Exploring the Limits to Instruction Level Parallelism

CIS.655.M001.FALL22.
Computer Architecture 14082.1231,
Syracuse University
Term Paper - Final Report

Team:
Ritika Radhakrishnan - (SUID: 7142420691)
Avanni Gudimetla - (SUID: 7212879711)

Content

Page No.

Abstract:

To further advance the implementation of heavy-pipelining in machines, there is a need to know more in depth about Instruction Level Parallelism and how much further it can be exploited.
The ability of a computer's central processing unit, known as the CPU to carry out multiple instructions concurrently is known as instruction-level parallelism, abbreviated as ILP.
This can improve the performance of a computer by allowing it to process more data in a given amount of time. However, there are limitations to the amount of ILP that can be achieved, and understanding these limits is essential for designing efficient and effective computer systems.

The study of ILP has been an imperative field of study in computer architecture for many years. Advances in technology have allowed for greater levels of parallelism, but there are still limitations that must be considered. For example, the amount of ILP that can be achieved is limited by the complexity of the instructions being executed and by the amount of available memory. Additionally, the execution of instructions in parallel can introduce additional challenges, such as the need for synchronization and the potential for conflicts.

The main objective of this paper is to explore the limits of ILP and to provide insight into how these limits can be overcome. To do this, we will first provide a brief overview of the current state of ILP and the challenges it presents. We will then present our methods for studying ILP, including the techniques explored and the metrics we used to evaluate performance. Finally, we will demonstrate how techniques like renaming memory references, reducing actual stack dependencies, branch prediction, VLIW, SMT and vector techniques can raise the amount of exploitable parallelism.

1. Introduction :

There are dependencies within different pairs of instructions in a program. This limits the extent to which parallelism can be exploited. Some dependencies are required and should not be trifled with as it may represent the natural flow in the data. Other dependencies are setbacks which should be erased for optimal parallelisation. Code 1) provides an example of parallelism, whereas Code 2) provides an example of dependecies. Assembly code portion in 1) consists of three instructions that can be executed individually at the same time as they do not depend on each other's results. Here, Registers r1, r2, r3, r4, r5, r6, r7, r8 are in action. The assembly code portion in 2) does have dependencies, and so cannot be executed at the same time. Here, registers r1, r3, r4, r7, r8 are in play. The number of cycles needed divided by the total number of instructions is known as parallelism.
In Code 1) 3 instructions require 1 cycle to execute. Hence, the parallelism is 3, whereas in code 2) 3 instructions require 3 different cycles to execute hence, parallelism is 1. *The greater the parallelism the better.*

```
1)  Parallelism = 3       2) Parallelism = 1
    lw r4 20 (r2)               lw r4 20 (r2)
    add r3 r1 r5                add r3 r1 r4
    nand r6 r7 r8               nand r3 r7 r8
```

Let us consider what the ILP would be for an ideal machine. An ideal machine would be one which has a perfect branch prediction, no stalls, infinite registers, infinite instruction and instruction window size and perfect cache (these terms are explained further in the paper). Suppose we consider different benchmark programs and test the ILP on all these programs. Consider gcc, espresso, tomcatv, etc from the SPEC CPU92 Benchmark suite. An upper limit to ILP on the ideal machines would be as follows:

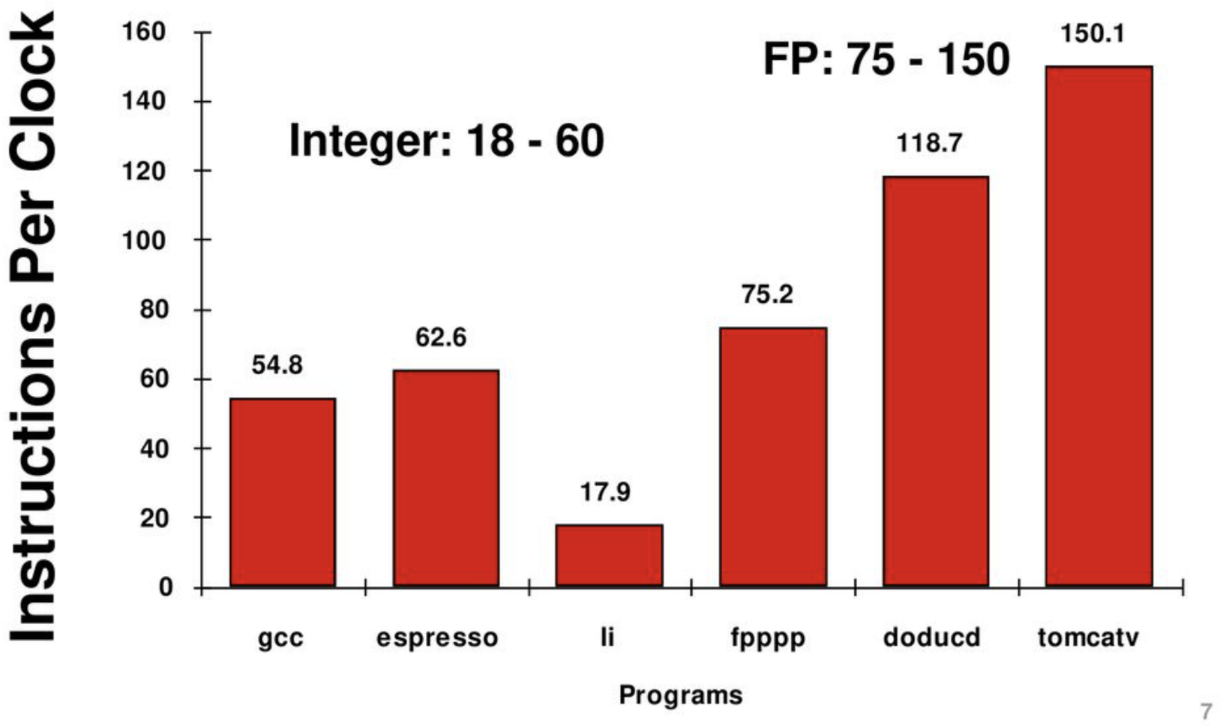


Figure 1: Upper limit to ILP on the ideal machines , Source : Chapter 3: Limits of Instruction-Level Parallelism , Published byDaniela Štěpánková

For an ideal machine, the number of instructions per cycle required is 39 on average and 112 for FP on average for these benchmark programs. In case, we start putting restrictions on the ideal machine and suppose we restrain the instruction window size, we get results as follows

