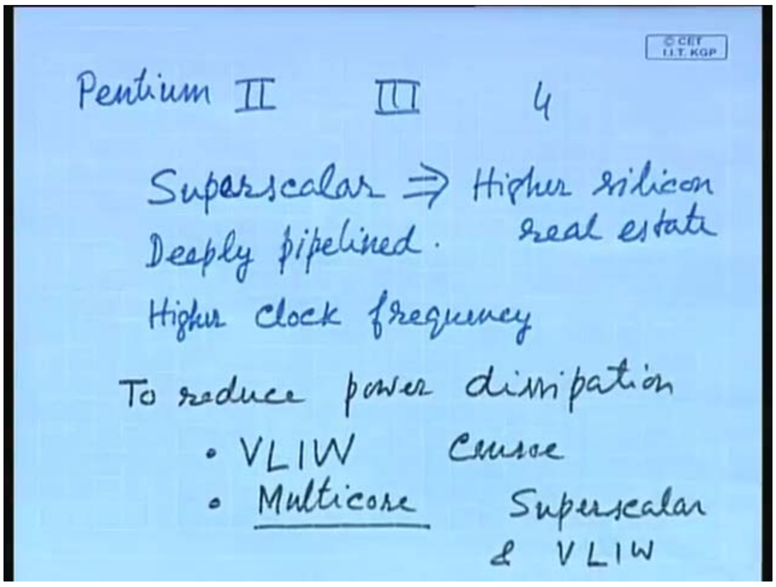
High Performance Computer Architecture

Prof. Ajit Pal
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

Lecture - 36
Case Studies (Contd.)

Hello and welcome to today's lecture on Case Studies, which I will continue our discussion on evolution of Intel processors.

(Refer Slide Time: 00:50)



Pentium II III 4

Superscalar ⇒ Higher silicon real estate
Deeply pipelined.
Higher clock frequency

To reduce power dissipation
- VLIW Crusoe
- Multicore Superscalar & VLIW

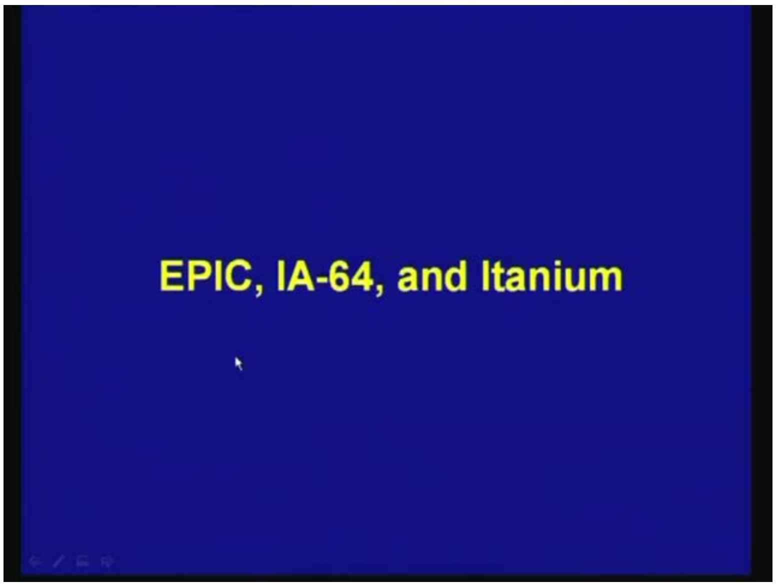
In the last couple of lectures, we have discussed about primarily Pentium series of processors, starting with Pentium 2, then Pentium 3 and Pentium 4. And we have seen that these processors are essentially super scalar, and deeply pipelined and as you know in super scalar architecture, the instruction scheduling is done that instruction issue is done with help of hardware. And because, of that the hardware complexity keeps on increasing, in super scalar architecture as the complexity of instructions scheduling increases.

Similarly, as you use deeply pipelined processor then the clock frequency becomes higher, so it leads to higher clock frequency. Similarly the super scalar frequency leads to higher silicon real estate, so both these things I mean this higher silicon real estate, as well as higher clock frequency leads to more power consumption. So, power consumptions become high, and question naturally arises how we can reduce the power dissipation.

So, to reduce power dissipation one approach is to use VLIW we have already seen, VLIW is essentially very large instruction ward, where the instruction scheduling is done with the help of compiler, instead of doing it by hardware, it is done by software. And as a consequence, the chip area reduces, power dissipation of the chip reduces that is one approach that we have already discussed. And we know that there are some processors based on this VLIW up to VLIW approach like, transmitters Crusoe processors, and another approach to reduce power dissipation is to use multi core.

So, later on we shall discuss about multi core instead of using a single processor with in a chip, and continuously reducing frequency to get higher and higher performance, you can use parallelism in terms of processor. So, we have to move from instruction level parallelism, to thread level parallelism, to get higher performance. So, we shall discuss about this in my next lecture, but today we shall focus on VLIW style approach and that has been implemented in that itanium processor.

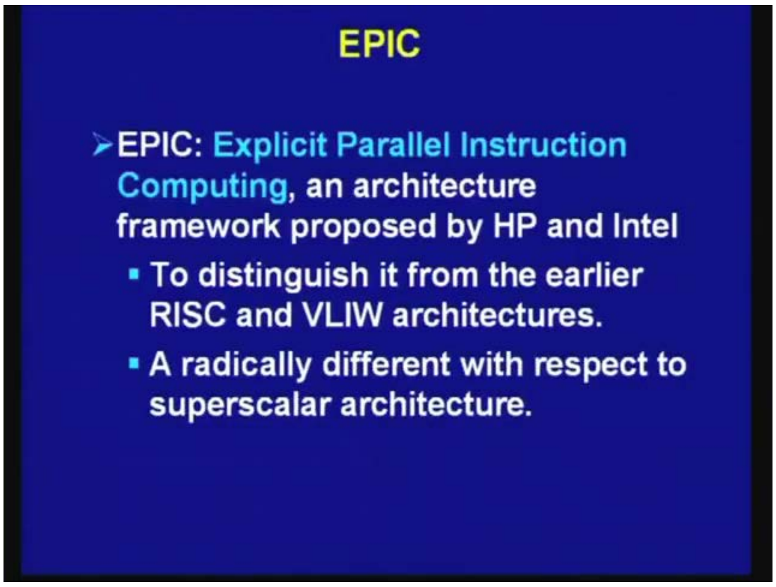
(Refer Slide Time: 04:33)



EPIC, IA-64, and Itanium

So, this itanium processor is essentially based on EPIC approach.

(Refer Slide Time: 04:37)



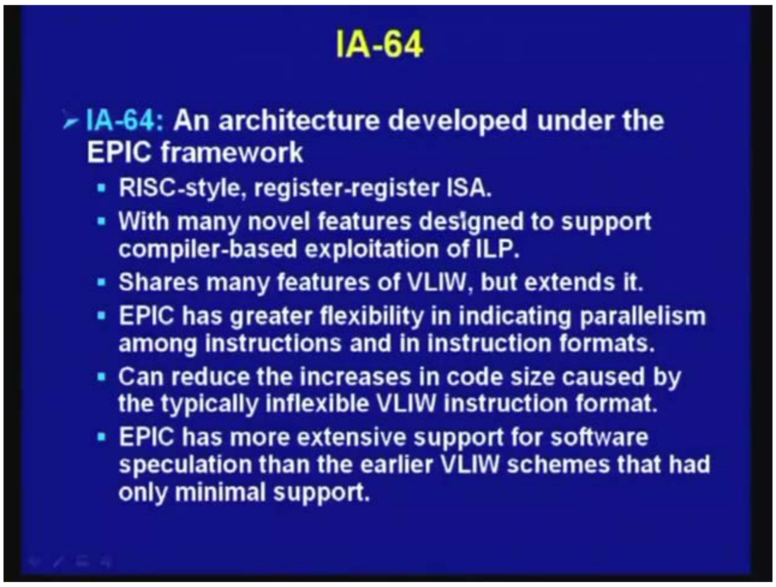
EPIC

- EPIC: Explicit Parallel Instruction Computing, an architecture framework proposed by HP and Intel
  - To distinguish it from the earlier RISC and VLIW architectures.
  - A radically different with respect to superscalar architecture.

This EPIC approach was is known as explicit parallel instruction computing, this explicit parallel instruction computing was developed Hewlett, Packard, HP and Intel. And this name was given to distinguish it from super scalar and VLIW; that means, we shall see although it will inherit many features of VLIW. But, there are many novel concepts which have been incorporated in this new this epic architecture.

So, to essentially to distinguish use it from super scalar or VLIW, a new approach was suggested and that new approach is known as epic, and which is radically different from superscalar architecture.

(Refer Slide Time: 05:28)

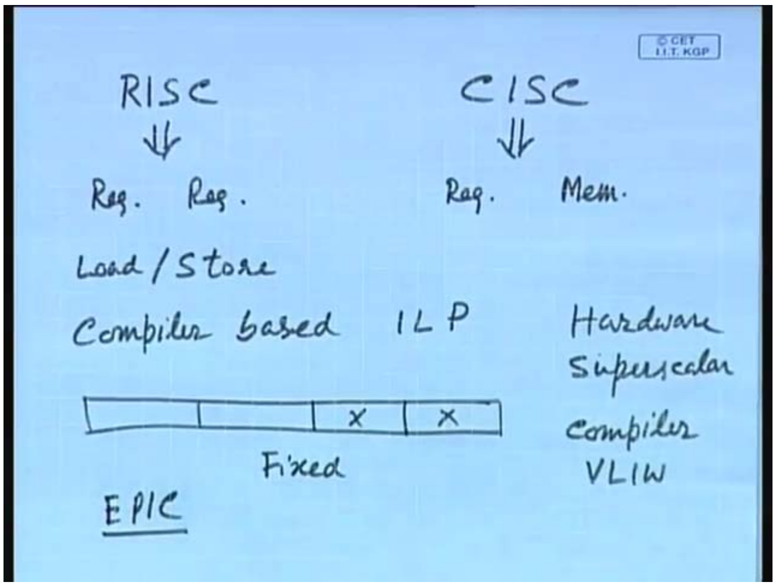


IA-64

- IA-64: An architecture developed under the EPIC framework
  - RISC-style, register-register ISA.
  - With many novel features designed to support compiler-based exploitation of ILP.
  - Shares many features of VLIW, but extends it.
  - EPIC has greater flexibility in indicating parallelism among instructions and in instruction formats.
  - Can reduce the increases in code size caused by the typically inflexible VLIW instruction format.
  - EPIC has more extensive support for software speculation than the earlier VLIW schemes that had only minimal support.

Then IA 64 essentially is architecture develop based on this epic approach, and it has many inherent features. And these inherent features are listed here, number one is it is RISC style register-register ISA.

(Refer Slide Time: 06:02)



We know that instruction set architecture can be of two types, one is your RISC another is your CISC, in CISC architecture you know that ALU can perform operation with the content of register and also with the content of memory. So, the instructions are available where you can perform say and add operation, with the content of a register with that of a content of memory location. And which is not done which is not allowed RISC processors are based on reduce instruction set computers, as based on that register-register operations.

That means, all the ALU operations are performed between the content of registers, and result is stored is also stored in the register, and to load the registers explicitly you have got load and store instruction that is why it is also called load store architecture RISC processors. So, this is one of the features there are many other features there in RISC processors because, they should have large number of registers and, so on, let us not get into that details.

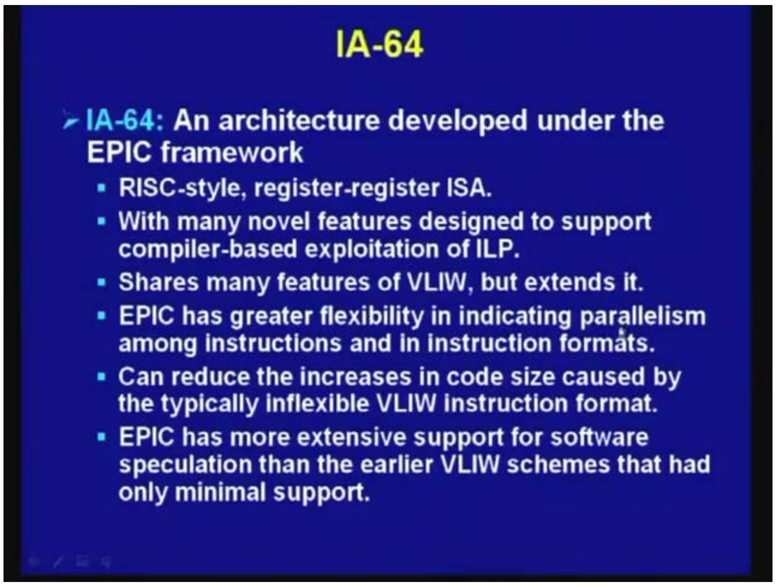
But, main feature is EPIC framework that EPIC architecture is based on RISC style, with register-register instruction set architecture. And with many novel feature design to support compiler based ILP, so another important feature is compiler based ILP, instruction and parallelism. We know that, both in super scalar processor and VLIW, VLIW processor we exploit instruction level parallelism, and in case of super scalar processor that instruction level parallelism is identified with the help of hardware.

