CS3006 - Parallel and Distributed Computing

Lecture 04 & 05: Parallel Computing Platforms

Syed Faisal Ali

email: faisal.ali@nu.edu.pk

National University of Computer and Emerging Sciences - FAST, Karachi Campus

Monday, Aug 28, FALL - 2023

Scope of Parallelism

- Conventional architectures coarsely comprise of a processor, memory system, and the datapath.
- Each of these components present significant performance bottlenecks.
- Parallelism addresses each of these components in significant ways.

Scope of Parallelism

- Different applications utilize different aspects of parallelism - e.g., data itensive applications utilize high aggregate throughput, server applications utilize high aggregate network bandwidth, and scientific applications typically utilize high processing and memory system performance.
- It is important to understand each of these performance bottlenecks.

Implicit Parallelism: Trends in Microprocessor Architectures

- Microprocessor clock speeds have posted impressive gains over the past two decades (two to three orders of magnitude).
- Higher levels of device integration have made available a large number of transistors.
- The question of how best to utilize these resources is an important one.

Implicit Parallelism: Trends in Microprocessor Architectures

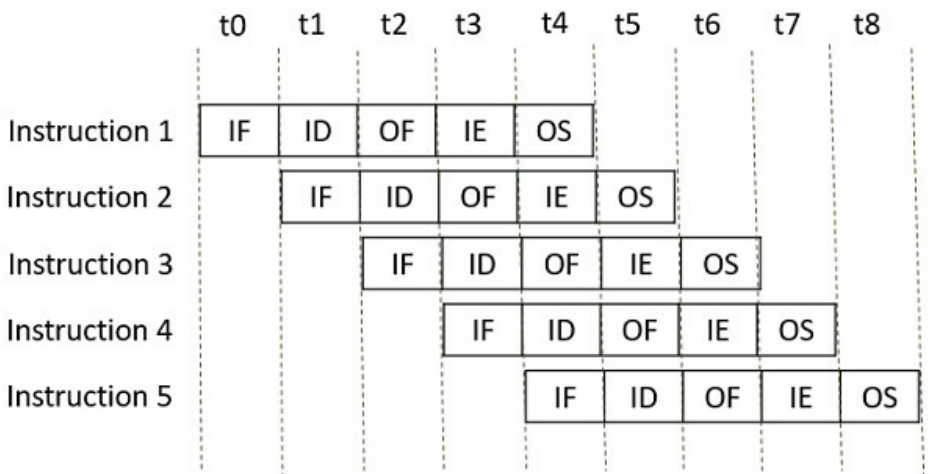
- Current processors use these resources in multiple functional units and execute multiple instructions in the same cycle.
- The precise manner in which these instructions are selected and executed provides impressive diversity in architectures.

Pipelining and Superscalar Execution

- Pipelining overlaps various stages of instruction execution to achieve performance.
- At a high level of abstraction, an instruction can be executed while the next one is being decoded and the next one is being fetched.
- This is similar to an assembly line for manufacture of cars.

Pipelining in Parallel Processing

- Pipelining is a technique where multiple instructions are overlapped during execution. Pipeline is divided into stages and these stages are connected with one another to form a pipe like structure. Instructions enter from one end and exit from another end. Pipelining increases the overall instruction throughput.