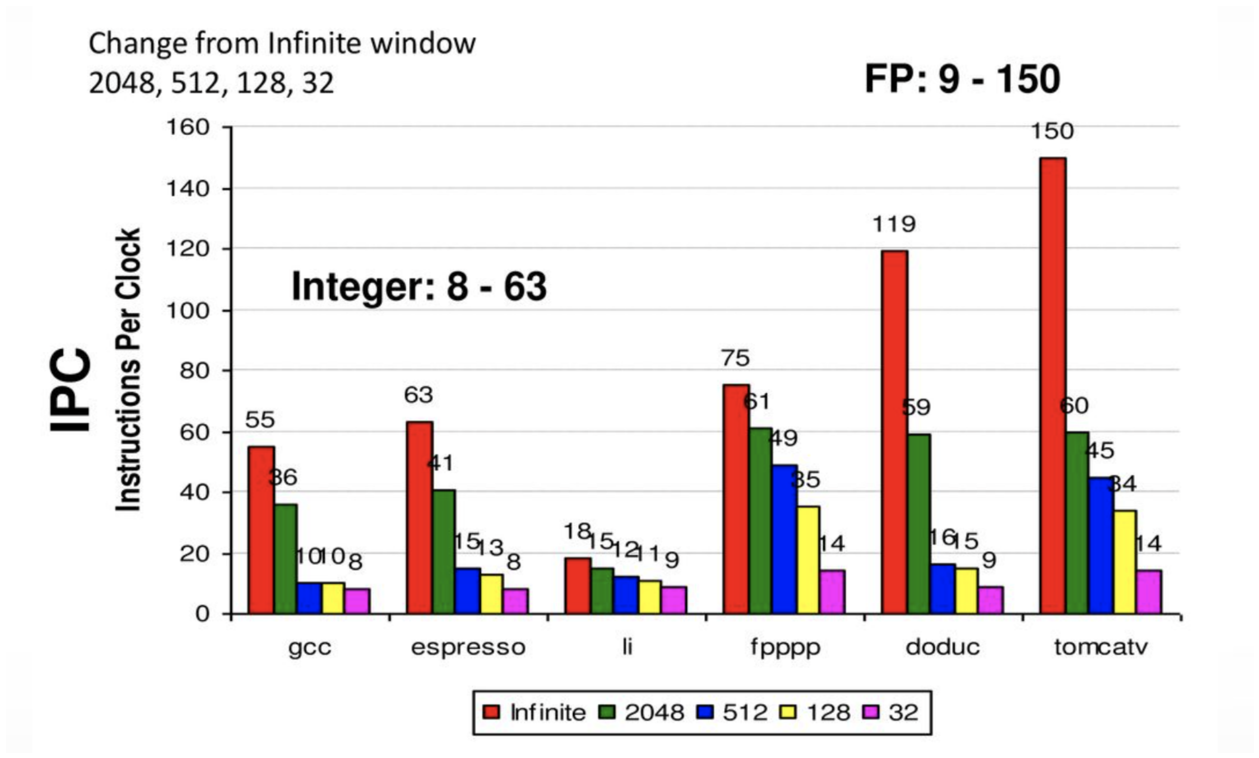


Figure 2: More realistic hardware: Window Impact , Source : Chapter 3: Limits of Instruction-Level Parallelism , Published byDaniela Štěpánková

The red portion denotes the number of cycles taken for an ideal machine, when the hardware is restricted to 2048, 512, 128 and 32 window size, we get different results for ILP. This goes to show even a slight restriction in our ideal model can lead to vastly different results. In real life, machines and processors are far from Ideal. In fact, we need to navigate Instruction level Parallelism in such a way that it considers the worst case condition for machines.

With the following analysis we understood that there is a need to come up with techniques to exploit Instruction level Parallelism, this is what we further discover in our paper.

2. Analysis into the techniques :

We will explore ways to increase the exploitation of instruction-level parallelism of programs by looking into various techniques that can be implemented individually as well as simultaneously.

2.1 Optimizing Block Parallelism :

2.1.1 Loop unrolling

Increasing parallelism within a block is one of the ways we can optimize block parallelism. There are software based approaches which would help to achieve this. One of the software techniques to do this would be Loop unrolling of the code portion within a block. This approach involves duplicating the instructions in a loop and executing them in a parallel manner. For example, if a loop consists of 'n' instructions, loop unrolling would duplicate these instructions and execute them in parallel, resulting in a n-fold increase in parallelism. This stretch has considerable advantages. Firstly, It can improve performance of a computer by increasing the Instruction-level parallelism achieved. Next, it can reduce the overhead associated with loop execution, such as the cost of branching and the time spent loading and storing loop variables. Additionally, it can improve the utilization of the CPU's functional units, allowing them to perform more work in a given amount of time.

However, there are limitations that come with loop unrolling. In computer science, it is said that the fewer lines in the code that have to be executed, the faster the system is. Loop unrolling increases the size of the code, which can not only reduce the amount of memory that is available but also increase the memory footprint. Finally, loop unrolling can cause us to deal with an increase in potential for conflicts and an ache for a need in synchronization.

2.1.2 Data-level parallelism

Another approach to increasing parallelism within blocks is to use 'data-level parallelism' also abbreviated as 'DLP'. This technique involves executing multiple data items in parallel, rather than multiple instructions. For example, if a block of code operates on an array of data, DLP would involve executing the operations on multiple elements of the array simultaneously. It can increase the amount of ILP achieved by a computer, which can improve its performance. Second, it can reduce the memory footprint, as it does not require duplicating instructions. Finally, it can improve the utilization of the CPU's functional units, allowing them to perform more work in a given amount of time.

However, DLP also has limitations. Data-level parallelism, just like loop unrolling, can cause us to deal with an increase in potential for conflicts and an ache for a need in synchronization. Another significant limitation is that the implementation is difficult to navigate, it requires careful and well thought partitioning of the data and maintaining the coordination of the execution of multiple threads. DLP is more advantageous over loop unrolling.

2.1.3 Register renaming

Now that the ways in which the software approaches can be optimized are discussed, there is a need to look into how the hardware can be exploited. Register Renaming comes into picture which is a hardware facility where the hardware will impose an ‘amount of indirection’ in between the actual register that is being used for the operations and the register number that appears in the instruction.

In a way, this approach will dynamically reassign the physical registers used by the CPU to store values, allowing multiple instructions to use the same register without causing conflicts. Once a register has been set by an instruction in the code, that existing register is used to do the operation by the hardware. This approach will allow hardware to incorporate more registers that will lead to a reduction in the False dependencies. The more the registers involved, the less this redundant problem can be avoided.Register renaming is thus, a dynamic approach of setting the register which can actually increase parallelism as compared to the usual static allocation done by the complier.

