

## VERY LARGE INSTRUCTION WORD (VLIW) PROCESSORS

1. Problems with Superscalar Architectures
2. VLIW Processors
3. Advantages and Problems
4. Loop Unrolling
5. Trace Scheduling
6. The Merced Architecture

## What is Good and what is Bad with Superscalars ?

### Good

- The hardware solves everything:
  - Hardware detects potential parallelism between instructions;
  - Hardware tries to issue as many instructions as possible in parallel.
  - Hardware solves register renaming.
- Binary compatibility
  - If functional units are added in a new version of the architecture or some other improvements have been made to the architecture (without changing the instruction sets), old programs can benefit from the additional potential of parallelism.

Why?  
Because the new hardware will issue the old instruction sequence in a more efficient way.

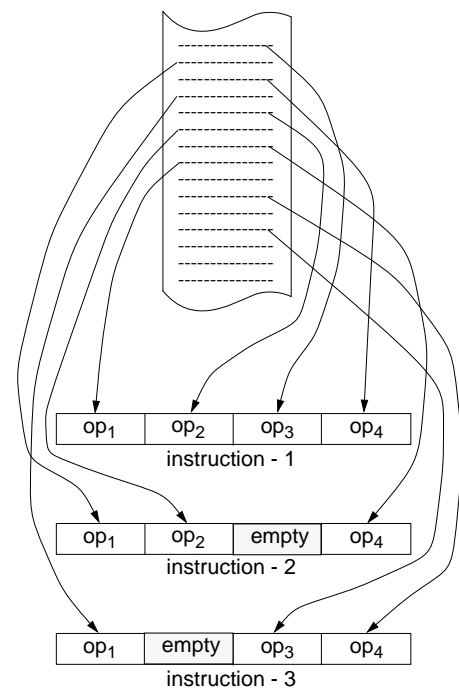
### Bad

- Very complex
  - Much hardware is needed for run-time detection. There is a limit in how far we can go with this technique.
- The *window of execution* is limited  $\Rightarrow$  this limits the capacity to detect potentially parallel instructions

## The Alternative: VLIW Processors

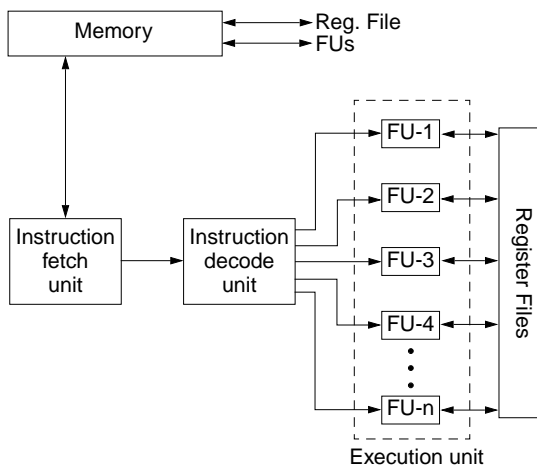
- VLIW architectures rely on compile-time detection of parallelism  $\Rightarrow$  the compiler analysis the program and detects operations to be executed in parallel; such operations are packed into one "large" instruction.
  - After one instruction has been fetched all the corresponding operations are issued in parallel.
- ↓
- No hardware is needed for run-time detection of parallelism.
  - The *window of execution* problem is solved: the compiler can potentially analyse the whole program in order to detect parallel operations.

## VLIW Processors



- Detection of parallelism and packaging of operations into instructions is done, by the compiler, off-line.

## VLIW Processors (cont'd)



## Advantages and Problems with VLIW Processors

### Advantages

- Simple hardware:
  - the number of FUs can be increased without needing additional sophisticated hardware to detect parallelism, like in superscalars.
- Good compilers can detect parallelism based on global analysis of the whole program (no window of execution problem).

### Problems

- Large number of registers needed in order to keep all FUs active (to store operands and results).
- Large data transport capacity is needed between FUs and the register file and between register files and memory.
- High bandwidth between instruction cache and fetch unit.  
Example: one instruction with 7 operations, each 24 bits  $\Rightarrow$  168 bits/instruction.
- Large code size, partially because unused operations  $\Rightarrow$  wasted bits in instruction word.

