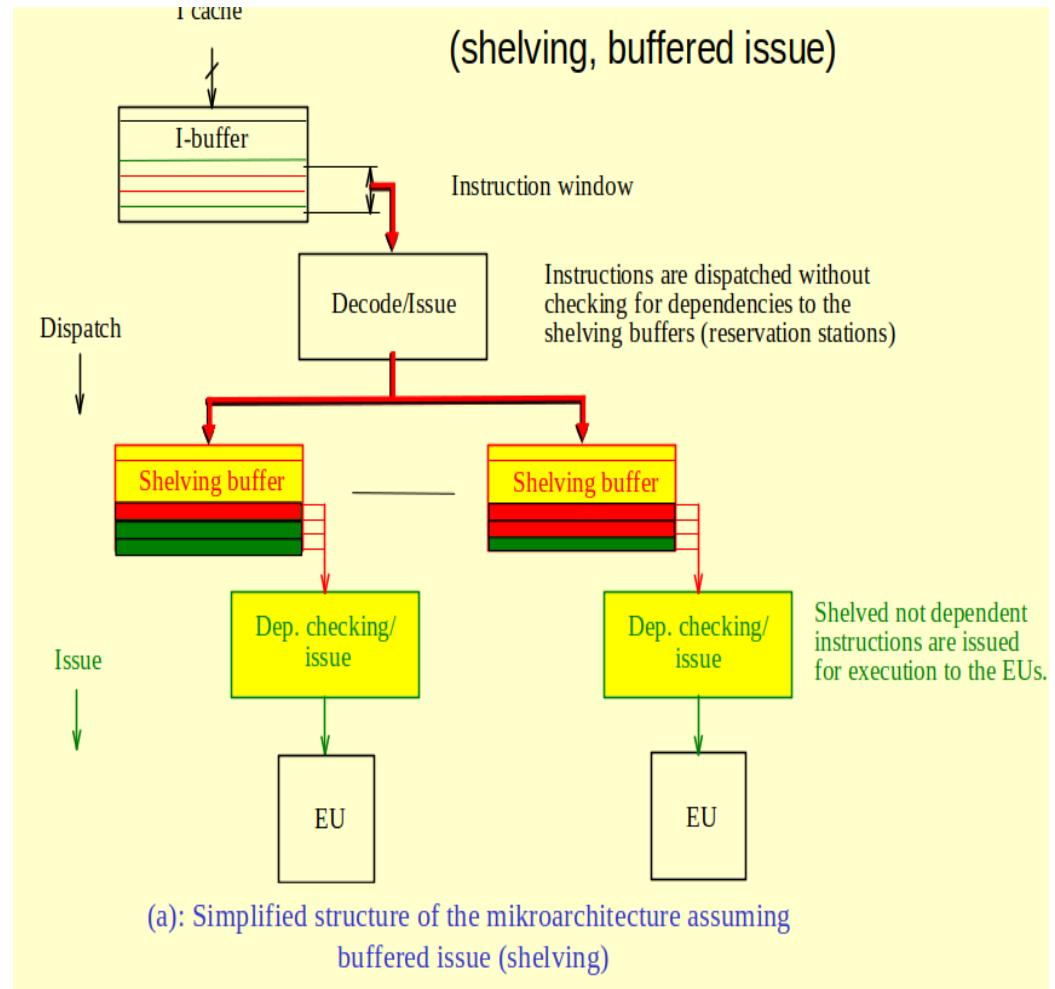


2: Data Hazard and Instruction Issue

The issue bottleneck



Dynamic instruction issue



2: Data Hazard and Instruction Issue

- **Data Dependency (Data Hazard)**

Dependency of instructions due to operand values unavailability

- **Control Dependency (Control Hazard)**

Dependency due to unresolved decision on control instructions.

2: Data Hazard and Instruction Issue

- **Data Dependency (Data Hazard)**

Dependency of instructions due to operand values unavailability

- Read After Read (RAR) : No dependency
- Read After Write (RAW) : Common Hazard
- Write After Read (WAR) : Hazard in Out of order execution
- Write After Write (WAW) : Hazard in Out of order execution