As register renaming allows the processor to execute instructions in parallel that would otherwise have to be executed sequentially, it can improve the performance of the computer. This performance enhancement is caused by allowing the CPU to process more data in a given amount of time. Another advantage of register renaming is that it can reduce the overhead associated with managing registers. In a traditional processor, each instruction is assigned a specific register to use, and this assignment must be tracked by the processor. With register renaming, the processor can dynamically reassign registers as needed, reducing the amount of tracking and management required.

However, register renaming also has limitations. One significant limitation is that it requires additional hardware resources, such as additional registers and logic for managing the register assignments. This can increase the cost and complexity of the processor. Additionally, register renaming can introduce additional challenges, such as the need for synchronization and the potential for conflicts. Further research is needed to explore the limitations of register renaming and to develop more effective approaches to increasing ILP.

2.2 Crossing the block bounds to optimize parallelism:

The average number of instructions between branches in a program is 6 to 7, it is not a very big value. Hence, we need to move beyond the boundaries of the loop and must be able to issue instructions from different basic blocks in parallel. Unfortunately, this comes with the need to know whether a conditional branch will be taken in advance, or else we will have to settle with the possibility that we do not know.

2.2.1 Branch prediction

Branch prediction is a technique used in computer architecture to improve the performance of a computer by allowing it to execute instructions in parallel. This paper will discuss how branch prediction helps to increase instruction-level parallelism (ILP) and the limitations of this approach.

Branch prediction is a way of guessing the outcome of a conditional branch in a computer program before it is actually executed. This allows the CPU to speculatively execute instructions based on the predicted outcome of the branch, which can improve performance by increasing the amount of ILP achieved. With pipelining you can organize multiple instructions simultaneously. The instruction is divided into the subtasks where each subtask will perform a dedicated task. The pipelining instructions are Fetch, Decode, Execute, Memory and Write back. With the help of branch prediction we can keep the pipeline busy at all times. Fetch and decode instructions of another instruction can be performed while execution of instructions before it takes place. In order to have greater parallelism, instructions should be executed across an unknown branch speculatively. Shadow registers can come into picture whose values are uncommitted until the branch is correctly predicted. Memory stores may not be executed speculatively. The compiler can take control and an instruction set can be designed with redundant instructions. These redundant instructions are squashable. An evaluation for a particular condition can be applied to the squashable instructions and these instructions will be bound to a condition that is being assessed in another instruction, and would be discarded by the hardware if the condition is not satisfied.

One limitation of branch prediction is that it can introduce additional overhead, as the CPU must spend time predicting branches and speculatively executing instructions. Additionally, incorrect predictions can result in wasted work, as the CPU must discard any instructions that were speculatively executed based on the incorrect prediction. Another limitation of branch prediction is that it is limited by the complexity of the branches being predicted. More complex branches are harder to predict accurately, which can reduce the effectiveness of branch prediction.

Branch prediction is a useful technique for increasing ILP and improving the performance of a computer. However, it is subject to limitations and must be used carefully to avoid introducing additional overhead or reducing the effectiveness of ILP. Further research is needed to explore the limitations of branch prediction and to develop more effective approaches to increasing ILP.

2.3. Out-of-order execution

Out-of-order execution is a technique that can be used to increase instruction-level parallelism (ILP) in a computer's central processing unit (CPU). This technique involves allowing the CPU to execute instructions in a different order than the order in which they were received, allowing the CPU to process more instructions in parallel.

To use out-of-order execution to increase ILP, the first step is to ensure that the CPU supports out-of-order execution. Most modern CPUs include support for out-of-order execution, but it may need to be enabled in the BIOS or through software settings.

Once out-of-order execution is enabled, the next step is to write or modify the code to take advantage of the additional parallelism provided by out-of-order execution. This can involve using techniques such as loop unrolling or data-level parallelism to increase the amount of ILP achieved by the CPU.

Finally, the performance of the code with out-of-order execution enabled can be evaluated using benchmarks or other performance measures to determine the extent to which ILP has been increased. This can help to identify any limitations or challenges associated with using out-of-order execution, and it can provide insight into how to further optimize the code to take advantage of the additional parallelism provided by out-of-order execution.

Out-of-order execution is a powerful technique for increasing ILP and improving the performance of a computer. By enabling out-of-order execution and writing or modifying code to take advantage of the additional parallelism provided by out-of-order execution, it is possible to achieve significant improvements in performance.

2.4. Simultaneous multithreading

Simultaneous multithreading (SMT) is a technique that can be used to increase instruction-level parallelism (ILP) in a computer's central processing unit (CPU).

Suppose we create a 6-wide superscalar machine for high performance for a program. But in reality, programs only execute a few instructions per cycle due to stall generated by dependencies. Hence, because of mispredictions, issue slots are wasted and most of the pipeline becomes empty and redundant. All this results in lesser optimization of parallelism because our end goal is to always run multiple programs at the same time in a pipeline. When there is lesser dependence between said programs/ threads we make better use of parallelism. Here is where SMT comes into picture, SMT involves allowing multiple threads to be executed on a single CPU core, allowing the CPU to process more instructions in parallel. It improves overall job throughput and gives better CPU utilization. Intel calls this ‘Hyperthreading’. Intel Hyper-Threading is a technology that allows a single CPU to appear as two virtual CPUs to the operating system. This allows the CPU to execute instructions from two different threads concurrently, improving the performance of the computer by increasing instruction-level parallelism (ILP). In SMT, instructions from more than one program can be processed in the same cycle, at any given stage. SMT will provide a single Virtual CPU for a single program. This ‘Virtual CPU’ is also called ‘context’ by hardware programmers. For some of this ‘context’, hardware can be replicated/shared depending on the size of the hardware and resources. For performance, Br History Buffer, Ret Addr Stack, load/store Q are replicated for each ‘context’, whereas hardware such as physical regfiles, caches and TLBs are shared among the contexts.
Thus with this approach we can get multiple programs to share the pipeline and increase the parallelism.