- Incomputability of binary code  
For example:

If for a new version of the processor additional FUs are introduced  $\Rightarrow$  the number of operations possible to execute in parallel is increased  $\Rightarrow$  the instruction word changes  $\Rightarrow$  old binary code cannot be run on this processor.

## An Example

Consider the following code in C:

```
for (i=959; i >= 0; i--)
    x[i] = x[i] + s;
```

Assumptions: **x** is an array of floating point values  
**s** is a floating point constant.

This sequence (for an ordinary processor) would be compiled to:

```
Loop: LDD    F0, (R1)   F0  $\leftarrow$  x[i] ;(load double)
      ADF    F4,F0,F2   F4  $\leftarrow$  F0 + F2 ;(floating pnt)
      STD    (R1),F4   x[i]  $\leftarrow$  F4 ;(store double)
      SBI    R1,R1,#8  R1  $\leftarrow$  R1 - 8
      BGEZ   R1,Loop
```

Assumptions:

- R1 initially contains the address of the last element in **x**; the other elements are at lower addresses; **x**[0] is at address 0.
- Floating point register F2 contains the value **s**.
- Each floating point value is 8 bytes long.

## An Example (cont'd)

Consider a VLIW processor:

- two memory references, two FP operations, and one integer operation or branch can be issued each clock cycle.
- The delay for a double word load is one additional clock cycle.
- The delay for a floating point operation is two additional clock cycles.
- No additional clock cycles for integer operations.

### An Example (cont'd)

LDD F0,(R1)				
		ADF F4,F0,F2		
				SBI R1,R1,#8
STD 8(R1),F4				BGEZ R1,Loop

The displacement of 8, in 8(R1), is needed because we have already subtracted 8 from R1.

- One iteration takes 6 cycles.  
The whole loop takes  $960 \cdot 6 = 5760$  cycles.
- Almost no parallelism there.
- Most of the fields in the instructions are empty.
- We have two completely empty cycles.



### Loop Unrolling

Let us rewrite the previous example:

```
for (i=959; i >= 0; i-=2){
    x[i] = x[i] + s;
    x[i-1] = x[i-1] + s;
}
```

This sequence (for an ordinary processor) would be compiled to:

```
Loop: LDD    F0, (R1)    F0 ← x[i] ;(load double)
      ADF    F4,F0,F2    F4 ← F0 + F2 ;(floating pnt)
      STD    (R1),F4    x[i] ← F4 ;(store double)
      LDD    F0, -8(R1)  F0 ← x[i-1] ;(load double)
      ADF    F4,F0,F2    F4 ← F0 + F2 ;(floating pnt)
      STD    -8(R1),F4  x[i-1] ← F4 ;(store double)
      SBI    R1,R1,#16  R1 ← R1 - 16
      BGEZ   R1,Loop
```



### Loop Unrolling (cont'd)

LDD F0,(R1)	LDD F6,-8(R1)			
		ADF F4,F0,F2	ADF F8,F6,F2	
				SBI R1,R1,#16
STD 16(R1),F4	STD 8(R1),F8			BGEZ R1,Loop

- There is an increased degree of parallelism in this case.
- We still have two completely empty cycles and empty operation.
- However, **we have a dramatic improvement in speed:**  
Two iterations take 6 cycles  
The whole loop takes  $480 \cdot 6 = 2880$  cycles



### Loop Unrolling (cont'd)

Loop unrolling is a technique used in *compilers* in order to increase the potential of parallelism in a program. This allows for more efficient code generation for processors with instruction level parallelism (which can execute several instructions in parallel).

Let us unroll three iterations in our example:

```
for (i=959; i >= 0; i-=3){
    x[i] = x[i] + s;
    x[i-1] = x[i-1] + s;
    x[i-2] = x[i-2] + s;
}
```

This sequence (for an ordinary processor) would be compiled to:

```
Loop: LDD    F0, (R1)    F0 ← x[i] ;(load double)
      ADF    F4,F0,F2    F4 ← F0 + F2 ;(floating pnt)
      STD    (R1),F4    x[i] ← F4 ;(store double)
      LDD    F0, -8(R1)  F0 ← x[i-1] ;(load double)
      ADF    F4,F0,F2    F4 ← F0 + F2 ;(floating pnt)
      STD    -8(R1),F4  x[i-1] ← F4 ;(store double)
      LDD    F0, -16(R1) F0 ← x[i-2] ;(load double)
      ADF    F4,F0,F2    F4 ← F0 + F2 ;(floating pnt)
      STD    -16(R1),F4 x[i-2] ← F4 ;(store double)
      SBI    R1,R1,#24  R1 ← R1 - 24
      BGEZ   R1,Loop
```



