



Master of Computer Applications

23MCAC105 – Advanced Computer Architecture

Credits: 3

L: T: P – 3-0-0

Prepared by
Dr. A. Rengarajan, Professor

Module – 4

PIPELINING and ILP

- Fundamentals of Computer Design
- Measuring and Reporting Performance
- Instruction Level Parallelism and Its Exploitation
- Concepts and Challenges
- Overcoming Data Hazards with Dynamic Scheduling
- Dynamic Branch Prediction

Fundamentals of Computer Design ^{1/5}

- ✓ Performance improvement due to
 - ❑ Advances in the technology
 - ❑ Innovation in computer design
- ✓ Before 1970's: above the concept made a major contribution to improve the performance
- ✓ 1970 – 1975: 25% to 30% per year performance improvement for the mainframes and minicomputers.
- ✓ After 1975,s: 35% per year performance improvement for microprocessors simply due to advances in the technology

Fundamentals of Computer Design ^{2/5}

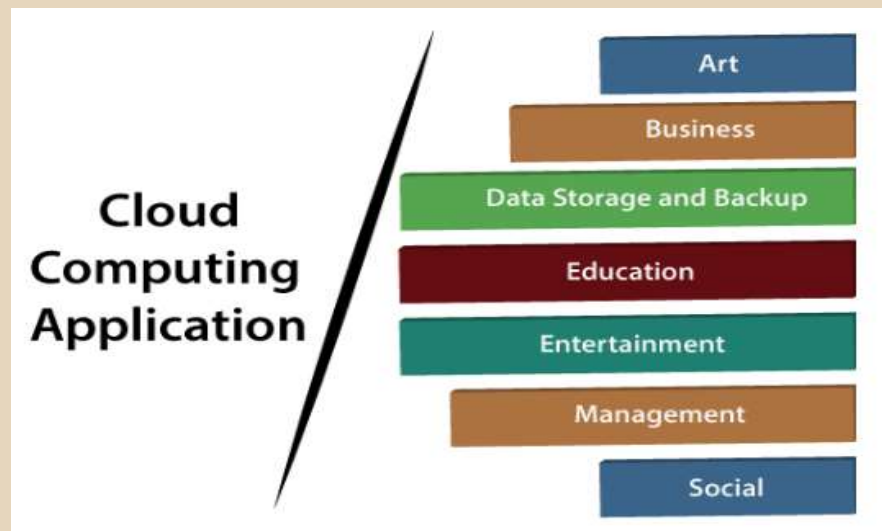
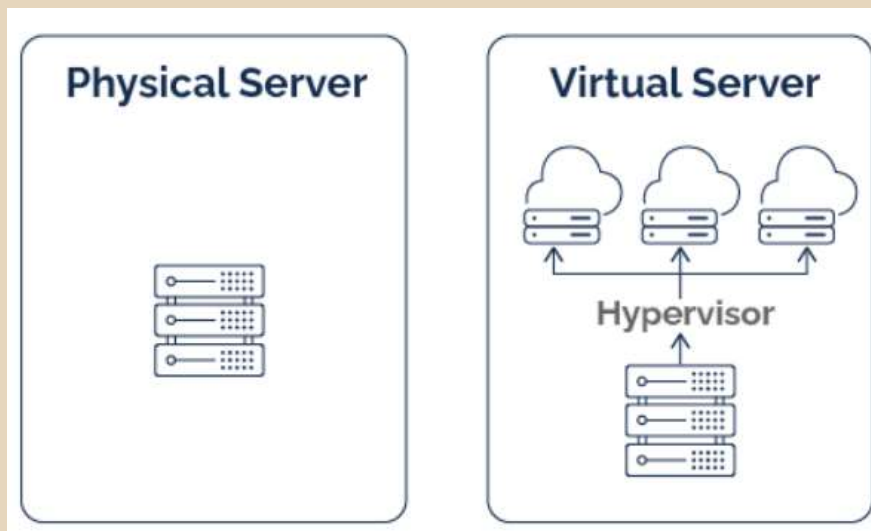
✓ Changing face of Computing