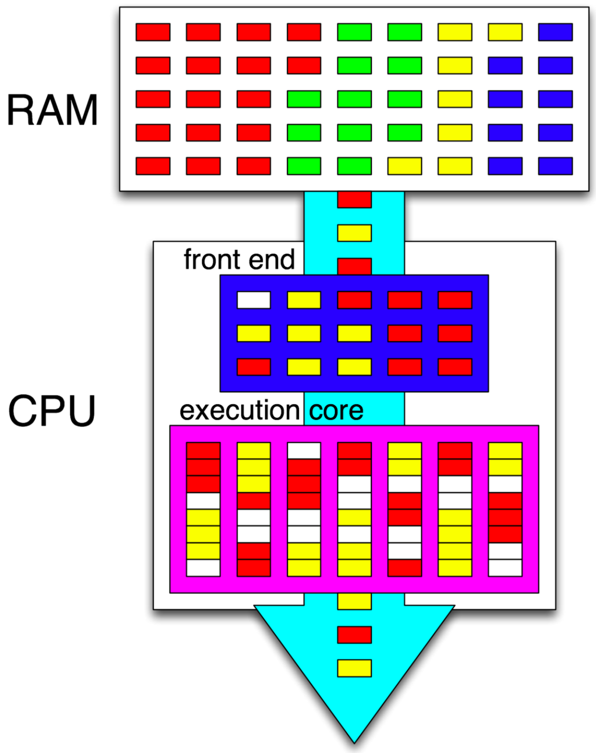


Figure 3: Hyperthreading, Source : Wikipedia

To use SMT to increase ILP, the first step is to ensure that the CPU supports SMT. Most modern CPUs include support for SMT, but it may need to be enabled in the BIOS or through software settings. Once SMT is enabled, the next step is to write or modify the code to take advantage of the additional parallelism provided by SMT. This can involve using techniques such as loop unrolling or data-level parallelism to increase the amount of ILP achieved by the CPU. Finally, the performance of the code with SMT enabled can be evaluated using benchmarks or other performance measures to determine the extent to which ILP has been increased. This can help to identify any limitations or challenges associated with using SMT, and it can provide insight into how to further optimize the code to take advantage of the additional parallelism provided by SMT, it is possible to achieve significant improvements in performance.

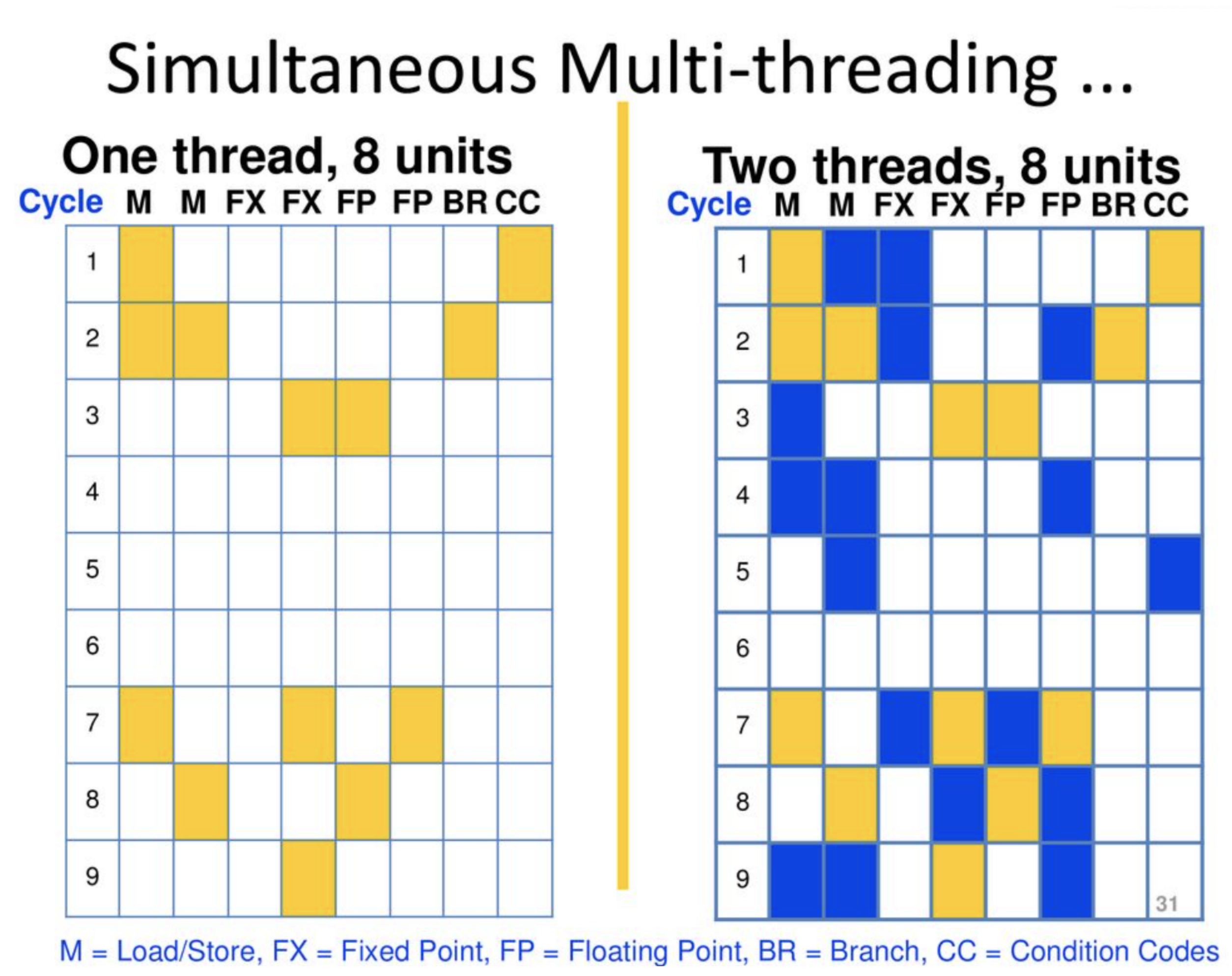

Simultaneous Multi-threading ...
One thread, 8 units
Two threads, 8 units
Cycle M M FX FX FP FP BR CC
1
2
3
4
5
6
7
8
9
31
M = Load/Store, FX = Fixed Point, FP = Floating Point, BR = Branch, CC = Condition Codes

Figure 4: SMT Source : Chapter 3: Limits of Instruction-Level Parallelism , Published byDaniela Štěpánková

In conclusion, SMT is a powerful technique for increasing ILP and improving the performance of a computer. By enabling SMT and writing or modifying code to take advantage of the additional parallelism provided by SMT, it is possible to achieve significant improvements in performance.