So, over a set of instructions that has been available that has been fetched in to the processor, on that within that set of pre fetch instructions, parallelism is detected with the help of hardware, in case of super scalar. On the other hand that ILP is detected with the help of compiler or software, in case of VLIW architecture, so a compiler whenever this instruction level parallelism is detected, with the help of compiler it has much more scope than it can be done by hardware as it is done in super scalar processor.

And, in fact, in this approach that epic approach this has been extended more, it shares many features of VLIW; that means, it is done with the help of compiler. But, extends it how it extends it, we have already discussed about VLIW architecture, we have seen that they have a rigid instruction format. In a single instruction format, few instructions accommodated and those are fixed the number of instructions that can be accommodated and, so on.

And parallelisms are not available, instruction level parallelism is not available some of the fields in instruction format may remain on empty, but in case of your EPIC architecture in case of EPIC approach it is much more flexible.

(Refer Slide Time: 10:21)



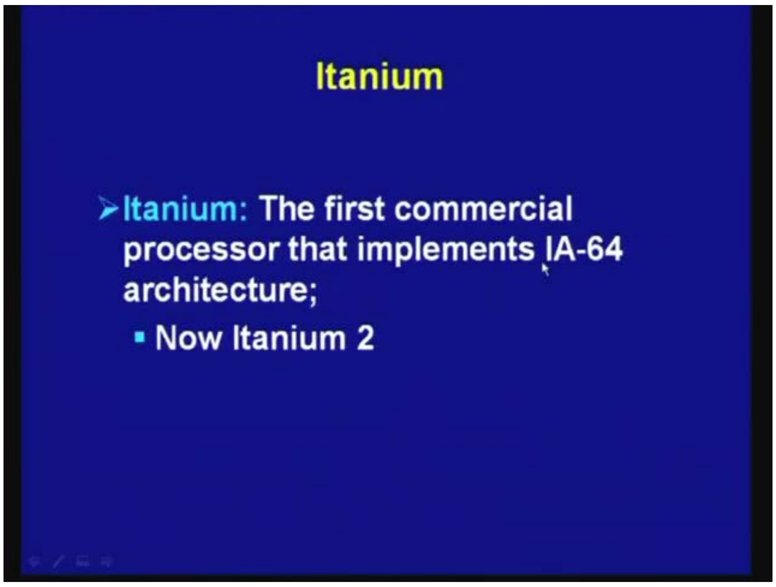
IA-64

- IA-64: An architecture developed under the EPIC framework
  - RISC-style, register-register ISA.
  - With many novel features designed to support compiler-based exploitation of ILP.
  - Shares many features of VLIW, but extends it.
  - EPIC has greater flexibility in indicating parallelism among instructions and in instruction formats.
  - Can reduce the increases in code size caused by the typically inflexible VLIW instruction format.
  - EPIC has more extensive support for software speculation than the earlier VLIW schemes that had only minimal support.

And it incorporates I mean data inflexibility in indicating parallelism among instruction, and instruction format both in identifying instruction level parallelism. And also in scheduling instructions, I mean in formatting the instructions that more flexibility is provided. And as a consequence it can reduce the increase in code size, caused by the typically inflexible VLIW instruction format; that means, the ultimate outcome is that whenever if it is done using simple VLIW.

Whatever the size of the code is possible, using this approach EPIC approach, you can achieve much better; that means, instruction code size will be smaller. And EPIC has more extensive support of the software speculation, so this software speculation it is done in a very limited way in a VLIW architecture. And as we shall see a little later the EPIC uses, much more extensive support of software speculation, like predication and, so on, we shall discuss it in more detail little later.

(Refer Slide Time: 11:41)

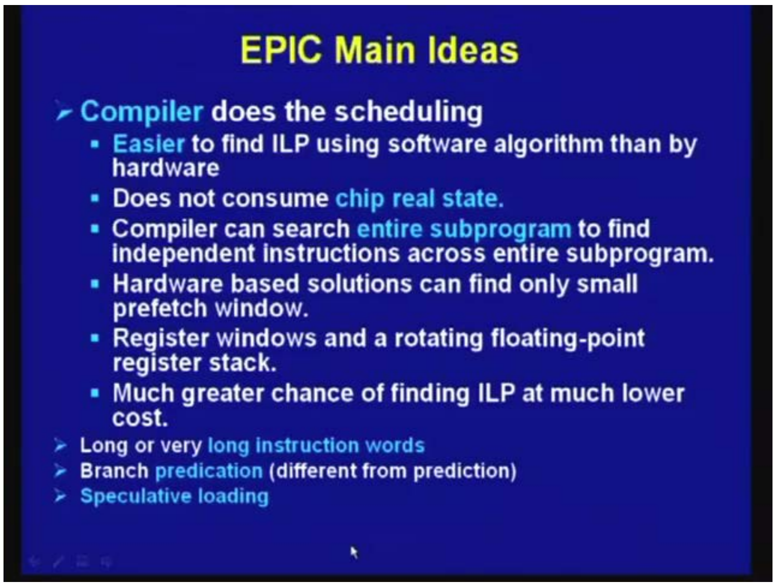


Itanium

- Itanium: The first commercial processor that implements IA-64 architecture;
  - Now Itanium 2

And based on this approach that IA 64 architecture the processor has been developed that is known as itanium. So, itanium is the first commercial processor that implements the IA 64 that is based on IA 64 architecture, and later on we shall see the itanium architecture has been extended also leading to itanium 2 architecture. Essentially these are up gradation of the basic itanium architecture.

(Refer Slide Time: 12:07)



EPIC Main Ideas

- Compiler does the scheduling
  - Easier to find ILP using software algorithm than by hardware
  - Does not consume chip real state.
  - Compiler can search entire subprogram to find independent instructions across entire subprogram.
  - Hardware based solutions can find only small prefetch window.
  - Register windows and a rotating floating-point register stack.
  - Much greater chance of finding ILP at much lower cost.
- Long or very long instruction words
- Branch predication (different from prediction)
- Speculative loading

So, this background, now let us focus on the main ideas of EPIC, as I have already mentioned in case of epic, the instruction scheduling is done with the help of compiler.

And whenever, we do it using the compiler it has been found that it is easier to find instruction level parallelism using software algorithm rather than hardware, so whenever it is done with the help of hardware, it is very restrictive you cannot use a sophisticated algorithm. And the instruction level parallelism is identified over a very small limited number of instructions.

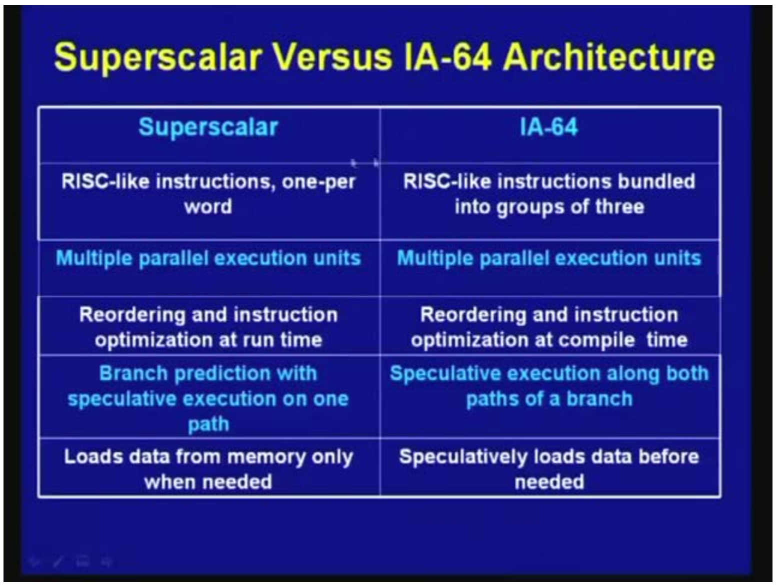
But, whenever you do it with the help of compiler you can develop sophisticated algorithm because, it is done in software the override is not much there. Whenever it is done in software, and then it can identify more I mean that instruction level parallelism can be more, whenever it is done in with the help of a compiler. So, whenever you do it with the help of a compiler it does not consume chip real estate, as I have already mentioned in super scalar architecture it is done by hardware; obviously, it will consume silicon real estate.

So, instead of consuming silicon real estate for scheduling instruction, it can be devoted for other purposes like you can have more number of general purpose registers, more number of floating point registers, and other special type of registers. So, that can be done, and so the compiler can search entire sub program to find independent instructions across the entire sub program. And that is another advantage of this compiler based approach and as I have already mentioned hardware based solutions can find only small pre fetch window.

And other novel techniques, like register windows and rotating floating point register tags have been used in this epic architecture, I shall discuss about that later. And whenever you use this compiler based approach with this enhancement or innovations, there is much greater chance of finding ILP at much lower cost. So, the cost is cost of implementation is lower, and you will get much better ILP, and it will lead to implementing that long or very long instruction words that is the basic approach of VLIW.

And as we shall see that EPIC will use branch predication, so earlier we had discussed about the technique of branch prediction, where I mean whether a branch will be taken or not taken is predicted based on past history. So, here we shall see we shall using a novel concept known as predication, which is different from prediction I shall explain about it in a little later. And also to reduce the latency of memory phase, some technique known as speculative loading is used I shall discuss about this speculative loading in detail. So, this is the main idea that has been used in epic architecture.

(Refer Slide Time: 15:46)



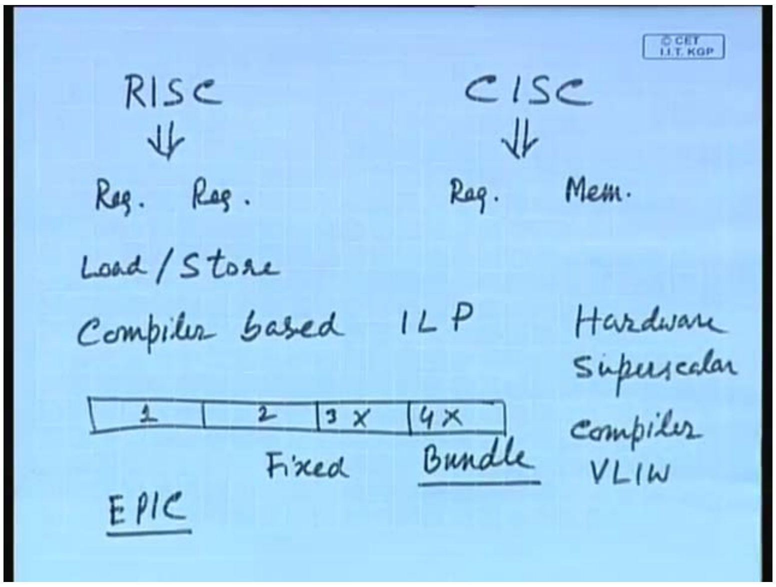
Superscalar Versus IA-64 Architecture

Superscalar	IA-64
RISC-like instructions, one-per word	RISC-like instructions bundled into groups of three
Multiple parallel execution units	Multiple parallel execution units
Reordering and instruction optimization at run time	Reordering and instruction optimization at compile time
Branch prediction with speculative execution on one path	Speculative execution along both paths of a branch
Loads data from memory only when needed	Speculatively loads data before needed

And here is the comparison between super scalar and IA W IA 64 approach, as you have already seen super scalar approach is best on RISC like instructions and one per word. So, one instruction per word is available in super scalar architecture, on the other hand in then you have to schedule multiple instructions to keep, multiple execution units or processing units busy.

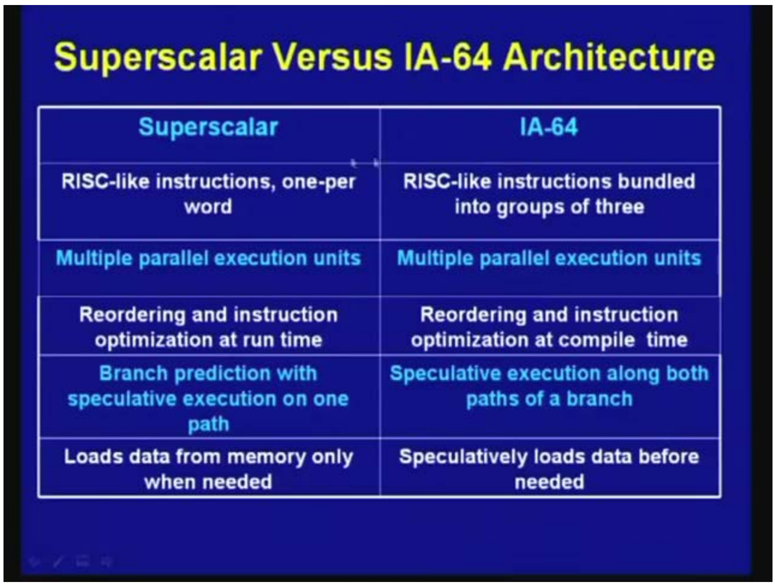
So, that is done by hardware you will fetch instruction one instruction at a time, but issue more than one operations instructions to keep the more functional units busy. On the other hand in RISC case of IA 64, RISC like instructions are bundled into groups of three, we have seen to VLIW architecture. In a single instruction you can bundle more than one instructions as I have already told this is the basic approach.

