UE18CS342
Heterogeneous Parallelism
Unit 1

Aronya Baksy

March 2021

1 Introduction

- **Parallelism** involves connecting multiple processing units with an interconnection network, as well as the necessary software needed to coordinate these processing units.
- The multiple processing units may all be the same (homogeneous) or may have different characteristics (heterogeneous)
- The two main types of hardware designs that acheive different goals are:

1.1 Latency-Oriented Design

- In general, aim to execute as many instructions as possible belonging to a single serial thread, in a given window of time
- Some features of Latency Oriented Design are:
  1. High clock frequency
  2. Large caches reduce the need for frequent memory accesses (that are high latency operations compared to cache access)
  3. Sophisitcated control logic includes features such as *branch prediction*, *predication*, *fusion*, *speculation* and *data forwarding*
  4. Powerful and specialized ALUs for integer and floating point operations

1.2 Throughput-Oriented Design

- Aim is to increase the throughput of the system, by parallelizing workloads.
- Some features of Throughput Oriented Design are:
  1. Moderate clock frequency compared to latency-oriented designs
  2. Small caches, increasing the memory bandwidth utilization and the memory throughput
  3. Simpler control logic, no branch prediction and data forwarding hardware
  4. Energy-efficient and simple ALUs that are deeply pipelined for higher throughput.
  5. Require massive number of parallel operations to tolerate latencies

1.3 Types of Parallelism

1.3.1 Bit-Level Parallelism

- Increasing processor word size, hence reduces number of instructions needed to process large data sizes (larger than word length)
- e.g.: An 8-bit processor will need 3 instructions to add 2 16-bit numbers but a 16-bit processor can do it in one instruction.

1.3.2 Instruction-Level Parallelism

- Concurrent execution of multiple instructions in the same stream.
- Generated and managed by hardware (superscalar) or by compiler (VLIW)
- Limited in practice by data and control dependences

1.3.3 Thread-Level Parallelism

- Multiple threads from the same application executed concurrently.
- Generated by the compiler, managed in hardware.
- Limited in practice by communication/synchronization overheads and by algorithm characteristics

1.3.4 Task-Level Parallelism

- Extraction of parallel tasks from control structures like **loops** and **functions**.
- A program exploiting loop-level parallelism uses multiple threads or processes which operate on some or all of the indices at the same time.
- The speedup generated by such parallelism is in line with Amdahl's Law

1.4 Flynn's Taxonomy

1.4.1 Single Instruction, Single Data

- Single control unit fetches single instruction stream from memory. The CU then generates appropriate control signals to direct single processing element to operate on single data stream i.e., one operation at a time.
- Can be classified as a serial scalar computer.
- e.g.: Intel 4004

1.4.2 Single Instruction, Multiple Data

- A single instruction executes on multiple data streams.
- A single operation on a vector can be thought of as applying the same instruction stream to each element of the vector (each element being a data stream)
- Instructions can be executed in sequence (using pipelining) or in parallel (with multiple functional units).
- e.g.: CRAY-1, ICL DAP

1.4.3 Multiple Instruction, Single Data

- Multiple instructions operate on one data stream.
- This is an uncommon architecture which is generally used for fault tolerance.
- e.g.: Space Shuttle flight computer, Colossus Mark II

1.4.4 Multiple Instruction, Multiple Data

- Multiple autonomous processors simultaneously executing different instructions on different data.
- e.g.: Multi processors and Multi computers