### Loop Unrolling (cont'd)

LDD F0,(R1)	LDD F6,-8(R1)			
LDD F10,-16(R1)				
		ADF F4,F0,F2	ADF F8,F6,F2	
		ADF F12,F10,F2		
STD (R1),F4	STD -8(R1),F8			SBI R1,R1,#24
STD 8(R1),F12				BGEZ R1,Loop

- The degree of parallelism is further improved.
- There is still an empty cycle and empty operations.
- Three iterations take 7 cycles  
The whole loop takes  $320 \times 7 = 2240$  cycles

### Loop Unrolling (cont'd)

With eight iterations unrolled:

```
for (i=959; i >= 0; i-=8){
    x[i] = x[i] + s; x[i-1] = x[i-1] + s;
    x[i-2] = x[i-2] + s; x[i-3] = x[i-3] + s;
    x[i-4] = x[i-4] + s; x[i-5] = x[i-5] + s;
    x[i-6] = x[i-6] + s; x[i-7] = x[i-7] + s;
}
```

LDD F0,(R1)	LDD F6,-8(R1)			
LDD F10,-16(R1)	LDD F14,-24(R1)			
LDD F18,-32(R1)	LDD F22,-40(R1)	ADF F4,F0,F2	ADF F8,F6,F2	
LDD F26,-48(R1)	LDD F30,-56(R1)	ADF F12,F10,F2	ADF F16,F14,F2	
		ADF F20,F18,F2	ADF F24,F22,F2	
STD (R1),F4	STD -8(R1),F8	ADF F28,F26,F2	ADF F32,F30,F2	
STD -16(R1),F12	STD -24(R1),F16			
STD -32(R1),F20	STD -40(R1),F24			SBI R1,R1,#64
STD 16(R1),F28	STD 8(R1),F32			BGEZ R1,Loop

- No empty cycles, but still empty operations
- Eight iterations take 9 cycles  
The whole loop takes  $120 \times 9 = 1080$  cycles

### Loop Unrolling (cont'd)

- Given a certain set of resources (processor architecture) and a given loop, there is a limit on how many iterations should be unrolled. Beyond that limit there is no gain any more.
- Loop unrolling increases the memory space needed to store the program.
- A good compiler has to find the optimal level of unrolling for each loop.

The example before illustrates some of the *hardware support needed to keep a VLIW processor busy*:

- large number of registers (in order to store data for operations which are active in parallel);
- large traffic has to be supported in parallel:
  - register file  $\leftrightarrow$  memory
  - register file  $\leftrightarrow$  functional units.

### Trace Scheduling

Trace scheduling is another technique used *in compilers* in order to exploit parallelism *across* conditional branches.

- The problem is that long instruction sequences are needed in order to detect sufficient parallelism  $\Rightarrow$  block boundaries have to be crossed.
- Trace scheduling is based on *compile time* branch prediction.

Trace scheduling is done in three steps:

1. Trace selection
2. Instruction scheduling
3. Replacement and compensation

### Trace Scheduling (cont'd)

Example:

```

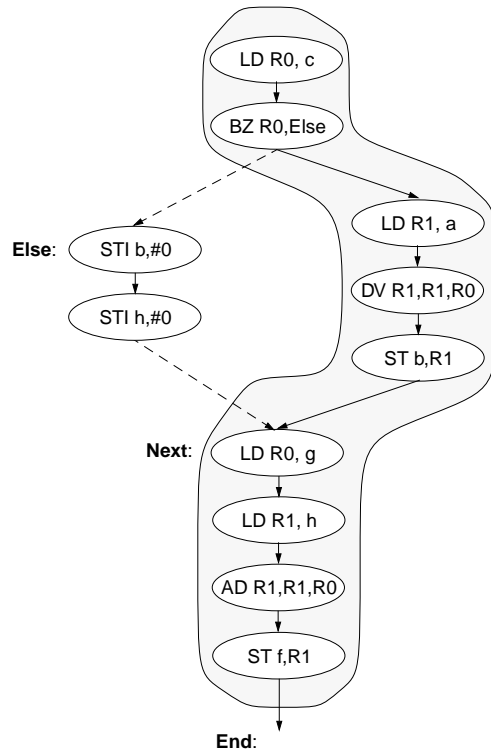
if (c != 0)
    b = a / c;
else
    b = 0; h=0;
f = g + h;
    
```

