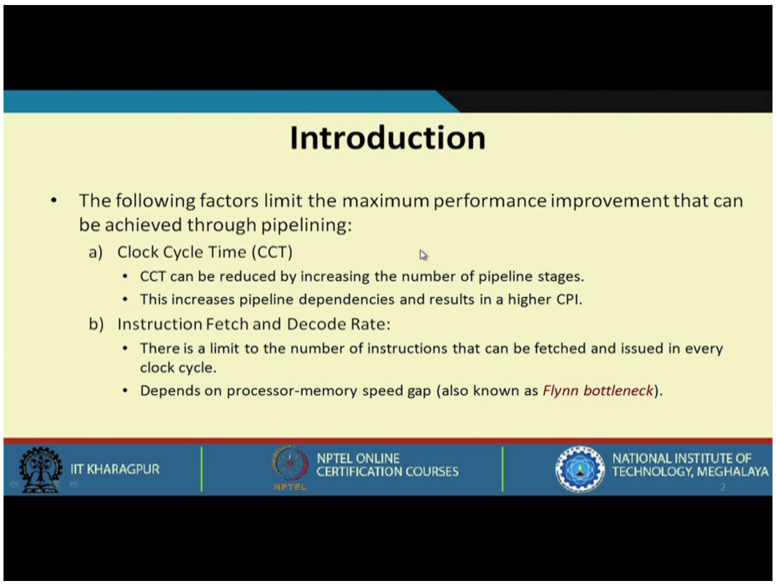
Computer Architecture and Organization
Prof. Indranil Sengupta
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

Lecture – 61
Vector Processors

In our last lecture we looked at various ways of increasing the instruction level parallelism in a program. We looked at a very well known technique called loop unrolling, which is used by compilers to expose more parallelism in a program, and also enables the compiler to do instruction scheduling or move instructions around to reduce the number of stall cycles. We also saw the so-called superscalar and VLIW architectures, which can enhance the performance of a pipeline by allowing more than one instructions to be issued in every clock cycle. In this lecture let us look one step further, and see how we can further parallelize computations, but we are looking at specific kinds of computations. Computations that are carried out on something called vectors.

(Refer Slide Time: 01:26)


Introduction

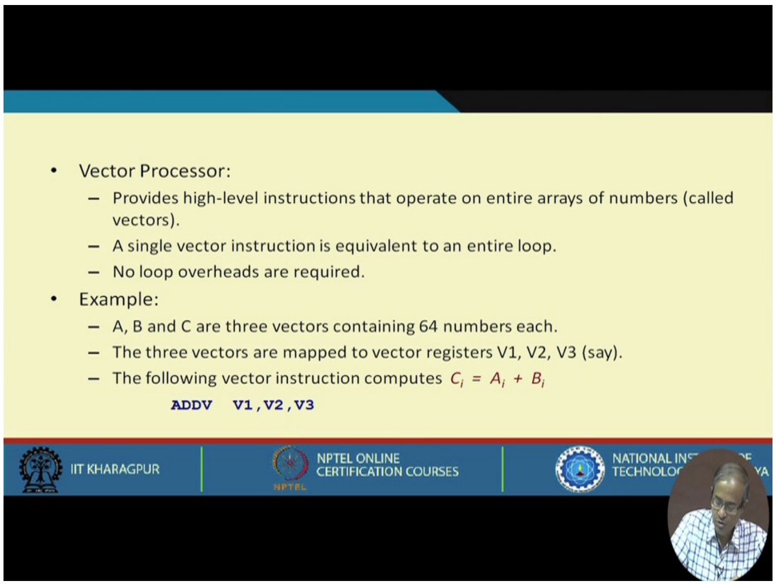
- The following factors limit the maximum performance improvement that can be achieved through pipelining:
  a) Clock Cycle Time (CCT)
     - CCT can be reduced by increasing the number of pipeline stages.
     - This increases pipeline dependencies and results in a higher CPI.
  b) Instruction Fetch and Decode Rate:
     - There is a limit to the number of instructions that can be fetched and issued in every clock cycle.
     - Depends on processor-memory speed gap (also known as *Flynn bottleneck*).

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

The topic of today’s lecture is vector processors. Let us try to see first that how we can look at the constraints in a pipeline and what are the factors that limit its performance. Broadly there are two factors: the first factor relates to the clock cycle time, like you see in a pipeline if you can make the clock cycle time faster than obviously, your instruction execution will become faster. Your instructions will execute faster, you can issue instructions at a faster rate, and this clock cycle time can be reduced by making the pipeline stages simpler, which means increasing the number of stages. But if you increase the number of stages, there will be one difficulty. You see here earlier in the MIPS32 pipeline even though there were 5 stages only, you saw that there were lot of dependencies and hazard creating scenarios that were coming. By using forwarding and other techniques we could reduce most of them, but still for certain dependencies we had to use some stall cycles.

So, if you increase the number of stages even further, this constraint will become even more pronounced, and more kinds of dependencies will be showing up. Clock cycle time reduction by increasing pipeline stages can increase dependencies in general, which can result in a higher CPI. The second constraint is the instruction fetch and decode rate. Here what we have saying is that the instructions have to be ultimately fetched from memory. Earlier we were saying one instruction has to fetched per cycle, but now when we are moving into superscalar and other kind of parallel computation, we are demanding that in every clock cycle we need to fetch more than one instructions. Well of course, this has something to do with the why we are organizing our memory. You have studied earlier that if we use memory interleaving, we can have some kind of parallel access. For instance for a 4-way memory interleaving we can access or read 4 words in every clock cycle. Organization of the memory also produces a constraint to the maximum memory-CPU bandwidth.

