# Computer Technology and Parallelism

## Computer Technology

- Performance improvements:
  - Improvements in semiconductor technology
    - Feature size, clock speed
  - Improvements in computer architectures
    - Enabled by HLL compilers, UNIX
    - Lead to RISC architectures
  - Improvement in cost-performance
    - Mobile devices, Personal computers and Workstations

  - Together have enabled:
    - Lightweight computers
    - Enhanced the capability available to computer users
    - Software development: Productivity-based managed/interpreted programming languages (Java, C#, Python etc) instead of performance oriented languages (C and C++)
    - Change in the nature of applications: Speech, sound, images, video, text and many more unstructured data

2

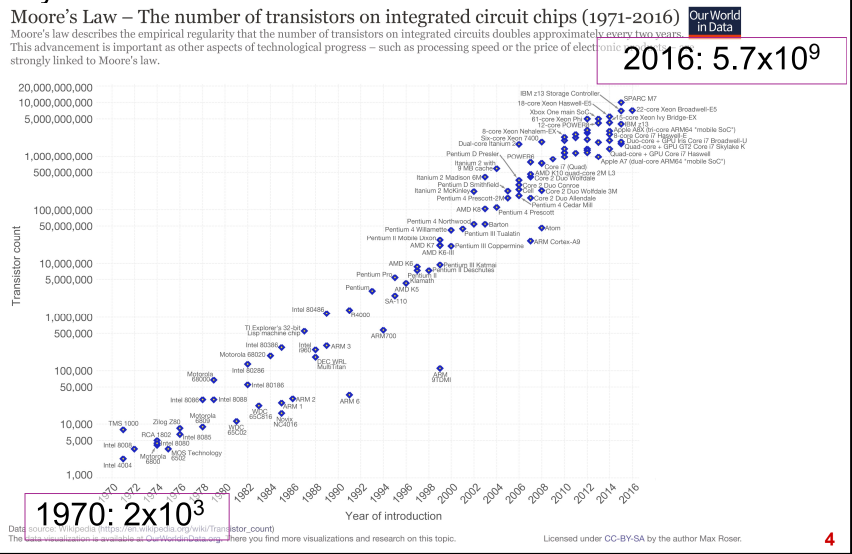
## Instruction Set Architectures

- **Complex Instruction Set Computer (CISC) processors:**
  - 2-operand instructions and 1-operand instructions
  - Any instruction can use memory operands
  - Many addressing modes
  - Complex instruction formats: Varying length instructions
  - Microprogrammed control unit
- **Reduced Instruction Set Computer (RISC) processors:**
  - 3-operand instructions, 2-operand instructions, and 1-operand instructions
  - Load-Store Architecture (LSA) processors:
    - Only memory transfer instructions (Load and Store) can use memory operands.
    - All other instructions can use register operands only.
  - A few addressing modes
  - Simple instruction formats: Fixed length instructions
  - Hardwired control unit
  - Suitable for pipelining

3

## Single Processor Performance: Moore's Law

- The performance of computer hardware doubles every 1.5 years



4

## Current Trends in Architecture

- Cannot continue to leverage Instruction-Level parallelism (ILP)
  - Single processor performance improvement ended in 2003
- New models for performance:
  - Data-level parallelism (DLP)
  - Thread-level parallelism (TLP)
  - Request-level parallelism (RLP)
- These require explicit restructuring of the application

5

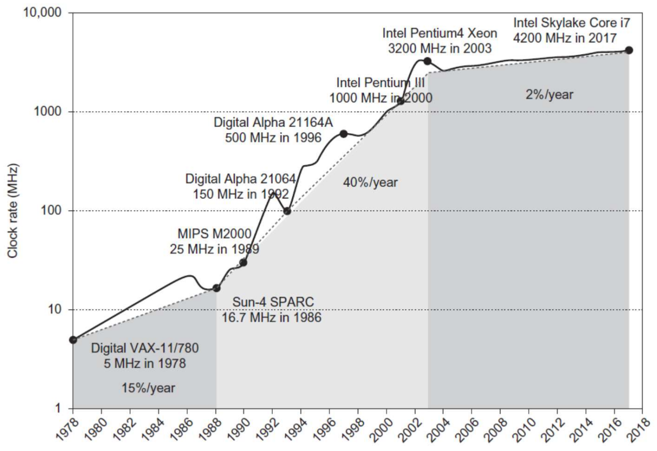
## Classes of Computers

- Personal Mobile Device (PMD)
  - e.g. smart phones, tablet computers
  - Emphasis on energy efficiency and real-time
- Desktop Computing
  - Emphasis on price-performance
- Servers
  - Emphasis on availability, scalability, throughput
- Clusters / Warehouse Scale Computers
  - Used for "Software as a Service (SaaS)"
  - Emphasis on availability and price-performance
  - Sub-class: Supercomputers, emphasis: floating-point performance and fast internal networks
- Internet of Things/Embedded Computers
  - Emphasis: price