This sequence (for an ordinary processor) would be compiled to:

```

LD    R0, c      R0 ← c ;(load word)
BZ    R0,Else
LD    R1, a      R1 ← a ;(load integer)
DV    R1,R1,R0   R1 ← R1 / R0 ;(integer)
ST    b,R1       b ← R1 ;(store word)
BR    Next
Else: STI b,#0    b ← 0
      STI h,#0    h ← 0
Next: LD R0, g    R0 ← g ;(load word)
      LD R1, h    R1 ← h ;(load word)
      AD R1,R1,R0 R1 ← R1 + R0 ;(integer)
      ST f,R1     f ← R1 ;(store word)
End:  -----
    
```

### Trace Scheduling (cont'd)



### Trace Scheduling (cont'd)

Trace selection:

- Selects a sequence of basic blocks, likely to be executed most of the time. This sequence is called a *trace*.
- Trace selection is based on compile time branch prediction
  - The quality of prediction can be improved by profiling:
    - execution of the program with several typical input sequences and collection of statistics concerning conditional branches.

Instruction scheduling:

- Schedules the instructions of the selected trace into parallel operations for the VLIW processor.

	LD R0,c	LD R1,a			
	LD R2,g	LD R3,h			BZ R0,Else
					DV R1,R1,R0
Next:	ST b,R1				AD R3,R3,R2
	ST f,R3				BR End

(suppose the same processor as described on slide 8)

### Trace Scheduling (cont'd)

Replacement and compensation

- The initial trace is replaced with the generated schedule.
- In the generated schedule, instructions have been moved across branches



In order to keep the code correct, regardless of the selected branches, *compensation code* has to be added!

In the example:

- the load of values *g* and *h* has been moved up, from the *next* sequence, before the conditional branch;
- the load of value *a* has been moved before the conditional branch;
- the store of value *b* after the division is now part of the *next* sequence.

## Trace Scheduling (cont'd)

Simply merging with the code from the *else* sequence is not enough:

	LD R0,c	LD R1,a			
	LD R2,g	LD R3,h			BZ R0,Else
					DV R1,R1,R0
Next:	ST b,R1				AD R3,R3,R2
	ST f,R3				BR End
Else:	STI b,#0	STI h,#0			BR Next
End:					

The store in the *next* sequence overwrites the STI in the *else* sequence (because the store of *b* has been moved down into the next sequence)

The value assigned to *h* in the *else* sequence is ignored for the addition (because the load of *h* has been moved up from the *next* sequence)

- **Compensation is needed!**



## Trace Scheduling (cont'd)

This is the correct sequence:

	LD R0,c	LD R1,a			
	LD R2,g	LD R3,h			BZ R0,Else
					DV R1,R1,R0
Next:	ST b,R1				AD R3,R3,R2
	ST f,R3				BR End
Else:	STI R1,#0	STI h,#0			
	STI R3,#0				BR Next
End:					

This is compensation code: it has been introduced because LD R3,h has been moved up at scheduling of the selected trace.



## Trace Scheduling (cont'd)

- This is different from speculative execution (as in superscalars, for example).
- This is a *compiler optimization* and tries to optimize the code so that the path which is most likely to be taken, is executed as fast as possible.

The *prize*: possible *additional instructions* (the compensation code) *to be executed when the less probable path is taken*.

- The correct path will be always taken; however, if this is not the one predicted by the *compiler*, execution will be slower because of the compensation code.
- Beside this, *at the hardware level*, a VLIW processor can also use branch prediction and speculative execution, like any processor in order to improve the use of its pipelines.



## Some VLIW Processors

Successful current VLIW processors:

- TriMedia of *Philips*
- TMS320C6x of *Texas Instruments*

Both are targeting the multi-media market.