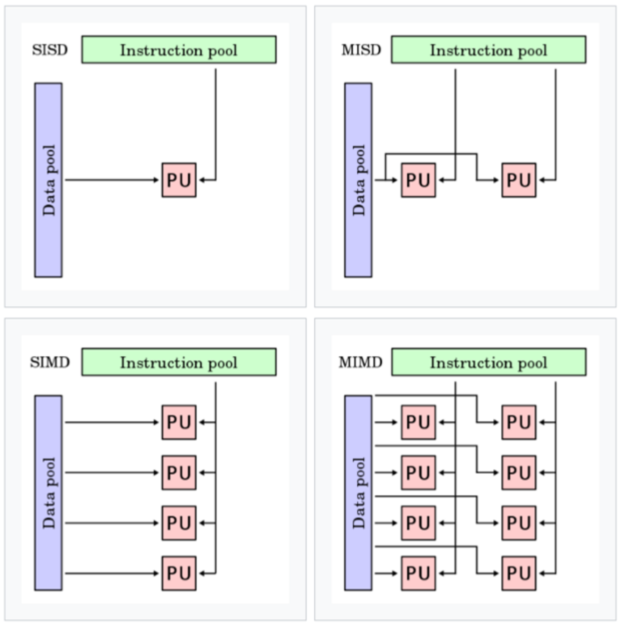

Figure 1: Flynn's Taxonomy

2 ILP Enhancement Techniques

- CPU performance is dependent on three factors as shown in the below equation:

$$T_{CPU} = Instr \times CPI \times t_{cycle}$$

- Improvement in CPU performance involves either
  - Reducing number of executed instructions (with better algorithms, compiler optimizations or better ISAs)
  - Reducing the cycle time (depends on technology, organization and logic design). Increasing clock speed is a solution but only till a physical limit
  - Reducing the CPI (introducing more overlap between instructions, depends on the ISA and the CPU organization). The key is **parallelism** (at instruction level using pipelining, or internally using superscalar processors with multiple functional units or multiple cores, or externally using multiple CPUs)
- Average CPI is calculated as

$$CPI = \sum_{i=1}^{n} F_i \times CPI_i$$

  where F_i is the probability of encountering instruction of type i having CPI equal to CPI_i

- Hazards prevent the next instruction in the instruction stream from being executing during its designated clock cycle.
- Hazards reduce the performance from the ideal speedup gained by pipelining.
- Types of hazards:
  - **Structural Hazards**: Resource conflicts when the hardware cannot support all possible combinations of instructions in simultaneous overlapped execution.
  - **Data Hazards**: Instruction depends on the result of a previous instruction in a way that is exposed by the overlapping of instructions in the pipeline. (e.g.: WAW, WAR, RAW)
  - **Control Hazards**: Arise from pipelining of branches and other instructions that change the PC.

2.1 Dependencies

- Dependencies are artifacts of programs while hazards are artifacts of pipeline architecture and organization
- Not all dependencies translate directly into pipeline hazards
- Software approaches to handle dependencies are:
  1. Reorder instructions at compile time
  2. Insert no-op instructions
  3. Loop unrolling

2.1.1 Data Dependencies

- Also called *true dependencies*
- Let instruction i occur before instruction j in the execution order. j is said to be data dependent on i if
  - j may use a result that is produced by i
  - j is dependent on instruction k and k is dependent on i

2.1.2 Name Dependencies

- When 2 instructions use the same name (i.e. register name) but there is no data flow between the instructions.
- Let instruction i occur before instruction j in the execution order:
  - An **anti-dependency** arises when j writes to a location that is read by i
  - An **output dependency** arises when both i and j write to the same register or memory location
- If the dependencies can be removed (using register renaming maybe) then such instructions *can* be parallelized

2.1.3 Control Dependencies

- Dependency of instructions to the sequential flow of execution and preserves flow-altering behavior of the program (such as branching)
- Two types of constraints:
  - An instruction that is control-dependent on a branch cannot be moved before the branch to make it control independent
  - An instruction that is control-independent of the branch cannot be moved after the branch to make it dependent on the branch

2.2 Dynamic Scheduling: Tomasulo's Algorithm

- This involves designing hardware that can reorder instructions at runtime while preserving data flow and exception behaviour
- Advantages of dynamic scheduling:
  - Simplifies the compiler as it no longer has to take care of reordering instructions
  - Allows instructions compiled for one pipeline to run on other pipelines efficiently as well
- Dynamic scheduling is implemented either as **scoreboarding** or using **Tomasulo's Algorithm**.
- In-order issue of instructions, but out-of-order execution (which leads to out-of-order completion).
- Register renaming is used by Tomasulo's Algorithm to correctly perform OO execution.
- Out-of-order completion leads to WAR/WAW data hazards, as well as complications in exception handling