2.5. VLIW

Very long instruction word (VLIW) is a type of computer architecture that can be used to increase instruction-level parallelism (ILP). VLIW architectures are designed to allow the CPU to execute multiple instructions in parallel, without the need for complex hardware support for instruction-level parallelism.

In a VLIW architecture, the instructions are encoded into a single, very long instruction word. This instruction word includes multiple fields, each of which corresponds to a different instruction. The CPU then decodes the instruction word and executes the instructions in parallel, without the need for sophisticated hardware support for parallelism.

To use a VLIW architecture to increase ILP, the first step is to design or modify the code to take advantage of the VLIW architecture. This can involve writing the code in a way that allows the instructions to be encoded into a single, very long instruction word.

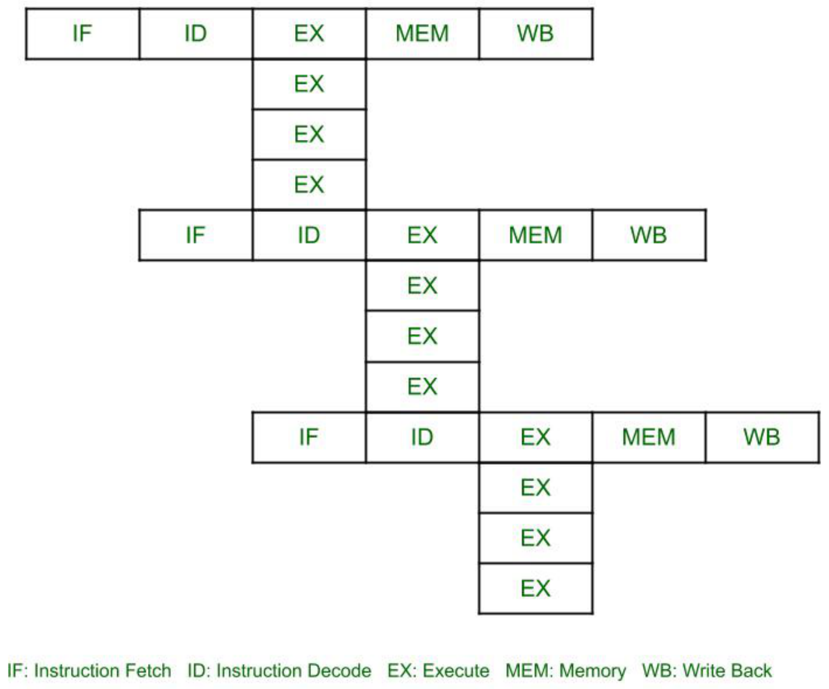


Figure 5: Time Space Diagram of VLIW Processor where four instructions are executed in parallel in a single instruction word Source : Blog, Geeks for Geeks, Very long instruction word vliw architecture

Once the code has been written or modified, the performance of the code can be evaluated using benchmarks or other performance measures to determine the extent to which ILP has been increased. This can help to identify any limitations or challenges associated with using a VLIW architecture, and it can provide insight into how to further optimize the code to take advantage of the additional parallelism provided by the VLIW architecture.

In conclusion, VLIW is a powerful technique for increasing ILP and improving the performance of a computer.

2.6. Speculative execution

Speculative execution is a technique that can be used to increase instruction-level parallelism (ILP) in a computer's central processing unit (CPU). This technique involves allowing the CPU to speculatively execute instructions that may be needed in the future, allowing the CPU to process more instructions in parallel.

To use speculative execution to increase ILP, the first step is to ensure that the CPU supports speculative execution. Most modern CPUs include support for speculative execution, but it may need to be enabled in the BIOS or through software settings.

Once speculative execution is enabled, the next step is to write or modify the code to take advantage of the additional parallelism provided by speculative execution. This can involve using techniques such as loop unrolling or data-level parallelism to increase the amount of ILP achieved by the CPU.

Finally, the performance of the code with speculative execution enabled can be evaluated using benchmarks or other performance measures to determine the extent to which ILP has been increased. This can help to identify any limitations or challenges associated with using speculative execution, and it can provide insight into how to further optimize the code to take advantage of the additional parallelism provided by speculative execution.

In conclusion, speculative execution is a powerful technique for increasing ILP and improving the performance of a computer. By enabling speculative execution and writing or modifying code to take advantage of the additional parallelism provided by speculative execution, it is possible to achieve significant improvements in performance.

2.7. Vectors to support ILP:

Instruction-level parallelism (ILP) is the degree to which a computer processor can execute multiple instructions in parallel. To increase ILP, a number of techniques can be used, including the use of vector instructions. Vector instructions allow a processor to perform the same operation on multiple data values simultaneously, which can increase the amount of parallelism within a program. For example, a vector instruction might allow a processor to add two vectors of numbers together in a single operation, rather than having to perform the addition one element at a time. Vector is technically a one-dimensional array of integers/digits/numbers, it is used in many scientific operations, multimedia, matrices, machine learning algorithms, and statistics to operate on vectors.

