# TCS 802
# Advanced Computer Architecture

Instruction Set Principle

Instruction Level Parallelism (ILP)

- Conditions of Parallelism
- Exploiting ILP
- Limits on ILP

# *Instruction Set Architecture*

- Elements of ISA
  - Programming Registers
  - Type and Size of Operands
  - Addressing Modes
  - Types of Operations
  - Instruction Encoding

- Role of Compilers

# *Instruction Set Architecture*

- Instruction set architecture is the structure of a computer that a machine language programmer must understand to write a correct (timing independent) program for that machine.
- The instruction set architecture is also the machine description that a hardware designer must understand to design a correct implementation of the computer.

# Components of an ISA

- Sometimes known as *The Programmer's Model* of the machine
- Storage cells
  - General and special purpose registers in the CPU
  - Many general purpose cells of same size in memory
  - Storage associated with I/O devices
- The machine instruction set
  - The instruction set is the entire repertoire of machine operations
  - Makes use of storage cells, formats, and results of the fetch/execute cycle
  - i.e., register transfers
- The instruction format
  - Size and meaning of fields within the instruction
- The nature of the fetch-execute cycle
  - Things that are done before the operation code is known

# An Instruction Specify

- Instruction add r0, r1, r2
- Operation to be perform add r0, r1, r2
  - Ans: Op code: add, load, branch, etc.
- Where to find the operands: add r0, r1, r2
  - In CPU registers, memory cells, I/O locations, or part of instruction
- Place to store result add r0, r1, r2
  - Again CPU register or memory cell
- Location of next instruction add r , r , r

  br endloop
  - Almost always memory cell pointed to by program counter(PC) or Instruction pointer (IP)

# Instructions May Be Divided into 3 Classes

- Data movement instructions
  - Move data from a memory location or register to another memory location or register without changing its form
  - *Load*—source is memory and destination is register
  - *Store*—source is register and destination is memory
- Arithmetic and logic (ALU) instructions
  - Change the form of one or more operands to produce a result stored in another location
  - *Add, Sub, Shift*, etc.
- Branch instructions (control flow instructions)
  - Alter the normal flow of control from executing the next instruction in sequence
  - *Br Loc, Brz Loc* ,—unconditional or conditional branches

# Classifying ISA

Stack | Accumulator | Register-Memory | Register-Register

Processor

TOS

...

Memory

# Classifying ISA (Instructions)

- **Stack Architectures** -
  operands are implicitly on the top of the stack

  address    add    tos <– tos + next

- **Accumulator Architectures** -
  one operand is implicitly accumulator

  address    add A    acc <- acc + mem[A]

- **General-Purpose Register Architectures** -
  only explicit operands, either registers or memory locations
  - Memory-Memory :
    access memory Locations as part of any instruction

  address    add A, B    mem[A] <- mem[A] + mem[B]
  address    add A, B, C    mem[A] <- mem[B] + mem[C]

# Classifying ISA (Instructions)

- General-Purpose Register Architectures -
  only explicit operands, either registers or memory locations
  - register-memory:
    access memory as part of any instruction

    address | add R , A | R <- R + mem[A]
    ---|---|---
    | load R , A | R <_ mem[A]

  - register-register:
    access memory only with load and store instructions

    address | add R , R , R | R <- R + R
    ---|---|---
    | load R , R | R <- mem[R
    | store R , R | mem[R <- R

# Operand Access

- Register-Register (0,3)

  (m, n) means m memory operands and n total operands in an ALU instruction

  - Pure RISC, register to register operations
  - Advantages
    - Simple, fixed length instruction encoding.
    - Simple code generation.
    - Instructions take similar number of clocks to execute. Uniform CPI
  - Disadvantages
    - Higher inst. count.
    - Some instructions are short and bit encoding may be wasteful.