- ❑ 1960's: Large Mainframes (Business data processing and scientific computing)
- ❑ 1970's: Mini Computers (Time sharing)
- ❑ 1980's: Desktop computers (PC)
- ❑ 1990's: Internet and WWW (Servers)
- ❑ 2000's: Embedded computing, Mobile computing, pervasive computing
- ❑ 2010's: Cloud computing
- ❑ 2012's: Ubiquitous computing (IoT)
- ❑ Recent: Quantum computing (AI, ML etc.,)

Fundamentals of Computer Design 3/5

✓ Cloud computing

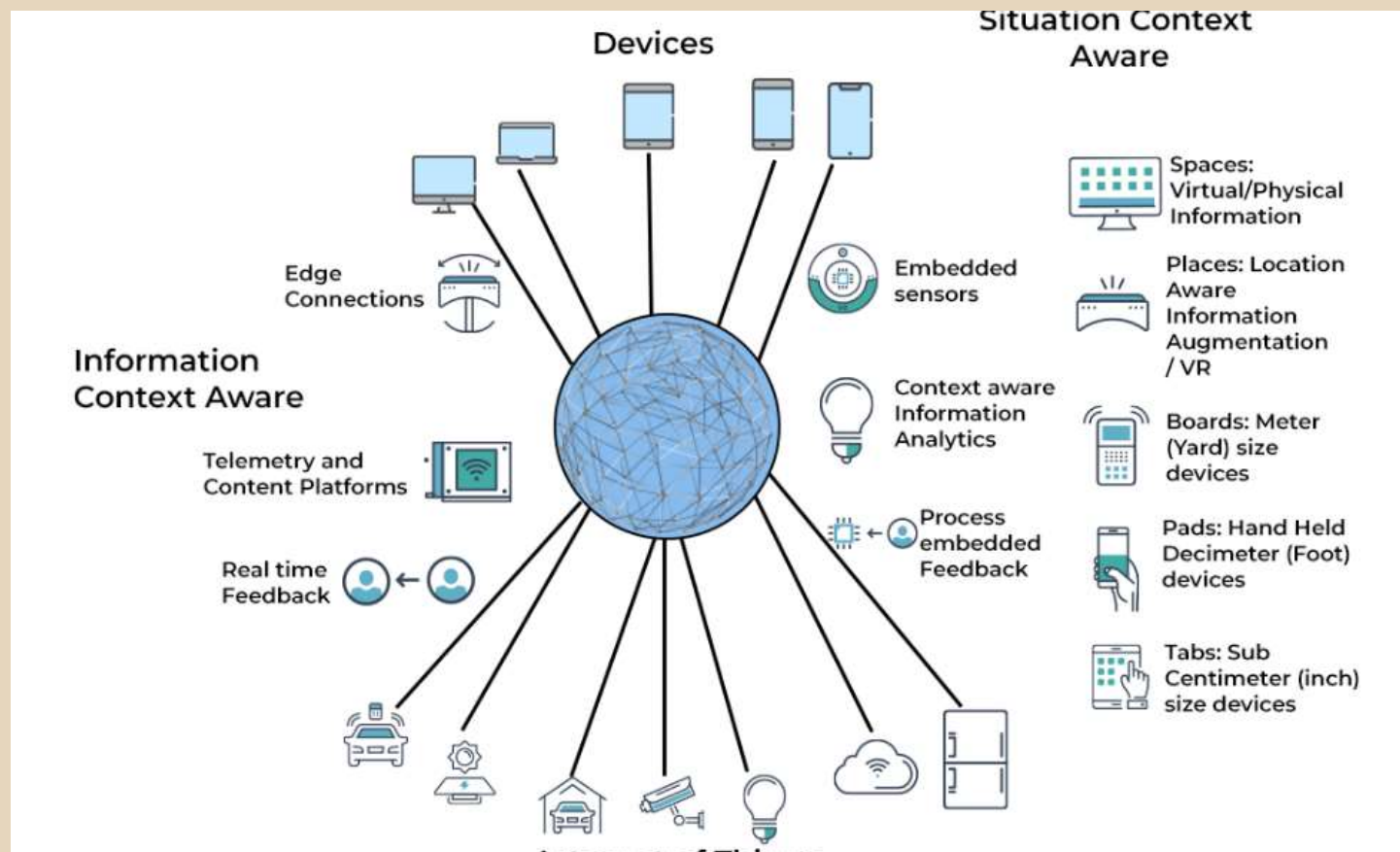
Cloud server	Physical server
A virtual or bare-metal server that a cloud provider hosts on its own infrastructure and delivers to users over a network	A physical server that an enterprise purchases, installs and maintains in its own data center
Best for enterprises that have variable workloads	Best for enterprises with predictable, demanding or sensitive workloads
Providers include AWS, Azure and Google	Providers include Dell EMC, HPE and IBM