(Refer Slide Time: 16:52)



This is one instruction, this is another instruction, this is another instruction, this is another instruction though this is a bundle. So, you can form a bundle of several instruction, in case of IA 64 three instructions are bundled together to form a single instruction.

(Refer Slide Time: 17:13)



Superscalar Versus IA-64 Architecture

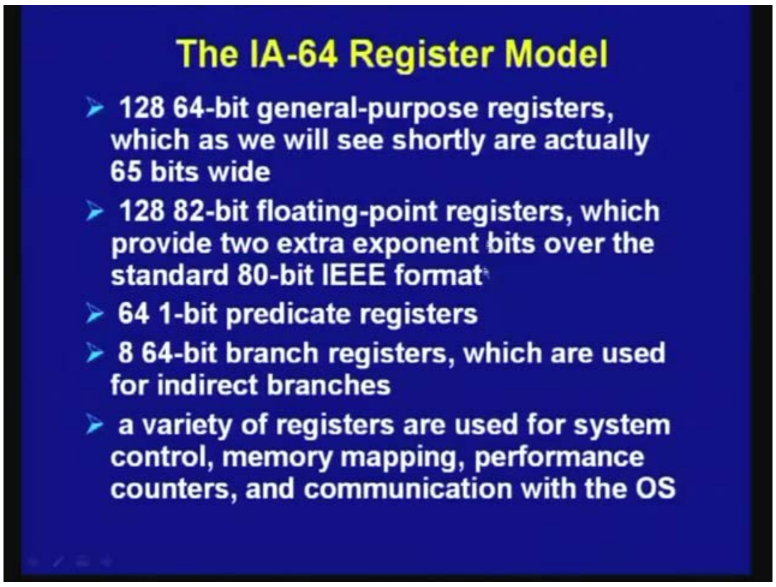
Superscalar	IA-64
RISC-like instructions, one-per word	RISC-like instructions bundled into groups of three
Multiple parallel execution units	Multiple parallel execution units
Reordering and instruction optimization at run time	Reordering and instruction optimization at compile time
Branch prediction with speculative execution on one path	Speculative execution along both paths of a branch
Loads data from memory only when needed	Speculatively loads data before needed

And in super scalar approach we use multiple parallel execution units that is also done in IA 64 multiple parallel execution units, and reordering and instruction optimization is done at run time with the help of hardware. On the other hand with the help of IA 64 reordering and instruction optimization is done at complied time with the help of a software, which is a compiler. And in case of super scalar architecture, branch prediction with speculative prediction is done on one path.

So, we have already discussed about it in detail earlier, and in case of IA 64 we shall see that speculative execution is done along both paths of a branch, which is known as predication I shall discuss about it in more details. Then loads data from memory only when they are needed; that means, in case of super scalar architecture that instructions are loaded that data is loaded from memory, only when they are needed, need based loading.

On the other hand in a 64 speculative loading is done; that means, speculatively loads data before they are needed. Of course, there is a possibility that the data which has been loaded, may not be used that possibility is there, but when they are used it is beneficial. So, this is the key difference between superscalar and IA 64 architecture.

(Refer Slide Time: 18:48)



The IA-64 Register Model

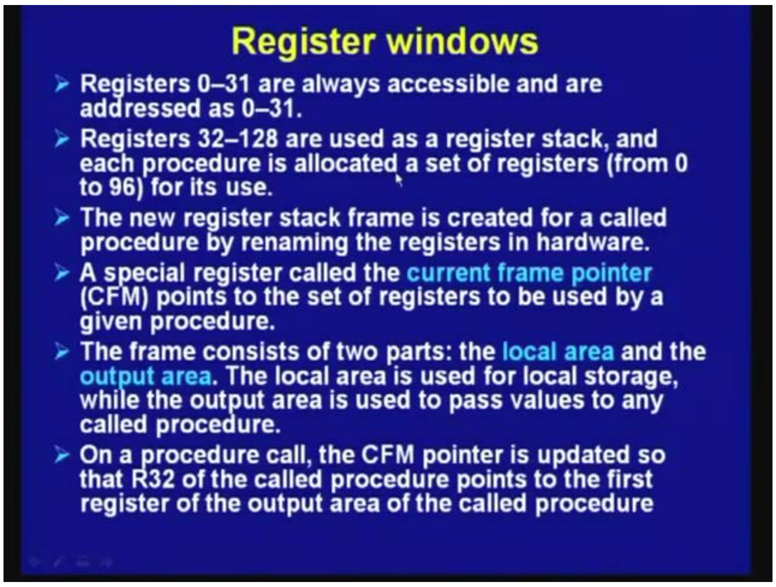
- 128 64-bit general-purpose registers, which as we will see shortly are actually 65 bits wide
- 128 82-bit floating-point registers, which provide two extra exponent bits over the standard 80-bit IEEE format
- 64 1-bit predicate registers
- 8 64-bit branch registers, which are used for indirect branches
- a variety of registers are used for system control, memory mapping, performance counters, and communication with the OS

Now, we shall focus on the details of IA 64 architecture, first we shall look at the register model, and as I have already mentioned if IA 64 uses a large number of registers, compared to super scalar or other VLIW architecture processors based architecture processors. So, it uses 128 64 bit general purpose registers, which as we shall see shortly are actually 64 bits there is one additional bit that is the additional bit is used that predicate bit, we shall discuss about it later.

And then 128 82 bit floating point registers, which provide two exponent bits over the standard 80 bit IEEE format, in the standard 80 bit IEEE format that instruction format uses 80 bit. And in case of IA 64 it has been extended with 82 bit with two additional bit for in the exponent field, and there is 64 1 bit predicate register, so predicate register concept is new in IA 64 I shall explain the operation of this predicate register details, and there are 8 64 branch registers.

So, you can see a variety of registers are used these branch registers are used for indirect branches, and apart from these registers a variety of registers are also used for system control, memory mapping, performance mapping and communication with the operating system. And all of them might not be visible to the programmers.

(Refer Slide Time: 20:38)

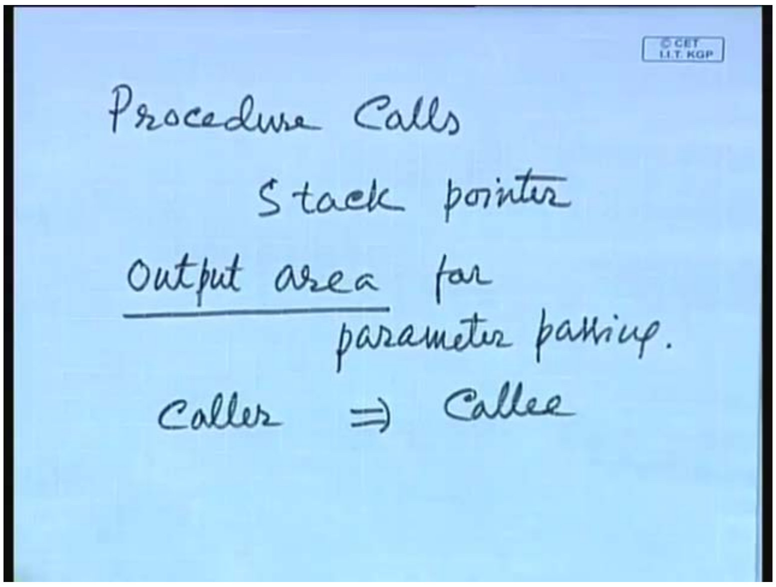


Register windows

- Registers 0–31 are always accessible and are addressed as 0–31.
- Registers 32–128 are used as a register stack, and each procedure is allocated a set of registers (from 0 to 96) for its use.
- The new register stack frame is created for a called procedure by renaming the registers in hardware.
- A special register called the current frame pointer (CFM) points to the set of registers to be used by a given procedure.
- The frame consists of two parts: the local area and the output area. The local area is used for local storage, while the output area is used to pass values to any called procedure.
- On a procedure call, the CFM pointer is updated so that R32 of the called procedure points to the first register of the output area of the called procedure

And it uses the concept of register windows, I mentioned that a new concept known as register windows is used in IA 64, what is this register window concept. It has go 128 fixed point registers, out of which 32 registers are I mean these registers 0 to 31 are always accessible, and are addressed as 0 to 31. The reaming registers that is 32 to 128 are used as register stack, and each procedure is allocated a set of registers; that means, we know that whenever we are dealing with procedures.

(Refer Slide Time: 21:22)


© CET I.I.T. KGP
Procedure Calls
Stack pointer
Output area for parameter passing.
Caller ⇒ Callee

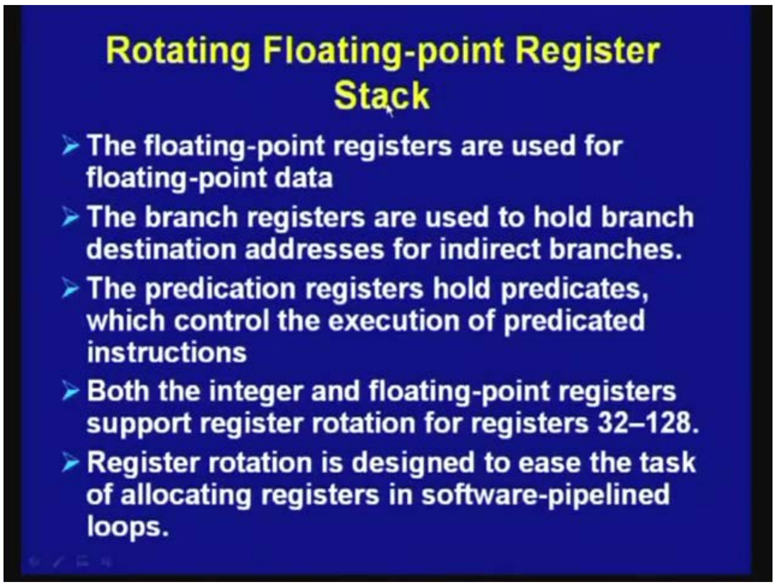
Procedure calls, we use stack and in the stack we store the information whenever we do the contact switching. So, when the contact switching is done a stack is used, normally a stack pointer is available as part of the processor, and then stack is available in the memory. So, whenever a contact switching occurs, we have to store the content I mean the status of the current context, and then we will reload the from the stack to where the jump is taking place.

So, when the contact switching occurs the restoring and storing of the status is time consuming, particularly if it is done through memory. So, instead of that here it is done with the help of registers, so just like a stack point we have we know that a stack pointer is used. So, instead of using a stack pointer a special register called the current frame register CFM points to the set of register to be used by a given procedure.

So, the frame consists of two parts the local area and the output area, so local area is used for local storage, and while the output area is used for to pass values to any called procedure; that means, we are using that output area for parameter passing. So, you have to pass parameter that caller will pass parameter to the callee.

So, whenever the caller passes parameter to callee this output area can be used to pass the parameter to the callee and vice versa. So, the local area is used for local storage and while the output area is used to pass values to any called procedure, so on a procedure called the CFM pointer is updated. So, that R 32 is called the procedure called call procedure points to the first procedure of the output area of the called procedure so; that means, you are using this register R 32 for this purpose for pointing to the area I mean that register window.

(Refer Slide Time: 24:20)



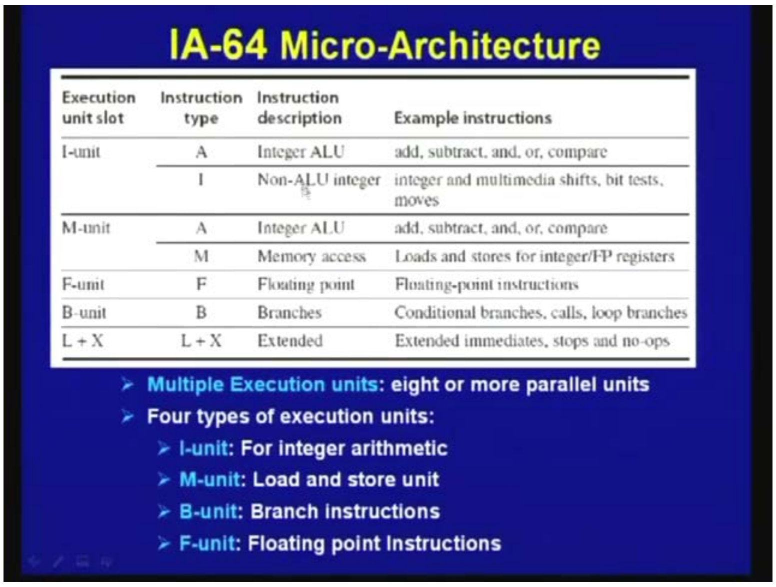
Rotating Floating-point Register Stack

- The floating-point registers are used for floating-point data
- The branch registers are used to hold branch destination addresses for indirect branches.
- The predication registers hold predicates, which control the execution of predicated instructions
- Both the integer and floating-point registers support register rotation for registers 32–128.
- Register rotation is designed to ease the task of allocating registers in software-pipelined loops.

Then it also uses rotating floating point register stack, so the floating point registers are used for floating point data, the branch registers are used to hold branch destination addresses for indirect branches. And the predication registers holds predicates, which controls the execution of predicated instructions, and both integer and floating point registers support register rotation for the registers 32 to 128.

So, those registers the register renaming is done, and by using the concept of register renaming that rotating of registers is done from register 32 to 128 for the purpose of this I mean sub routing calls and other things procedure calls. So, register rotation is designed to ease the task of allocating registers in software pipelined loops. So, this is the novel concept that is also used.