6

## Power

- Intel 80386 consumed ~ 2 W
- 3.3 GHz Intel Core i7 consumes 130 W
- Heat must be dissipated from 1.5 x 1.5 cm chip
- This is the limit of what can be cooled by air



7

## Power and Energy

- Problem: Get power in, get power out
- Thermal Design Power (TDP)
  - Characterizes sustained power consumption
  - Used as target for power supply and cooling system
  - Lower than peak power (1.5X higher), higher than average power consumption
- Clock rate can be reduced dynamically to limit power consumption
- Reducing clock rate reduces power, not energy

8

## Classes of Parallelism and Parallel Architecture

- Parallelism at multiple level is the driving force of computer design
  - Energy and cost being the constraints
- Classes of parallelism in applications:
  - Data-Level Parallelism (DLP)
  - Task-Level Parallelism (TLP)
- Classes of architectural parallelism:
  - Instruction-Level Parallelism (ILP)
  - Vector architectures/Graphic Processor Units (GPUs)
  - Thread-Level Parallelism
  - Request-Level Parallelism

9

## Categories based on Flynn's Taxonomy

- Single instruction stream, single data stream (SISD):
  - A sequential computer which exploits instruction level parallelism - uniprocessors
- Single instruction stream, multiple data streams (SIMD):
  - A single instruction operates on multiple different data streams
- Multiple instruction streams, multiple data streams (MIMD)
  - Multiple autonomous processors simultaneously executing different instructions on different data
- Multiple instruction streams, single data stream (MISD)
  - Multiple instructions operate on one data stream
  - No commercial implementation

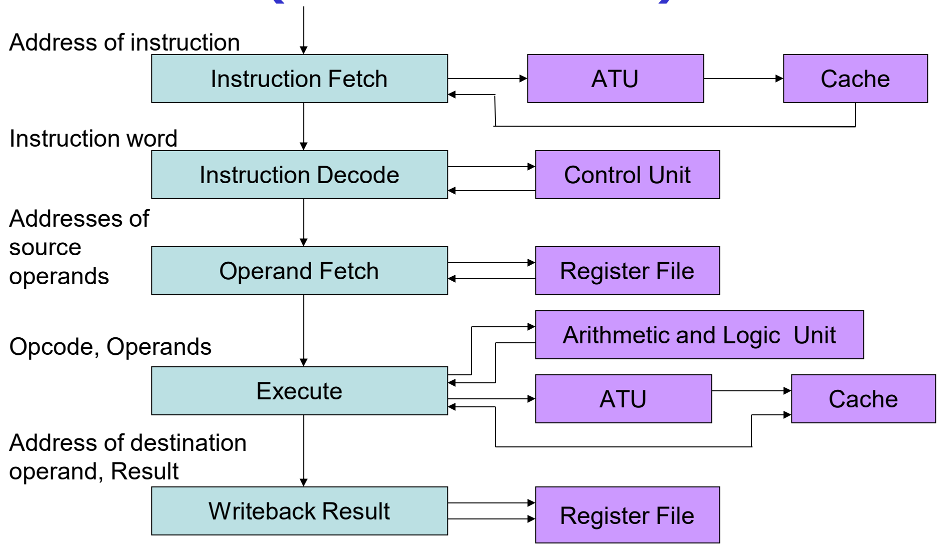
10

## Single Instruction Stream, Single Data Stream (SISD)

- Uniprocessor
- Programmer thinks of this as sequential computer
- SISD exploits instruction-level parallelism (ILP)
- ILP exploits data level parallelism
  - With compiler help using pipelining
    - Overlapped execution of instructions, One instruction per clock cycle (1 IPC)
  - Using techniques like superscalar and speculative execution
    - Instruction-level parallelism, Multiple instructions per clock cycle (> 1 IPC)
- Reason for slowdown in uniprocessor:
  - Diminishing retunes in exploiting ILP
  - Growing concern over power

11

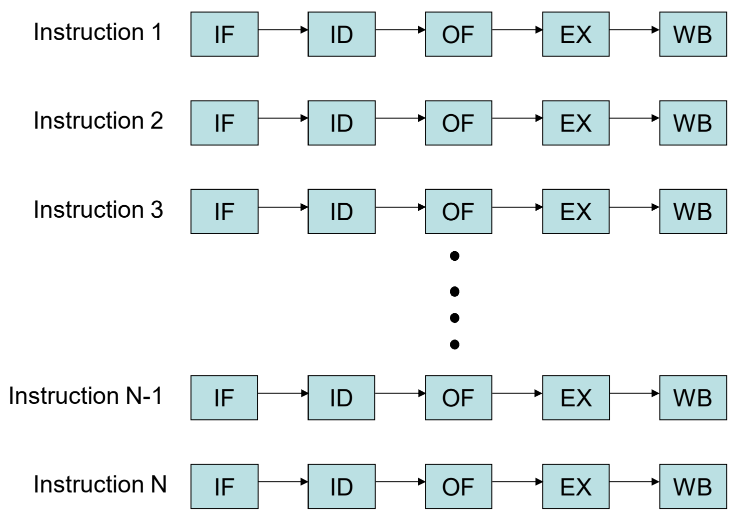
## Instruction Cycle in Processors (RISC Processor)