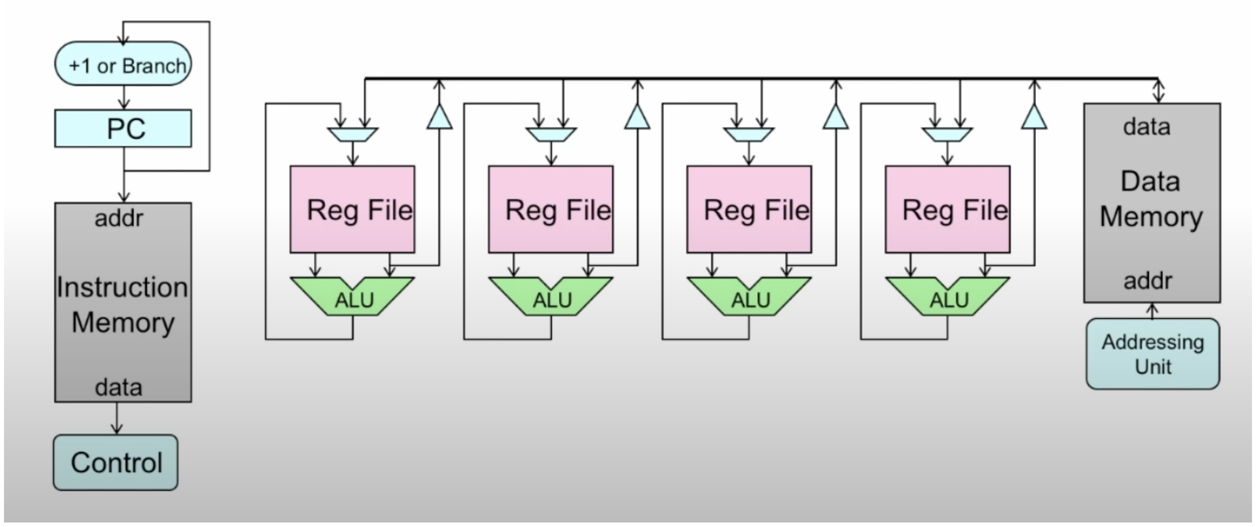
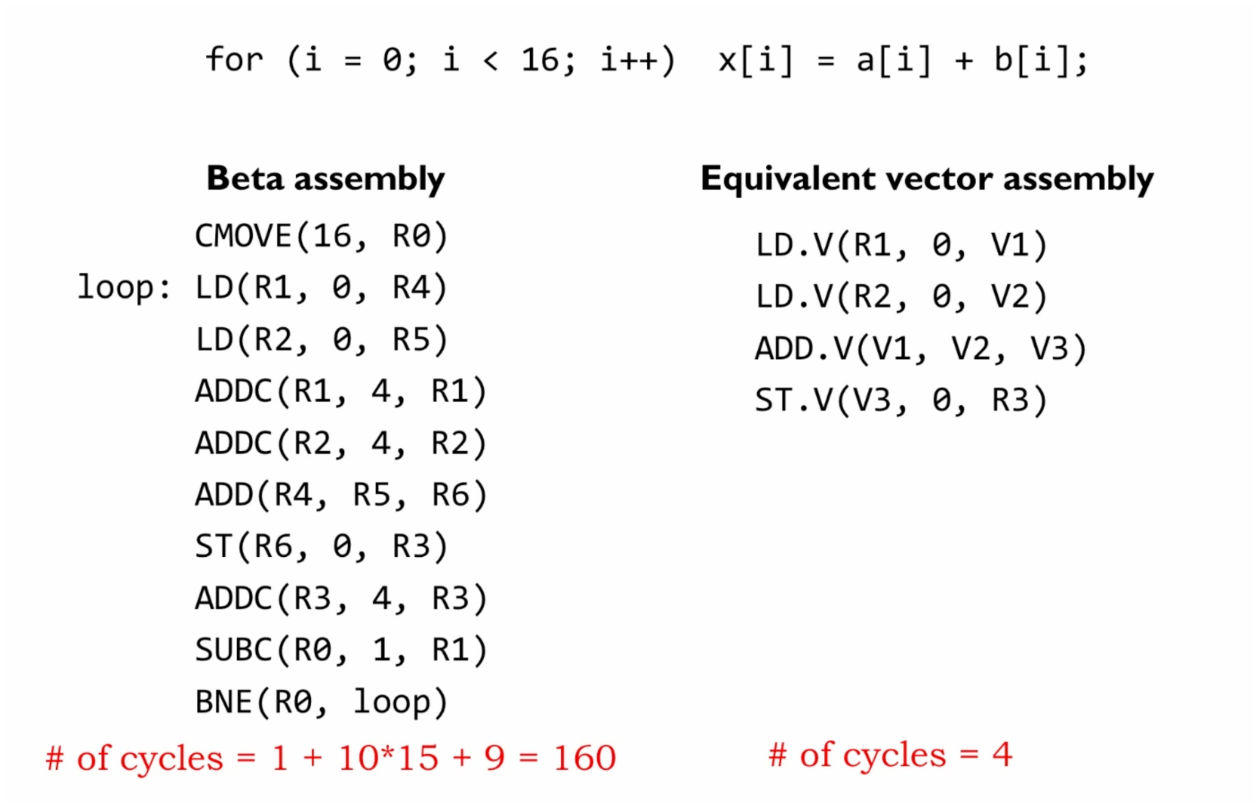


Figure 6: Vector Processors, Source : MIT OpenCourseWare

As mentioned before, one vector instruction will be operating on each and every element of the vector. Suppose we have an instruction lw x3, x1, x2. Here, x3 will perform on both x1 and x2. Now, comes the question - Why choose vectors? What benefits are we getting from using vectors? The answer to this lies with our need to want deeper pipelines. More occupied pipelines are difficult to achieve because of conditions like rename issue, bypass logic, stalls due to increase in data hazard, hard to issue multiple instructions per cycle, fetch and issue bottleneck that is, Flynn bottleneck, etc. This is where vector instructions come into picture, with Vectors, intra-vector data hazards are eliminated, intra vector interlocks are eliminated, no inner loop control hazards, need to issue multiple instructions to receive multiple operations is eliminated. In a way, vectors are the key with which the hardware can get a memory access pattern.



```
for (i = 0; i < 16; i++)  x[i] = a[i] + b[i];
```

Beta assembly

```
      CMOVE(16, R0)
loop: LD(R1, 0, R4)
      LD(R2, 0, R5)
      ADDC(R1, 4, R1)
      ADDC(R2, 4, R2)
      ADD(R4, R5, R6)
      ST(R6, 0, R3)
      ADDC(R3, 4, R3)
      SUBC(R0, 1, R1)
      BNE(R0, loop)
```

# of cycles = 1 + 10*15 + 9 = 160

Equivalent vector assembly

```
LD.V(R1, 0, V1)
LD.V(R2, 0, V2)
ADD.V(V1, V2, V3)
ST.V(V3, 0, R3)
```

# of cycles = 4

Figure 7: Vector Code , Source : MIT OpenCourseWare

In the figure above, we see that if beta assembly instructions are used we will require 160 cycles to run the program whereas if we convert the same instructions into equivalent vector assembly, only 4 cycles are required for the program. The high-level code can be converted to vector instructions with the help of the compiler. This is known as ‘automatic vectorization’.

Let us talk about the different vector architectures.Vector-register machines are a load store type of computer architecture that is designed to support efficient vector processing of data. Vector operations use vector registers except load and store. In a vector-register machine, the main memory is organized into a large number of registers, each of which can hold a vector of data elements. The instructions executed by the processor operate on these registers and can perform vector operations such as element-wise addition, multiplication, and other operations on the data vectors stored in the registers.

Memory-memory vector machines are a type of computer architecture that is designed to support efficient vector processing of data. In a memory-memory vector machine, the main memory is organized into a large number of registers, each of which can hold a vector of data elements. The instructions executed by the processor operate on these registers and can perform vector operations such as element-wise addition, multiplication, and other operations on the data vectors stored in the registers. Memory-memory vector machines are similar to vector-register machines, but they differ in the way that they access and operate on the data vectors stored in the main memory. Some examples of memory-memory vector machines include TI ASC, CDC STAR-100.