(Refer Slide Time: 25:23)



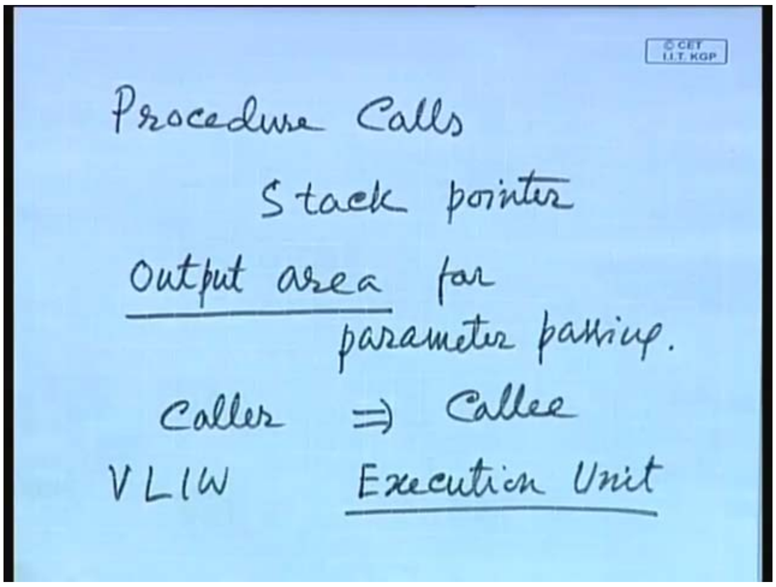
IA-64 Micro-Architecture

Execution unit slot	Instruction type	Instruction description	Example instructions
I-unit	A	Integer ALU	add, subtract, and, or, compare
	I	Non-ALU integer	integer and multimedia shifts, bit tests, moves
M-unit	A	Integer ALU	add, subtract, and, or, compare
	M	Memory access	Loads and stores for integer/FP registers
F-unit	F	Floating point	Floating-point instructions
B-unit	B	Branches	Conditional branches, calls, loop branches
L + X	L + X	Extended	Extended immediates, stops and no-ops

- **Multiple Execution units**: eight or more parallel units
- Four types of execution units:
  - **I-unit**: For integer arithmetic
  - **M-unit**: Load and store unit
  - **B-unit**: Branch instructions
  - **F-unit**: Floating point Instructions

Now, let us focus on the different types of instructions types and execution units that is available in the IA 64 micro architecture.

(Refer Slide Time: 25:39)



Procedure Calls

Stack pointer

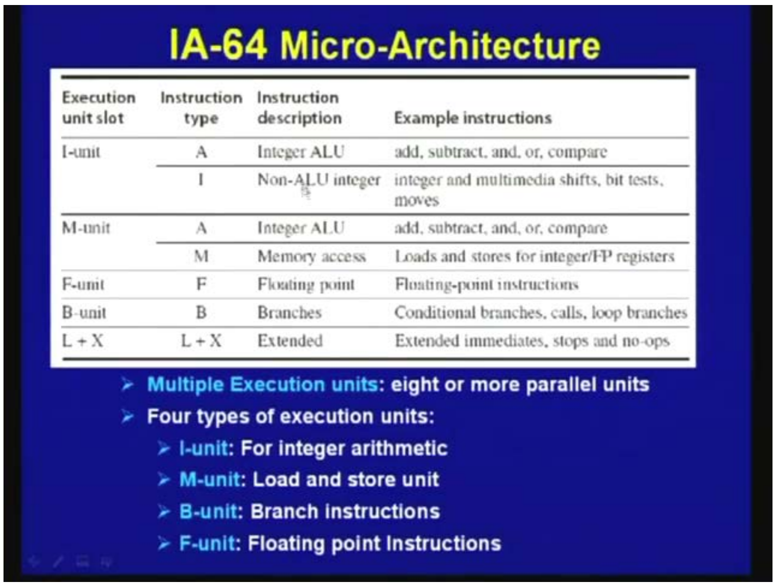
Output area for parameter passing.

Caller ⇒ Callee

VLIW Execution Unit

So, since it is a super scalar processor I mean VLIW processor, there are a number of execution units or functional units and different types of execution units are available.

(Refer Slide Time: 25:55)



IA-64 Micro-Architecture

Execution unit slot	Instruction type	Instruction description	Example instructions
I-unit	A	Integer ALU	add, subtract, and, or, compare
	I	Non-ALU integer	integer and multimedia shifts, bit tests, moves
M-unit	A	Integer ALU	add, subtract, and, or, compare
	M	Memory access	Loads and stores for integer/FP registers
F-unit	F	Floating point	Floating-point instructions
B-unit	B	Branches	Conditional branches, calls, loop branches
L + X	L + X	Extended	Extended immediates, stops and no-ops

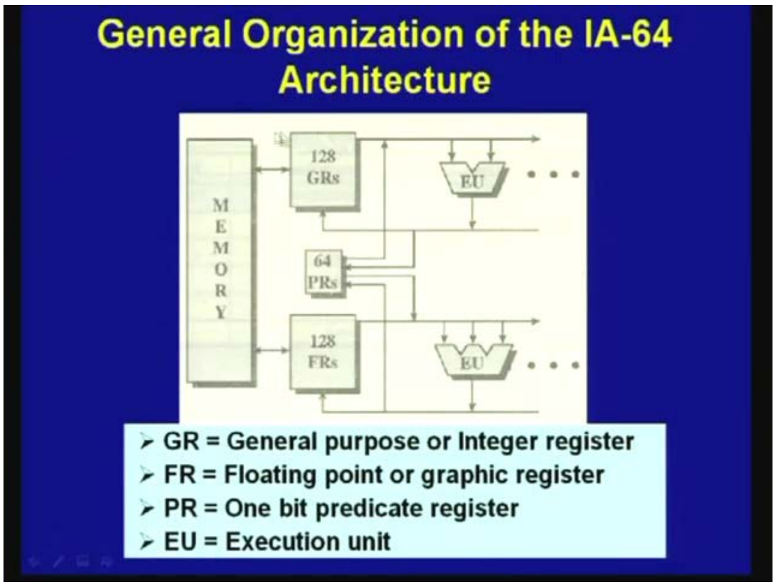
- Multiple Execution units: eight or more parallel units
- Four types of execution units:
  - I-unit: For integer arithmetic
  - M-unit: Load and store unit
  - B-unit: Branch instructions
  - F-unit: Floating point Instructions

First one is I unit I stands for integer unit which you can perform operate on A type and I type instructions, A type instructions are essentially integer ALU type instructions like integer type additions, subtraction and or compare and or, so on. And I type instructions are not ALU integers, like integer and multimedia shifts, bit, tests and moves and, so on. Then M units perform that M unit stands for loads and stores used for the purpose of loads and stores, and it can perform the A type instructions or M type instructions.

So, integer ALU add, subtract and or compare, so which can be done with the help of M unit, and also for memory access like loads and stores for integer point resistor that is M type instruction can be performed with the help of M units. Then you have got F unit floating point unit, floating point instructions can be executed with the help of this F unit. And then B unit is you exclusively used for the purpose of branches, like conditional branches, calls, loop branches and, so on.

Then you have got L plus X type of instructions of course, there is no separate execution units available for them. But, I believe that I type and B type units can be used to for the purpose of L plus X type of instructions, which can be used for extended immediate, 64 bit extended immediate and stops and no ops instruction.

(Refer Slide Time: 27:56)



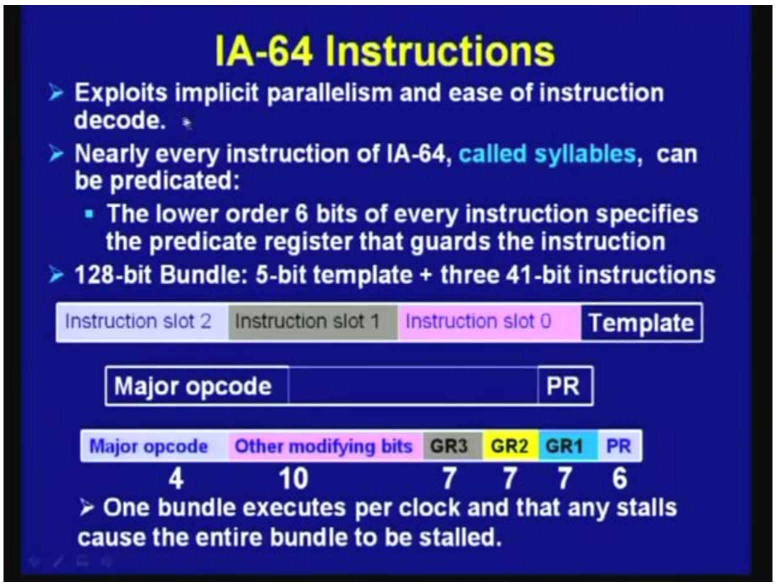
General Organization of the IA-64 Architecture

- GR = General purpose or Integer register
- FR = Floating point or graphic register
- PR = One bit predicate register
- EU = Execution unit

And here is the general organization of the IA 64 architecture, and it has got a large number of execution units, we have seen the number of execution units 8 or more parallel units are available. So, here those execution units are shown both for fixed point and floating point, you have got several execution units, then 128 general purpose registers or integer registers are shown.

And 128 floating point registers are available, and 64 bit predicate registers are available, I shall discuss it is use a little later. So, you have two predicate registers in addition to general purpose registers and floating point registers.

(Refer Slide Time: 28:45)



IA-64 Instructions

- Exploits implicit parallelism and ease of instruction decode.
- Nearly every instruction of IA-64, called syllables, can be predicated:
  - The lower order 6 bits of every instruction specifies the predicate register that guards the instruction
- 128-bit Bundle: 5-bit template + three 41-bit instructions

Instruction slot 2	Instruction slot 1	Instruction slot 0	Template

Major opcode		PR

Major opcode	Other modifying bits	GR3	GR2	GR1	PR
4	10	7	7	7	6

- One bundle executes per clock and that any stalls cause the entire bundle to be stalled.

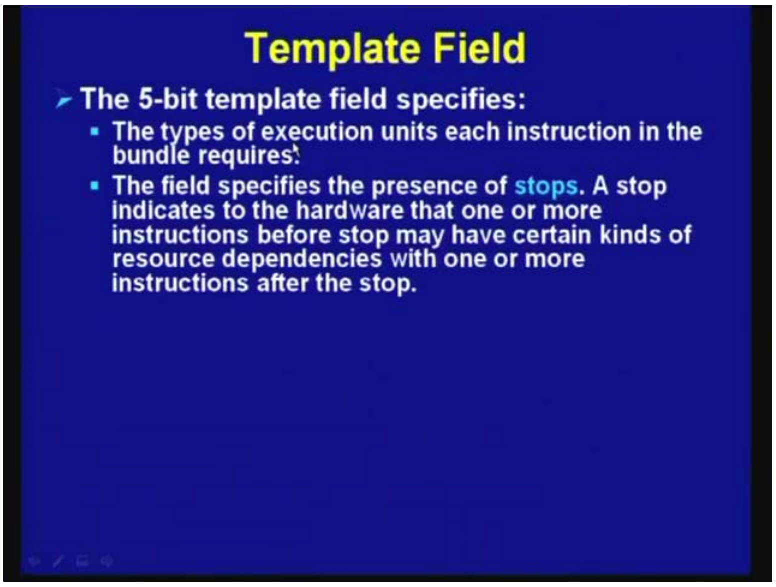
So, let us look at the instruction format of the IA 64 instructions, here it explicit parallelism, implicit parallelism and each of instruction decode. Actually by using the compiler to do it that implicit parallelism available in the large number of instruction is exploited, and instruction decoding each of instruction decoding is done, with the help of those template part. And nearly every instruction of IA 64 is called syllables, so here sometimes instead of instructions they are called syllables, can be predicated.

Predicated means, the lower 6 bits specifies the predicate that is lower order of 6 bits of every instructions specifies the predicate register that guards the instruction. So, here you can see there is a predicate register, 6 bit predicate register available in each and every register. And, so each instruction is of 41 bit, so 41 bit is divided in this way, major op code is 4 bit and it has got 10 modifying bits.

And there are three fields which can specify the general purpose registers, general purpose registers 3, general purpose registers 2, and general purpose registers 1. Since you have got 128 registers, you require 7 bits for each of these fields, and then you have got predicate registers PR. And one bundle you can see these instructions these three instructions together is called a bundle, so one bundle executes per cycle; that means, whenever instruction execution is done, one bundle will be fetched from the memory and it will be executed in a single cycle.

So, if it is stored then all the entire all the instructions in a single bundle will be stored, so any stalls cause the entire bundle to be stalled. So, now, let us focus on the template field I mentioned about that there is a template field present here, so 41 into 3, so out of 128 41 into 3 those bit close for specifying the three instructions, and you have got five bit template field.