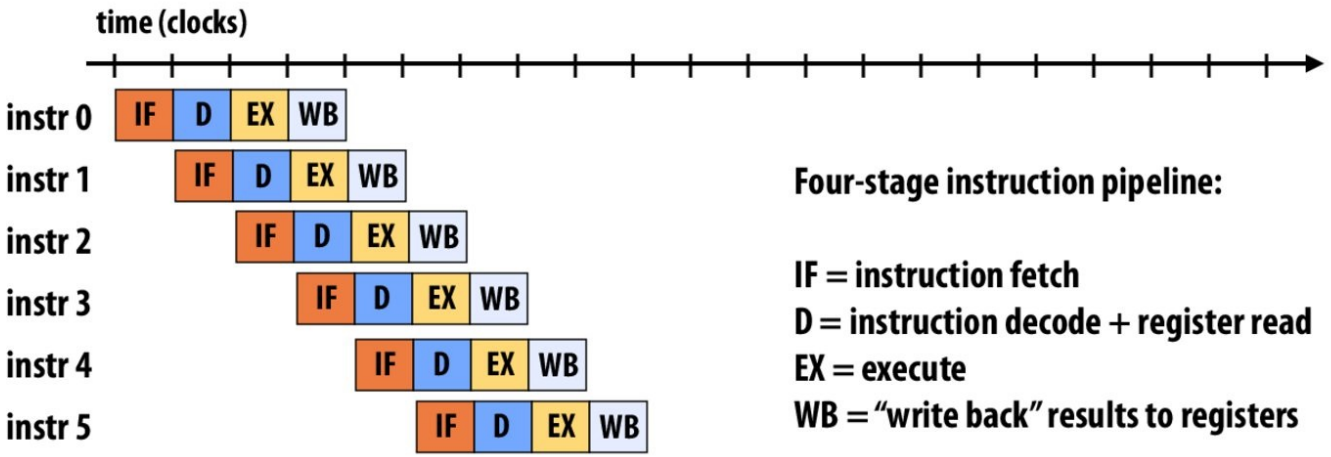
Pipelining of 5 Instructions

Pipelining in Parallel Processing

Another example: an instruction pipeline

Break execution of each instruction down into several smaller steps

Enables higher clock frequency (only a simple, short operation is done by each part of pipeline each clock)



Four-stage instruction pipeline:

IF = instruction fetch

D = instruction decode + register read

EX = execute

WB = "write back" results to registers

Latency: 1 instruction takes 4 cycles

Throughput: 1 instruction per cycle

(Yes, care must be taken to ensure program correctness when back-to-back instructions are dependent.)

Intel Core i7 pipeline is variable length (it depends on the instruction) ~15-20 stages

Figure: Pipeline in Parallel Processing

Pipelining and Superscalar Execution

Limitations of Pipelining

- Pipelining, however, has several limitations.
- The speed of a pipeline is eventually limited by the slowest stage.
- For this reason, conventional processors rely on very deep pipelines (20 stage pipelines in state-of-the-art Pentium processors).

Pipelining and Superscalar Execution

Limitations of Pipelining

- However, in typical program traces, every 5-6th instruction is a conditional jump! This requires very accurate branch prediction.
- The penalty of a misprediction grows with the depth of the pipeline, since a larger number of instructions will have to be flushed.
- One simple way of alleviating these bottlenecks is to use multiple pipelines.
- The question then becomes one of selecting these instructions.

Pipelining and Superscalar Execution

Superscalar Execution

- Superscalar architecture is a method of parallel computing used in many processors.
- In a superscalar computer, the central processing unit (CPU) manages multiple instruction pipelines to execute several instructions concurrently during a clock cycle.