2.2.1 Components of Tomasulo's Organization

- **Reservation Station**:
  - Allow the CPU to make use of a value as soon as it has been computed instead of waiting for it to be written to a register and reading from there
  - It checks if the operands are available (avoid RAW) and if execution unit is free (avoid Structural hazard) before starting execution.
  - Instructions are stored with available parameters, and executed when ready.
  - Results are identified by the unit that will execute the corresponding instruction. Implicitly *register renaming* solves WAR and WAW hazards.
- Registers in instructions are replaced with pointers to the reservation stations that will produce those values. This act is called *register renaming.*
- As there are more reservation stations than there are registers, it allows for optimizations that compilers cannot do, as well as elimiation of WAR and WAW hazards.
- Instructions are issued in sequence to Reservation Stations which buffer the instruction as well as the operands of the instruction.
- If the operand is not available, the Reservation Station listens on a **Common Data Bus** for the operand to become available. When the operand becomes available, the Reservation Station buffers it, and the execution of the instruction can begin.
- Functional Units (e.g.: FP Adder, FP multiplier), each have their corresponding Reservation Station. The output of the Functional Unit connects to the Common Data Bus, where Reservation Stations are listening for the operands they need.
- The data values stored in a Reservation Station are:
  - **Op**: The operand (ADD, SUB, MUL, DIV, etc.) for the instruction
  - V_j, V_k: The actual register **values** (i.e. values of the source operands) are stored here
  - Q_j, Q_k: The reservation station that will produce the relevant source operand, in case it is not yet available (A value of 0 indicates that the relevant result is already in one of the V fields)
  - **Busy**: Indicates whether that station is currently busy or not.
- **Load/Store Buffers** maintain 2 fields, a busy bit (indicating whether that buffer entry is full or not) and the memory address to be loaded.

2.2.2 Tomasulo's Algorithm Stages

1. **Issue**: get instruction from FP operation queue. If a reservation station is free then the control issues the instruction and sends the operands.
2. **Execute**: Watch CDB to pick up operands from prior instructions. When both operands ready, can execute
3. **Write Back**: Write to all waiting units via CDB, and mark the reservation station as free

2.2.3 Drawbacks of Tomasulo's Algorithm

- High complexity of the hardware and control logic
- Many associative stores (CDB) at high speed
- The CDB speed is a bottleneck on the performance of the algorithm. If each CDB must reach multiple functional units, it implies high capacitance and high wiring density (which ofc leads to higher heat dissipation)
- Only one functional unit can complete execution in each cycle. For multiple FUs to complete at once, more CDBs are needed, which means extra control logic for parallel associative stores
- Imprecise exceptions (exceptions wherein the processor cannot restart directly from the point where the exception was thrown) cannot be handled.

3 Speculative Execution

- Speculative execution is an optimization technique where the CPU executes an instruction before it is known whether it is actually needed.
- This is done so as to prevent a delay that would have to be incurred by executing the instruction after it is known that it is needed.
- If it turns out the instruction's result was not needed after all, most changes made by the instruction are reverted and the results are ignored.

3.1 Hardware Support for Precise exceptions

- A *reorder buffer* allows instructions to be committed in-order.
- An extra **Commit** stage is added to the Tomasulo Algorithm.
- In this stage, the results of instructions will be stored in a register or memory. In the "Write Result" stage, the results are just put in the re-order buffer. All contents in this buffer can then be used when executing other instructions depending on these.

3.2 Tomasulo's Algorithm Stages

- **Issue**: get instruction from FP op queue. If reservation station **and** reorder buffer slot are free, issue instr, send operands and reorder buffer no.
- **Execution**: When both operands ready then execute. If not ready, watch CDB for result. When both in reservation station, execute
- **Write Result**: Write on Common Data Bus to all awaiting FUs and reorder buffer. Mark reservation station available.
- **Commit**: update register with reorder result. When instr. at head of reorder buffer & result present, update register with result (or store to memory) and remove instr from reorder buffer. A mispredicted branch flushes the reorder buffer