Adobe Creative Cloud, Paypal, Google G Suite, Google Apps for Education, Online games, Video Conferencing, Facebook, Twitter, LinkedIn

Fundamentals of Computer Design ^{4/5}

✓ Ubiquitous computing (IoT)

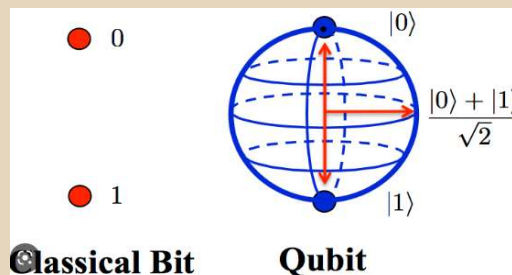


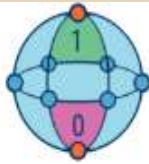
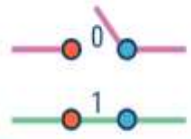



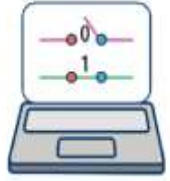
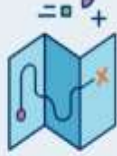

Laptops, notebooks, smartphones, tablets, wearable devices, and sensors are ubiquitous computing devices

Fundamentals of Computer Design 5/5

✓ Quantum computing

- ❖ In quantum computing, a **qubit** or **quantum bit** is a basic unit of quantum information—the quantum version of the classic binary bit physically realized with a two-state device.
- ❖ A qubit is a two-state (or two-level) quantum-mechanical system, one of the simplest quantum systems displaying the peculiarity of quantum mechanics.



 <p>Calculates with qubits, which can represent 0 and 1 at the same time</p>	 <p>Calculates with transistors, which can represent either 0 or 1</p>
 <p>Power increases exponentially in proportion to the number of qubits</p>	 <p>Power increases in a 1:1 relationship with the number of transistors</p>
 <p>Quantum computers have high error rates and need to be kept ultracold</p>	 <p>Classical computers have low error rates and can operate at room temp</p>
 <p>Well suited for tasks like optimization problems, data analysis, and simulations</p>	 <p>Most everyday processing is best handled by classical computers</p>

Pipelining

- ✓ Pipelining is a general-purpose efficiency technique
 - It is not specific to processors

- ✓ Pipelining is used in:
 - Assembly lines
 - Bucket groups
 - Fast food restaurants

- ✓ Pipelining is used in other CS disciplines:
 - Networking
 - Server software architecture

- ✓ Useful to increase throughput in the presence of long latency
 - More on that later...

Pipelining Processors

- ✓ We've seen two possible implementations of the MIPS architecture.
 - A **single-cycle datapath** executes each instruction in just one clock cycle, but the cycle time may be very long.
 - A **multicycle datapath** has much shorter cycle times, but each instruction requires many cycles to execute.
- ✓ **Pipelining** gives the best of both worlds and is used in just about every modern processor.
 - Cycle times are short so clock rates are high.
 - But we can still execute an instruction in about one clock cycle!