Superscalar Execution

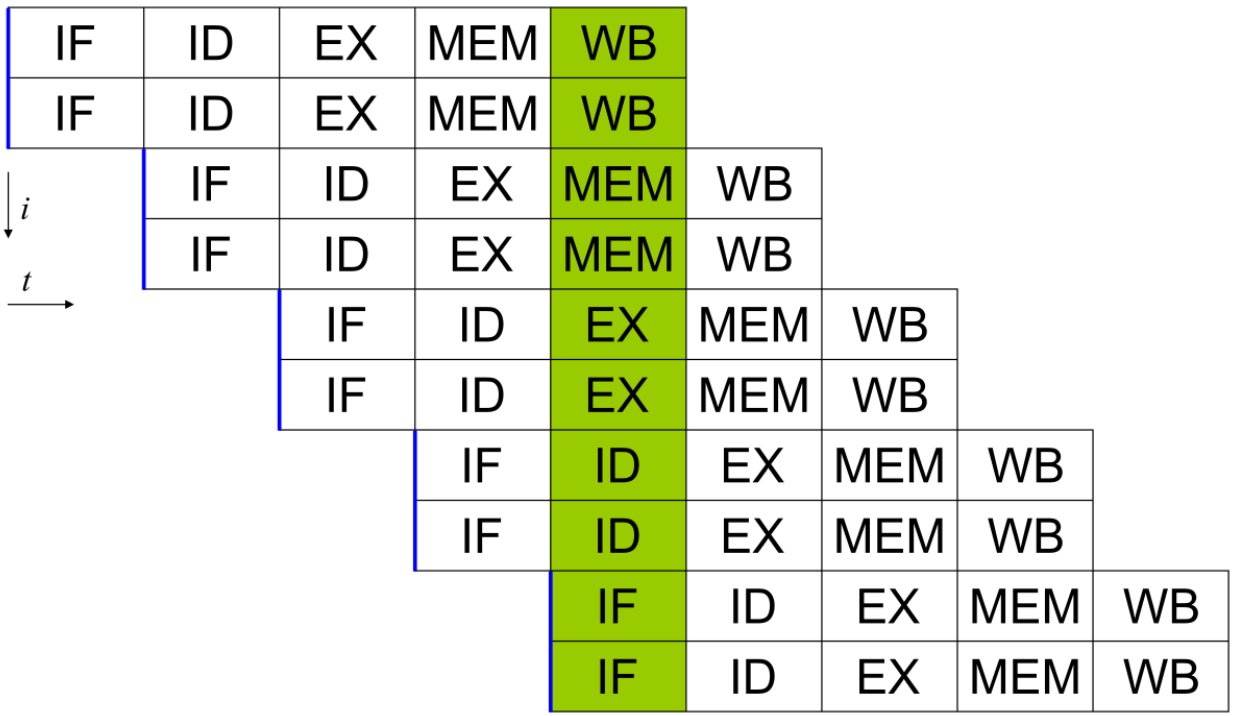


Figure: Superscalar Execution of Instruction Stream

Superscalar Execution: An Example

- Three different code fragments

```
1. load R1, @1000        1. load R1, @1000        1. load R1, @1000
2. load R2, @1008        2. add R1, @1004         2. add R1, @1004
3. add R1, @1004         3. add R1, @1008         3. load R2, @1008
4. add R2, @100C         4. add R1, @100C         4. add R2, @100C
5. add R1, R2            5. store R1, @2000       5. add R1, R2
6. store R1, @2000                                6. store R1, @2000
```