- The IA-64 architecture from *Intel* and *Hewlett-Packard*.
  - This family uses many of the VLIW ideas.
  - It is not "just" a multi-media processor, but intended to become "the new generation" processor for servers and workstations.
  - The first planned product (for 2000/2001): *Merced* (recently renamed *Itanium*).



## The Merced (Itanium) Architecture

The Merced is not a pure VLIW architecture, but many of its features are typical for VLIW processors.

### Basic concepts with Merced

These are typical VLIW features:

- Instruction-level parallelism fixed at compile-time.
- (Very) long instruction word.

These are specific for the IA-64 Architecture (Merced/Itanium and its followers):

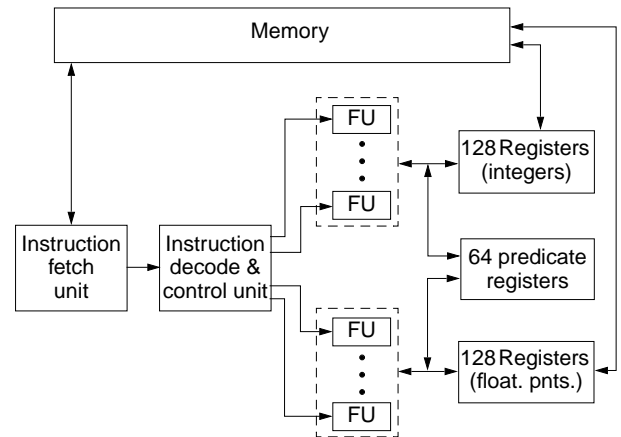
- Branch predication.
- Speculative loading

Intel calls the Merced an EPIC (explicitly parallel instruction computing) processor:

- because the parallelism of operations is explicitly there in the instruction word.



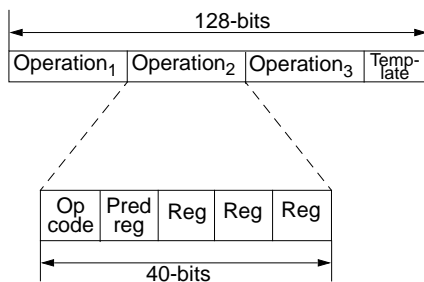
## The General Organization



- Registers (both integer and fl. pnt.) are 64-bit.
- Predicate registers are 1-bit.
- 8 or more functional units.



## Instruction Format



- 3 operations/instruction word
  - This does not mean that max. 3 operations can be executed in parallel!
  - The three operations in the instruction are not necessarily to be executed in parallel!
- The *template* indicates what can be executed in parallel.
  - The encoding in the template shows which of the operations in the instruction can be executed in parallel.
  - The template connects also to neighbouring instructions ⇒ operations from different instructions can be executed in parallel.



## Instruction Format (cont'd)

- The template provides high flexibility and avoids some of the problems with classical VLIW processors
  - Operations in one instruction have not necessarily to be parallel ⇒ no places have to be left empty when no operation is there to be executed in parallel.
  - The number of operations to be executed in parallel is not restricted by the instruction size ⇒ successive processor generations can have different number of functional units without changing the instruction word ⇒ improved binary compatibility.
  - If, according to the template, more operations can be executed in parallel than functional units are available ⇒ the processor is able to take them sequentially.



## Predicated Execution

- Any operation can refer to a predicate register (see slide 27)

<Pi> operation      $i$  is the number of a predicate register (between 0 and 63)

- This means that the respective operation is to be committed (the results made visible) only when the respective predicate is true (the predicate register gets value 1).
- If the predicate value is known when the operation is issued, the operation is executed only if this value is *true*.

If the predicate is not known at that moment, the operation is started; if the predicate turns out to be *false*, the operation is discarded.

- If no predicate register is mentioned, the operation is executed and committed unconditionally.



## Predicated Execution (cont'd)

### Predicate assignment

$P_j, P_k = \text{relation}$       $j$  and  $k$  are predicate registers (between 0 and 63).

- Will set the value of predicate register  $j$  to *true* and that of predicate register  $k$  to *false* if the relation is evaluated to *true*;  $j$  will be set to *false* and  $k$  to *true* if the relation evaluates to *false*.

### Predicated predicate assignment

<Pi>  $P_j, P_k = \text{relation}$       $i, j$  and  $k$  are predicate registers (between 0 and 63).