Overall, the use of vector instructions and the study of vector architecture are just one of the many techniques that can be used to increase instruction-level parallelism and improve the performance of a computer processor.

2.8. DLXV Architecture

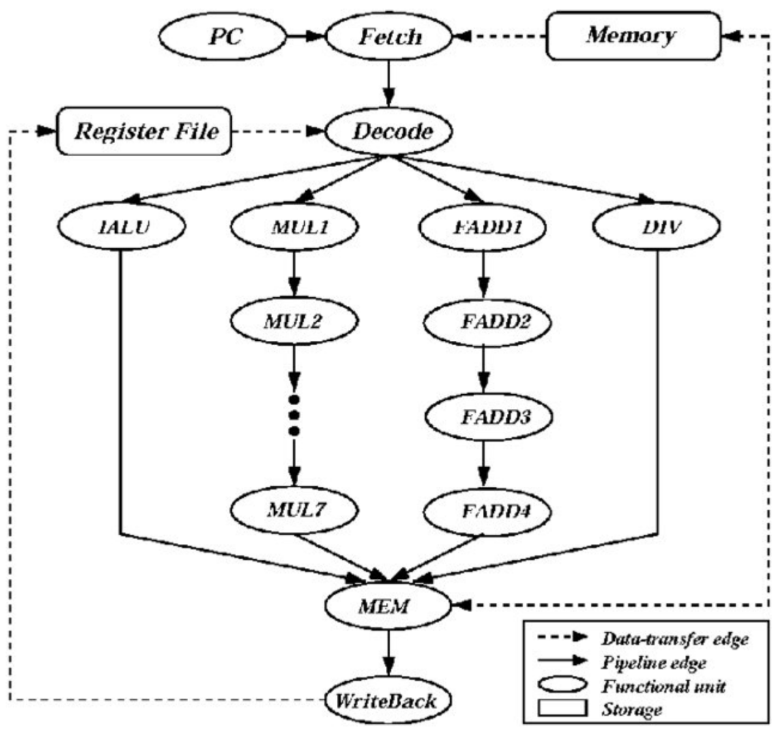


Figure 8: DLXV Architecture, Source : Magdy S Abadir from Researchgate publication - 220309598, A methodology for validation of microprocessors using symbolic simulation

DLXV is a simplified, educational architecture that was designed to demonstrate the principles of instruction-level parallelism (ILP) and superscalar processors. It is not a real, commercially-available processor architecture. It is strongly based on CRAY-1. It requires 8 registers from V1-V8 (vectors) that are a total of 2^12 bytes and each one is 64 double-precision FP.

DLXV extends DLX (baby pipeline). The DLXV architecture is based on the DLX architecture, which was a reduced instruction set computing (RISC) architecture developed in the early 1990s. Like DLX, DLXV uses a simple, fixed-length instruction set and a load/store architecture, which allows the processor to access memory directly only for load and store operations. The DLXV architecture consists of vector functional units which are five in total. They are as follows: FP*, FP/, FP+, logical and integer that will be deeply pipelined with stages in the range of 2 to 25. Vector load and store units will also be present with fully pipelined implementation in the range of 10 to 55.

The DLX architecture consists of vector-vector instructions, vector-scalar instructions, vector load-store instructions, load/store vector with stride and vector load/store indexed.
In vector-vector instructions, operation is done on tw vectors and a third new vector is produced. Suppose the loop prior to vectorization was as follows, e.g.

```
for (i ranges from 1 to n) :
        x3[i] = x2[i], x1[i]
```

We can transform it to ‘add x3, x2, x1’ which consists of having no branches and no hazards and the entire loop is saturated to a single instruction.
In vector ~ scalar instructions, operation is carried out on one scalar and one vector individually. Suppose the loop, e.g.

```
 for (i ranges from 1 to n) :
        a[i] = f5 + b[i]
```

We can transform it to ‘addv a, f5, b’ which will justify vector-scalar instructions.
In vector load-store instructions, the instructions operate on contiguous addresses. A vector from memory is either loaded or stored (lv, sv) into a vector register. Suppose, consider the following vector load-store instructions, e.g.

```
lv x, y ; x[i] = M[Y+i]
sv y, x ; M[Y+i] = x[i]
```

For load and store vector with stride, a non-unit stride is added on each access and the vectors are not always contiguous in memory, e.g.

```
lvws x, (a,b) ; x[i] = M[a+i*b]
svws (a,b), v1 ; M[a+i*b] = x[i]
```

In indexed vector load and store, through an index vector there are indirect accesses, e.g.

```
lvi x, (a+y) ; x[i] = M[a+y[i]]
svi (a+y), x ; M[a+y[i]] = x[i]
```

Hence, these are the different kinds of instructions in DLXV architecture.

To increase ILP, the DLXV architecture uses a number of techniques, including instruction pipelining and out-of-order execution. Instruction pipelining allows the processor to fetch, decode, and execute instructions simultaneously, which can increase the throughput of the processor. Out-of-order execution allows the processor to execute instructions in a different order than they appear in the program, which can help to hide delays and improve the efficiency of the processor. Overall, the DLXV architecture is a useful educational tool for demonstrating the principles of ILP and superscalar processors, but it is not a real, commercially-available processor architecture.

3. Future Scope :

The scope for Instruction Level Parallelism in the future is very promising, as there is still a lot of room for improvement in terms of how effectively processors can utilize this technique to make the processor run faster.

One potential area of future development for ILP is in the use of more advanced techniques for detecting and exploiting parallelism in a program's instruction stream. For example, machine learning and deep learning algorithms could be used to learn the patterns in a program that indicate the presence of parallelism, and to automatically generate parallel versions of the program that can be executed on a processor with ILP support. The integration of Artificial intelligence and hardware is currently being streamlined and has a very promising reward in terms of faster results. Another potential area of development is in the use of ILP to support new types of workloads and applications. For example, ILP could be used to improve the performance of machine learning algorithms, which are becoming increasingly important for a wide range of applications.

Additionally, ILP could be used to enable the use of new types of parallel programming models, such as dataflow or stream processing. The demand to streamline these models have been rapidly increasing each year due to the high demand of high quality video games, video streaming services, Automated reality, and other real time applications that require a high amount of real time data. Hence, ILP can improve the performance of real-time applications like gaming or virtual reality.