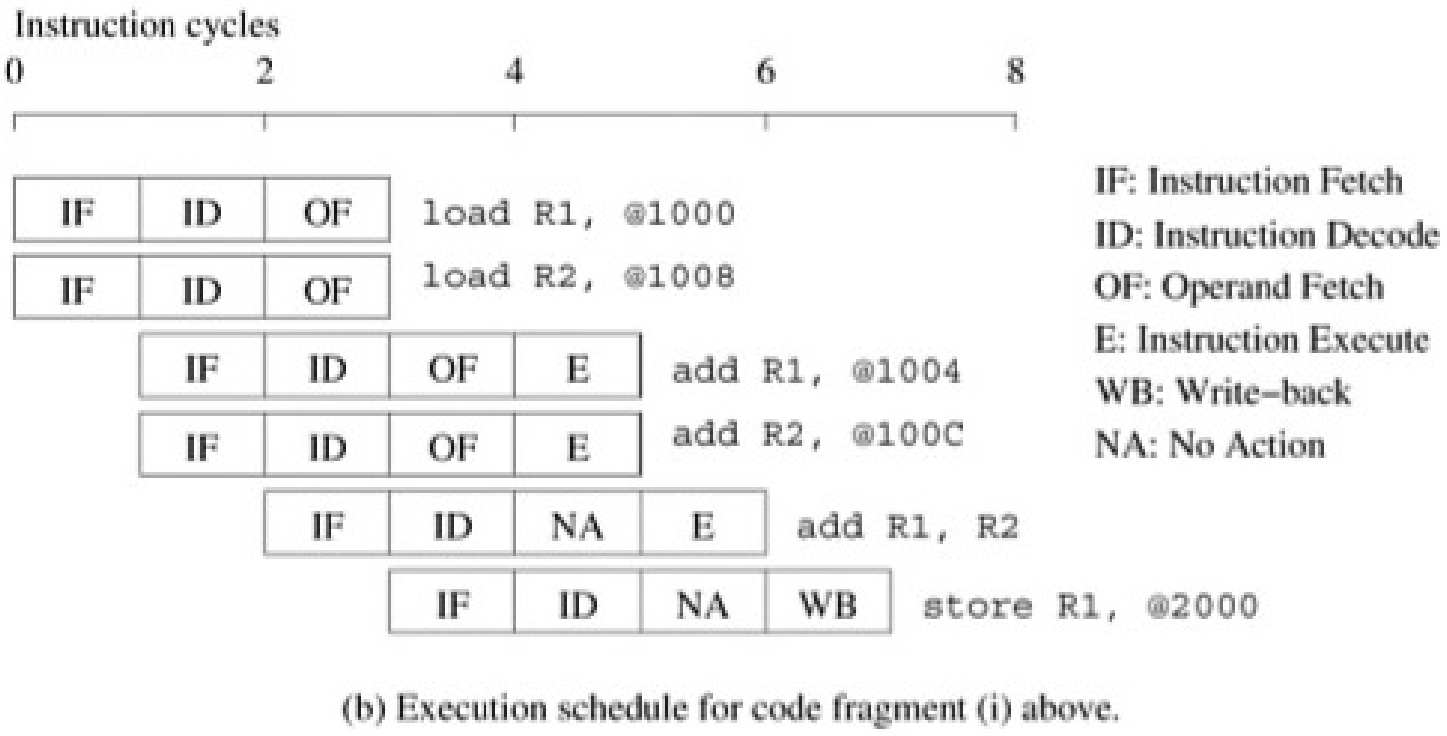
(i) (ii) (iii)

(a) Three different code fragments for adding a list of four numbers.

Figure: Three different code fragments for adding a list of four numbers.

Superscalar Execution: An Example

- Execution schedule.



(b) Execution schedule for code fragment (i) above.

Figure: Executing schedule for code fragment (i) above.

Superscalar Execution: An Example

- Hardware Utilization.