- Predicate registers  $j$  and  $k$  will be updated if and only if predicate register  $i$  is true.

The instruction format of the predicate assignment and the predicated predicate assignment operation is slightly different from that shown on slide 27. These operations have two and three fields for predicate registers, respectively.



## Branch Predication

- Branch predication* is a very aggressive compilation technique for generation of code with instruction level parallelism (code with parallel operations).
- Branch predication goes one step further than trace scheduling. In order to exploit all potential of parallelism in the machine it lets *operations from both branches of a conditional branch to be executed in parallel*.
- Branches can be very often eliminated and replaced by conditional execution. In order to do this hardware support is needed.
- Branch predication is based on the instructions for *predicated execution* provided by the Merced architecture.

**The idea is:** let instructions from both branches go on in parallel, before the branch condition has been evaluated. The hardware (predicated execution) takes care that only those are committed which correspond to the right branch.



## Branch Predication (cont'd)

*Branch predication* is not *branch prediction*:

- Branch prediction:** guess for one branch and then go along that one; if the guess is bad, undo all the work;
- Branch predication:** both branches are started and when the condition is known (the predicate registers are set) the right instructions are committed, the others are discarded.



There is no lost time with failed predictions.



### Branch Predication (cont'd)

#### Example:

```

if (a && b)
    j = j + 1;
else{
    if (c)
        k = k + 1;
    else
        k = k - 1;
    m = k * 5;
}
i = i + 1;

```

#### Assumptions:

The values are stored in registers, as follows:  
*a*: R0; *b*: R1; *j*: R2; *c*: R3; *k*: R4; *m*: R5; *i*: R6.

This sequence (for an ordinary processor) would be compiled to:

```

      BZ    R0, L1    branch if a == 0
      BZ    R1, L1    branch if b == 0
      ADI   R2, R2, #1 R2 ← R2 + 1;(integer)
      BR    L4
L1:   BZ    R3, L2    branch if c == 0
      ADI   R4, R4, #1 R4 ← R4 + 1;(integer)
      BR    L3
L2:   SBI   R4, R4, #1 R4 ← R4 - 1;(integer)
L3:   MPI   R5, R4, #5 R5 ← R4 * 5;(integer)
L4:   ADI   R6, R6, #1 R6 ← R6 + 1;(integer)

```



### Branch Predication (cont'd)

Let us read it in this way:

```

if not(a == 0) and not(b == 0)
if not(not(a == 0) and not(b == 0)) and not(c == 0)
if not(not(a == 0) and not(b == 0)) and not(not(c == 0))
if not(not(not(a == 0) and not(b == 0)))

```

ADI R2, R2, #1  
 ADI R4, R4, #1  
 SBI R4, R4, #1  
 MPI R5, R4, #5  
 ADI R6, R6, #1



### Branch Predication (cont'd)

The same with predicated execution:  
 (all predicates are initialised as *false*)

- (1) P1, P2 = EQ(R0, #0)
- (2) <P2> P1, P3 = EQ(R1, #0)
- (3) <P3> ADI R2, R2, #1
- (4) <P1> P4, P5 = NEQ(R3, #0)
- (5) <P4> ADI R4, R4, #1
- (6) <P5> SBI R4, R4, #1
- (7) <P1> MPI R5, R4, #5
- (8) ADI R6, R6, #1

- The compiler can plan all these instructions to be issued in parallel, except (5) and (6) which are data-dependent.
- Instructions can be started before the particular predicate on which they depend is known. When the predicate will be known, the particular instruction will or will not be committed.



### Speculative Loading

You remember when we discussed "delayed loading"  
 (Fö. 5/6):

The load is placed so that memory latency is avoided  
 (the value is already there when it's needed):

```

LOAD  R1,X    loads from address X into R1
ADD  R2,R1    R2 ← R2 + R1
ADD   R4,R3    R4 ← R4 + R3
SUB   R2,R4    R2 ← R2 - R4

```



```

LOAD  R1,X    loads from address X into R1
ADD   R4,R3    R4 ← R4 + R3
ADD  R2,R1    R2 ← R2 + R1
SUB   R2,R4    R2 ← R2 - R4

