VERY LONG INSTRUCTION WORD (VLIW) PROCESSORS

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

What is Good with Superscalars?

- 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.

What is Bad with Superscalars?

- Very complex
  - Much hardware is needed for run-time detection. There is a limit in how far we can go with this technique.
  - Power consumption can be very large!

- The *instruction window* is limited $\Rightarrow$ this limits the capacity to detect potentially parallel instructions.

The Alternative: VLIW Processors

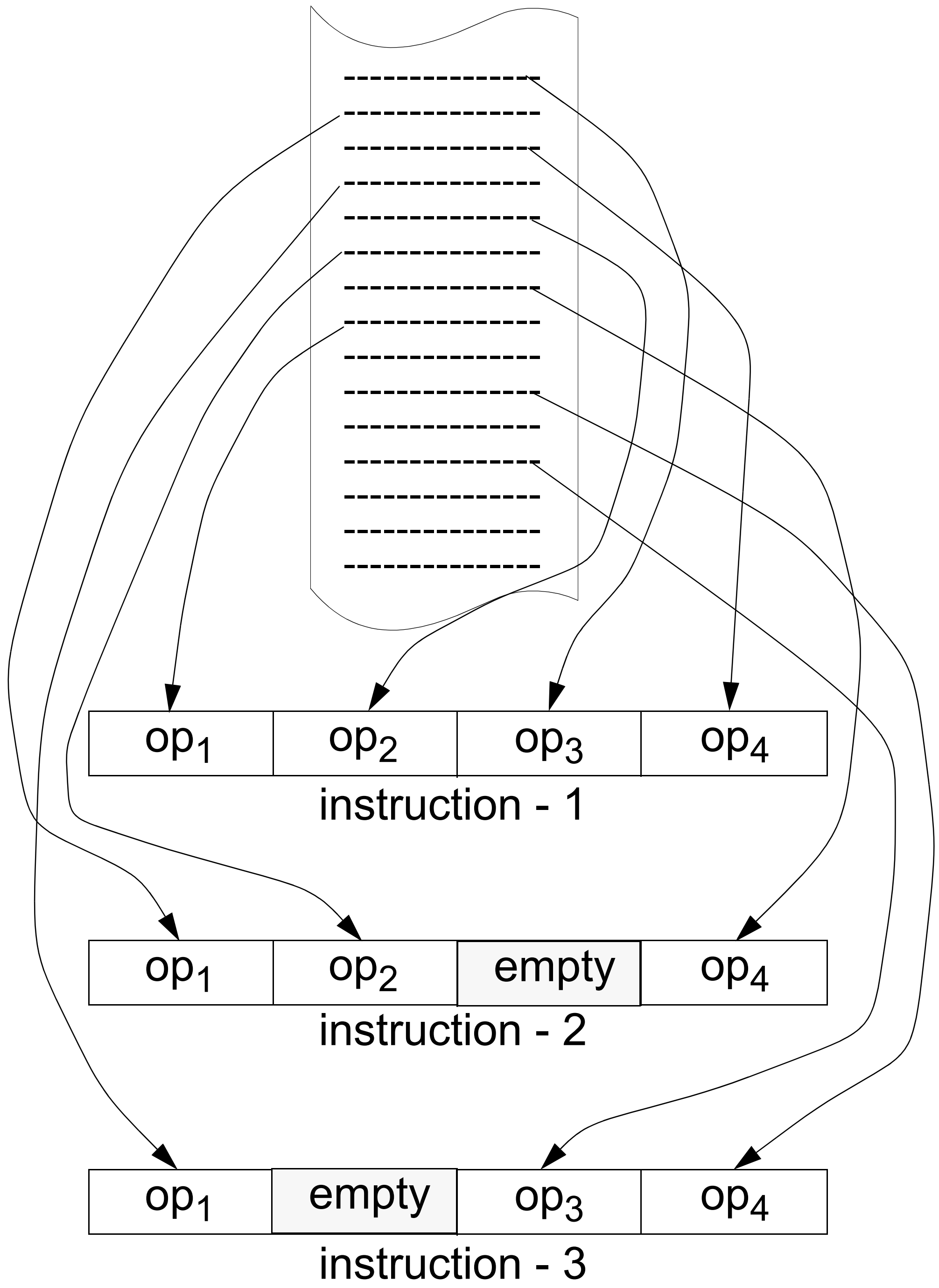
- **VLIW architectures rely on compile-time detection of parallelism $\Rightarrow$ the compiler analysis the program and detects operations to be executed in parallel; such operations are packed into one "large" instruction.**