(Refer Slide Time: 04:41)

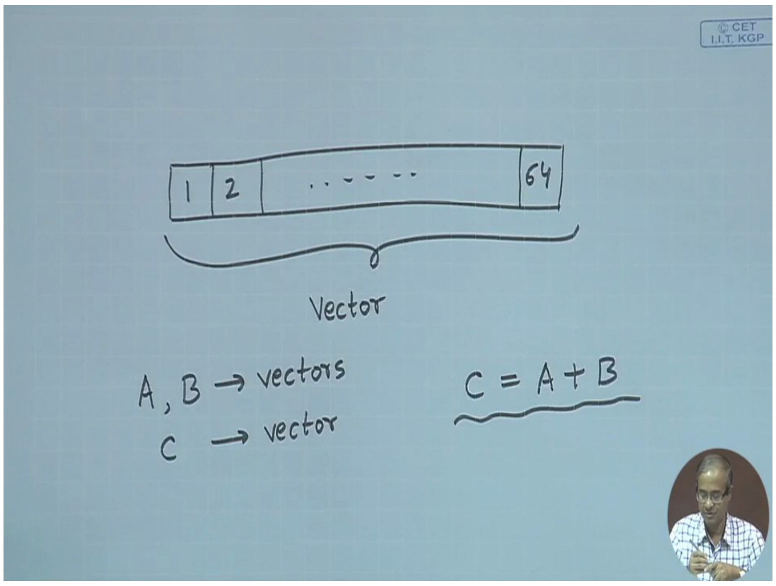


- Vector Processor:
  - Provides high-level instructions that operate on entire arrays of numbers (called vectors).
  - A single vector instruction is equivalent to an entire loop.
  - No loop overheads are required.
- Example:
  - A, B and C are three vectors containing 64 numbers each.
  - The three vectors are mapped to vector registers V1, V2, V3 (say).
  - The following vector instruction computes $C_i = A_i + B_i$

```
ADDV   V1,V2,V3
```

This processer-memory speed gap is sometimes also referred to as Flynn bottleneck. In general there is a limit to the number of instructions that can be fetched in every clock cycle. Now that we are moving on to vector processors, let us see what a vector processor is. Well here we are operating on entire arrays of numbers called vectors. Let us say there are a total of 64 numbers. All the 64 numbers taken together, we are referring this as a vector.

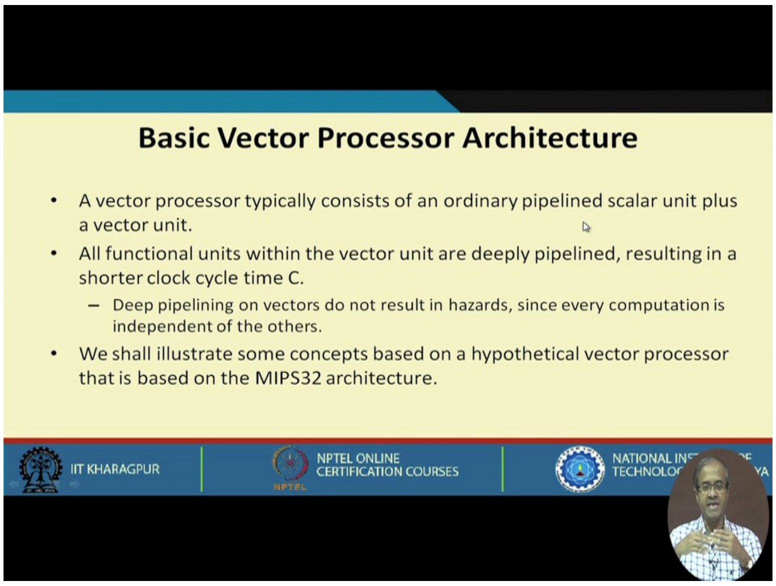
(Refer Slide Time: 05:15)



Let us say A is a vector, B is also a vector. Let us say C is also a vector, where you want to store the result. I can simply write an instruction like C = A + B. This is an example of vector instruction where although we had using a single instruction, but actually 64 additions are taking place. The first element of A is added to the first element of B, second element of A is added to the second element of B, and so on, and the results are stored in corresponding elements of the vector C. In a conventional processer where these kinds of vector operations are not there, we had to use a loop. In every loop we have to add one pair of numbers, then decrement a loop counter, check for 0 or some condition, then again branch back.

There you see some loop overheads were there, decrementing a counter, branching, but here it is a single instruction and there are no loop overheads. So, expectedly it should run faster. As I had said single vector instruction is equivalent to an entire loop, and I had said no loop overheads are required. For the example for adding two vectors, let us say the instruction will look like this ADDV V1,V2,V3. The vectors V2 and V3 are added element by element, and result stored in V1. The single instruction can compute a set of 64 additions. Each of the elements can be, for example, a double precision 64-bit number.

(Refer Slide Time: 07:40)



Basic Vector Processor Architecture

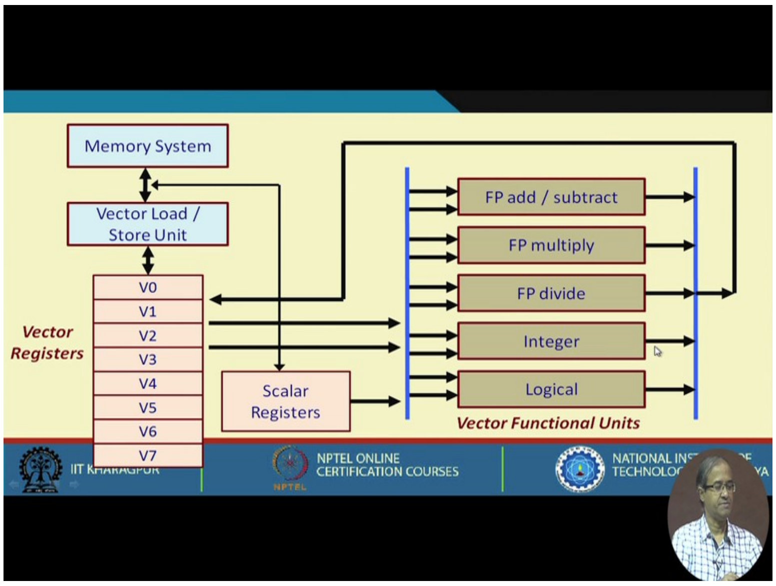
- A vector processor typically consists of an ordinary pipelined scalar unit plus a vector unit.
- All functional units within the vector unit are deeply pipelined, resulting in a shorter clock cycle time C.
  - Deep pipelining on vectors do not result in hazards, since every computation is independent of the others.
- We shall illustrate some concepts based on a hypothetical vector processor that is based on the MIPS32 architecture.