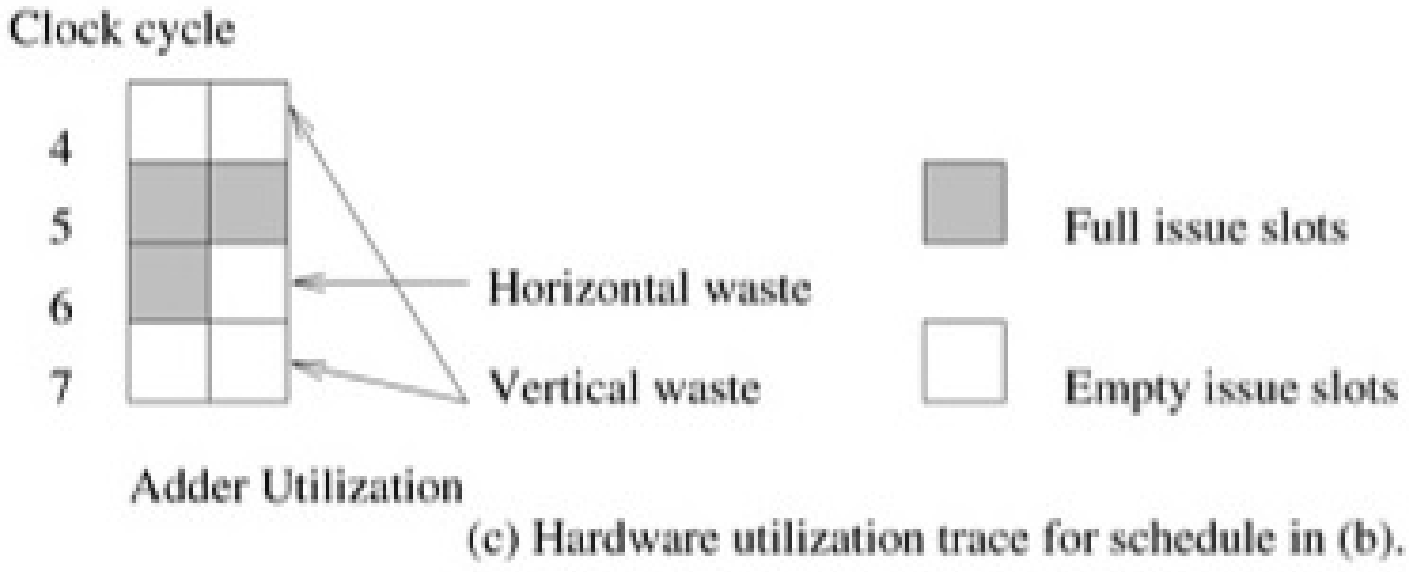


Figure: Hardware utilization trace for schedule in (b).

Superscalar Execution

- Consider the execution of the first code fragment in Figure for adding four numbers.
- The first and second instructions are independent and therefore can be issued concurrently.
- This is illustrated in the simultaneous issue of the instructions load R1, @1000 and load R2, @1008 at t = 0.
- The instructions are fetched, decoded, and the operands are fetched.
- The next two instructions, add R1, @1004 and add R2, @100C are also mutually independent, although they must be executed after the first two instructions.

Superscalar Execution

- Consequently, they can be issued concurrently at t = 1 since the processors are pipelined.
- These instructions terminate at t = 5. The next two
- instructions, add R1, R2 and store R1, @2000 cannot be executed concurrently since the result of the former (contents of register R1) is used by the latter.
- Therefore, only the add instruction is issued at t = 2 and the store instruction at t = 3.

Superscalar Execution

- Note that the instruction add R1, R2 can be executed only after the previous two instructions have been executed.
- The instruction schedule is illustrated in Figure 2.1(b).
- The schedule assumes that each memory access takes a single cycle.
- In reality, this may not be the case.
- The implications of this assumption are discussed in Section 2.2 on memory system performance.

Superscalar Execution

- In the above example, there is some wastage of resources due to data dependencies.
- The example also illustrates that different instruction mixes with identical semantics can take significantly different execution time.

Scheduling of Instructions

- Scheduling of instructions is determined by a number of factors:
- True Data Dependency: The result of one operation is an input to the next.
- Resource Dependency: Two operations require the same resource.
- Branch Dependency: Scheduling instructions across conditional branch statements cannot be done deterministically a-priori.

Scheduling of Instructions

- The scheduler, a piece of hardware looks at a large number of instructions in an instruction queue and selects appropriate number of instructions to execute concurrently based on these factors.
- The complexity of this hardware is an important constraint on superscalar processors.

Superscalar Execution: Issue Mechanisms

- In the simpler model, instructions can be issued only in the order in which they are encountered. That is, if the second instruction cannot be issued because it has a data dependency with the first, only one instruction is issued in the cycle. This is called in-order issue.
- In a more aggressive model, instructions can be issued out of order. In this case, if the second instruction has data dependencies with the first, but the third instruction does not, the first and third instructions can be co-scheduled. This is also called dynamic issue.
- Performance of in-order issue is generally limited.