# Operand Access

- Register-Memory (1,2)
  - Register – Memory ALU Architecture
  - In later evolutions of RISC and CISC
  - Advantages
    - Data can be accessed without loading first.
    - Instruction format easy to encode
    - Good instruction density
  - Disadvantages
    - Source operand also destination, data overwritten
    - Need for memory address field may limit # registers
    - CPI varies by operand location

# Operand Access

- Memory-Memory (3,3)
  - True memory-memory ALU model, e.g. full orthogonal CISC architecture
  - Advantages
    - Most compact instruction density, no temporary registers needed
  - Disadvantages
    - Memory access create bottleneck
    - Variable CPI
    - Large variation in instruction size
    - Expensive to implement
  - Not used in today's architectures

# Memory Addressing

- What is accessed - byte, word, multiple words?
  - today's machine are byte addressable, due to legacy issues
- But main memory is organized in 32 - 64 byte lines
  - matches cache model
  - Retrieve data in, say, 4 byte chunks
- Alignment Problem
  - accessing data that is not aligned on one of these boundaries will require multiple references
    - E.g. fetching 16 bit integer at byte offset 3 requires two four byte chunks to be read in (line 0, line 1)
  - Can make it tricky to accurately predict execution time with mis-aligned data
  - Compiler should try to align! Some instructions auto-align too

# Addressing Modes

- The addressing mode specifies the address of an operand we want to access
  - Register or Location in Memory
  - The actual memory address we access is called the effective address
- Effective address may go to memory or a register array
  - typically dependent on location in the instruction field
  - multiple fields may combine to form a memory address
  - register addresses are usually simple - needs to be fast
- Effective address generation is important and should be fast!
  - Falls into the common case of frequently executed instructions

# Memory Addressing

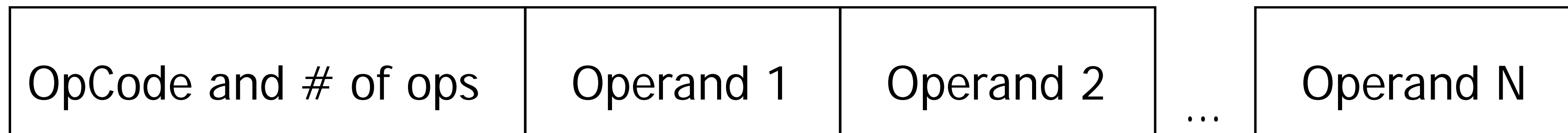
| Mode | Example | Meaning | When used |
|---|---|---|---|
| Register | Add R4, R3 | Regs[R4]←Regs[R4] + Regs[R3] | Value is in a register |
| Immediate | Add R4, #3 | Regs[R4] ← Regs[R4] + 3 | For constants |
| Displacement | Add R4, 100(R1) | Regs[R4] ← Regs[R4] + Mem[100+Regs[R1]] | Access local variables |
| Indirect | Add R4, (R1) | Regs[R4]←Regs[R4] + Mem[Regs[R1]] | Pointers |
| Indexed | Add R3, (R1+R2) | Regs[R3]←Mem[Regs [R1] + Regs[R2]] | Traverse an array |
| Direct | Add R1, $1001 | Regs[R1] ← Regs[R1] + Mem[1001] | Static data, address constant may be large |

# Memory Addressing

| Mode | Example | Meaning | When used |
|---|---|---|---|
| Memory Indirect | Add R1, @(R3) | Regs[R1]←Regs[R1] + Mem[Mem[Regs[R3]]] | *p if R3=p |
| Autoinc | Add R1, (R2)+ | Regs[R1]←Regs[R1]+ Mem[Regs[R2]], Regs[R2]←Regs[R2]+1 | Stepping through arrays in a loop |
| Autodec | Add R1, (R2)- | Regs[R1]←Regs[R1]+ Mem[Regs[R2]], Regs[R2]←Regs[R2]-1 | Same as above. Can push/pop for a stack |
| Scaled | Add R1, 100(R2)[R3] | Regs[R1]← Regs[R1]+ Mem[100+Regs[R2] + Regs[R3] * d] | Index arrays by a scaling factor, e.g. word offsets |