Single Cycle Datapath	CPI = 1	Long Cycle Time
Multi-cycle Datapath	CPI = ~4	Short Cycle Time
Pipelined Datapath	CPI = ~1	Short Cycle Time

Instruction execution review

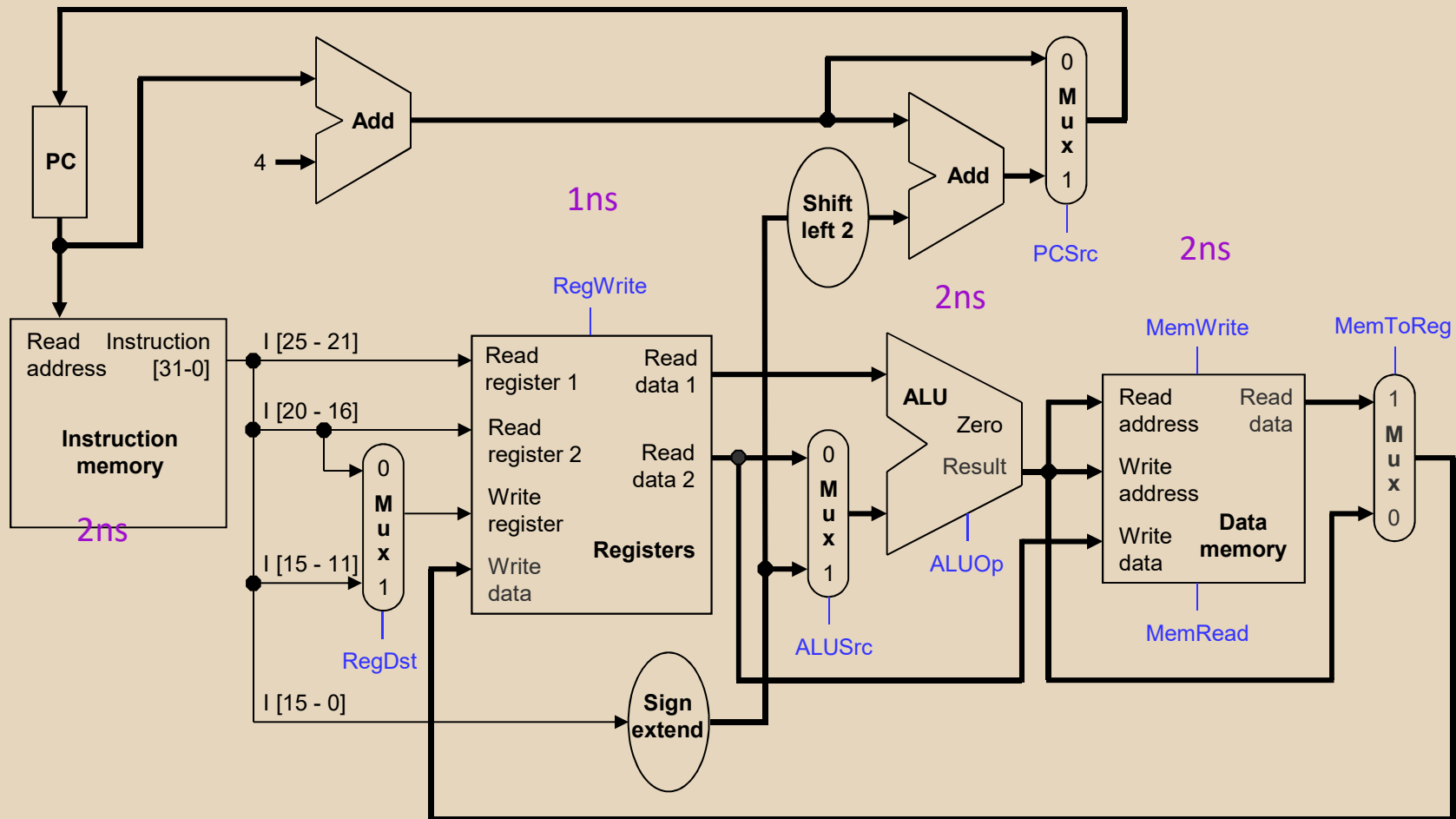
- ✓ Executing a MIPS instruction can take up to five steps.

Step	Name	Description
Instruction Fetch	IF	Read an instruction from memory.
Instruction Decode	ID	Read source registers and generate control signals.
Execute	EX	Compute an R-type result or a branch outcome.
Memory	MEM	Read or write the data memory.
Writeback	WB	Store a result in the destination register.

- ✓ However, as we saw, not all instructions need all five steps.