```



### Speculative Loading (cont'd)

```

ADI    R0,#1    R0 ← R0 + 1;(integer)
BZ     R0, L1
ADI    R2, R2,#1 R2 ← R2 + 1;(integer)
BR     L2
L1:    LD     R3, x    R3 ← x
ADI    R3, R3,#1 R3 ← R3 + 1;(integer)
L2:    SBI    R4, R4,#1 R4 ← R4 - 1;(integer)

```

In order to avoid memory latency the load for value  $x$  is moved by the compiler like this:

```

LD     R3, x    R3 ← x
ADI    R0,#1    R0 ← R0 + 1;(integer)
BZ     R0, L1
ADI    R2, R2,#1 R2 ← R2 + 1;(integer)
BR     L2
L1:    ADI    R3, R3,#1 R3 ← R3 + 1;(integer)
L2:    SBI    R4, R4,#1 R4 ← R4 - 1;(integer)

```

**Attention:** the load has been moved ahead of the conditional branch!

### Speculative Loading (cont'd)

What if a load is moved across a branch?

- The load will be executed for both branches. This shouldn't be a problem if parallel resources are available.

But it is a problem if, for example, a page fault is generated. *Handling such an exception takes extremely much time and resources.*



We would like to handle the exception only on the branch which really needs the loaded value!  
*This is solved by speculative loading!*

### Speculative Loading (cont'd)

With speculative loading a load instruction in the original program can be replaced by two instructions:

- A *speculative load* (LD.s) which performs the memory fetch and detects an exception (like page fault) if generated. The exception, however is not signalled to the operating system). The speculative load is the one which is moved if needed for latency reduction.
- A *checking instruction* (CHK.s) which signals the exception if one has been detected by the speculative load. If no exception has been detected, nothing happens. The checking instruction remains in place (is not moved).

By this technique, even if loads are moved across branch boundaries, exceptions are handled only on the branch which really needs the value.

### Speculative Loading (cont'd)

The sequence on slide 37 will look like this, with speculative loading:

```

LD.s   R3, x    R3 ← x
ADI    R0,#1    R0 ← R0 + 1;(integer)
BZ     R0, L1
ADI    R2, R2,#1 R2 ← R2 + 1;(integer)
BR     L2
L1:    CHK.s   R3
ADI    R3, R3,#1 R3 ← R3 + 1;(integer)
L2:    SBI    R4, R4,#1 R4 ← R4 - 1;(integer)

```

The compiler for a Merced processor will make use of both branch predication and speculative loading. For the sequence above, only speculative loading has been illustrated.

## Summary

- In order to keep up with increasing demands on performance, superscalar architectures become excessively complex.
- VLIW architectures avoid complexity by relying exclusively on the compiler for parallelism detection.
- Operation level parallelism is explicit in the instructions of a VLIW processor.
- Not having to deal with parallelism detection, VLIW processors can have a high number of functional units. This, however, generates the need for a large number of registers and high communication bandwidth.
- In order to keep the large number of functional units busy, compilers for VLIW processors have to be aggressive in parallelism detection.
- Loop unrolling is used by compilers in order to increase the degree of parallelism in loops: several iterations of a loop are unrolled and handled in parallel.
- Trace scheduling applies to conditional branches. The compiler tries to predict which sequence is the most likely to be selected, and schedules operations so that this sequence is executed as fast as possible. Compensation code is added in order to keep the program correct. This can add some overhead to other sequences.



- The IA-64 family of Intel and Hewlett-Packard uses VLIW techniques for the future generation of high-end processors. Merced (Itanium) is the first planned member of this family.
- Beside typical VLIW features, the Merced architecture uses branch predication and speculative loading.
- The instruction format avoids empty operations and allows more flexibility in the specification of parallelism. This is solved using the *template* field.
- Branch predication is based on predicated execution. The execution of an instruction can be connected to a certain predicate register. The instruction will be committed only if the specified predicate register becomes *true*.
- Branch predication allows instructions from different branches to be started in parallel. Finally, only those in the right branch will be committed.
- Speculative loading tries to reduce latencies generated by load instructions. It allows load instructions to be moved across branch boundaries, without page exceptions being handled if not needed.
- For speculative loading, a load instruction is split into two: the speculative load, which is moved, and the checking instruction which is kept on place.