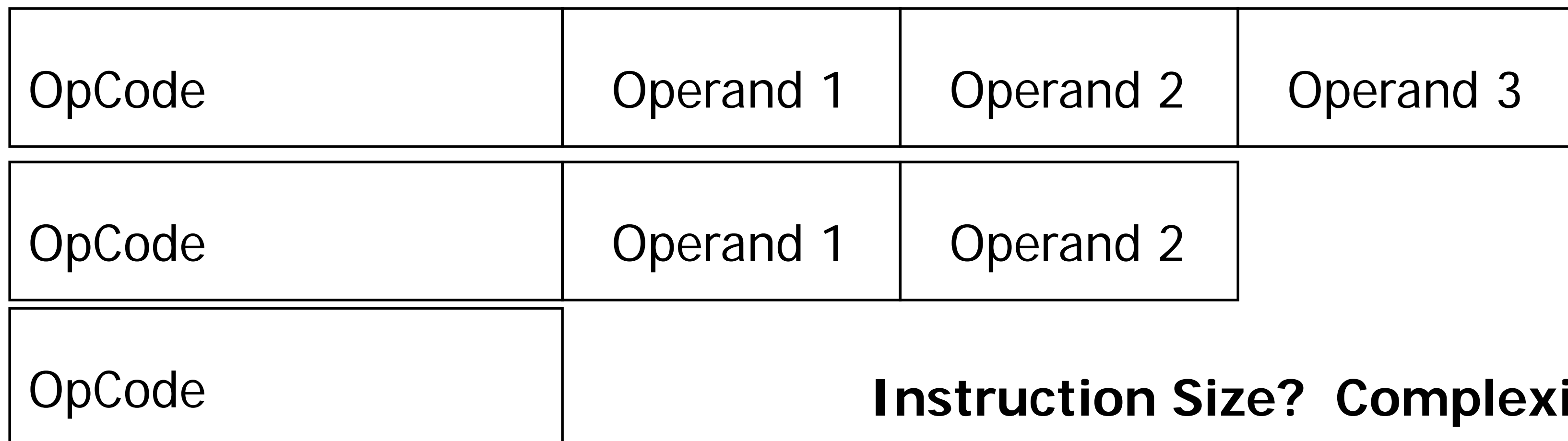
# Instruction Set Encoding Options

**Variable (e.g. VAX)**




**Fixed (e.g. DLX, SPARC, PowerPC)**

**Hybrid (e.g. x86, IBM 360)**




**Instruction Size? Complexity?**

# Role of the Compiler

- **Role of the compiler is critical**
  - Difficult to program in assembly, so nobody does it
  - Certain ISA's make assembly even more difficult to optimize
  - Leave it to the compiler
- **Compiler writer's primary goal:**
  - correctness
- **Secondary goal:**
  - speed of the object code
- **More minor goals:**
  - speed of the compilation
  - debug support
  - Language interoperability

# Compiler Optimizations

- **High-Level**
  - Done on source with output fed to later passes
  - E.g. procedure call changed to inline
- **Local**
  - Optimize code only within a basic block (sequential fragment of code)
  - E.g. common subexpressions – remember value, replace with single copy. Replace variables with constants where possible, minimize boolean expressions
- **Global**
  - Extend local optimizations across branches, optimize loops
  - E.g., remove code from loops that compute same value on each pass and put it before the loop. Simplify array address calculations.

# Compiler Optimizations (cont)

- **Register Allocation**
  - What registers should be allocated to what variables?
  - NP Complete problem using graph coloring.  Must use an approximation algorithm
- **Machine-dependent Optimization**
  - Use SIMD instructions if available
  - Replace multiply with shift and add sequence
  - Reorder instructions to minimize pipeline stalls