Instruction	Steps required				
beq	IF	ID	EX		
R-type	IF	ID	EX		WB
sw	IF	ID	EX	MEM	
lw	IF	ID	EX	MEM	WB

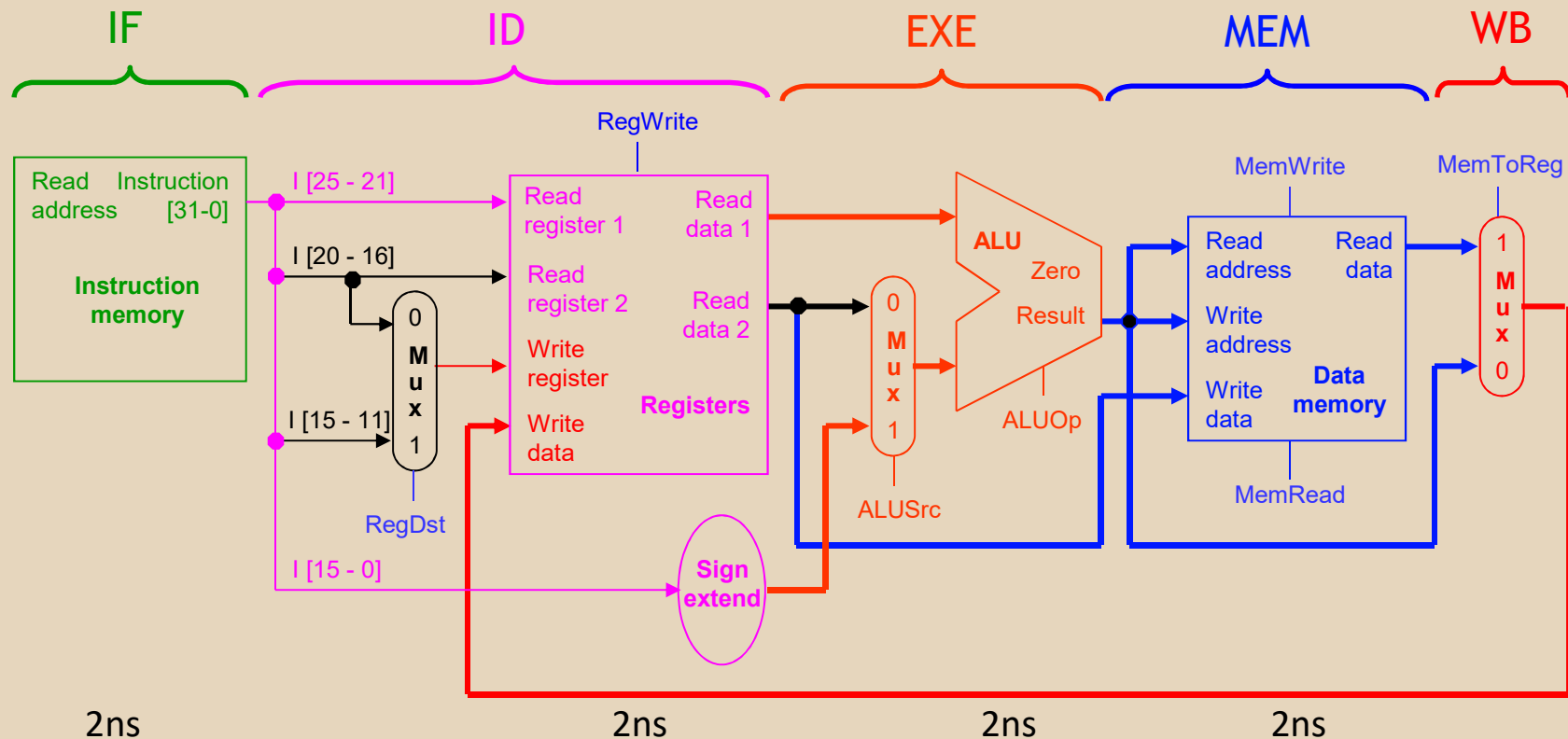
Single-cycle datapath diagram



- How long does it take to execute each instruction?