(Refer Slide Time: 31:29)

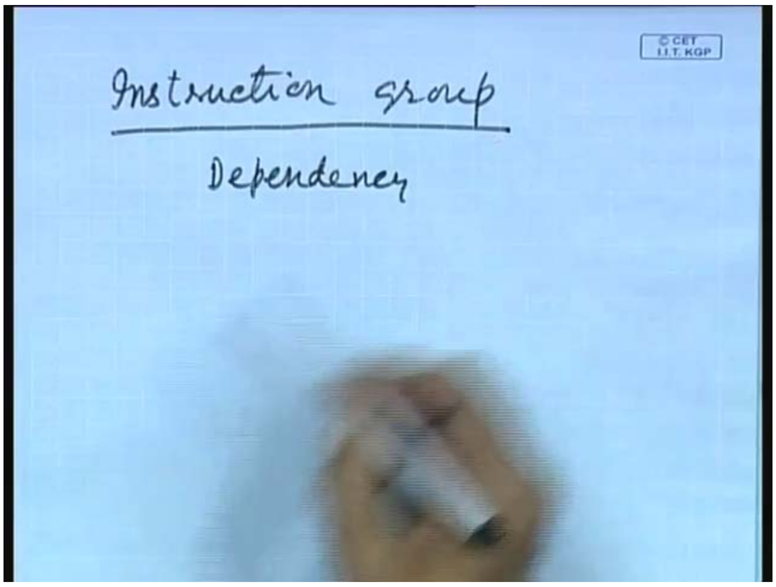


Template Field

- The 5-bit template field specifies:
  - The types of execution units each instruction in the bundle requires.
  - The field specifies the presence of stops. A stop indicates to the hardware that one or more instructions before stop may have certain kinds of resource dependencies with one or more instructions after the stop.

This template field specifies the type of execution units each instruction in the bundle requires. So, we have seen each instruction comprises three instructions in a single bundle, and you will require different types of execution units for executing those instructions in a bundle. And this template field specifies, what type of execution is needed by different instruction of that bundle, and this filed specifies the presence of stops.

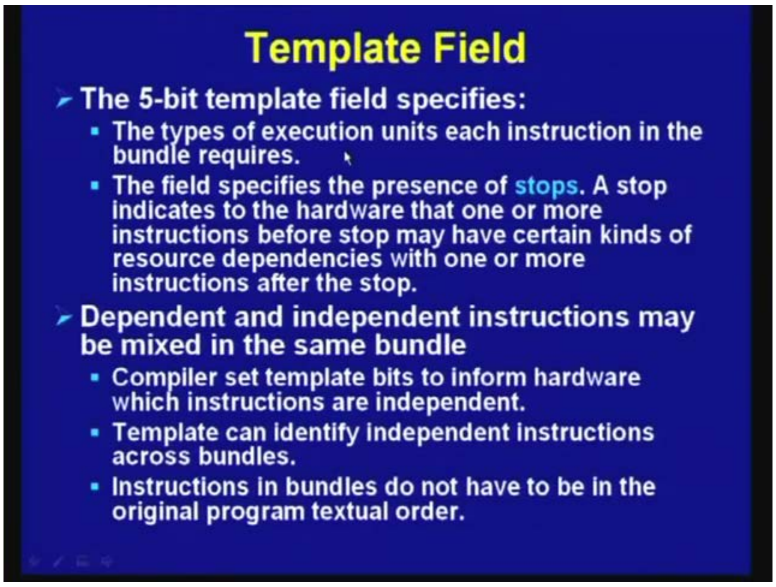
(Refer Slide Time: 32:16)


© CET I.I.T. KGP
Instruction group
Dependency

Stop is another novel concept that is being used you know there is a concept called instruction group, with instruction groups should not have any dependency register-register type of dependency. That means, this instructions within a instruction group, can be executed in a parallel, so since they do not have that dependency because, of register one is writing in to a register, then subsequent instructions reading it like that.

That kind of dependency is not present, and also it acquires the memory based dependencies, then those instructions can form a group.

(Refer Slide Time: 33:12)

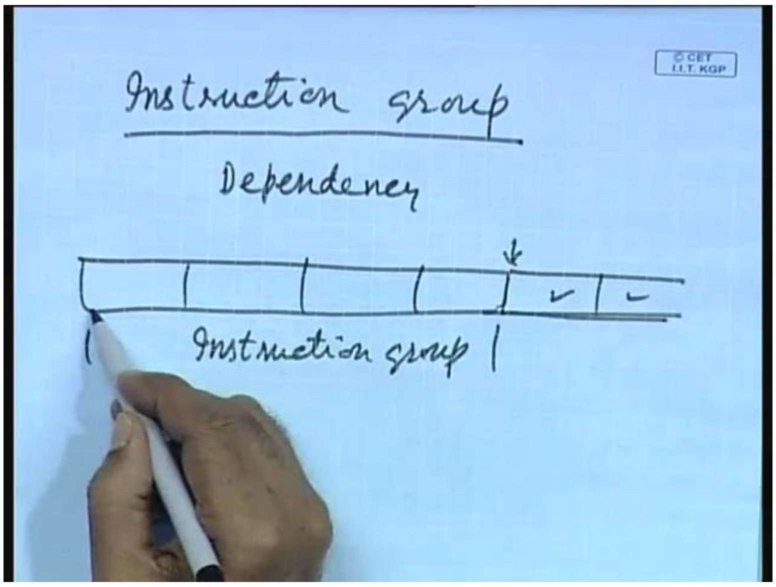


Template Field

- The 5-bit template field specifies:
  - The types of execution units each instruction in the bundle requires.
  - The field specifies the presence of stops. A stop indicates to the hardware that one or more instructions before stop may have certain kinds of resource dependencies with one or more instructions after the stop.
- Dependent and independent instructions may be mixed in the same bundle
  - Compiler set template bits to inform hardware which instructions are independent.
  - Template can identify independent instructions across bundles.
  - Instructions in bundles do not have to be in the original program textual order.

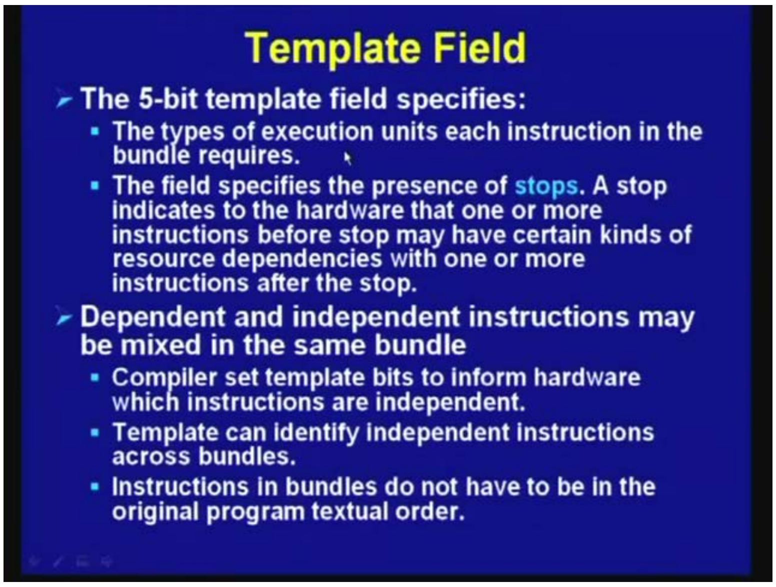
Now, the field specifies the presence of stops.

(Refer Slide Time: 33:19)



That means, you can have several instructions and suppose up to these they form a instruction group; however, this instruction, and this instruction does not fall in this group; that means, they cannot be executed in parallel because, of the dependencies. Then the stop bits specifies that there is a stop bit here; that means, these instructions, and these instructions cannot be executed in parallel because, of this dependency.

(Refer Slide Time: 34:00)



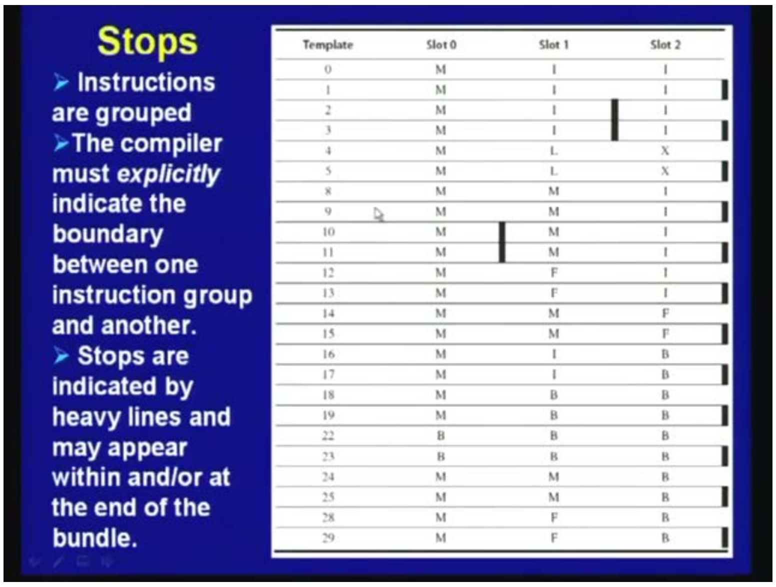
Template Field

- The 5-bit template field specifies:
  - The types of execution units each instruction in the bundle requires.
  - The field specifies the presence of stops. A stop indicates to the hardware that one or more instructions before stop may have certain kinds of resource dependencies with one or more instructions after the stop.
- Dependent and independent instructions may be mixed in the same bundle
  - Compiler set template bits to inform hardware which instructions are independent.
  - Template can identify independent instructions across bundles.
  - Instructions in bundles do not have to be in the original program textual order.

So, a stop indicates to the hardware that one or more instructions before stops may have a certain kinds of resource dependencies, with one or more instructions after the stop. So, that is how, it separates the instruction groups, in a single bundle, and; however, whenever you forming a bundle for execution by a processor. You will see that in IA 64 it is possible to bundle, instructions not belonging to the same instruction group.

That means, dependent and independent instructions may be mixed in a same bundle, and compiler set template bits to inform hardware which instructions are independent, and template can identify independent instructions across bundles. And instructions in bundles do not have to be in the original program textual order; that means, the way these instructions are available in your original text, they may not be in the same in your instruction bundle that you prepare.

(Refer Slide Time: 35:12)



Stops

- Instructions are grouped
- The compiler must *explicitly* indicate the boundary between one instruction group and another.
- Stops are indicated by heavy lines and may appear within and/or at the end of the bundle.

Template	Slot 0	Slot 1	Slot 2
0	M	I	I
1	M	I	I ▌
2	M	I ▌	I
3	M	I ▌	I ▌
4	M	L	X
5	M	L	X ▌
8	M	M	I
9	M	M	I ▌
10	M ▌	M	I
11	M ▌	M	I ▌
12	M	F	I
13	M	F	I ▌
14	M	M	F
15	M	M	F ▌
16	M	I	B
17	M	I	B ▌
18	M	B	B
19	M	B	B ▌
22	B	B	B
23	B	B	B ▌
24	M	M	B
25	M	M	B ▌
28	M	F	B
29	M	F	B ▌

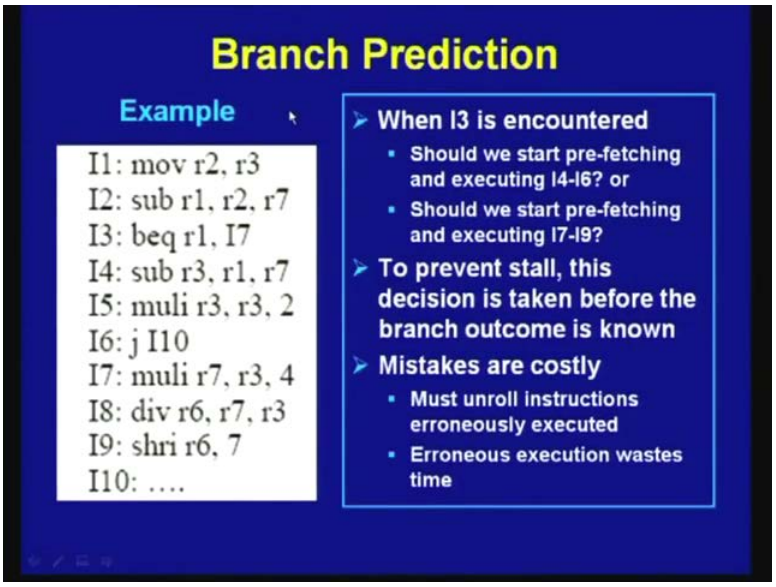
And let me illustrate this with the help of an example, so here as you can see you have got a number of bundles, and this template field specifies the type of execution units they require, I mean required by different instruction in a bundle. As I have already mentioned a single bundle, we will have three instructions, so this template 0 specifies that the three instructions, will require three execution units one is M type slot 0, slot 1 will require I type, and slot 2 will require I type.

So, since you have got multiple execution units you can execute these instructions simultaneously. Similarly, you have got another instruction M I I that template 1 specifies that second bundle, and here you can see there is a stop; that means, they form a single bundle, single instruction group. Here, you see that after the first bundle there was no stop, but you can see here there was a stop; that means, these instructions, and these instructions from a single instruction group.

Although they are bundle in two separate, I mean they form two separate bundles for execution purpose by IA 64, but here you can see the stop is present in a single bundle. So, M I and I, so here you have got a stop, so the compiler specifies must explicitly indicate the boundary between one instruction group and the another, stops are indicate by heavy lines may appear within and or end of the bundle. So, you can see conventionally it is present at the end of the bundle, but in some cases it is present in the middle also in the middle of the bundle.