Hardware Speculatio	**Software Speculation**
Runtime memory disambiguation done using Tomasulo's Algorithm, allows loads to move past stores at runtime	Runtime memory disambiguation is difficult at compile time when pointers are involved
Works better when control flow is not predictable	Hardware branch pred beats software branch pred that is done at compile time
Maintains completely precise execution model even for speculated instructions	Need added spl support for this in software
Do not require compensation or bookkeeping code	Compensation and bookkeeping are needed
Lookahead in code is limited	Better possible scheduling due to the ability to see the entire code at compile time

3.3 Thread-Level Speculation

- Speculative execution of a sequence of code that is anticipated to be executed in parallel on an independent thread
- Assumptions need to be made about input values. If assumptions are correct then thread can run faster
- If assumptions are wrong then thread needs to be squashed and discarded.
- TLS allows the compiler to optimistically create threads at runtime despite ambiguity about data dependencies, as well as detect such violations and recover from them at runtime.

4 Predicative Execution

- Associate a Boolean expression (i.e. a predicate) with the issue/execution/commit of an instruction
- The result of the instruction is either retained or invalidated depending on whether the predicate evaluates to a Boolean true or a Boolean false.
- In **partial predication**, only a few specific opcodes have predicates associated with them. In **full predication**, all instructions are predicated.
- Central idea behind predication is that the compiler converts control dependence into data dependence, meaning that branch is eliminated (allows for sequential execution rather than changes in control flow)
- Advantages of predcation:
  - Eliminate mis-predictions for hard-to-predict branches
  - Reduce number of branches tested per cycle
  - When misprediction cost is larger than the cost for doing useless work, predication works very well
  - Instructions can be freely reordered without regard for any control dependencies
- Disadvantages of predication:
  - Increased fetch utilization, increased register consumption
  - Easy-to-predict branches become costly due to extra useless work
  - Static predication is not adaptive to run-time branch behavior.
  - Increased FU utilization if predicates are tested at commit time
  - All hard-to-predict branches (such as loop branches) are not completely eliminated
  - Additional hardware and ISA support, larger instruction cache needed.

5 Dependency Analysis

5.1 Types of Dependencies

- **True Dependency**: Instruction S_i precedes instruction S_j, and instruction S_i computes a value that is used by S_i. This means that S_i **must** execute before S_j
- True dependency in the above example is denoted as $S_i\ \delta^+\ S_j$
- **Anti Dependency**: Instruction S_i precedes instruction S_j, and instruction S_i uses a value that is computed by S_i. This means that S_i **must** execute before S_j
- Anti dependency is denoted as $S_i\ \delta^a\ S_j$
- **Output Dependency**: Instruction S_i precedes instruction S_j, and instruction S_i computes a value that is also computed by S_i. This means that S_i **must** execute before S_j
- Output dependency is denoted as $S_i\ \delta^o\ S_j$
- **Input Dependency**: Instruction S_i precedes instruction S_j, and instruction S_i uses a value that is also used by S_i. **Not necessary** that S_i must execute before S_j
- Input dependency is denoted as $S_i\ \delta^I\ S_j$
- Every dependency except true dependency can be eliminated using renaming.

5.2 Iteration Space

- Iteration space is the set of iterations , whose ID's are given by the values held by the loop index variables
- e.g.: the iteration space for the loop

  ```
  for(int i=2; i<100; i = i+3)
  {
      z[i] = 0;
  }
  ```

  is the set $\{2, 5, 8, 11, ..., 95, 98\}$
- The *normalized iteration space* is given as:

$$i_n = \frac{i - i_{min}}{step} \; \forall \, i \; \in \; IterSpace$$

5.3 Dependence Testing

- Given the loop below:

  ```
  for (i in IterSpace1)
  {
      for(j in IterSpace2)
      {
          a[f(i, j)] = expr1;
          expr2 = a[g(i, j)];
      }
  }
  ```
- A dependence is said to exist if there exist 2 iteration vectors $\vec{k}, \vec{j}$ such that $f(\vec{k}) = g(\vec{j})$
- An algorithm that reports dependence only when one exists is called an **exact dependence test**. Else it is called an **in-exact dependence test**.
- Dependence tests must be conservative: if dependence cannot be ascertained, the test must still report that a dependence exists
- NOTE: The equation $a_1 i_1 + a_2 i_2 + a_3 i_3 + ... + a_n i_n = c$ has a solution only if c is divisible exactly by $gcd(a_1, a_2, a_3, ..., a_n)$. This form of linear equation is called a **Diophantine Equation**.