Break datapath into 5 stages

- ✓ Each stage has its own functional units.
- ✓ Each stage can execute in 2ns
 - Just like the multi-cycle implementation



Pipelining Loads

	Clock cycle								
	1	2	3	4	5	6	7	8	9
lw \$t0, 4(\$sp)	IF	ID	EX	MEM	WB				
lw \$t1, 8(\$sp)		IF	ID	EX	MEM	WB			
lw \$t2, 12(\$sp)			IF	ID	EX	MEM	WB		
lw \$t3, 16(\$sp)				IF	ID	EX	MEM	WB	
lw \$t4, 20(\$sp)					IF	ID	EX	MEM	WB

- ✓ Execution time on ideal pipeline:
 - time to fill the pipeline + one cycle per instruction
 - What is the execution time for N instructions?
- ✓ Compare with other implementations:
 - Single Cycle: (8ns clock period)
- ✓ How much faster is pipelining for N=1000 ?

Pipeline – Summary

- ✓ Pipelining attempts to maximize instruction throughput by overlapping the execution of multiple instructions.
- ✓ Pipelining offers amazing speedup.
 - In the best case, one instruction finishes on every cycle, and the speedup is equal to the pipeline depth.
- ✓ The pipeline datapath is much like the single-cycle one, but with added pipeline registers
 - Each stage needs its own functional units

Measuring and Reporting Performance

- ✓ Benchmark programs should be derived from how actual applications will execute. However, performance is often the result of combined characteristics of a given computer architecture.
- ✓ The system software/hardware components in addition to the microprocessor. Other factors such as the operating system, compilers, libraries, memory design and I/O subsystem characteristics may also have impacts on the results and make comparisons difficult.

Measuring Performance

- ✓ Two ways to measure the performance are:
 - The speed measure - which measures how fast a computer completes a single task. For example, the SPECint95 is used for comparing the ability of a computer to complete single tasks.
 - The throughput measure - which measures how many tasks a computer can complete in a certain amount of time. The SPECint_rate95 measures the rate of a machine carrying out a number of tasks.

•
The **speed** of your data is more of an appreciation of how fast data is being sent. It isn't very precise. **Bandwidth** refers to the theoretical rate of speed that data on your network can travel, which is probably a speed you won't see very often. **Throughput** is the actual rate of speed that data on your network travels.

Interpreting Results

- ✓ There are three important guidelines to remember when interpreting benchmark results:
 1. Be aware of what is being measured.
 2. Representativeness is key.
 3. Avoid single-measure metrics.

Reporting Performance

- ✓ There are some points to remember when reporting results obtained from running benchmarks.
 1. Use newer version over the older.
 2. Use all programs in a suite.
 3. Report compilation mode.
 4. Use a variety of benchmarks when reporting performance.
 5. List all factors affecting performance. Have enough information about performance measurements to allow readers to duplicate the results. These include: program input, version of the program, version of compiler, optimizing level of compiled code, version of operating system, amount of main memory, number and types of disks, version of the CPU

Instruction Level Parallelism (ILP)

- ✓ Almost all processors since 1985 use pipelining to overlap the execution of instructions and improve performance. This potential overlap among instructions is called instruction level parallelism
- ✓ First introduced in the IBM Stretch (Model 7030) in about 1959
- ✓ Later the CDC 6600 incorporated pipelining and the use of multiple functional units
- ✓ The Intel i486 was the first pipelined implementation of the IA32 architecture

Instruction Level Parallelism (ILP)

- ✓ Instruction level parallel processing is the concurrent processing of multiple instructions
- ✓ Difficult to achieve within a basic code block
 - Typical MIPS programs have a dynamic branch frequency of between 15% and 25%
 - That is, between three and six instructions execute between a pair of branches, and data hazards usually exist within these instructions as they are likely to be dependent
- ✓ Given basic code block size in number of instructions, ILP must be exploited across multiple blocks
- ✓ The current trend is toward very deep pipelines, increasing from a depth of < 10 to > 20 .