# PARALLELISM

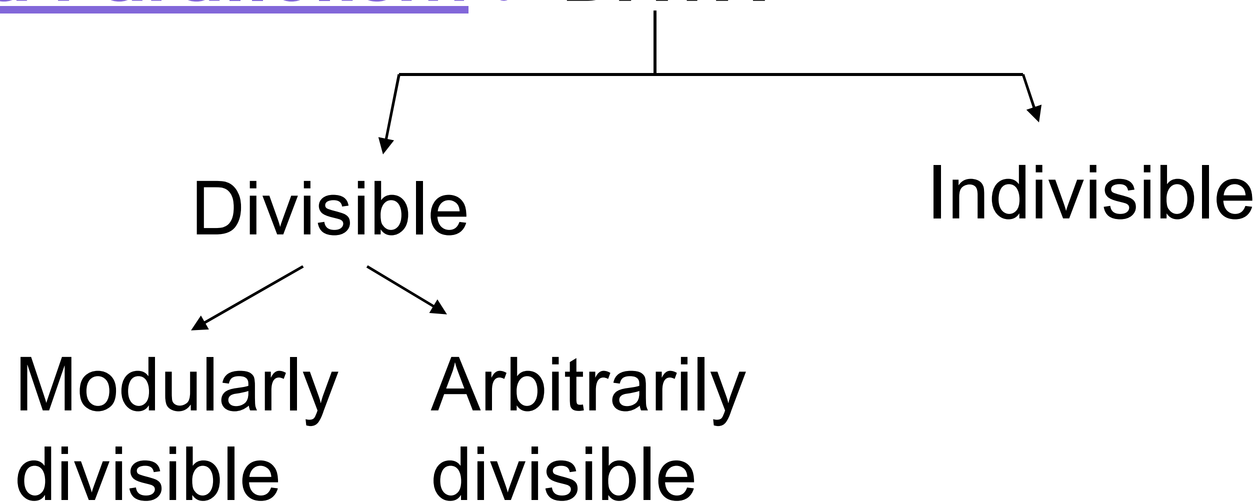
## Attributes of parallelism

- Computational granularity,
- Time and space complexities,
- Communication latencies,
- Scheduling policies,
- Load balancing, etc.

# *Types of Parallelism*

- Program
- Data

  Here we deal with program parallelism and some aspects of data parallelism



# Conditions of Parallelism

- The exploitation of parallelism in computing requires understanding the basic theory associated with it. Progress is needed in several areas:
  - computation models for parallel computing
  - interprocessor communication in parallel architectures
  - integration of parallel systems into general environments

# Data and Resource Dependencies

- Program segments cannot be executed in parallel unless they are independent.
- Independence comes in several forms:
  - Data dependence: data modified by one segement must not be modified by another parallel segment.
  - Control dependence: if the control flow of segments cannot be identified before run time, then the data dependence between the segments is variable.
  - Resource dependence: even if several segments are independent in other ways, they cannot be executed in parallel if there aren't sufficient processing resources (e.g. functional units).

# Data Dependence

- Flow dependence: S1 precedes S2, and at least one output of S1 is input to S2.
- Antidependence: S1 precedes S2, and the output of S2 overlaps the input to S1.
- Output dependence: S1 and S2 write to the same output variable.
- I/O dependence: two I/O statements (read/write) reference the same variable, and/or the same file.

# Data Dependence

- Unknown dependence: Dependence relationships cannot be determined in the following situations:
  - Indirect addressing
  - The subscript of a variable is itself subscripted.
  - The subscript does not contain the loop index variable.
  - A variable appears more than once with subscripts having different coefficients of the loop variable (that is, different functions of the loop variable).
  - The subscript is nonlinear in the loop index variable.
- Parallel execution of program segments which do not have total data independence can produce non-deterministic results.

# Control Dependence

- When the order of the execution cannot be determined before run time, such a situation arises. Conditional statements like IF will not be resolved until run time.
- Control-independent example:

```
for (i=0;i<n;i++) {
a[i] = c[i];
        if (a[i] < 0) a[i] = 1;
}
```

- Control-dependent example:

```
for (i=1;i<n;i++) {
        if (a[i-1] < 0) a[i] = 1;
}
```

- Compiler techniques are needed to get around control dependence limitations.

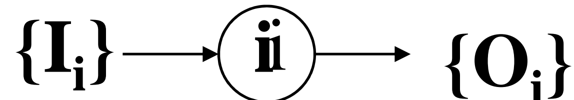
# Resource Dependence

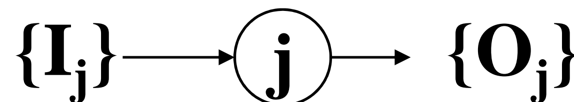
- Data and control dependencies are based on the independence of the work to be done.
- Resource independence is concerned with conflicts in using shared resources, such as registers, integer and floating point ALUs, etc.
- ALU conflicts are called ALU dependence.
- Memory (storage) conflicts are called storage dependence.