Overall, the future scope for ILP is very promising, and we can expect to see continued improvements in the effectiveness of this technique as research and development in this area continues.

4. Conclusions :

If the software or hardware does good branch prediction it paramounts to a high level of instruction-level parallelism exploitation. The better the prediction the stronger the concept of ‘branch prediction’ is. Register renaming plays a great role in the streamlining of our paper as well, with the assistance of static analysis, the compiler might be able to do a requisite task as well if we tune it in a way to exploit as much parallelism as we want. We can also significantly reduce the penalty for indirect jumps which can help us with parallelism. Unfortunately, for non-penalty cycles, the effect is not so much.

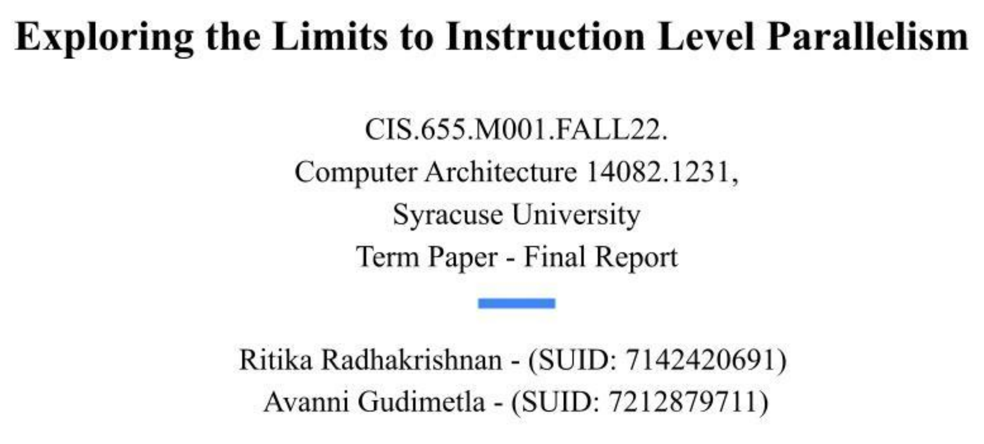
With the help of the techniques we have explored in our paper, we get a good understanding of the factors which have a very good effect on parallelism and it’s Exploitation. Instruction-level parallelism is considered one of the most important topics in computer architecture and hardware studies as it affects the question of ‘how fast a system can be’, positively to a great extent.

5. References:

1) D. W. Wall. “Limits of instruction-level parallelism”, In Proc. ASPLOS-4, volume 26, pages 176-189, April 1991.
2) Bernard Goossens, David Parello, “Limits of Instruction-Level Parallelism Capture” at the International Conference on Computational Science, ICCS 2013
3) David W. Wall. Limits of instruction-level parallelism. Technical Report DEC-WRL-93-6, Digital Equipment Corpora-tion, Western Research Lab, November 93.
4) Norman P.Jouppi David W. Wall, “Available Instruction-Level Parallelism for Superscalar and Superpipelined Machines” In Proc. ASPLOS-3, volume 24, pages 272-282, May 1989.

5) M. Butler, T.Y. Yeh, Y. Patt, M. Alsup, H. Scales, and M. She- banow. Single instruction steam parallelism is greater than two. In the 18th Annual International Symposium on Computer Architecture, pages 276–286, May 1991.
6) Dean M. Tullsen, Susan J. Eggers, and Henry M. Levy, "Simultaneous Multithreading: Maximizing On-Chip Parallelism", Department of Computer Science and Engineering University of Washington.
7) Exploiting ILP with Software Approaches by Dr A. P. Shanthi
8) An Approach for Compiler Optimization to Exploit Instruction Level Parallelism by Rajendra Kumar & P. K. Singh
9) Exploiting Instruction-Level Parallelism for Memory System Performance by Vijay S. Pai
10) Tjaden, Garold S., and Flynn, Michael J. ''Detection andParallel Execution ofIndependent Instructions. " IEEE Transactions on Computers C-l 9, 10 (October 1970), 889-895.
11) https://engineering.purdue.edu/~ee565/slides/ch3.pdf

Appendix:


Exploring the Limits to Instruction Level Parallelism

CIS.655.M001.FALL22.
Computer Architecture 14082.1231,
Syracuse University
Term Paper - Final Report

Ritika Radhakrishnan - (SUID: 7142420691)
Avanni Gudimetla - (SUID: 7212879711)

Outline

- Introduction
- Techniques
    - Optimising Block Parallelism
    - Crossing the block bounds to optimise parallelism
    - Out of Order Execution
    - Simultaneous Multithreading
    - VLIW
    - Speculative Execution
    - Vectors to support ILP
    - DLXV Architecture
- Future Work

Introduction

- ILP is the simultaneous processing of multiple instructions.
- Difficult to do in a single code block and is typically done in MIPS programs by having a dynamic branch frequency of between 15% and 25%
- Data hazards are likely to exist within these instructions, since they are likely to be dependent on each other.
- To take advantage of ILP, it is necessary to exploit it across multiple blocks of code.

Optimising Block Parallelism

- **Loop unrolling -** Duplicating the instructions in a loop and executing them in a parallel manner.
-
- **Data-level parallelism -** Executing multiple data items in parallel, rather than multiple instructions.
-
- **Register renaming -** Dynamically reassign the physical registers used by the CPU to store values and allow multiple instructions to use the same register without causing conflicts

Crossing the black bounds to optimise parallelism

Branch prediction : Improves the performance of a computer by allowing it to execute instructions in parallel.

Out-of-order execution & Simultaneous multithreading

- **Out-of-order execution** allows CPU to execute instructions in a different order than the order in which they were received, allowing the CPU to process more instructions in parallel.

- **SMT** involves allowing multiple threads to be executed on a single CPU core, allowing the CPU to process more instructions in parallel.

VLIW & Speculative execution

- **VLIW** - **Very long instruction word** architectures are designed to allow the CPU to execute multiple instructions in parallel, without the need for complex hardware support for instruction-level parallelism.

- **Speculative execution** involves allowing the CPU to speculatively execute instructions that may be needed in the future, allowing the CPU to process more instructions in parallel.

Vectors to support ILP

Vector instructions allow a processor to perform the same operation on multiple data values simultaneously, which can increase the amount of parallelism within a program.

Future Scope

- Looks very promising
- Lot of room for improvement to speed up the processor
- Exploiting parallelism in a program's instruction stream
- Combining machine learning algorithms that will automatically generate parallel versions of the program
- ILP can improve the performance of machine learning algorithms