- ATU: Address Translation Unit

12

## Non-Pipelined Execution of Instructions



13

## Pipelining

- Overlapped execution of instructions
- Different phases of an instruction cycle use different functional units in a processor
- Different phases of multiple instructions can be overlapped
- Target of pipelined processor design
  - Throughput of one instruction per clock cycle

14

## Pipelined Execution of Instructions

Clock cycle →

Instruction ↓

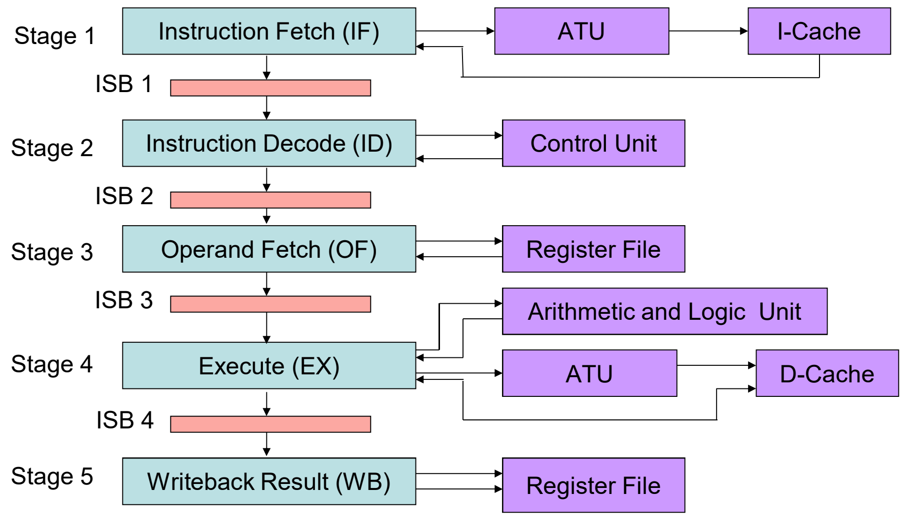
| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| I1 | IF | ID | OF | EX | WB | | | | | | | | | |
| I2 | | IF | ID | OF | EX | WB | | | | | | | | |
| I3 | | | IF | ID | OF | EX | WB | | | | | | | |
| I4 | | | | IF | ID | OF | EX | WB | | | | | | |
| I5 | | | | | IF | ID | OF | EX | WB | | | | | |
| I6 | | | | | | IF | ID | OF | EX | WB | | | | |
| I7 | | | | | | | IF | ID | OF | EX | WB | | | |
| I8 | | | | | | | | IF | ID | OF | EX | WB | | |
| I9 | | | | | | | | | IF | ID | OF | EX | WB | |
| I10 | | | | | | | | | | IF | ID | OF | EX | WB |

↔ Pipeline Fill Phase

↔ Pipeline Drain Phase

15

## Pipelined Processor Architecture (RISC Processor)



- ISB: Inter Stage Buffer
- ATU: Address Translation Unit
- Split Level I-Cache, Multiport Register File

16

## Pipelining Idealism and Realism

- Pipelining Idealism
  - Uniform subcomputations: Uniform latency
  - Identical subcomputations: Fixed latency
  - Independent subcomputations

- Instruction Pipelining Realism
  - Non-uniform subcomputations: Different phases of instruction cycle have different latencies
    - How to decide the period of the pipeline clock cycle?
    - Balancing the pipeline stages
  - Non-identical subcomputations: A phase of instruction cycle has different latency for different instructions
    - Example: Execute (EX) phase for arithmetic instructions
    - Stalls in the pipeline due to structural hazards
  - Dependent subcomputations: Source operand of an instruction is the destination operand of a previous instruction
    - Stalls in the pipeline due to data hazards

17

## Structural Hazards

- Stalls in the pipeline due to high latency of memory access operations
  - a cache miss in I-cache during IF phase for I2
  - a cache miss in D-cache during EX phase of I4, a LOAD instruction

Clock cycle →

Instruction ↓

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| I1 | IF | | | | | | | | | | | | | |
| I2 | | | | | | | | | | | | | | |
| I3 | | | | | | | | | | | | | | |
| I4 | | | | | | | | | | | | | | |
| I5 | | | | | | | | | | | | | | |
| I6 | | | | | | | | | | | | | | |

- Loss of clock cycles due to these stalls can be reduced by using Level 1 caches with higher hit rate and lower miss penalty

18

## Structural Hazards

- Stalls in the pipeline due to arithmetic operations that require multiple clock cycles during the EX phase of I2 and I4 instructions

```
I1: ADD  R2, R0, R1
I2: DIV   R5, R3, R4
I3: ADD  R11, R9, R0
I4: FADD F2, F0, F1
I5: SUB   R8, R6, R7
```