# Bernstein's Conditions

- Bernstein's conditions are a set of conditions which must exist if two processes can execute in parallel.
- Notation
  - $I_i$ is the set of all input variables for a process $P_i$.
  - $O_i$ is the set of all output variables for a process $P_i$.
- If $P_1$ and $P_2$ can execute in parallel (which is written as $P_1 \| P_2$), then:

$$I_1 \cap O_2 = \varnothing$$
$$I_2 \cap O_1 = \varnothing$$
$$O_1 \cap O_2 = \varnothing$$



# *Bernstein's Conditions*

- In terms of data dependencies, Bernstein's conditions imply that two processes can execute in parallel if they are flow-independent, antiindependent, and output-independent.
- The parallelism relation || is commutative ($P_i$ || $P_j$ implies $P_j$ || $P_i$), but not transitive ($P_i$ || $P_j$ and $P_j$ || $P_k$ does not imply $P_i$ || $P_k$) . Therefore, || is not an equivalence relation.
- Intersection of the input sets is allowed.

# Detection of Parallelism

- Example

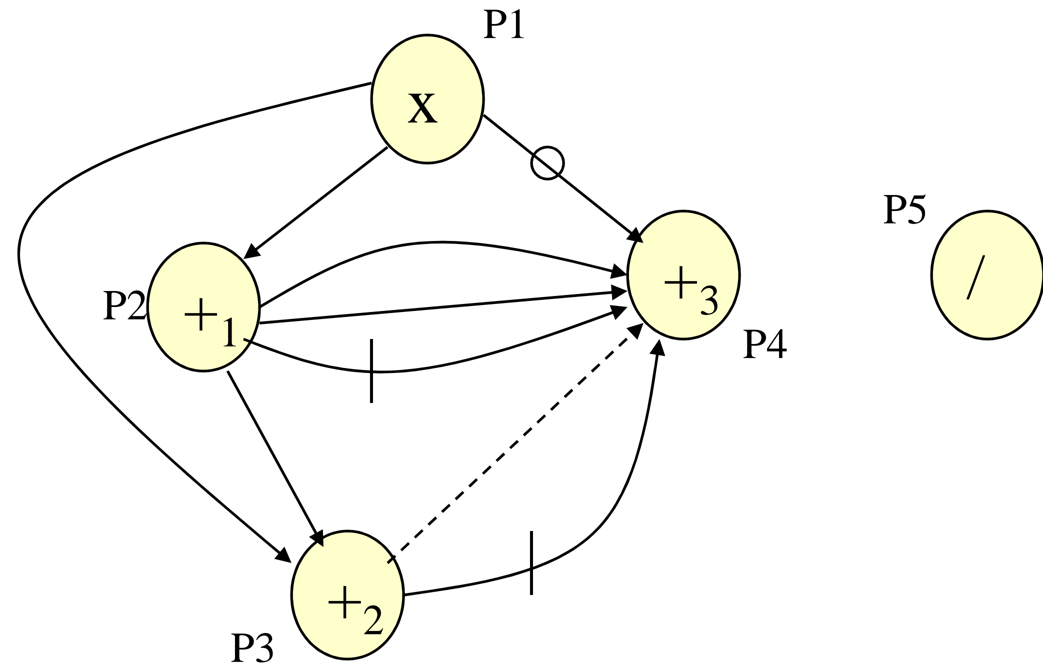
  $P_1$: C = D x E

  $P_2$: M = G + C

  $P_3$: A = B + C

  $P_4$: C = L + M

  $P_5$: F = G / E




Dependence Graph