2: Data Hazard and Instruction Issue

- **Data Dependency (Data Hazard)**

Dependency of instructions due to operand values unavailability

- **Read After Read (RAR) : No dependency**

ADD R1, **R2**, R6 ; R1 <- **R2**+R6

SUB R4, **R2**, R7; R4 <- **R2**+R7

2: Data Hazard and Instruction Issue

- **Data Dependency (Data Hazard)**

Dependency of instructions due to operand values unavailability

- **Read After Write (RAW) : Common Hazard**

ADD R1, R2, R6 ; R1 <- R2+R6

SUB R4, R1, R7; R4 <- R1+R7

Solve Using Data Forwarding

2: Data Hazard and Instruction Issue

- **Data Dependency (Data Hazard)**

Dependency of instructions due to operand values unavailability

- **Write After Read (WAR)** : Not Common in in-order pipeline

ADD R1, **R2**, R3 ; R1 <- R2+R3

SUB **R2**, R4, R5; R2 <- R4+R5

For Out of Order execution, Use dependency check for execution

2: Data Hazard and Instruction Issue

- **Data Dependency (Data Hazard)**

Dependency of instructions due to operand values unavailability

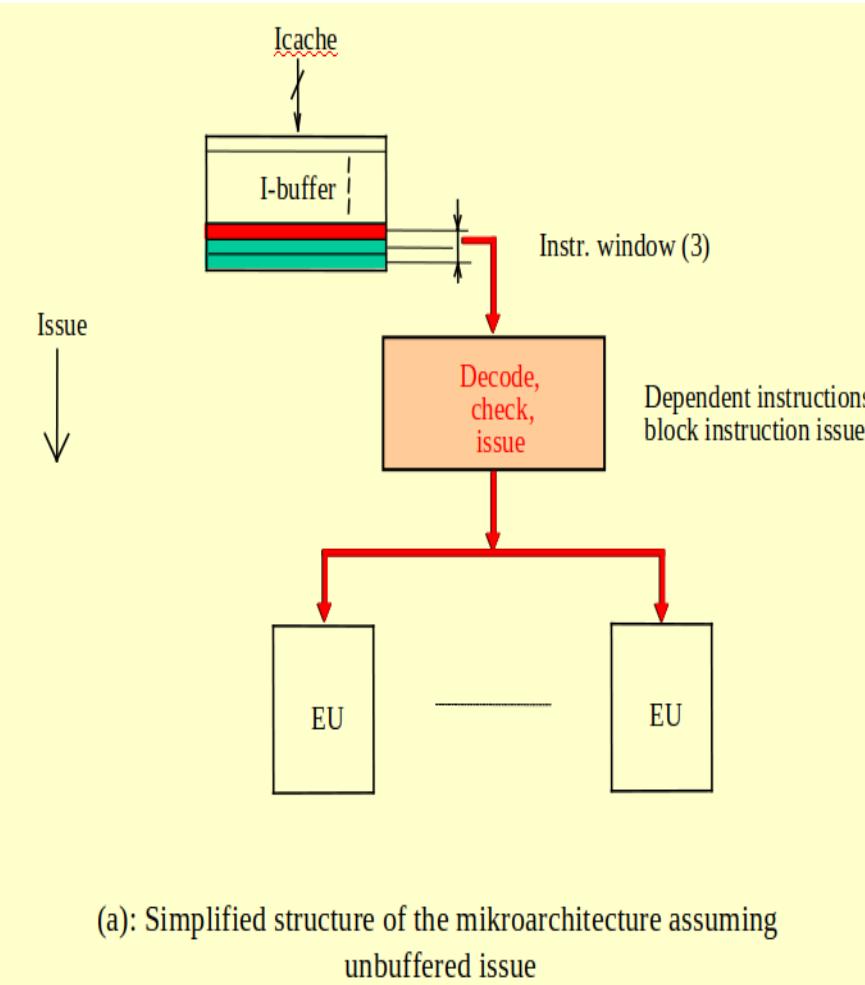
- **Write After Write (WAW) :** Not hazard in in-order pipeline