5.3.1 Affine Array Access

- Every array access can be formulated as a matrix equation. Given a loop of length d, the iteration space is represented as

$$\{i \in \mathbb{Z}^d | Bi + b \geq 0\}$$

  Where B is a $d \times d$ matrix, b is a vector of length d and 0 is a zero vector of length d.
- e.g.: For the loop:

  ```
  for(i = 0; i <= 5; ++i)
      for(j=1; j<=7; ++j)
          z[j, i] = 0;
  ```

  we have the following:

$$\begin{aligned}
i \geq 0 &\implies i + 0j + 0 \geq 0 \\
i \leq 5 &\implies -i + 0j + 5 \geq 0 \\
j \geq 1 &\implies 0i + j - 1 \geq 0 \\
j \leq 7 &\implies 0i - j + 7 \geq 0
\end{aligned}$$

- An array access is called Affine if the bounds of the loop and the index of each dimension of the array are affine expressions of loop variables and symbolic constants
- Affine accesses can be represented as linear equations (matrix-vector equations)
- Affine functions map from iteration space to data space, hence easier to identify iterations that map to the same data.

5.4 Indices and Subscripts

- An index is a variable that appears in a subscript of an array (commonly denoted as i, j, k...)
- A subscript is the set of all expressions that occur at one particular position within an array access. (e.g.: in the expression $A[i, j] = A[i, k] + c$ the first subscript is the set $\{i, i\}$ and the second subscript is the set $\{j, k\}$
- A subscript is said to be a:
  1. **Zero Index Variable** if it contains no index variables
  2. **Single Index variable** if it contains exactly one index variable
  3. **Multiple Index Variable** if it contains more than 1 index variable
- e.g.: The expression $A(5, i+1, j) = A(1, i, k)+c$ contains the subscripts $\{5, 1\}$ (zero index), $\{i+1, i\}$ (single index) and $\{j, k\}$ (multiple index)
- Two subscripts are said to be **coupled** if the index variables appearing in one subscript also appear in the other.
- Two subscripts that are not coupled are called **separable**

5.5 Conservative Testing

- Considers only linear subscript expressions, tries to assert that "no dependence exists between 2 subscript expressions of one array reference"
- It is never incorrect (as it is conservative) but it is sub-optimal
- Partition subscripts into separable and minimally coupled groups.
- For separable subscripts, apply single subscript tests. For coupled groups apply multi-subscript tests like the Delta test.

5.6 Delta Test

- Let the source iteration be denoted as I_0 and the sink iteration denoted as $I_0 + \Delta I$. Solve the equation $f(I_0) = g(I_0 + \Delta I)$
- For the expression $A(i + 1) = A(i) + B$ we have $I_0 + 1 = I_0 + \Delta I$, which yields $\Delta I = 1$. Hence the distance vector is (1) and the direction is (+)
- For the expression $A(i + 1, j, k) = A(i, j, k + 1) + B$ we have the equations

$$\begin{aligned} I_0 + 1 &= I_0 + \Delta I \\ J_0 &= J_0 + \Delta J \\ K_0 &= K_0 + \Delta K + 1 \end{aligned}$$

  which give $\Delta I = 1, \Delta J = 0, \Delta K = -1$. Hence the distance vector is $(1, 0, 1)$ with directions $(+, 0, -)$
- For the expression $A(i + 1) = A(i) + B(j)$ we have $I_0 + 1 = I_0 + \Delta I$ which gives $\Delta I = 1$. Since j has no source, its direction is unconstrained. The distance vector is given as $(+1, *)$

5.7 Challenges of Dependency testing

- Unknown loop bounds can lead to false dependences
- Need to be conservative about aliasing
- Triangular loops and new constraints