# *Execution*

D
E X P1
C
G + P2
B + P3
A
L + P4 M
C
G
E / P5
F

E D
G X P1 B
C
P2 + P3 + 
G E
/ P5
L M
+ P4
C A F

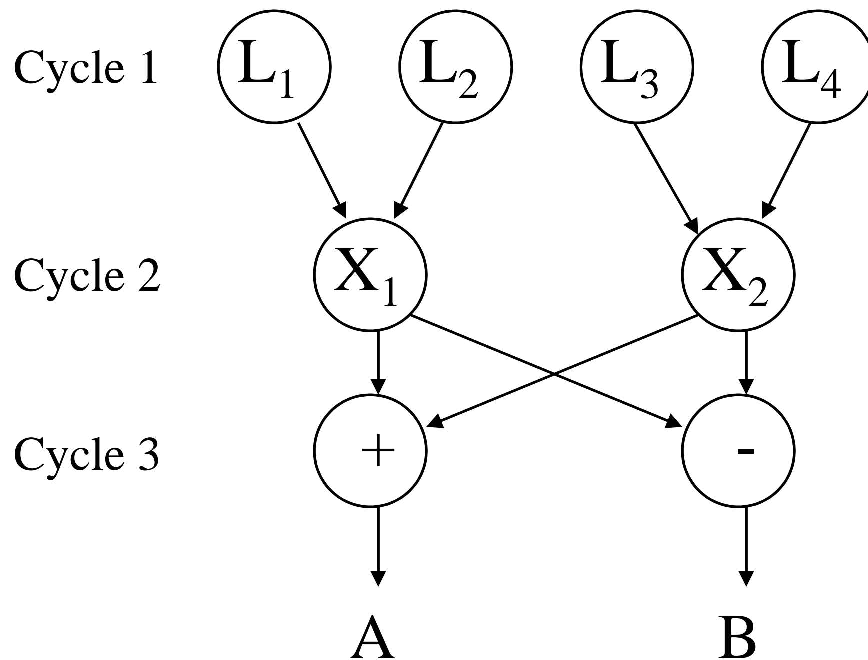
# *Hardware Parallelism*

- Hardware parallelism is defined by machine architecture and hardware multiplicity.
- It can be characterized by the number of instructions that can be issued per machine cycle. If a processor issues *k* instructions per machine cycle, it is called a *k-issue* processor. Conventional processors are *one-issue* machines.
- Examples. Intel i960CA is a three-issue processor (arithmetic, memory access, branch). IBM RS-6000 is a four-issue processor (arithmetic, floating-point, memory access, branch).
- A machine with *n* *k*-issue processors should be able to handle a maximum of *nk* threads simultaneously.

# Software Parallelism

- Software parallelism is defined by the control and data dependence of programs, and is revealed in the program's flow graph.
- It is a function of algorithm, programming style, and compiler optimization.

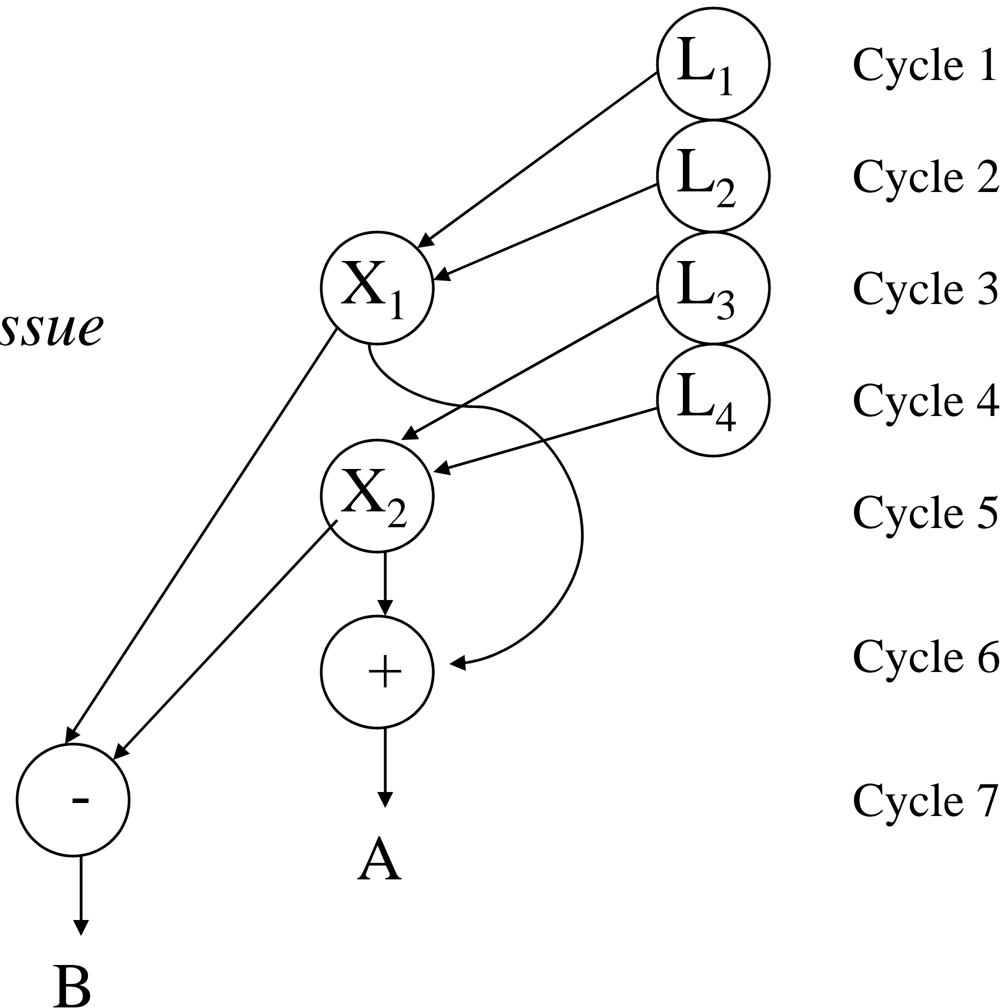
# *Mismatch between software and hardware parallelism*