If you can map the vectors to the so-called vector registers, we can use a single instruction to add them. Let us now look at the basic vector processor architecture. The basic hardware is nothing but heavily pipelined scalar unit. The vector processor is an extension of a pipelined processing unit, where in the execution unit is also pipelined.

Here the arithmetic units are separately pipelined, they can be separately fed with data, and results can be picked up separately. So, all the functional units will be deeply pipelined to the extent possible so that you can feed data at a faster rate. There will be two things; there will be an ordinary pipeline scalar unit, and in addition there will be a vector unit. Vector unit will consist of the vector registers, the arithmetic units and so on.

Inside the vector unit all the functional units like adder, multiplier, and divider etc., they are all deeply pipelined so that for execution in the functional units the pipeline clock cycle time will be as less as possible. Now another thing you see here, in a normal instruction set pipelining when we are feeding a number of instructions sequentially in the pipe, there were the situations of hazards, but for vector operation suppose I am adding two vectors A and B storing the results in C, there are 64 operations that are going on in parallel in overlapped fashion. But all the operations are independent. There is no dependency across successive computation. So no question of hazards is coming in, this is another very good thing for vector processing.

(Refer Slide Time: 10:29)

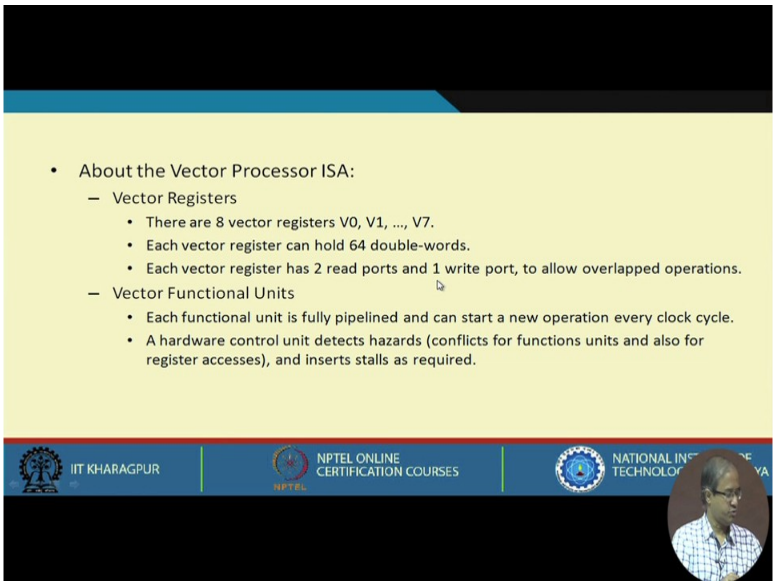


This kind of deep pipelining does not result in hazards, because the computations are independent. Let us look at a hypothetical vector processor that is an extension of the MIPS32 architecture. This is a very high-level schematic; let us say there are 8 vector registers V0 to V7. Let us assume each of this vector registers can hold 64 double precision numbers.

Each of the registers is actually a set of 64 64-bit registers, there are 8 such. And in the vector registers, two of them can be read and you can write into one of them every cycle. And in addition there are the standard scalar registers as you saw in MIPS32, the integer and the floating point registers, there is a memory system. The memory will be interfaced to the scalar register just like as you saw earlier, but in addition there will also be a vector load/store unit, through which you can load an entire vector from memory into one of the registers, or you can store one of the vector registers into memory, and on this side you have the vector functional units.

So, you see there are many vector functional units, each of them are having two inputs and one output, and they can be fed concurrently. The bandwidth of this bus should be high enough so that while some vector addition is going on in parallel, a vector multiply should also be going on. It really does not mean that these arrows will mean that one and two data are coming together; it may be more than two. In general if there are 5 such functional units, then 10 such numbers that can come in every clock, and similarly 5 of the results can be stored in the registers in every clock.

(Refer Slide Time: 12:38)

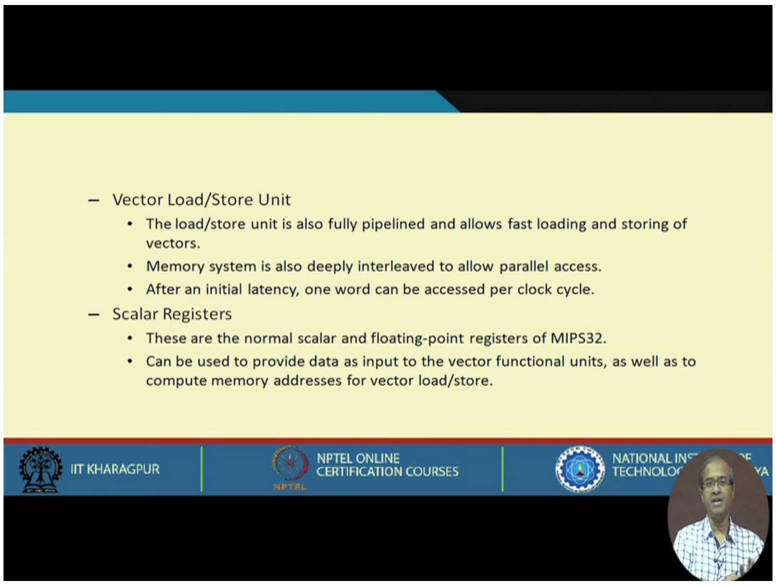


- About the Vector Processor ISA:
  - Vector Registers
    - There are 8 vector registers V0, V1, ..., V7.
    - Each vector register can hold 64 double-words.
    - Each vector register has 2 read ports and 1 write port, to allow overlapped operations.
  - Vector Functional Units
    - Each functional unit is fully pipelined and can start a new operation every clock cycle.
    - A hardware control unit detects hazards (conflicts for functions units and also for register accesses), and inserts stalls as required.