Clock cycle $\longrightarrow$

Instruction $\downarrow$

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| I1 | IF | ID | OF | EX | WB | | | | | | | | | |
| I2 | | IF | ID | OF | EX | EX | EX | WB | | | | | | |
| I3 | | | IF | ID | OF | 🟥 | 🟥 | EX | WB | | | | | |
| I4 | | | | IF | ID | 🟥 | 🟥 | OF | EX | EX | EX | EX | WB | |
| I5 | | | | | IF | 🟥 | 🟥 | ID | OF | 🟥 | 🟥 | 🟥 | EX | WB |

- Loss of clock cycles due to these stalls can be reduced by using faster circuits for complex arithmetic operations

19

## Data Hazards

- Stalls in the pipeline due to Read-After-Write (RAW) data dependencies among instructions

```
I1: ADD  R2, R0, R1
I2: SUB  R5, R3, R4
I3: MUL  R6, R2, R5;  R2 and R5 are destination operands of I1 and I2
I4: ADD  R9, R7, R8
I5: SUB  R10, R6, R9; R6 and R9 are destination operands of I3 and I4
```

Clock cycle $\longrightarrow$

Instruction $\downarrow$

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| I1 | IF | ID | OF | EX | WB | | | | | | | | |
| I2 | | IF | ID | OF | EX | | | | | | | | |
| I3 | | | IF | ID | OF | | | | | | | | |
| I4 | | | | IF | ID | | | | | | | | |
| I5 | | | | | IF | | | | | | | | |

- Loss of clock cycles due to these stalls can be reduced by operand forwarding

20

## Control Hazards

- Stalls in the pipeline due to the presence of conditional branch instructions

I1: INC R4
I2: CMP R4, R5
I3: BEQ NEXT (Address of I10); *If equal, jump to* I10 *at* NEXT
I4:
I5:
I6:
I7:
I8:
I9:
I10:

Instruction ↓

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| I1 | IF | ID | OF | EX | WB | | | | | | | |
| I2 | | IF | ID | OF | EX | WB | | | | | | |
| I3 | | | IF | ID | OF | EX | WB | | | | | |
| I4 | | | | IF | ID | OF | EX | | | | | |
| I5 | | | | | IF | ID | OF | | | | | |
| I6 | | | | | | IF | ID | | | | | |
| I7 | | | | | | | IF | | | | | |
| I10 | | | | | | | | IF | ID | OF | EX | WB |

↑ Flush

- The pipeline is flushed in clock cycle 7 because the condition of the branch instruction is TRUE and the branch to the instruction I10 has to take place

21

## Static Branch Prediction

- Branch prediction can be used to reduce the loss of clock cycles due to control hazards
- Branch prediction for conditional branch instructions due to loop statements can be made by the compiler
  - FOR statement:
    - Predict that the branch WILL NOT take place
  - DO… WHILE statement:
    - Predict that the branch WILL take place
- Instructions are fetched into the pipeline according to the prediction made
  - Whenever the prediction is correct, flushing of the pipeline is avoided
- Prediction is not possible for conditional branch instructions due to IF…THEN…ELSE statements and SWITCH statements.

22

## Limitations of Scalar Pipelines

- Scalar pipelines are characterized by a single-instruction pipeline of K stages
- All instructions, regardless of type, traverse through the same set of pipeline stages
- At most, one instruction can be resident in each pipeline stage at any one time
- The instructions advance through the pipeline in a lockstep fashion
- The maximum throughput for a scalar pipeline is bounded by one instruction per clock cycle
- The stalling of a lockstep or rigid pipeline induces unnecessary pipeline bubbles

23

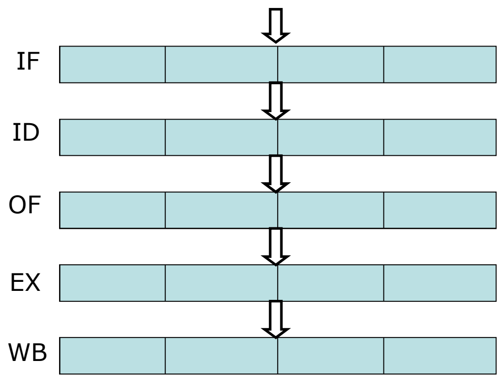
## Superscalar Organization

- Target of superscalar processor design
  - Throughput of multiple instructions per clock cycle

24

## Superscalar Organization

- Target of superscalar processor design
  - Throughput of multiple instructions per clock cycle
- Parallel pipelines: Initiate the processing of multiple instructions in every clock cycle



- Parallel pipeline of width 4
- 4 instructions can be present in each stage of the pipeline
- Functional units in each stage should be capable of processing 4 instructions in each clock cycle
- Wider I-cache, Multiport register file, Multiple execute units

25

## Superscalar Organization

- Parallel pipelines: Initiate the processing of multiple instructions in every clock cycle