LLP exploitation among Iterations of a loop

- ✓ Loop adding two 1000 element arrays

- Code

```
for (i=1; i<= 1000; i=i+1)
    x[i] = x[i] + y[i];
```

- ✓ If we look at the generated code, within a loop there may be little opportunity for overlap of instructions, but each iteration of the loop can overlap with any other iteration

Concepts and Challenges Approaches to Exploiting ILP

- ✓ Two major approaches
 - **Dynamic** – these approaches depend upon the hardware to locate the parallelism
 - **Static** – fixed solutions generated by the compiler, and thus bound at compile time
- ✓ These approaches are not totally disjoint, some requiring both
- ✓ Limitations are imposed by data and control hazards

Dynamic Approach

- ✓ Hardware intensive approach
- ✓ Dominate desktop and server markets
 - Pentium III, 4, Athlon
 - MIPS R10000/12000
 - Sun UltraSPARC III
 - PowerPC 603, G3, G4
 - Alpha 21264

Static Approach

- ✓ Compiler intensive approach
- ✓ Embedded market and IA-64

Terminology and Ideas

- ✓ Cycles Per Instruction
 - $\text{Pipeline CPI} = \text{Ideal Pipeline CPI} + \text{Structural Stalls} + \text{Data Hazard Stalls} + \text{Control Stalls}$
- ✓ Ideal Pipeline CPI is the max that we can achieve in a given architecture. Stalls and/or their impacts must be minimized.
- ✓ During 1980s $\text{CPI} = 1$ was a target objective for single chip microprocessors
- ✓ 1990's objective: reduce CPI below 1
 - *Scalar* processors are pipelined processors that are designed to fetch and issue at most one instruction every machine cycle
 - *Superscalar* processors are those that are designed to fetch and issue multiple instructions every machine cycle

Approaches to Exploiting ILP that we will explore

Technique	Reduces
Forwarding and bypassing	Potential data hazards and stalls
Delayed branches and simple branch scheduling	Control hazard stalls
Basic dynamic scheduling (score boarding)	Data hazard stalls from true dependences
Dynamic scheduling with renaming	Data hazard stalls and stalls from antidependences and output dependences
Branch prediction	Control stalls
Issuing multiple instructions per cycle	Ideal CPI
Hardware Speculation	Data hazard and control hazard stalls
Dynamic memory disambiguation	Data hazard stalls with memory
Loop unrolling	Control hazard stalls
Basic computer pipeline scheduling	Data hazard stalls
Compiler dependence analysis, software pipelining, trace scheduling	Ideal CPI, data hazard stalls
Hardware support for Compiler speculation	Ideal CPI, data, control stalls.

Pipeline Hazards

- ✓ Hazards make it necessary to *stall* the pipeline.
 - Some instructions in the pipeline are allowed to proceed while others are delayed
 - For this example pipeline approach, when an instruction is stalled, all instructions further back in the pipeline are also stalled
 - No new instructions are fetched during the stall
 - Instructions issued earlier in the pipeline must continue

Data Hazard

- ✓ **Data Dependences** – an instruction j is data dependent on instruction i if either of the following holds
 - Instruction i produces a result that may be used by instruction j
 - Instruction j is data dependent on instruction k , and instruction k is data dependent on instruction i – that is, one instruction is dependent on another if there exists a chain of dependencies of the first type between two instructions.

Data Hazard – Other Reason

- ✓ How much parallelism exists in a program and how it can be exploited
- ✓ If two instructions are *parallel*, they can execute simultaneously in a pipeline without causing any stalls (assuming no structural hazards exist)
- ✓ There are no dependencies in parallel instructions
- ✓ If two instructions are not parallel and must be executed in order, they may often be partially overlapped.