ADD **R1**, R2, R3 ; R1 <- R2+R3

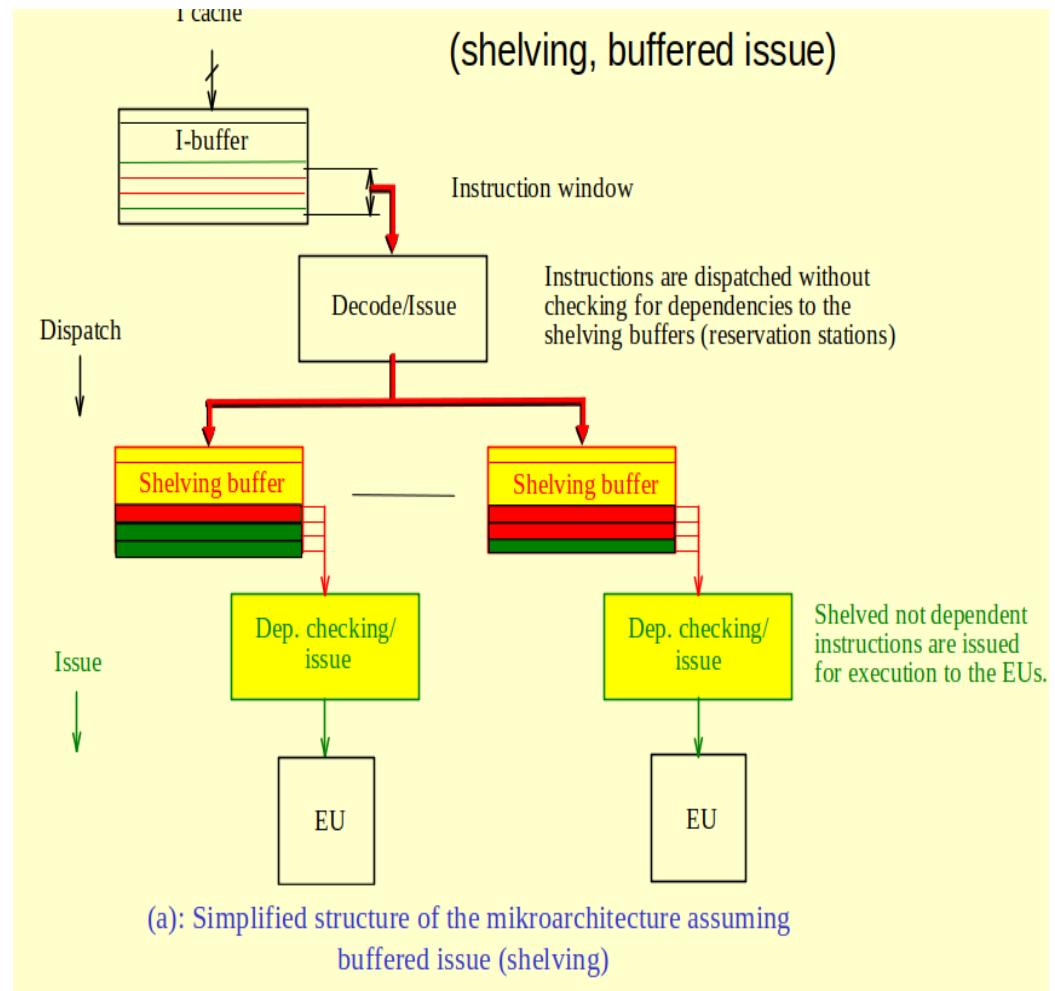
SUB **R1**, R4, R5; R1 <- R4+R5

2: Data Hazard and Instruction Issue

The issue bottleneck



Dynamic instruction issue



3. Inorder Vs Out of order execution

- **In-order instruction execution**

- Instructions are fetched, executed and completed in compiler generated order
- One instruction stall, stalls all other instructions
- Instructions are **Statistically Scheduled**

- **Out-of-order instruction execution**

- Instructions are fetched in compiler-generated order
- Instruction completion may be in-order or out-of-order
- Independent instruction behind a stalled instruction can pass it
- Instructions are **Dynamically executed**

3. Inorder Vs Out of order execution

- **Out-of-order procesors**

- After instruction decode

- Check for **structural hazards**

- An instruction can be issued when a functional unit is available
- An instruction stalls if no appropriate functional unit

- Check for **Data Hazards**

- An instruction can execute when its operands have been calculated or loaded from memory
- An instruction stalls if operands are not available.

3. Inorder Vs Out of order execution

- **Out-of-order processors**

- After instruction decode

- Independent ready instructions can execute before earlier instructions that are stalled

in-order processors

lw \$3, 100(\$4)

in execution, cache miss

add \$2, \$3, \$4

waits until the miss is satisfied

sub \$5, \$6, \$7

waits for the add

out-of-order processors

lw \$3, 100(\$4)

in execution, cache miss

sub \$5, \$6, \$7

can execute during the cache miss

add \$2, \$3, \$4

waits until the miss is satisfied

3. Inorder Vs Out of order execution

- **Out-of-order procesors**

- After instruction decode
 - Independent ready instructions can execute before earlier instructions that are stalled
 - When path instructions are waiting for a branch condition to be computed
 - The instructions that are issued from the predicted path are issued speculatively, called speculative execution.
 - Speculative isntructions are execute (but not commit) before the branch is resolved
 - If the prediction was wrong, speculative instructions are flushed from the pipeline
 - If prediction is right, instructions are no longer speculative.

3. Inorder Vs Out of order execution

- **Speculative Execution**

- Executing an instruction before it is known that it should be executed
 - All instructions that are fetched because of a prediction are speculative
 - Inorder pipeline : branch is executed before the path
 - Out of order pipeline
 - Path can be executed before the branch
 - Speculative instructions can execute but not committed
 - Getting rid of wrong path instructions is not just a matter of flushing them from the pipeline.

Thank You

Reference

Textbooks and/or Reference Books:

1. Professional CUDA C Programming – John Cheng, Max Grossman, Ty McKercher, 2014
2. Wilkinson, M. Allen, "Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers", Pearson Education, 1999
3. I. Foster, "Designing and building parallel programs", 2003
4. Parallel Programming in C using OpenMP and MPI – Micheal J Quinn, 2004
5. Introduction to Parallel Programming – Peter S Pacheco, Morgan Kaufmann Publishers, 2011
6. Advanced Computer Architectures: A design approach, Dezso Sima, Terence Fountain, Peter Kacsuk, 2002
7. Parallel Computer Architecture : A hardware/Software Approach, David E Culler, Jaswinder Pal Singh Anoop Gupta, 2011
8. Introduction to Parallel Computing, Ananth Grama, Anshul Gupta, George Karypis, Vipin Kumar, Pearson, 2011