Superscalar Execution: Efficiency Considerations

- Not all functional units can be kept busy at all times.
- If during a cycle, no functional units are utilized, this is referred to as vertical waste.
- If during a cycle, only some of the functional units are utilized, this is referred to as horizontal waste.
- Due to limited parallelism in typical instruction traces, dependencies, or the inability of the scheduler to extract parallelism, the performance of superscalar processors is eventually limited.
- Conventional microprocessors typically support four-way superscalar execution.

Very Long Instruction Word (VLIW) Machine

- It consists of many functional units connected to a large central register file.
- Each functional unit have two read ports and one write port.
- Register file would have enough memory bandwidth to balance the operand usage rate of functional units.

Ideal VLIW Machine

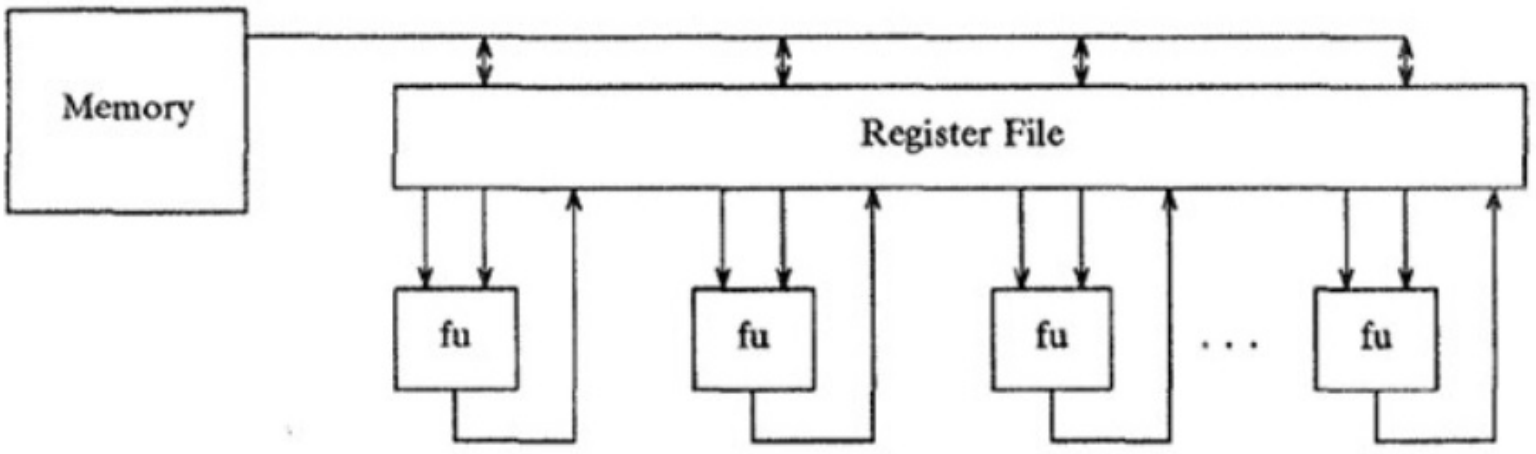


Figure: Block Diagram of an Ideal VLIW Execution Engine

Single VLIW Instruction

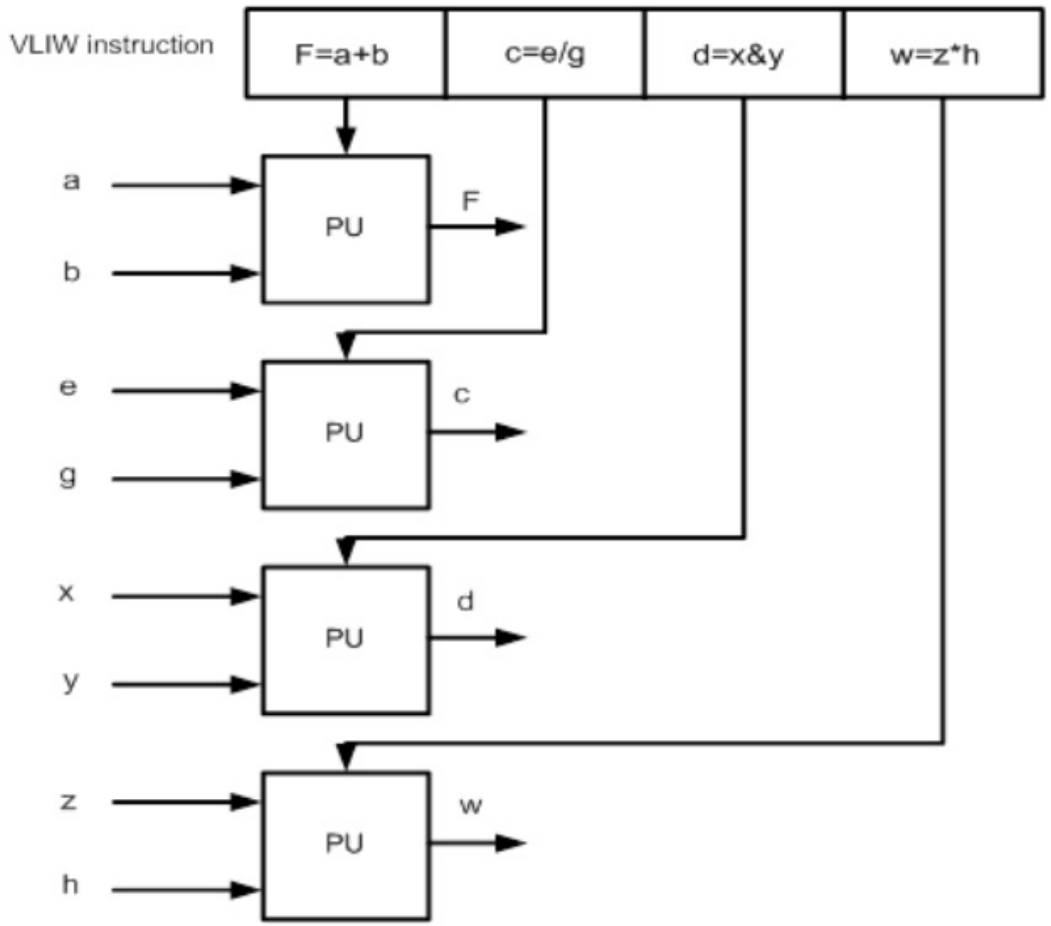


Figure: Example of a single VLIW instruction: F = a+b; c = e/g; d = x&y; w = z*h;

Very Long Instruction Word (VLIW) Processors

- The hardware cost and complexity of the superscalar scheduler is a major consideration in processor design.
- To address this issues, VLIW processors rely on compile time analysis to identify and bundle together instructions that can be executed concurrently.
- These instructions are packed and dispatched together, and thus the name very long instruction word.
- This concept was used with some commercial success in the Multiflow Trace machine (circa 1984).
- Variants of this concept are employed in the Intel IA64 processors.

Very Long Instruction Word (VLIW) Processors: Considerations

- Issue hardware is simpler.
- Compiler has a bigger context from which to select co-scheduled instructions.
- Compilers, however, do not have runtime information such as cache misses. Scheduling is, therefore, inherently conservative.
- Branch and memory prediction is more difficult.
- VLIW performance is highly dependent on the compiler. A number of techniques such as loop unrolling, speculative execution, branch prediction are critical.
- Typical VLIW processors are limited to 4-way to 8-way parallelism.

Limitations of Memory System Performance

- Memory system, and not processor speed, is often the bottleneck for many applications.
- Memory system performance is largely captured by two parameters, latency and bandwidth.
- Latency is the time from the issue of a memory request to the time the data is available at the processor.
- Bandwidth is the rate at which data can be pumped to the processor by the memory system.