Clock cycle →

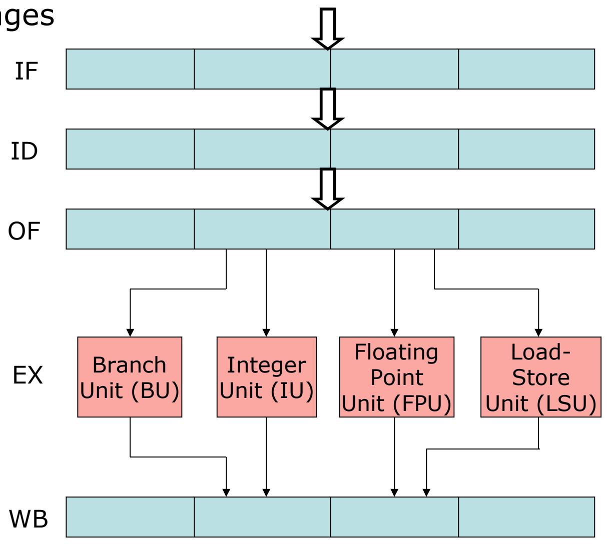
Instruction ↓

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| I1 | IF | ID | OF | EX | WB | | | | | | | | | |
| I2 | IF | ID | OF | EX | WB | | | | | | | | | |
| I3 | IF | ID | OF | EX | WB | | | | | | | | | |
| I4 | IF | ID | OF | EX | WB | | | | | | | | | |
| I5 | | IF | ID | OF | EX | WB | | | | | | | | |
| I6 | | IF | ID | OF | EX | WB | | | | | | | | |
| I7 | | IF | ID | OF | EX | WB | | | | | | | | |
| I8 | | IF | ID | OF | EX | WB | | | | | | | | |
| I9 | | | IF | ID | OF | EX | WB | | | | | | | |
| I10 | | | IF | ID | OF | EX | WB | | | | | | | |
| I11 | | | IF | ID | OF | EX | WB | | | | | | | |
| I12 | | | IF | ID | OF | EX | WB | | | | | | | |

26

## Superscalar Organization

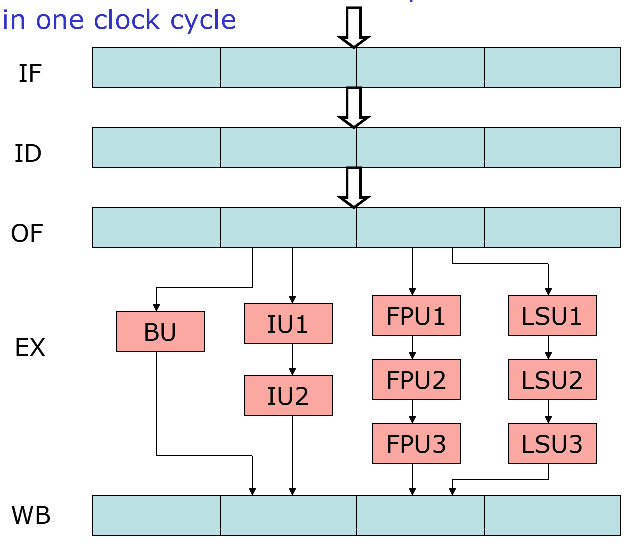
- Diversified pipelines: Use multiple and heterogeneous functional units in their Perform Operation (Execute) stages



27

## Superscalar Organization

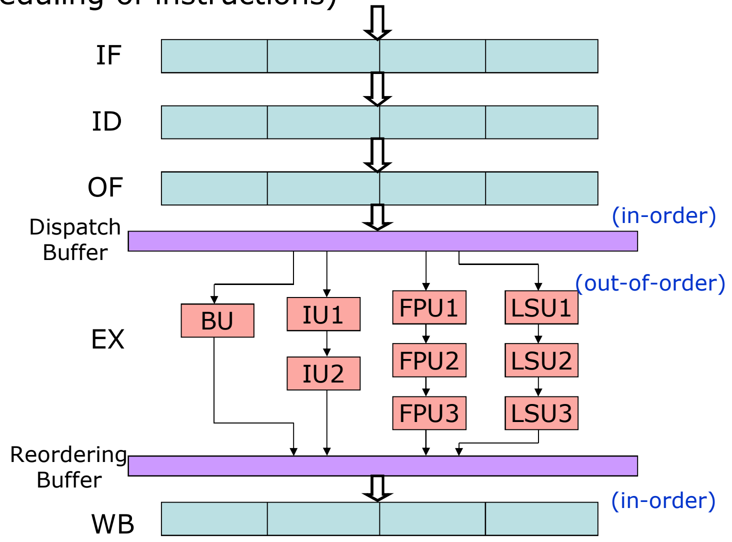
- **Pipelined Execute Units:**
  - More than 4 instructions are processed in the EX stage in one clock cycle



28

## Superscalar Organization

- Dynamic pipelines: Perform out-of-order execution of instructions dynamically during the run-time (Dynamic scheduling of instructions)



29

## Superscalar Pipeline Design

- Instruction Fetch: Wide I-cache is used to fetch multiple instructions in a single read operation
- Instruction Decode: RISC instruction set simplifies the decoding task. Multiple instructions can be decoded in a clock cycle
- Operand Fetch: Number of read ports for the multiport register file is increased
- Execute: One or more execute units of each type are used. Execute units of some types are also pipelined
- Writeback Result: Number of write ports for the register file is increased
- Support for out-of-order execution and in-order completion of instructions is to be provided