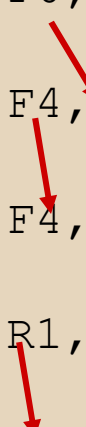
Data Hazard – Example

✓ Data Dependences –

– Code Example

```

LOOP:   L.D      F0, 0(R1)           ;F0=array element
        ADD.D    F4, F0, F2          ;add scalar in F2
        S.D      F4, 0(R1)          ;store result
        DADDUI   R1, R1, #-8         ;decrement pointer 8
        BNE      R1, R2, LOOP;
  
```



- ✓ The above dependencies are in floating point data for the first two arrows, and integer data in the last two instructions
- ✓ If two instructions are dependent, they cannot be simultaneously executed or be completely overlapped

Overcoming Data Hazards

✓ Two Ways

1. Maintain dependence but avoid the hazard
 - Schedule the code dynamically
2. Transform the code

Name dependences: Two Categories

- ✓ Two instructions use the same register or memory location, called a name, but there is actually no flow of data between the instructions associated with that name. In cases where i precedes j .
 - 1. An **anti dependence** between instructions i and j occurs when instruction j writes a register or memory location that instruction i reads. The original ordering must be preserved
 - 2. An **output dependence** occurs when instruction i and instruction j write the same register or memory location, the order again must be preserved

Name dependences: Two Categories

- ✓ 1. An anti dependence
 - i DADD R1,R2.#-8
 - j DADD R2,R5,0
- ✓ 2. An output dependence
 - i DADD R1,R2.#-8
 - j DADD R1,R4,#10

Name dependences

- ✓ Not true data dependencies, and therefore we could execute them simultaneously or reorder them if the name (register or memory location) used in the instructions is changed so that the instructions do not conflict
- ✓ Register renaming is easier
 - i DADD R1,R2,#-8
 - j DADD R2,R4,#10
 - i DADD R1,R2,#-8
 - j DADD R5,R4,#10

Data Hazards – Types

- ✓ Two instructions i and j, with i occurring before j in program order, possible hazards are:
 - RAW (read after write) – j tries to read a source before i writes it, so j incorrectly gets the old value
 - The most common type
 - Program order must be preserved
 - In a simple common static pipeline a load instruction followed by an integer ALU instruction that directly uses the load result will lead to a RAW hazard

Data Hazards – Types

✓ Second type:

- WAW (write after write) – j tries to write an operand before it is written by i, with the writes ending up in the wrong order, leaving value written by i
 - Output dependence
 - Present in pipelines that write in more than one pipe or allow an instruction to proceed even when a previous instruction is stalled
 - In the classic example, WB stage is used for write back, this class of hazards avoided.
 - If reordering of instructions is allowed this is a possible hazard
 - Suppose an integer instruction writes to a register after a floating point instruction does

Data Hazards – Types

✓ Third type:

- WAR (write after read) – j tries to write an operand before it is read by i, so i incorrectly gets the new value.
 - Anti dependence
 - Cannot occur in most static pipelines – note that reads are early in ID and writes late in WB

Dynamic Branch Prediction

- ✓ Branch prediction buffer
 - Simplest scheme
 - A small memory indexed by the lower portion of the address of the branch instruction
 - Includes a bit that says whether the branch was taken recently or not
 - No other tags
 - Useful only to reduce the branch delay when it is longer than the time to compute the possible target PCs
 - Since we only use low order bits, some other branch instruction could have set the tag
 - The prediction is a hint that is assumed to be correct, if it turns out wrong, the prediction bit is inverted and stored back

Dynamic Branch Prediction

- ✓ Branch prediction buffer is a cache
- ✓ The 1 bit scheme has a shortcoming
 - Even if a branch is almost always taken, we will usually predict incorrectly twice, rather than once, when it is not taken
 - Consider a loop branch that is taken nine times in a row then not taken. What is the prediction accuracy for this branch, assuming the prediction bit for this branch remains in the prediction buffer
 - Mispredict on the first and last predictions, as the loop branch was not taken on the first one as is set to 0. Then on the last loop it will not be taken and the prediction will be wrong again.
 - Down to 80% accuracy here

Dynamic Branch Prediction

- ✓ To remedy this situation, 2 bit branch prediction schemes are often used. A prediction must miss twice before it is changed.
- ✓ A specialization of a more general scheme that has a n-bit saturating counter for each entry in the prediction buffer. With n bits, we can take on the values 0 to 2^n-1 . When the counter is $\geq \frac{1}{2}$ of its max value, branch is predicted as taken
- ✓ Count is incremented on a taken branch and decremented on a not taken one
- ✓ 2 bits work almost as well as larger numbers

Dynamic Branch Prediction

The States in a 2 Bit Prediction Scheme