Memory System Performance: Bandwidth and Latency

- It is very important to understand the difference between latency and bandwidth.
- Consider the example of a fire-hose. If the water comes out of the hose two seconds after the hydrant is turned on, the latency of the system is two seconds.
- Once the water starts flowing, if the hydrant delivers water at the rate of 5 gallons/second, the bandwidth of the system is 5 gallons/second.
- If you want immediate response from the hydrant, it is important to reduce latency.
- If you want to fight big fires, you want high bandwidth.

Memory Latency: An Example

- Consider a processor operating at 1 GHz (1 ns clock) connected to a DRAM with a latency of 100 ns (no caches). Assume that the processor has two multiply-add units and is capable of executing four instructions in each cycle of 1 ns. The following observations follow:
- The peak processor rating is 4 GFLOPS.
- Since the memory latency is equal to 100 cycles and block size is one word, every time a memory request is made, the processor must wait 100 cycles before it can process the data.

Memory Latency: An Example

- On the above architecture, consider the problem of computing a dot-product of two vectors.
- A dot-product computation performs one multiply-add on a single pair of vector elements, i.e., each floating point operation requires one data fetch.
- It follows that the peak speed of this computation is limited to one floating point operation every 100 ns, or a speed of 10 MFLOPS, a very small fraction of the peak processor rating!

Imporving Effective Memory Latency Using Caches

- Caches are small and fast memory elements between the processor and DRAM.
- This memory acts as a low-latency high-bandwidth storage.
- If a piece of data is repeatedly used, the effective latency of this memory system can be reduced by the cache.
- The fraction of data references satisfied by the cache is called the cache hit ratio of the computation on the system.
- Cache hit ratio achieved by a code on a memory system often determines its performance.

Impact of Caches: Example

- Consider the architecture from the previous example. In this case, we introduce a cache of size 32 KB with a latency of 1 ns or one cycle. We use this setup to multiply two matrices A and B of dimensions 32×32. We have carefully chosen these numbers so that the cache is large enough to store matrices A and B, as well as the result matrix C.

Impact of Caches: Example

- The following observations can be made about the problem:
- Fetching the two matrices into the cache corresponds to fetching 2K words, which takes approximately 200 µs.
- Multiplying two n $\times$ n matrices takes $2n^3$ operations. For our problem, this corresponds to 64K operations, which can be performed in 16K cycles (or 16 µs) at four instructions per cycle.
- The total time for the computation is therefore approximately the sum of time for load/store operations and the time for the computation itself, i.e., 200 + 16 µs.
- This corresponds to a peak computation rate of 64K/216 or 303 MFLOPS.

Impact of Caches

- Repeated references to the same data item correspond to temporal locality. In our example, we had $O(n^2)$ data accesses and $O(n^3)$ computation. This asymptotic difference makes the above example particularly desirable for caches.
- Data reuse is critical for cache performance.

Impact of Memory Bandwidth

- Memory bandwidth is determined by the bandwidth of the memory bus as well as the memory units.
- Memory bandwidth can be improved by increasing the size of memory blocks.
- The underlying system takes l time units (where l is the latency of the system) to deliver b units of data (where b is the block size).

Reading

1. Reading Assignment: Read Chapter 2: Parallel Programming Platforms from Introduction to Parallel Computing 2nd Edition by Ananth Grama.
2. Types of pipelining in parallel computing with diagrams?
3. What are the hazards in pipelining. Write them in detail.
4. When to use two way superscalar execution of instructions? Give suitable example.
5. Write down the formulas you have read in this chapter.

End of Lecture

Dua for Parents

رَّبِّ ارْحَمْهُمَا كَمَا رَبَّيَانِي صَغِيرًا

Rabbi irhamhuma kama rabbayanee sagheera

My Lord, have mercy on them, as they raised me when I was a child.

Quran 17:24