30

## Data Dependencies

- Consider the following instruction sequence:

```
I1: DIV    R8, R0, R1
I2: MUL    R9, R2, R3
I3: SUB    R10, R4, R5
I4: ADD    R8, R8, R9
I5: ADD    R6, R8, R10
I6: MUL    R9, R0, R2
I7: SUB    R7, R9, R3
```

- True data dependencies (RAW dependencies):

| | | |
|---|---|---|
| I4 | $\longrightarrow$ | I1: R8 |
| I4 | $\longrightarrow$ | I2: R9 |
| I5 | $\longrightarrow$ | I3: R10 |
| I5 | $\longrightarrow$ | I4: R8 |
| I7 | $\longrightarrow$ | I6: R9 |

- False data dependencies (WAW and WAR dependencies):

| | | | |
|---|---|---|---|
| I4 | $\longrightarrow$ | I1: R8 | WAW (Output dependence) |
| I6 | $\longrightarrow$ | I2: R9 | WAW (Output dependence) |
| I6 | $\longrightarrow$ | I4: R9 | WAR (Anti-dependence) |

31

## Dependency Detection and Resolution

- A 'valid'(V) bit is associated with each register
- RAW dependency
  - Let $R_i$ be the destination operand of an instruction $I_m$
    - The 'valid' bit of $R_i$ is set to 0 while decoding $I_m$
    - The 'valid' bit of $R_i$ is set to 1 when the Writeback for $I_m$ is done, i.e., when $I_m$ is completed
  - Let $R_i$ be the source operand of an instruction $I_n$ with $n > m$
    - If the 'valid' bit of $R_i$ is 1, take the contents of $R_i$
    - If the 'valid' bit of $R_i$ is 0, then it is a case of RAW dependency
      - The instruction $I_n$ will wait in the reservation station till the result of $I_m$ is available on the operand forwarding path

32

## Dependency Detection and Resolution

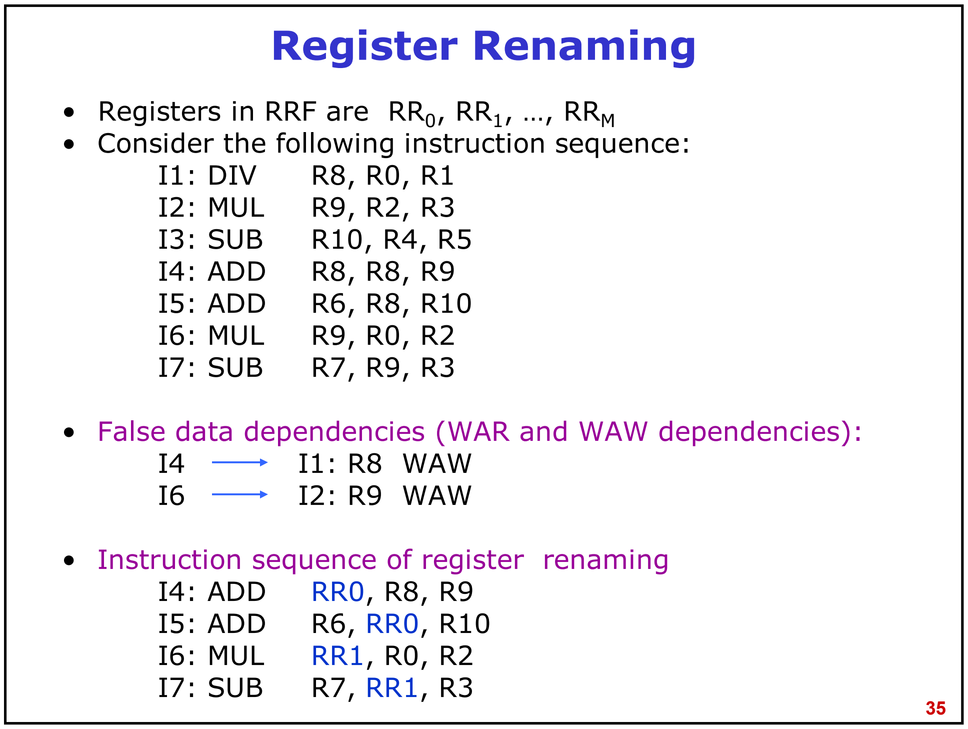
- WAW dependency (Output dependency)
  - Let $R_i$ be the destination operand of an instruction $I_n$ with $n > m$
    - If the 'valid' bit of $R_i$ is 1, there is no WAW dependency
    - If the 'valid' bit of $R_i$ is 0, then it is a case of WAW dependency
      - The register $R_i$ is renamed
- WAR dependency (Anti-dependency)
  - There is no need to check for it when the instructions are dispatched and completed in-order

33

## Register Renaming Techniques

- Register renaming using Rename Registers
  - A non-architectural register file called Rename Register File (RRF) is used
- Method 1: Rename the destination operand of every instruction
  - It requires large number of rename registers
- Method 2: Rename the destination operand of an instruction that has WAW dependency
  - It is necessary to check for WAW dependency while decoding an instruction