*Maximum software parallelism (L=load, X/+/- = arithmetic).*

# *Mismatch between software and hardware parallelism*

*Same problem, but considering the parallelism on a two-issue superscalar processor.*



# *Mismatch between software and hardware parallelism*

*Same problem, with two single-issue processors*

= *inserted for synchronization*

# Types of Software Parallelism

- Control Parallelism – two or more operations can be performed simultaneously. This can be detected by a compiler, or a programmer can explicitly indicate control parallelism by using special language constructs or dividing a program into multiple processes.
- Data parallelism – multiple data elements have the same operations applied to them at the same time. This offers the highest potential for concurrency (in SIMD and MIMD modes). Synchronization in SIMD machines handled by hardware.

# *Solving the Mismatch Problems*

- Develop compilation support
- Redesign hardware for more efficient exploitation by compilers
- Use large register files and sustained instruction pipelining.
- Have the compiler fill the branch and load delay slots in code generated for RISC processors.

# The Role of Compilers

- Compilers used to exploit hardware features to improve performance.
- Interaction between compiler and architecture design is a necessity in modern computer development.
- It is not necessarily the case that more software parallelism will improve performance in conventional scalar processors.
- The hardware and compiler should be designed at the same time.

# Program Partitioning & Scheduling

- The size of the parts or pieces of a program that can be considered for parallel execution can vary.
- The sizes are roughly classified using the term “granule size,” or simply “granularity.”
- The simplest measure, for example, is the number of instructions in a program part.
- Grain sizes are usually described as *fine*, *medium* or *coarse*, depending on the level of parallelism involved.

# Latency

- *Latency* is the time required for communication between different subsystems in a computer.
- *Memory latency*, for example, is the time required by a processor to access memory.
- *Synchronization latency* is the time required for two processes to synchronize their execution.
- Computational granularity and communication latency are closely related.

# Levels of Parallelism

| Increasing communication demand and scheduling overhead<br><br>Higher degree of parallelism (↓) | Level | Grain |
|---|---|---|
| | Jobs or programs | Coarse grain |
| | Subprograms, job steps or related parts of a program | Coarse grain / Medium grain |
| | Procedures, subroutines, tasks, or coroutines | Medium grain |
| | Non-recursive loops or unfolded iterations | Fine grain |
| | Instructions or statements | Fine grain |

# Instruction Level Parallelism

- This fine-grained, or smallest granularity level typically involves less than 20 instructions per grain. The number of candidates for parallel execution varies from 2 to thousands, with about five instructions or statements (on the average) being the average level of parallelism.
- Advantages:
  - There are usually many candidates for parallel execution
  - Compilers can usually do a reasonable job of finding this parallelism

# Loop-level Parallelism

- Typical loop has less than 500 instructions.
- If a loop operation is independent between iterations, it can be handled by a pipeline, or by a SIMD machine.
- Most optimized program construct to execute on a parallel or vector machine
- Some loops (e.g. recursive) are difficult to handle.
- Loop-level parallelism is still considered fine grain computation.