As I said there are 8 vector registers, each of which can hold 64 numbers, each are double precision or double words. Each vector register has two read ports and one write port as this diagram shows. It is not just for the whole bank, but for each of the vectors and for the vector functional units they are all fully pipelined; they can start an operation every clock cycle. And if there is any dependency across the functional units, like I am carrying out a vector addition, the next instruction is a vector multiplication that uses one of the vectors that is produce as the result. There will be latency or a stall.

All the 64 operations can go on in an overlapped way without any stalls. If there any dependencies across instructions then stall cycles can be inserted.

(Refer Slide Time: 14:07)

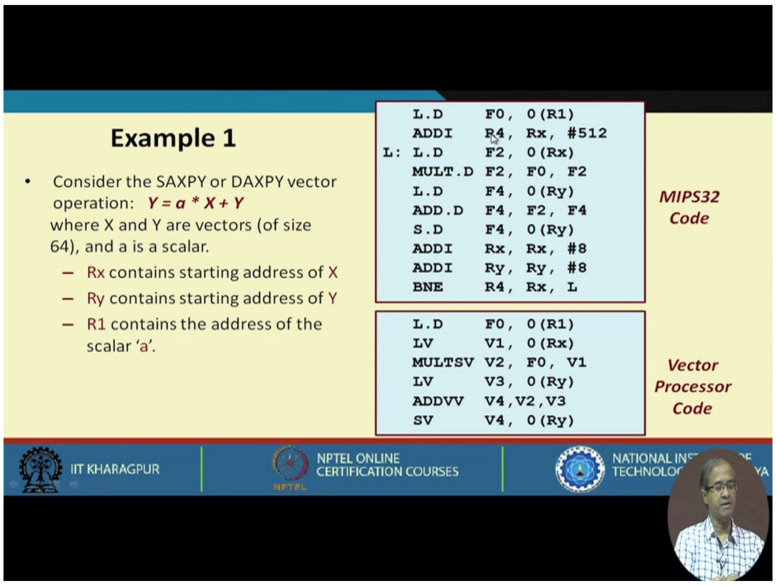


- Vector Load/Store Unit
  - The load/store unit is also fully pipelined and allows fast loading and storing of vectors.
  - Memory system is also deeply interleaved to allow parallel access.
  - After an initial latency, one word can be accessed per clock cycle.
- Scalar Registers
  - These are the normal scalar and floating-point registers of MIPS32.
  - Can be used to provide data as input to the vector functional units, as well as to compute memory addresses for vector load/store.

Regarding the vector load/store unit as I said that is also fully pipelined, and it allows fast loading and storing of the vectors, and to allow this the memory system has to be very deeply interleaved. So, 2-way or 4-way interleaving will not do; may be it will be 16-way or 32-way or 64-way interleaved. So, memory organization has to be much more complex if we have to read a whole 64 words vector in one cycle or in a very few number of cycles.

So, you need to modify the memory system as well. After the initial latency that can indicate the access time of the memory, you can access the words one per cycle. And in addition to that the scalar registers are also there like MIPS32, these are the normal integer and floating point registers. These registers can be used to carry out integer operations of course, but they can also be used to provide data as input to the vector functional units. There are some operations where you may want to add say a scalar number to all the elements of a vector, then the scalar number can come from one of the scalar registers. Moreover you can also use the scalar registers to compute the memory address, which will be used by the vector load store unit, from which memory address you want to load or store.

(Refer Slide Time: 15:48)



Example 1

- Consider the SAXPY or DAXPY vector operation: *Y = a * X + Y* where X and Y are vectors (of size 64), and a is a scalar.
  - Rx contains starting address of X
  - Ry contains starting address of Y
  - R1 contains the address of the scalar 'a'.

```
    L.D     F0, 0(R1)
    ADDI    R4, Rx, #512
L:  L.D     F2, 0(Rx)
    MULT.D  F2, F0, F2
    L.D     F4, 0(Ry)
    ADD.D   F4, F2, F4
    S.D     F4, 0(Ry)
    ADDI    Rx, Rx, #8
    ADDI    Ry, Ry, #8
    BNE     R4, Rx, L
```

MIPS32 Code

```
    L.D     F0, 0(R1)
    LV      V1, 0(Rx)
    MULTSV  V2, F0, V1
    LV      V3, 0(Ry)
    ADDVV   V4,V2,V3
    SV      V4, 0(Ry)
```