34



## Register Renaming

- Registers in RRF are $RR_0$, $RR_1$, …, $RR_M$
- Consider the following instruction sequence:

```
I1: DIV    R8, R0, R1
I2: MUL    R9, R2, R3
I3: SUB    R10, R4, R5
I4: ADD    R8, R8, R9
I5: ADD    R6, R8, R10
I6: MUL    R9, R0, R2
I7: SUB    R7, R9, R3
```

- False data dependencies (WAR and WAW dependencies):

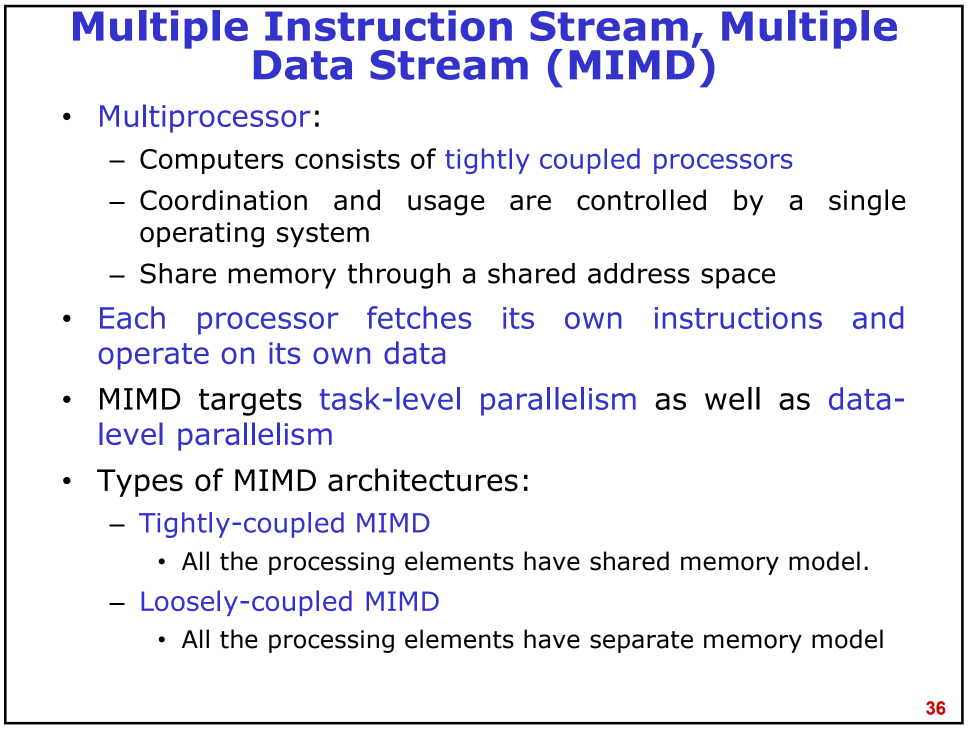
  I4 $\longrightarrow$ I1: R8 WAW

  I6 $\longrightarrow$ I2: R9 WAW

- Instruction sequence of register renaming

```
I4: ADD    RR0, R8, R9
I5: ADD    R6, RR0, R10
I6: MUL    RR1, R0, R2
I7: SUB    R7, RR1, R3
```

35



## Multiple Instruction Stream, Multiple Data Stream (MIMD)

- Multiprocessor:
  - Computers consists of tightly coupled processors
  - Coordination and usage are controlled by a single operating system
  - Share memory through a shared address space
- Each processor fetches its own instructions and operate on its own data
- MIMD targets task-level parallelism as well as data-level parallelism
- Types of MIMD architectures:
  - Tightly-coupled MIMD
    - All the processing elements have shared memory model.
  - Loosely-coupled MIMD
    - All the processing elements have separate memory model

36

## Tightly Coupled MIMD Architecture

- Exploits thread-level parallelism
- Thread is a light-weight process
- Threads of a process share the same virtual address space
- Multiple cooperating threads operate in parallel
- Parallel processing: Tightly coupled set of threads collaborating on a single task
- This architecture ranges from dual processor to dozens of processors
  - Communicate and coordinate through the sharing memory
- Multiprocessors include in
  - Computers consists of single chip with multiple cores (multicore)
  - Computers consisting of several chips, each may be multicore design

37

## Loosely Coupled MIMD Architecture

- Exploits request-level parallelism
  - Exploits parallelism among largely decoupled tasks specified by programmer or the operating system
- Many independent tasks processed in parallel with little need for communication and synchronization
- Examples: Clusters and Warehouse-scale computers (Cloud computers)

38

## Single Instruction Stream, Multiple Data Stream (SIMD)

- Same instruction is executed by multiple processors using different data streams
- SIMD exploits data-level parallelism by applying the same operation to multiple items of data in parallel
- Data parallelism for
  - Matrix-oriented scientific computing
  - Media-oriented image and sound processors
- Each processor has its own data memory, but there is a single instruction memory and control processor
- SIMD vs MIMD:
  - MIMD is more flexible and expensive than SIMD
  - SIMD is more energy efficient than MIMD
  - In SIMD programmer continues to think sequentially, yet achieves parallel speedup

39

## Single Instruction Stream, Multiple Data Stream (SIMD)

- Types of SIMD architecture:
  - Vector architectures
  - Multimedia extensions to standard instruction sets
  - Graphics processor units (GPUs)
- For x86 processors:
  - Expect two additional cores per chip per year
  - Potential speedup from SIMD to be twice that from MIMD!