So, the compiler will specify to the this facilitates the coding and execution of the instructions.

(Refer Slide Time: 37:30)



Branch Prediction

Example

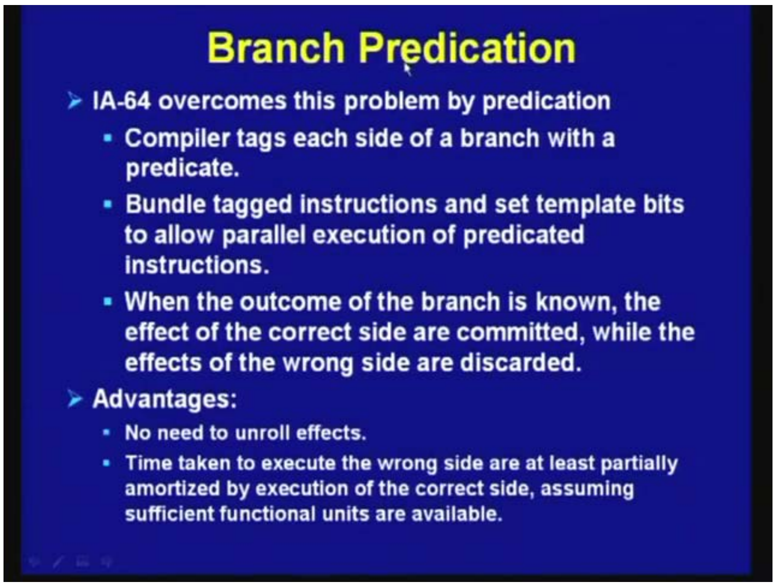
```
I1: mov r2, r3
I2: sub r1, r2, r7
I3: beq r1, I7
I4: sub r3, r1, r7
I5: muli r3, r3, 2
I6: j I10
I7: muli r7, r3, 4
I8: div r6, r7, r3
I9: shri r6, 7
I10: ....
```

- When I3 is encountered
  - Should we start pre-fetching and executing I4-I6? or
  - Should we start pre-fetching and executing I7-I9?
- To prevent stall, this decision is taken before the branch outcome is known
- Mistakes are costly
  - Must unroll instructions erroneously executed
  - Erroneous execution wastes time

Now, coming to the branch prediction we know that earlier super scalar processors, uses branch prediction. So, here is a sequence of instructions I 1 to I 10, so here, now here you have got the branch instructions I 3 branch instruction, now whenever we reach this branch instruction, some prediction is required whether these instruction will be fetched, I 4, I 5 and I 6 will be fetched or I 7, I 8 or I 9 are to be fetched that is decided based on branch prediction. That means, if a branch is taken, prediction is taken, then you will fetch I 7, I 8 and I 9.

On the other hand, if the branch is not taken then you will fetch I 4, I 5 and I 6, so that is based on branch prediction that is done in traditional processors. So, in your Pentium 3, Pentium 4 in all these processor that is how it is being done.

(Refer Slide Time: 38:52)



Branch Predication

- IA-64 overcomes this problem by predication
  - Compiler tags each side of a branch with a predicate.
  - Bundle tagged instructions and set template bits to allow parallel execution of predicated instructions.
  - When the outcome of the branch is known, the effect of the correct side are committed, while the effects of the wrong side are discarded.
- Advantages:
  - No need to unroll effects.
  - Time taken to execute the wrong side are at least partially amortized by execution of the correct side, assuming sufficient functional units are available.

But, in case of this IA 64 processor it goes beyond it, you are introducing a concept known as branch predication. So, this problem that which one to be fetched whether you will fetch those instructions, in the branch is taken or the branch is not taken that problem is overcome that problem of prediction is overcome, with the help of this concept of predication. So, the compiler tags each side of a branch with a predicate, so instructions are tagged with the help of a predicate field.

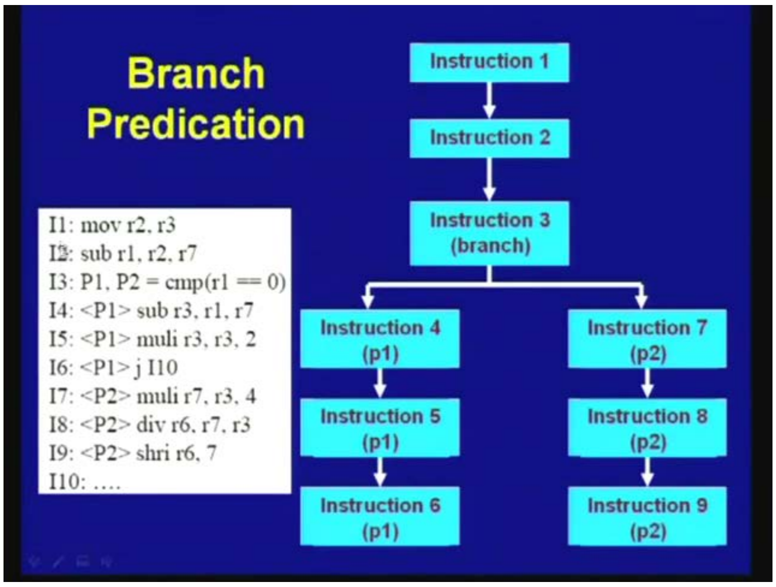
And bundle tagged instructions and set template bits to allow parallel execution of predicated instructions. So, here because, we have got large number of execution units, instead of executing only one side of the branch taken side or not taken side, the execution is carried out for both the sides. However, you have to keep a track of which instructions belong to which side, so when the outcome of the branch is known, the effect of the correct side is committed while the effect of the wrong side is discarded.

That means, the execution is carried out for both sides, but only when it is know that the branch will be taken or not taken, only results of one side is committed by committed we mean that writing into the registers and all this things will be take place. But, the other side will be completely discarded, I shall explain with the help of an example little later and it has advantage that there is no need to unroll earlier we have seen.

If the prediction is wrong, then you have to do you have to unroll it, so you have to discard some instructions, then you have to fetch a new set of instructions, and you have to execute that. So, this kind of unrolling is completely avoided in this predication concept, so the time taken to execute the wrong side are at least partially amortized by the execution of the correct side. Assuming sufficient functional units are available as I mentioned time taken to execute, as I mentioned d time taken to execute the wrong side is also there.

But, since we have got sufficient number of functional units, you can execute them parallel, you can execute them parallel because, they are independent they can be independently executed. So, there is no dependency between the right side, the correct side or wrong side, so they can be executed in parallel I shall illustrate that with the help of an example.

(Refer Slide Time: 41:45)



Branch Predication

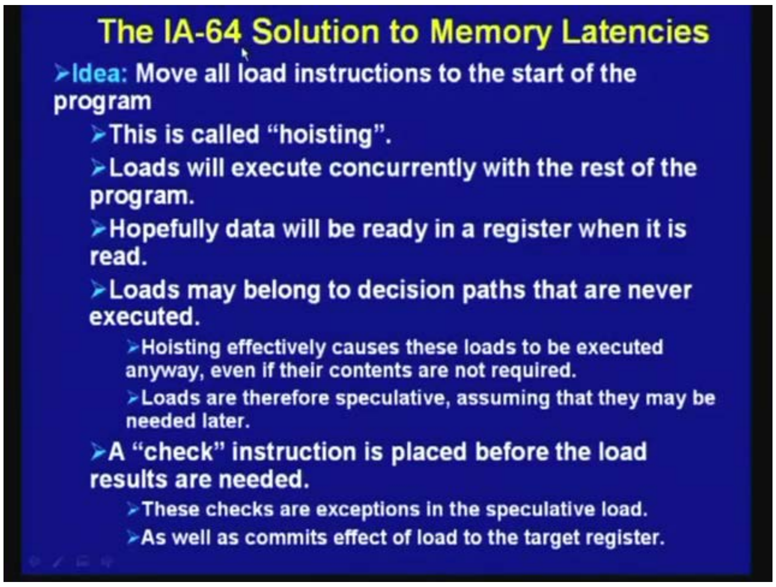
```
I1: mov r2, r3
I2: sub r1, r2, r7
I3: P1, P2 = cmp(r1 == 0)
I4: <P1> sub r3, r1, r7
I5: <P1> muli r3, r3, 2
I6: <P1> j I10
I7: <P2> muli r7, r3, 4
I8: <P2> div r6, r7, r3
I9: <P2> shri r6, 7
I10: ....
```

Here for example, you have got a sequence of instructions I 3, I 1, I 2, I 3, I 3 is a branched instruction, and there are two directions. So, I 4, I 5, I 6, I 4, I 5, I 6, from one path and I 7, I 8, and I 9 from another path, so these are tagged with the help of that there is a predicate register. So, for this instruction 4, instruction 5 and instruction 6 in the predicate register write p 1 that path 1 is written here p 1, p 1 and p 1. On the other hand for the instructions 7, 8 and 9 in the predicate register p 2 is written.

So, each instruction is predicated, so you are predicating the instructions and as a consequence, when the outcome of a branch is known, whether I mean that r 1 is 0 or not. Because, that may take a little time the comparison of instructions may take time when it is known, you can discard either the p 1 instructions I mean the instructions predicated by p 1 or you can discard the instructions predicated by p 2.

However, the beauty of this instructions that 4, 5, 6 and 7, 8, 9 they can be executed parallel because, they are independent. So, this is the basics concept of branch predication.

(Refer Slide Time: 43:30)



The IA-64 Solution to Memory Latencies

- Idea: Move all load instructions to the start of the program
  - This is called "hoisting".
  - Loads will execute concurrently with the rest of the program.
  - Hopefully data will be ready in a register when it is read.
  - Loads may belong to decision paths that are never executed.
    - Hoisting effectively causes these loads to be executed anyway, even if their contents are not required.
    - Loads are therefore speculative, assuming that they may be needed later.
  - A "check" instruction is placed before the load results are needed.
    - These checks are exceptions in the speculative load.
    - As well as commits effect of load to the target register.

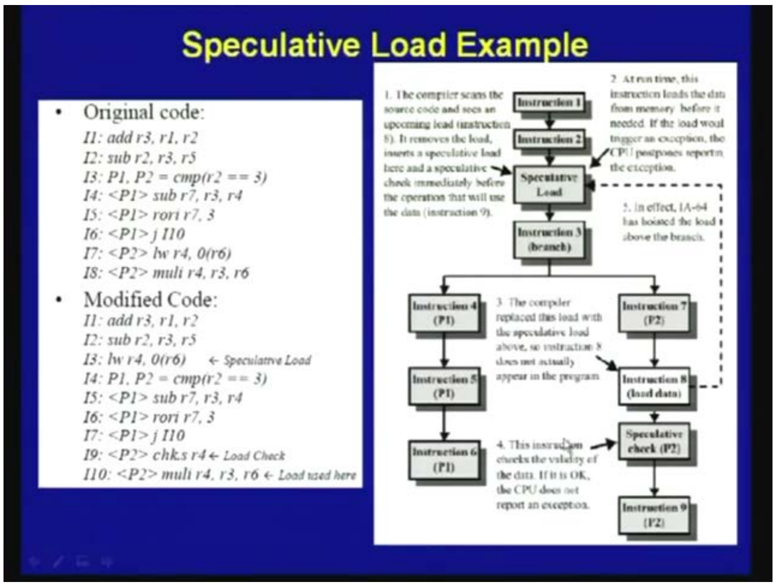
Now, coming to another important aspect of IA 64 that is the solution to memory latency. So, we have already seen that memory you have to fetch instructions from the memory, so what IA 64 does, it does speculative fetching of instructions actually the technique is known as hoisting.

So, move all load instructions to the start of the program, this is called hoisting; that means, you are load instructions which may be present in the later part of the instruction, they are moved to the beginning part. So, they are fetched on the assumption that they will be needed in future, but they may not be needed, but I mean hopefully data will be ready in the register when it is read; that means, when it will be needed.

And loads may belong to the decision path that are never executed, as I mentioned that some of the loads I mean where you load the data from the memory to the register that path may not be executed. And hoisting effectively causes these loads to be executed anyway, even if their contents are never required, so loads are there for speculative assuming that they may be needed later, so you are doing speculative loading of instructions.

So, a check instruction is placed before the load results are needed, and these checks are exceptions in the speculative load, and as well as commits effect to the load register. So, whenever you are doing the committing, then that has got some effect on the loading in to the target register.

(Refer Slide Time: 45:31)



Speculative Load Example

- Original code:

```
I1: add r3, r1, r2
I2: sub r2, r3, r5
I3: P1, P2 = cmp(r2 == 3)
I4: <P1> sub r7, r3, r4
I5: <P1> rori r7, 3
I6: <P1> j I10
I7: <P2> lw r4, 0(r6)
I8: <P2> muli r4, r3, r6
```

- Modified Code:

```
I1: add r3, r1, r2
I2: sub r2, r3, r5
I3: lw r4, 0(r6)      ← Speculative Load
I4: P1, P2 = cmp(r2 == 3)
I5: <P1> sub r7, r3, r4
I6: <P1> rori r7, 3
I7: <P1> j I10
I9: <P2> chk.s r4 ← Load Check
I10: <P2> muli r4, r3, r6 ← Load used here
```

1. The compiler scans the source code and sees an upcoming load (instruction 8). It removes the load, inserts a speculative load here and a speculative check immediately before the operation that will use the data (instruction 9).