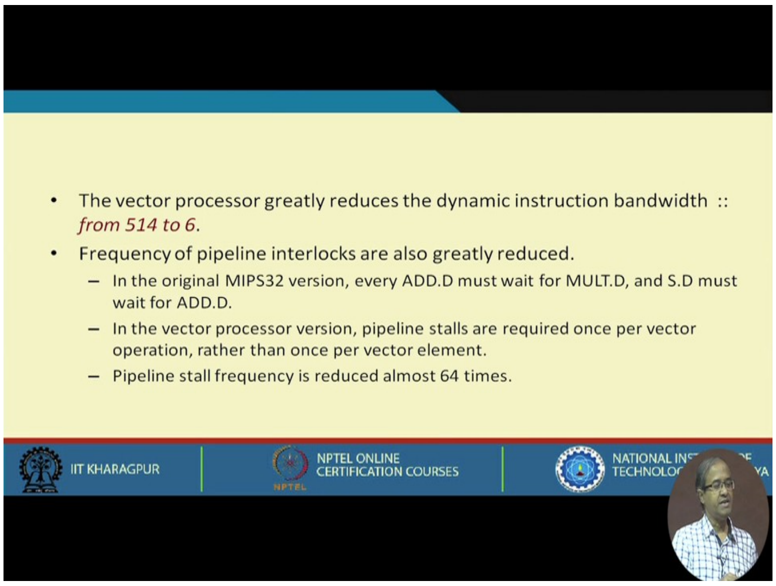
Vector Processor Code

So, scalar and vector registers can be used together. Let us take an example. We look at a computation like this: Y = a * X + Y, where X and Y are vectors and a is a scalar, this is sometimes called single precision SAXPY or double precision DAXPY loop. Here we are assuming that X and Y are both vectors of size 64 and a is a scalar. For convenience let us assume Rx is a register, that contains the starting address of vector X in memory, Ry is a register containing the address of Y, and the scalar register R1 contains the address of the scalar number a. The conventional MIPS32 code will look like this.

The equivalent vector computing code will look like this, here we are loading the scalar number into F0 as usual, this is the load vector instruction and so on.

One big advantage of vector processor is that there will be only 6 instructions, which drastically reduces the dynamic instruction bandwidth.

(Refer Slide Time: 20:01)



- The vector processor greatly reduces the dynamic instruction bandwidth :: *from 514 to 6.*
- Frequency of pipeline interlocks are also greatly reduced.
  - In the original MIPS32 version, every ADD.D must wait for MULT.D, and S.D must wait for ADD.D.
  - In the vector processor version, pipeline stalls are required once per vector operation, rather than once per vector element.
  - Pipeline stall frequency is reduced almost 64 times.

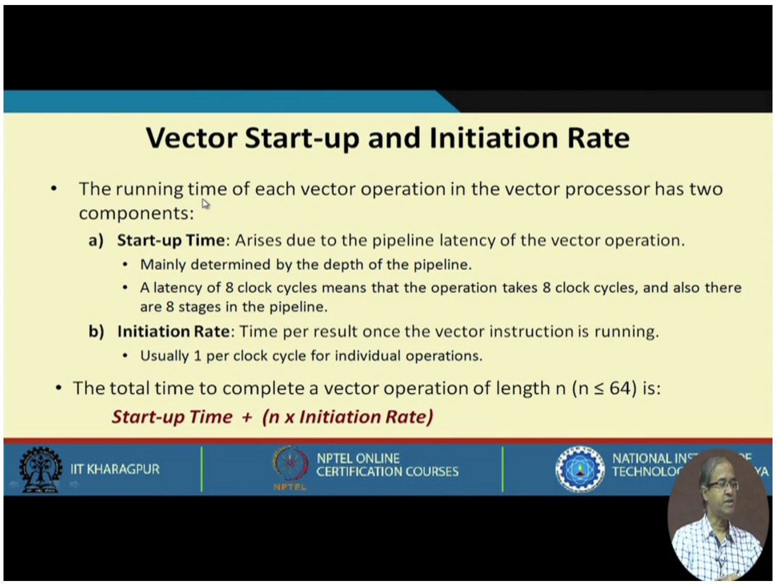
Dynamic instruction bandwidth is equal to the number of instructions that are actually getting executed.

Let us look at the MIPS32 code version here the loop was this 1 2 3 4 5 6 7 8 there were 8 instructions in the loop, and the loop was going 64 times. So, 8 x 64 = 512, and outside the loop there are two more instructions 514. But here there are only 6. So, earlier 514 instructions were fetched and executed; now only 6 instructions will be fetched and executed. This is a great reduction in the instruction bandwidth; similarly the stalls and pipeline interlocks are also greatly reduced because in the original MIPS32 version there will be lot of dependencies.

But here within the vector instructions there are no dependencies, but across there can be. Like here you will loading V1, next instruction V1 is used like that. So, in vector processor, the stalls are required once per vector operation; rather than once per vector element. In the original version stalls were occurring in every iteration; here within a vector instruction there are no stalls, but between vector instructions that can be stalls.

In general, the pipeline stall frequency is also reduced by almost 64 times, these are the main advantages. Now there are something called vector start up and initiation rate.

(Refer Slide Time: 22:59)



Vector Start-up and Initiation Rate

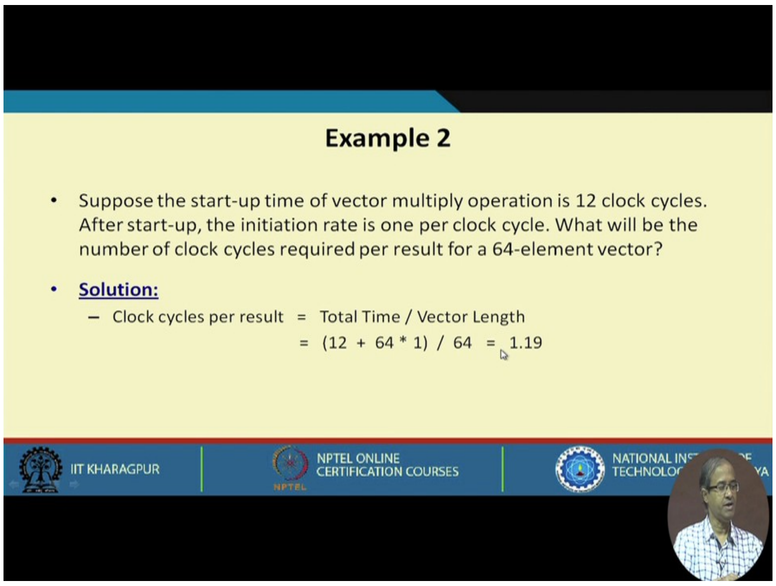
- The running time of each vector operation in the vector processor has two components:
  - a) **Start-up Time**: Arises due to the pipeline latency of the vector operation.
    - Mainly determined by the depth of the pipeline.
    - A latency of 8 clock cycles means that the operation takes 8 clock cycles, and also there are 8 stages in the pipeline.
  - b) **Initiation Rate**: Time per result once the vector instruction is running.
    - Usually 1 per clock cycle for individual operations.
- The total time to complete a vector operation of length n ($n \leq 64$) is:

  Start-up Time + (n x Initiation Rate)

You see the running time for the vector operations has two components, one is called the start up time. Start up time means after how much time the first result will be available, this depends on the depth of the pipeline. This is determined by the depth of the pipeline. If I say that my start up time is 8, this will mean that my pipeline depth is 8, and it will take 8 clock cycles for the first result to come out. And initiation rate is that once the vectors are being operated on, what is the delay with which I can feed the successive elements of the vectors?

It is usually one per clock cycle because the execution units are deeply pipelined. In general if you have a vector operation that is operating on n elements, typically n <= 64, then it will be equal to the start up time plus number of elements multiplied by initiation rate, as I said initiation rate is typically one. So, this will be the total time to compute the operation.

(Refer Slide Time: 24:00)



Example 2

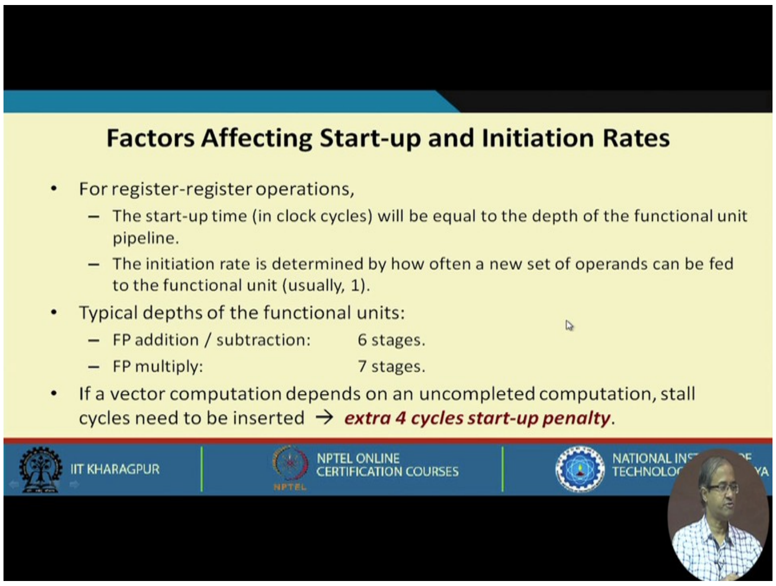
- Suppose the start-up time of vector multiply operation is 12 clock cycles. After start-up, the initiation rate is one per clock cycle. What will be the number of clock cycles required per result for a 64-element vector?
- **Solution:**
  - Clock cycles per result = Total Time / Vector Length
    = (12 + 64 * 1) / 64 = 1.19

IIT KHARAGPUR
NPTEL ONLINE CERTIFICATION COURSES
NATIONAL INSTITUTE OF TECHNOLOGY

Let us take a simple example. Let us say this start up time of a multiply operation is 12 clock cycles, which means it is a 12 stage pipeline. So, after starting up initiation rate is 1 per clock cycle. We are trying to estimate what will be the number of clock cycles required per result for a 64-element vector.

You see for all the 64 elements you can use the previous formula to compute the total time. Now I want time per element. So, we are computing 64 results divide by 64. It is 1.19.

(Refer Slide Time: 25:00)



Factors Affecting Start-up and Initiation Rates

- For register-register operations,
  - The start-up time (in clock cycles) will be equal to the depth of the functional unit pipeline.
  - The initiation rate is determined by how often a new set of operands can be fed to the functional unit (usually, 1).
- Typical depths of the functional units:
  - FP addition / subtraction: 6 stages.
  - FP multiply: 7 stages.
- If a vector computation depends on an uncompleted computation, stall cycles need to be inserted → ***extra 4 cycles start-up penalty***.

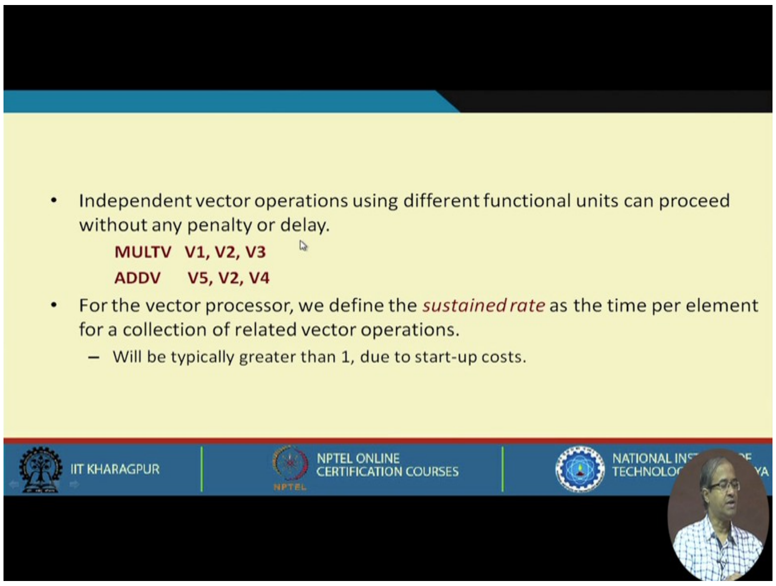
IIT KHARAGPUR
NPTEL ONLINE CERTIFICATION COURSES
NATIONAL INSTITUTE OF TECHNOLOGY

There are certain factors that affect the start time and initiation rates. Like for register to register operations the start up time will be equal to the number of stages in the pipeline, and initiation rate is typically one.