# Procedure-level Parallelism

- Medium-sized grain; usually less than 2000 instructions.
- Detection of parallelism is more difficult than with smaller grains; interprocedural dependence analysis is difficult and history-sensitive.
- Communication requirement less than instruction-level
- SPMD (single procedure multiple data) is a special case
- Multitasking belongs to this level.

# Subprogram-level Parallelism

- Job step level; grain typically has thousands of instructions; medium- or coarse-grain level.
- Job steps can overlap across different jobs.
- Multiprograming conducted at this level
- No compilers available to exploit medium- or coarse-grain parallelism at present.

# Job or Program-Level Parallelism

- Corresponds to execution of essentially independent jobs or programs on a parallel computer.
- This is practical for a machine with a small number of powerful processors, but impractical for a machine with a large number of simple processors (since each processor would take too long to process a single job).

# Summary

- Fine-grain exploited at instruction or loop levels, assisted by the compiler.
- Medium-grain (task or job step) requires programmer and compiler support.
- Coarse-grain relies heavily on effective OS support.
- Shared-variable communication used at fine- and medium-grain levels.
- Message passing can be used for medium- and coarse-grain communication, but fine-grain really need better technique because of heavier communication requirements.

# Communication Latency

- Balancing granularity and latency can yield better performance.
- Various latencies attributed to machine architecture, technology, and communication patterns used.
- Latency imposes a limiting factor on machine scalability. Ex. Memory latency increases as memory capacity increases, limiting the amount of memory that can be used with a given tolerance for communication latency.

# Interprocessor Communication Latency

- Needs to be minimized by system designer
- Affected by signal delays and communication patterns
- Ex. $n$ communicating tasks may require $n(n - 1)/2$ communication links, and the complexity grows quadratically, effectively limiting the number of processors in the system.

# Communication Patterns

- Determined by algorithms used and architectural support provided
- Patterns include
  - permutations
  - broadcast
  - multicast
  - conference
- Tradeoffs often exist between granularity of parallelism and communication demand.

# Grain Packing and Scheduling

- Two questions:
  - How can I partition a program into parallel “pieces” to yield the shortest execution time?
  - What is the optimal size of parallel grains?
- There is an obvious tradeoff between the time spent scheduling and synchronizing parallel grains and the speedup obtained by parallel execution.
- One approach to the problem is called “grain packing.”

# Program Graphs and Packing

- A program graph is similar to a dependence graph
  - Nodes = { (n,s) }, where n = node name, s = size (larger s = larger grain size).
  - Edges = { (v,d) }, where v = variable being "communicated," and d = communication delay.
- Packing two (or more) nodes produces a node with a larger grain size and possibly more edges to other nodes.
- Packing is done to eliminate unnecessary communication delays or reduce overall scheduling overhead.

# Scheduling

- A schedule is a mapping of nodes to processors and start times such that communication delay requirements are observed, and no two nodes are executing on the same processor at the same time.
- Some general scheduling goals
  - Schedule all fine-grain activities in a node to the same processor to minimize communication delays.
  - Select grain sizes for packing to achieve better schedules for a particular parallel machine.

# Node Duplication

- Grain packing may potentially eliminate interprocessor communication, but it may not always produce a shorter schedule (see figure 2.8 (a)).
- By duplicating nodes (that is, executing some instructions on multiple processors), we may eliminate some interprocessor communication, and thus produce a shorter schedule.

# Example 2.5

- Example 2.5 illustrates a matrix multiplication program requiring 8 multiplications and 7 additions.
- Using various approaches, the program requires:
  - 212 cycles (software parallelism only)
  - 864 cycles (sequential program on one processor)
  - 741 cycles (8 processors) - speedup = 1.16
  - 446 cycles (4 processors) - speedup = 1.94