2. At run time, this instruction loads the data from memory before it needed. If the load woul trigger an exception, the CPU postpones reportin the exception.

3. The compiler replaced this load with the speculative load above, so instruction 8 does not actually appear in the program.

4. This instruction checks the validity of the data. If it is OK, the CPU does not report an exception.

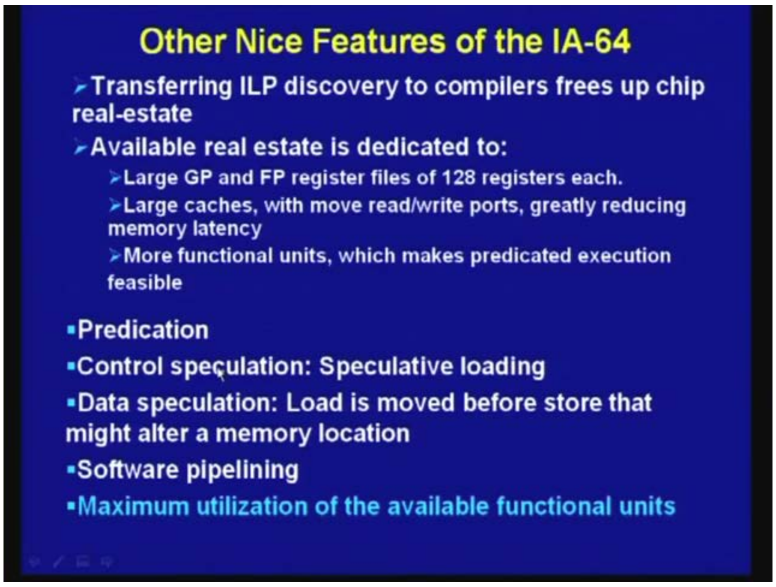
5. In effect, IA-64 has hoisted the load above the branch.

So, this is the speculative loading that is being done, and here is the example of speculative loading. Here for example, you have got the instructions sequence, this is the original code, where you have got the load instruction here instruction 7 is the load instruction. So, what is done that instruction 7 is shifted before the branch, so before the branch takes place; that means, I 3 is your branch, before the branch this instruction reference is shifted in the modified code as a code you can see, this instruction 3 is now load instruction load r 4 0 r 6.

So, you are doing speculative loading then you are going for the branch instruction so; however, whenever you do that here you can see a performing load check. So, after executing this you are doing speculative check, so this is how the instructions sequence are executed. So, compiler scans the source code and identifies the upcoming load instructions, identifies the load instructions, and at run time you know these instructions load the data from the memory, before it is needed.

And the if load would trigger and exception this CPU postpones reporting that exception. So, it may lead to some exception and that is not being informed, so it is not reported to the processor, and as I mentioned the compiler replaces the load with the help of speculative load above, and here you have got a speculative check. So, the instructions check the validity of the data, if it is then the CPU does not report an exception, otherwise an exception is reported. So, this is how you do speculative loading and you can see how the code is modified.

(Refer Slide Time: 47:44)



Other Nice Features of the IA-64

- Transferring ILP discovery to compilers frees up chip real-estate
- Available real estate is dedicated to:
  - Large GP and FP register files of 128 registers each.
  - Large caches, with move read/write ports, greatly reducing memory latency
  - More functional units, which makes predicated execution feasible

- Predication
- Control speculation: Speculative loading
- Data speculation: Load is moved before store that might alter a memory location
- Software pipelining
- Maximum utilization of the available functional units

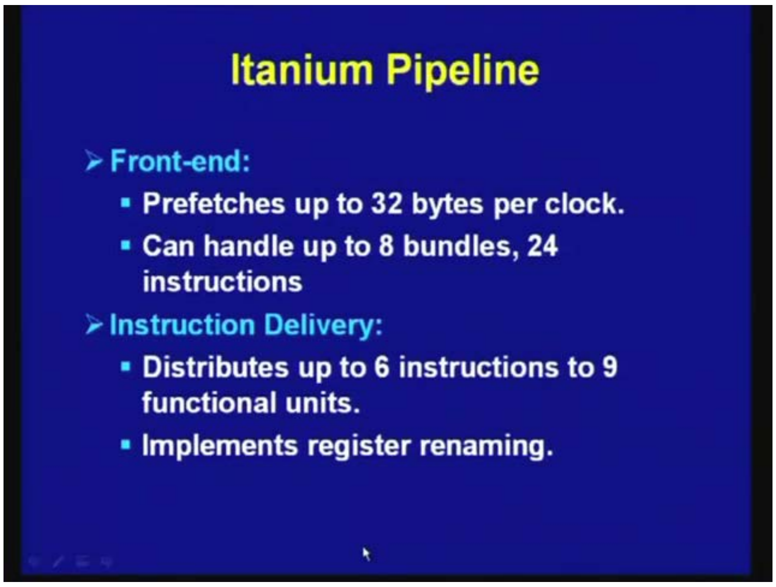
And here are some other nice features of IA 64, so transferring ILP discovery to compiler frees up the chip real estate, as I have already told since it is done by software, silicon real estate is available which can be used for more beneficial purposes. Like it can be used for implementing large number of general purpose registers and floating point register files and we have seen, each of them is 128 bit register files have in IA 64. And also it can be used for large cache memories, with more read write and also it more read write port, this greatly reduces memory latency.

That means, instead of using single ported memory you can have read write ports, move read write ports and this can reduce the memory latency. And you can have more functional units, which makes predicated execution feasible, we have seen whenever we do predicated execution of instructions, you require large number of functional units because, we have to execute both the paths. So, this approach of identifying instruction level parallelism, by software has got many benefits.

Because, the real estate that is being saved can be used for these beneficial purposes, and I have already mentioned about the nice features of IA 64 predication, instead of prediction. And that definitely enhances the efficiency of the program execution, then the control speculation and that speculative loading that is being done of data, and data speculation load is moved before the store that might alter the memory location. So, this data speculation is done, and it facilitates software pipelining, earlier we have discussed about software pipelining.

And whenever these features are used, we have seen in case of software pipelining we have in addition to those loop portion, we have got some I mean post and I mean there are some course in the beginning and at the end. So, those course can be reduced or completely overcome with the help of these features, I am not going into the details of that and this leads maximum utilization of the available functional units.

(Refer Slide Time: 50:33)



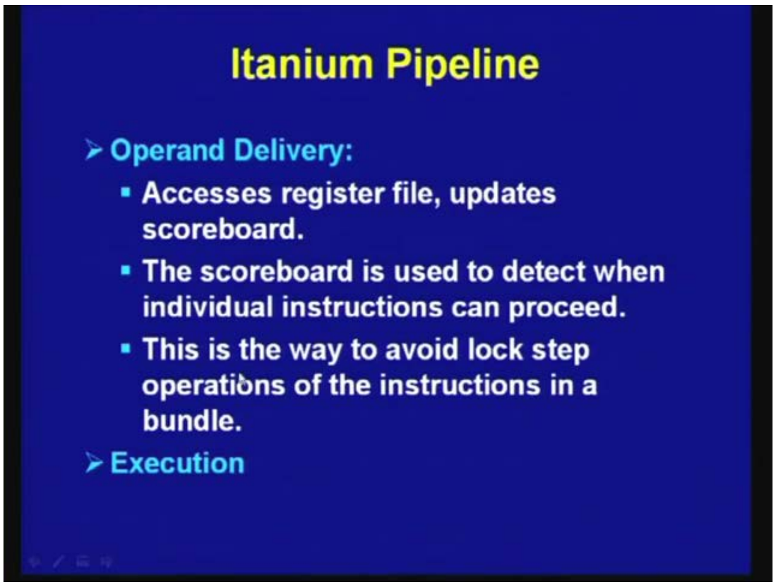
Itanium Pipeline

- Front-end:
  - Prefetches up to 32 bytes per clock.
  - Can handle up to 8 bundles, 24 instructions
- Instruction Delivery:
  - Distributes up to 6 instructions to 9 functional units.
  - Implements register renaming.

Coming to the itanium processors, the pipelining let us look at the pipelining that is used in itanium processors, in the front end it does the perfecting of instructions. So, the perfecting of instructions is done up to 32 bites per clock, so you can see, so up to 32 bytes per clock perfecting is done, by the front end of the processor and it can handle up to 8 bundles. So, each bundle as you have seen comprises 128 bits, and those bundles can be; that means, 8 bundles; that means, each bundle has got free instructions.

So, 24 instructions, so front end we will do that then the instruction delivery is performed by the in a pipelined way, it distributes up to 6 instructions to 9 functional units. So, it uses up to 9 functional units, and up to 6 instructions can be distributed to the 9 functional units. And it uses register renaming I have already discussed about that register renaming is used particularly for that procedural calls, we are using with the help of registers instead of memory for stack.

(Refer Slide Time: 52:08)

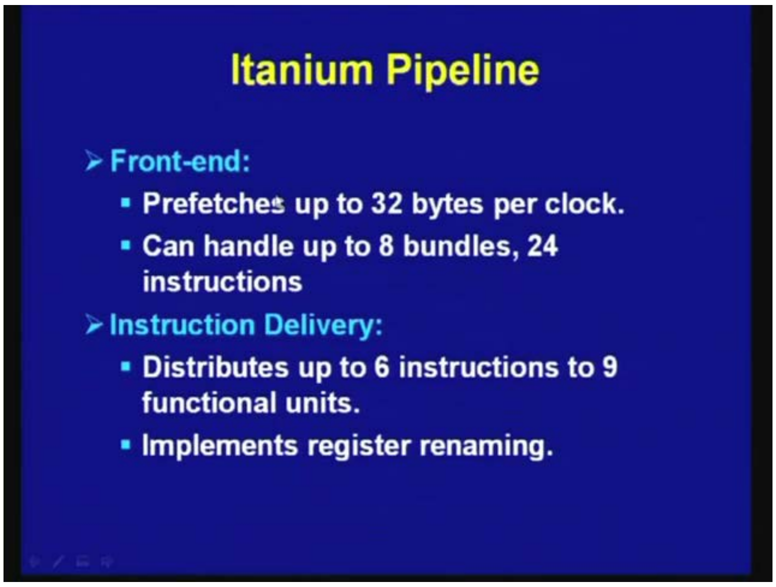


Itanium Pipeline

- Operand Delivery:
  - Accesses register file, updates scoreboard.
  - The scoreboard is used to detect when individual instructions can proceed.
  - This is the way to avoid lock step operations of the instructions in a bundle.
- Execution

Then operant delivery is performed it accesses register file updates scoreboard and the scoreboard is used to detect when the individual instruction can proceed, we have discussed in detail about the score boarding approach. And this score boarding approach is used here, used to detect when individual instruction can proceed and this is the way to avoid lock step operations in the instructions in a bundle. So, this is how the execution of operant delivery is done to perform to get a better efficiency and then execution of instruction is done.

(Refer Slide Time: 52:53)

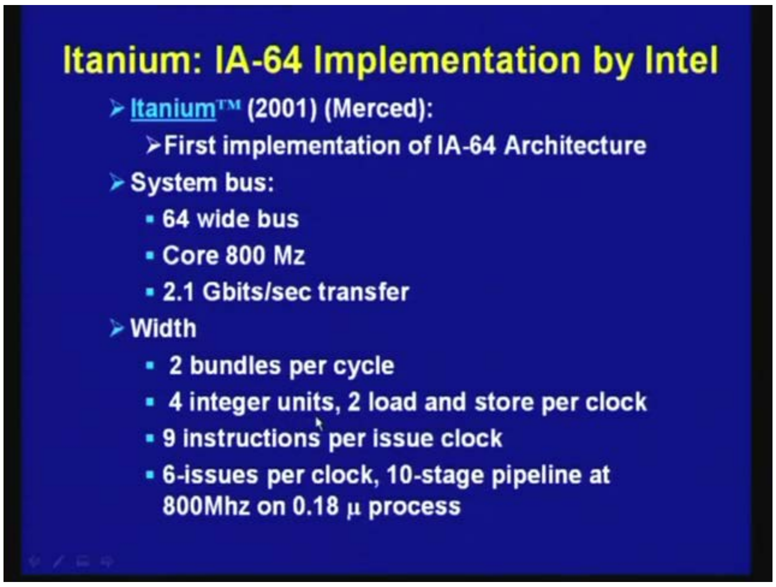


Itanium Pipeline

- Front-end:
  - Prefetches up to 32 bytes per clock.
  - Can handle up to 8 bundles, 24 instructions
- Instruction Delivery:
  - Distributes up to 6 instructions to 9 functional units.
  - Implements register renaming.

So, these are the pipelining front end performs this.