40

## Detecting and Enhancing Loop-Level Parallelism

- This is critical for exploiting Data-Level Parallelism and Task-Level Parallelism
- Compiler technology for discovering the amount of parallelism that we can exploit in a program as well as hardware support for these compiler techniques
- Focus is on understanding
  - When a loop is parallel
  - How dependence can prevent a loop from being parallel
  - Techniques for eliminating some types of dependences
- Loop-level parallelism is normally analyzed at the source level or close to it
- Loop-level analysis involves determining what dependences exist among the operands in a loop across the iterations of the loop
- We focus on data dependences (RAW dependence)

41

## Loop-Carried Dependence

- Dependence exists among the operand (data value) in one iteration with the data value produced in earlier iterations
- Loop-carried dependence generally prevent parallelism
- Example:

```
for (i=999; i>=0; i--)
   x[i] = x[i] + s;
```

- Loop-carried dependence between successive uses of induction variable `i` in different iterations
  - Handled using loop unrolling techniques
- Finding loop-level parallelism involves recognizing structures such as
  - Loops
  - Array references
  - Induction variable compositions

42

## Loop-Carried Dependence

- Example:

```
    for (i=0; i< 100; i++){
       A[i+1] = A[i] + C[i];    // S1
       B[i+1] = B[i] + A[i+1];  // S2
    }
```

- Statement S1 and S2 has loop-carried dependence
- Statement S2 is also depending on S1 within an iteration
  - This is intra-loop dependence

## Loop-Carried Dependence

- Example:

```
    for (i=0; i< 100; i++){
       A[i] = A[i] + B[i];   // S1
       B[i+1] = C[i] + D[i]; // S2
    }
```

- Loop-carried dependence between S2 and S1
- Loop is parallel if it can be written without a cycle in dependence
- Absence of a cycle means that the dependence give a partial ordering on the statements
- It must be transformed to confirm to the partial ordering and expose parallelism
- Two observations:
  - There is no dependence from S1 and S2
  - On the first iteration of loop, S2 depends on the value B[0] computed prior to the initiating the loop

## Loop-Carried Dependence

- Replace with code sequence

```
A[0]=A[0]+B[0];
for (i=0; i< 99; i++){
   B[i+1] = C[i] + D[i];
   A[i+1] = A[i+1] + B[i+1];
}
B[100]=C[99] + D[99];
```

- Example:

```
for (i=0; i< 100; i++){
   A[i] = B[i] + C[i];   // S1
   D[i] = A[i] * E[i];   // S2
}
```

45

## Loop-Carried Dependence

- Often loop-carried dependence are in the form of recurrence
- Detecting recurrence (loop-carried dependence) is important as some architectures have special support for executing recurrences

46

## Name Dependence

- Name dependence:
  - Eliminated by renaming and copying
- Example:

```
for (i=0; i< 100; i++){
   y[i] = x[i] / c;    // S1
   x[i] = x[i] + c;    // S2
   z[i] = y[i] + c;    // S3
   y[i] = c - y[i];    // S4
}
```

  - True dependence
  - Anti-dependence
  - Output dependence

47

## Eliminating Dependent Computations

- Recurrences
- Example: Dot product

```
for (i=0; i< 1000; i++){
   sum = sum + x[i]*y[i];
}
```

- Loop-carried dependence on variable `sum`
- Transform into a set of loops, one is completely parallel and another is partial
  - Scalar expansion: This transformation makes loop completely parallel
  - Reduce step: Sums up the elements of vector
  - Reductions are handled by special hardware in vector and SIMD architectures
  - It allows reduction step to be done much faster than it could be done in scalar mode

48

## Dependence Analysis

- Dependence Analysis is critical for exploiting parallelism
- For detecting loop-level parallelism dependence analysis is a basic tool
- However, it applies only under a limited circumstances
- CUDA/OpenMP programmers write explicitly parallel loop

49

## Text Books

- J. L. Hennessy and D. A. Patterson, *Computer Architecture – A Quantitative Approach*, 5th Edition, Morgan Kaufmann, 2012
- J. P. Shen and M. H. Lipasti, *Modern Processor Design – Fundamentals of Superscalar Processors*, Tata McGraw-Hill, 2005.
- W. Stallings, *Computer Organization and Architecture – Designing for Performance*, 8th Edition, Prentice Hall of India, 2011.
- D. Sima, T. Fountain and P. Kacsuk, *Advanced Computer Architecture – A Design Space Approach*, Addison-Wesley, 1997.

50