Typical depths are as follows. For floating point operations, addition/subtraction require 6 stages, taking into account the mantissa alignment, normalization everything and floating point multiply requires 7 stages.

But if there is an a dependency between vector operations; that means, if a computation depends on an uncompleted computation due to a previous instruction, then some stall cycles may have to be inserted. Typically extra 4 cycles are required.

(Refer Slide Time: 26:10)



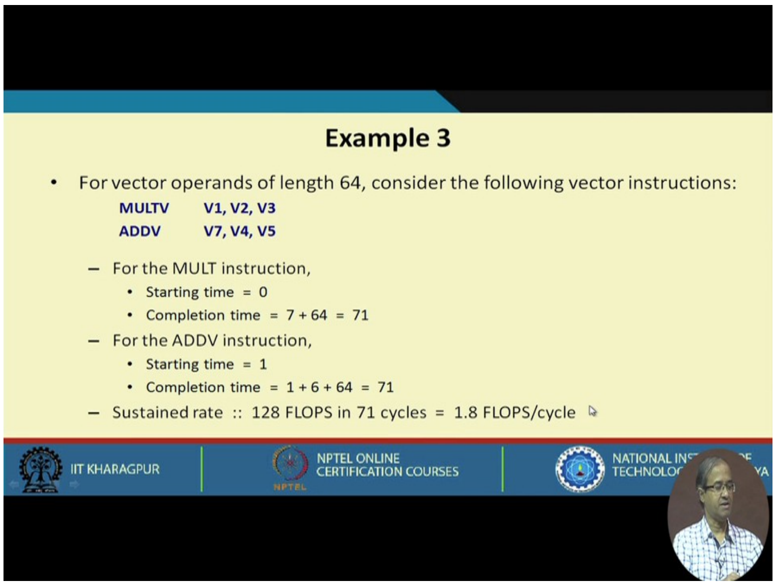
- Independent vector operations using different functional units can proceed without any penalty or delay.
  MULTV V1, V2, V3
  ADDV V5, V2, V4
- For the vector processor, we define the *sustained rate* as the time per element for a collection of related vector operations.
  - Will be typically greater than 1, due to start-up costs.

IIT KHARAGPUR
NPTEL ONLINE CERTIFICATION COURSES
NATIONAL INSTITUTE OF TECHNOLOGY

Let us take an example like this where the operands are all independent. There is no dependency, and you can proceed without any penalty or delay. But if there is a dependency then in the previous slide I mentioned some extra cycle start up penalty is there, that penalty have to be given for the second instruction. Even with forwarding we have to wait for this operation to be complete, then you can forward it here.

The sustained rate is defined across a number of operations, what is the time per element for a collection of operations. Let us take an example.

(Refer Slide Time: 27:00)



Example 3

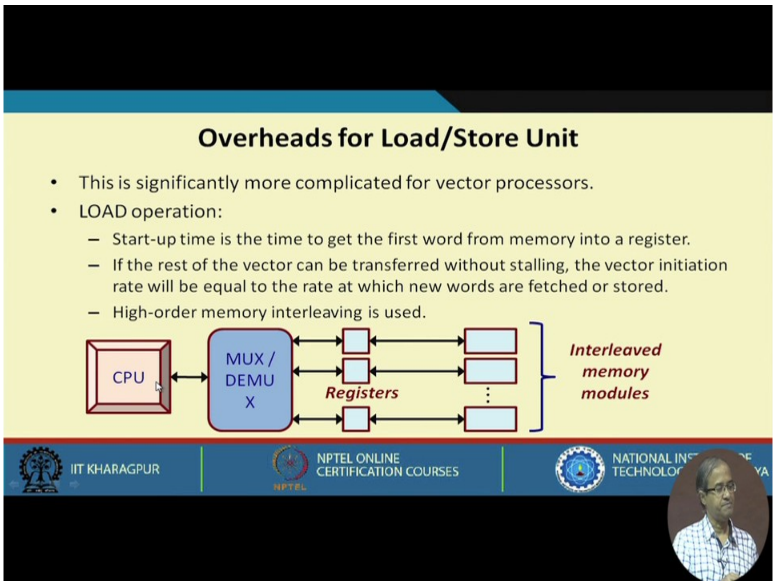
- For vector operands of length 64, consider the following vector instructions:

```
MULTV    V1, V2, V3
ADDV     V7, V4, V5
```

  - For the MULT instruction,
    - Starting time = 0
    - Completion time = 7 + 64 = 71
  - For the ADDV instruction,
    - Starting time = 1
    - Completion time = 1 + 6 + 64 = 71
  - Sustained rate :: 128 FLOPS in 71 cycles = 1.8 FLOPS/cycle

Let us say there are two instructions like this, multiply followed by add when the operands are of length 64. For the multiply instruction the starting time is 0 because in the clock cycle 0 it is starting. So, completion time will be 71. And add will start after one clock cycle, its starting time will be 1. So, completion time will be again 71. In total 71 clock cycles we are completing 64 + 64 = 128 floating point operations.