(Refer Slide Time: 53:03)



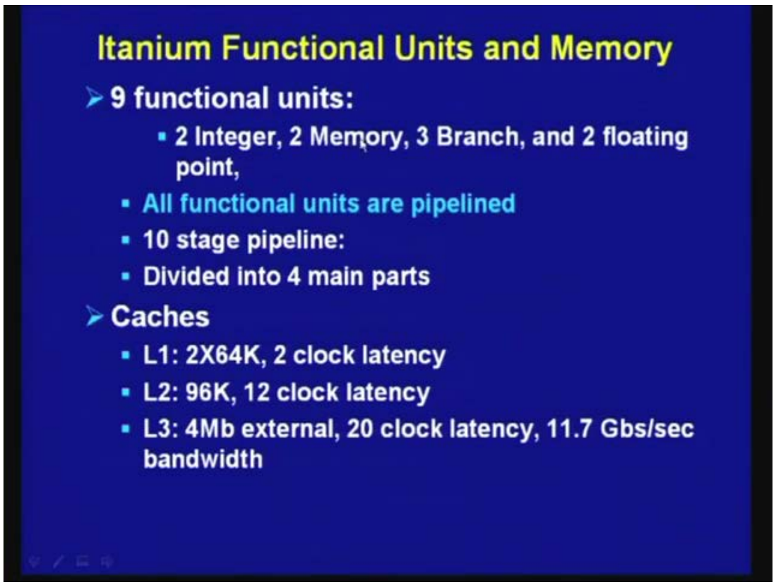
Itanium: IA-64 Implementation by Intel

- Itanium™ (2001) (Merced):
  - First implementation of IA-64 Architecture
- System bus:
  - 64 wide bus
  - Core 800 Mz
  - 2.1 Gbits/sec transfer
- Width
  - 2 bundles per cycle
  - 4 integer units, 2 load and store per clock
  - 9 instructions per issue clock
  - 6-issues per clock, 10-stage pipeline at 800Mhz on 0.18 μ process

Then instruction delivery, then the operand delivery, then execution of the instructions and this is the example itanium IA 64 implementation by Intel. So, this was done back in 2001 itanium, it was code named as Merced, so this was the first implementation of IA 64 architecture. And used 64 wide bus using 800 megahertz core, and 2.1 gigabytes per second transfer through the system bus, and the width of the bus was 2 bundles per cycle, and it used 4 integer units to load and store per clock 4 integer units and 2 load and store per clock.

And 9 instructions per issue clock and 6 issue per clock, and 10 stage pipeline using 800 megahertz on 0.18 micron process.

(Refer Slide Time: 54:00)

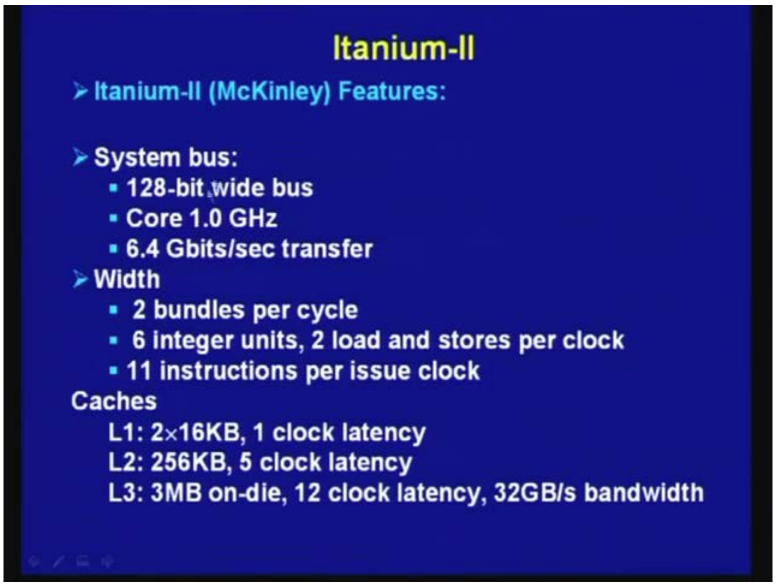

Itanium Functional Units and Memory

- 9 functional units:
    - 2 Integer, 2 Memory, 3 Branch, and 2 floating point,
  - All functional units are pipelined
  - 10 stage pipeline:
  - Divided into 4 main parts
- Caches
  - L1: 2X64K, 2 clock latency
  - L2: 96K, 12 clock latency
  - L3: 4Mb external, 20 clock latency, 11.7 Gbs/sec bandwidth

And subsequently it was enhanced, so these are 9 functional units 2 integer, 2 memory, 3 branch and 2 floating point, and all these functional units are pipelined using 10 stage pipeline. And divided into 4 main parts, which I already have mentioned then it uses 3 different types of caches L1 2 in to 64 that is instruction and data, it has got two separate caches.

And the L 2 cache is you know unified cache 96 kilobyte, using having the latency of 2 clocks and your L 1 has latency of 2 clocks and L 3 cache memory has 4 megabyte which is external in case of itanium and 20 clock latency and using 11.6 gigabytes per second bandwidth.

(Refer Slide Time: 54:57)



Itanium-II

- Itanium-II (McKinley) Features:

- System bus:
  - 128-bit wide bus
  - Core 1.0 GHz
  - 6.4 Gbits/sec transfer
- Width
  - 2 bundles per cycle
  - 6 integer units, 2 load and stores per clock
  - 11 instructions per issue clock

Caches

L1: 2×16KB, 1 clock latency

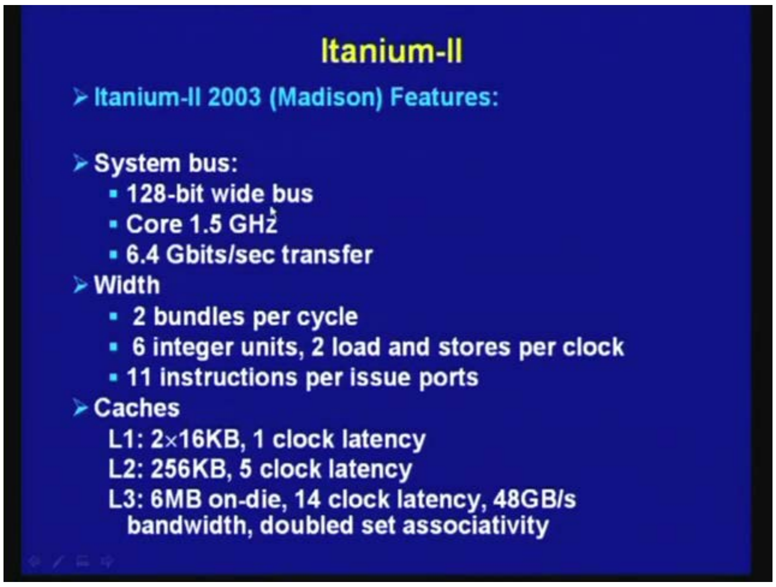
L2: 256KB, 5 clock latency

L3: 3MB on-die, 12 clock latency, 32GB/s bandwidth

And itanium 2 which was nicknamed as McKinley, it has the feature of 128 bit wide bus. So, width capacity bus is increased core of 1.0 gigahertz and 6.4 gigabytes per second transfer and this is the width 2 bundles per cycle, 6 integer units 2 loads and store per clock, and 11 instructions per issue clock. And, so far the cache memory is concerned, the L 1 cache is again separate for data instruction 2 into 64 kilobyte, with 1 clock latency instead of 2 that was there in the earlier processor.

And L 2 cache has got 256 kilobytes lodger, and it has got the latency of 5 clock cycles and L 3 is on die 3 mega byte and 12 clock latency and 32 gigabytes per second bandwidth.

(Refer Slide Time: 55:47)



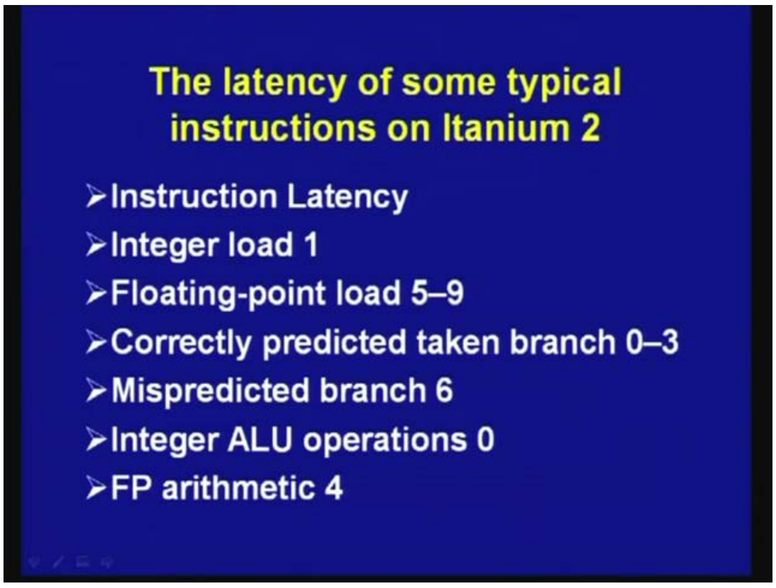
Itanium-II

- Itanium-II 2003 (Madison) Features:

- System bus:
  - 128-bit wide bus
  - Core 1.5 GHz
  - 6.4 Gbits/sec transfer
- Width
  - 2 bundles per cycle
  - 6 integer units, 2 load and stores per clock
  - 11 instructions per issue ports
- Caches
  - L1: 2×16KB, 1 clock latency
  - L2: 256KB, 5 clock latency
  - L3: 6MB on-die, 14 clock latency, 48GB/s bandwidth, doubled set associativity

So, then itanium 2 that was introduced in 2003 and these are the features a little bit enhanced compared to the earlier.

(Refer Slide Time: 55:59)

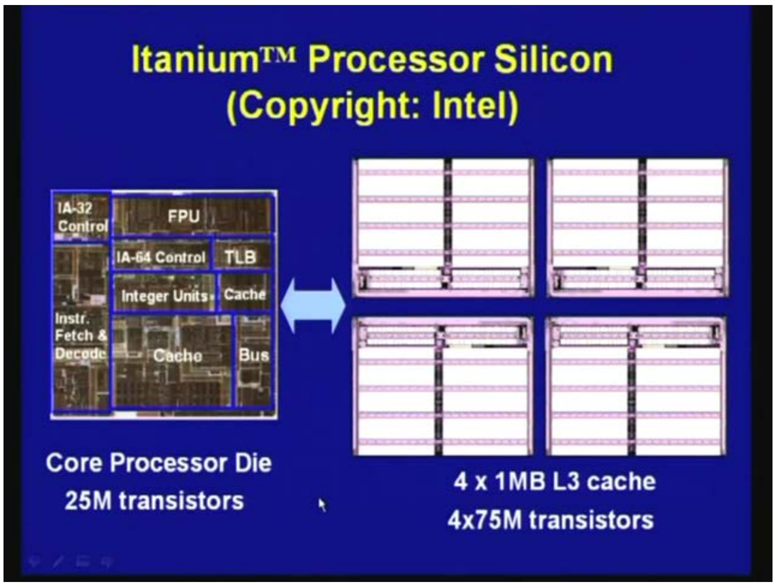


The latency of some typical instructions on Itanium 2

- Instruction Latency
- Integer load 1
- Floating-point load 5–9
- Correctly predicted taken branch 0–3
- Mispredicted branch 6
- Integer ALU operations 0
- FP arithmetic 4

And, these are the latency or some of the typical instructions I have already mentioned about, integer load 1, floating.5 to 9 cycles and that correctly predicted taken branch 0 to 3. And mispredicted branch 6 cycles, integer ALU operations does not require any latency because, it is available in the registers and floating point arithmetic require 4 cycles.

(Refer Slide Time: 56:25)

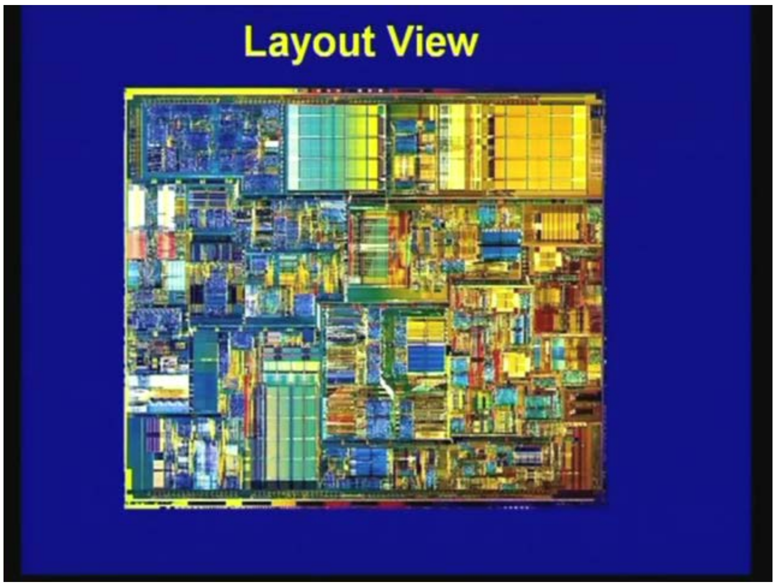

Itanium™ Processor Silicon
(Copyright: Intel)

Core Processor Die
25M transistors

4 x 1MB L3 cache
4x75M transistors

And these are the you know that core processor die that it looks like.

(Refer Slide Time: 56:36)


Layout View

This is the layout view, so with this we have come to the end of our lecture on itanium, and particularly on various types of processors based on instruction level parallelism. In my next lecture, we shall start our discussion on another type of parallelism that is your thread level parallelism.

Thank you.