(Refer Slide Time: 27:54)

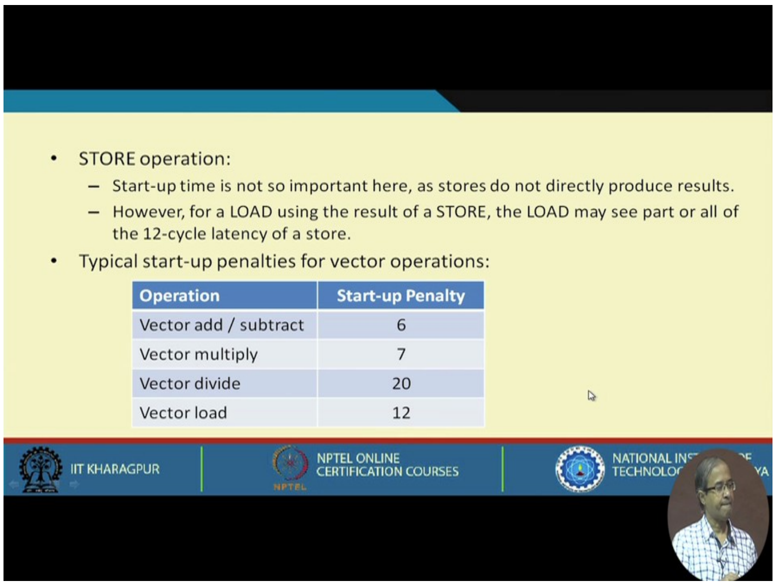


Overheads for Load/Store Unit

- This is significantly more complicated for vector processors.
- LOAD operation:
  - Start-up time is the time to get the first word from memory into a register.
  - If the rest of the vector can be transferred without stalling, the vector initiation rate will be equal to the rate at which new words are fetched or stored.
  - High-order memory interleaving is used.

So, 128 floating point operation in 71 cycles mean 1.8 flops per cycle, this is the sustained rate. Talking about the overheads of load store unit we just mentioned it earlier, we have the memory here. The memory has to be very heavily interleaved, you can access these banks in parallel, you can store them temporarily in registers, and through a MUX/DEMUX network the registers can be read or written at a very fast rate to the CPU from the CPU. For load operation this start up time will be the time to get the first word from memory into register; it will be equal to the access time of the memory. So, how much time is required to get the data here and the MUX to forward the first data, and since the remaining data are already registered you can feed them one by one at successive clock. So, vector initiation rate will be equal to the rate at which new words can be fetched. This memory organization has to be done very carefully such that very high data transfer rate can be sustained.

(Refer Slide Time: 29:04)

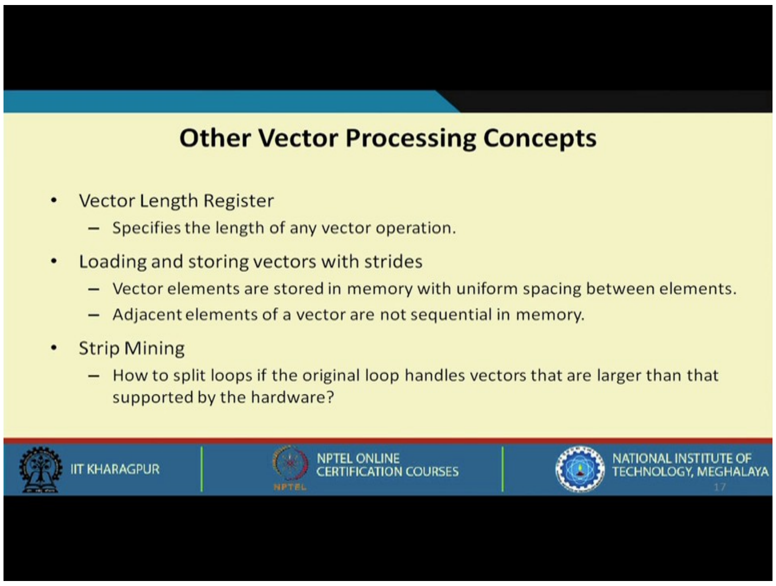


- STORE operation:
  - Start-up time is not so important here, as stores do not directly produce results.
  - However, for a LOAD using the result of a STORE, the LOAD may see part or all of the 12-cycle latency of a store.
- Typical start-up penalties for vector operations:

Operation	Start-up Penalty
Vector add / subtract	6
Vector multiply	7
Vector divide	20
Vector load	12

For store operation start up time is not that important, because store does not produce results in some vector register, it is storing into memory. But if there is a load instruction which follows this store and uses the same data, then the load may have to wait. These are the typical start up penalties.

(Refer Slide Time: 29:39)



Other Vector Processing Concepts

- Vector Length Register
  - Specifies the length of any vector operation.
- Loading and storing vectors with strides
  - Vector elements are stored in memory with uniform spacing between elements.
  - Adjacent elements of a vector are not sequential in memory.
- Strip Mining
  - How to split loops if the original loop handles vectors that are larger than that supported by the hardware?

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

Sometimes you may have a situation where you do not have to operate on the whole 64 element vector, may be we have vector of size 30, then there is a register called Vector Length Register. You can load it with 30, only the first 30 elements would be operated on. Second one is loading and storing vectors with strides. This is something like this, normally when you load a vector from memory you load them from consecutive memory locations. That is one way, other way is that you can specify something called a stride or a gap, let us say the gap is 10. The first vector element is loaded from say memory location 0, the second location second is loaded from 10, third is loaded from 20, 30, 40 and so on; that means, at gap of 10. This is sometimes used, for example, if you have a two dimensional array, they are typically stored in memory in row major or column major order. Suppose you want to load a row or a column in a vector register.

If we used stride you can do both. For column if it is stored in row major order, if we have a gap you can access the elements in column by specifying the stride, you can also load the column in a vector register.

The third is called strip mining, which says that well your vector register is of length 64, but suppose my original program is running for let us say 200 cycles. So, what do we do? So, 64, 64, 64 we can do it for three times. So, it will be 192, and 8 vector elements will be left. This is called strip mining. I will be factoring it out 64, 64, 64 and whatever is left out I will be using as a separate 4th vector operation.

These are some concepts that are there along with vector processing, which helps a programmer to write vector programs. With this we come to the end of this lecture. I tried to give you a brief overview about what vector processing is and how vector computing works. If you look at a real vector processor that is available commercially, you will see there are lot of other details involved there, but here we are not going into those details.

Thank you.