- **At execution, after one instruction has been fetched all the corresponding operations are issued in parallel.**

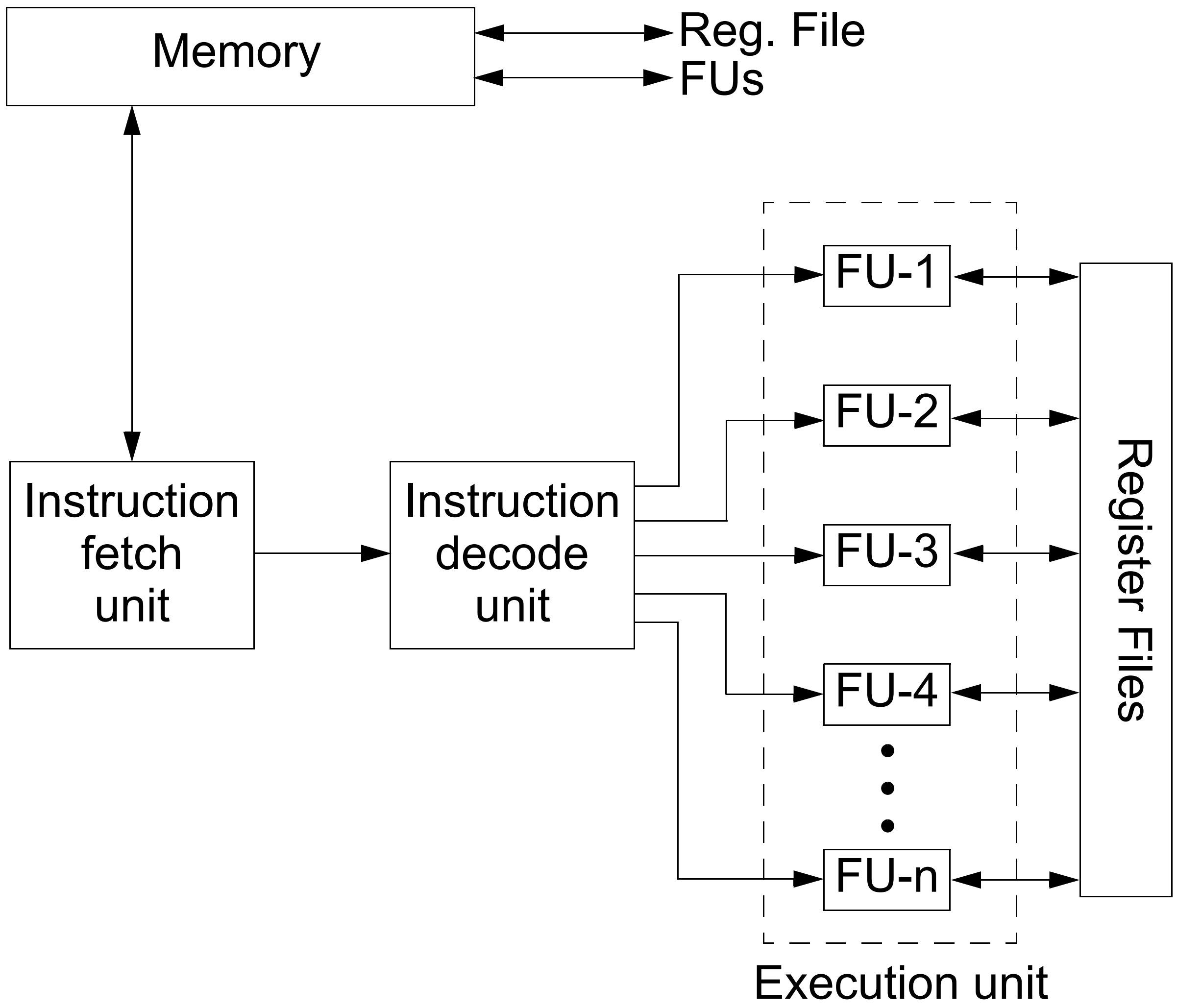
- **No hardware is needed for run-time detection of parallelism.**

- **The *instruction window* problem is solved: the compiler can potentially analyse the whole program in order to detect parallel operations.**

VLIW Processors



VLIW Processors



Advantages with VLIW Processors

- **Simpler hardware:**
  - **Does not need additional sophisticated hardware to detect parallelism, like in superscalars.**
  - **Power consumption is reduced, compared to superscalar.**

- **Good compilers can detect parallelism based on global analysis of the whole program (no instruction window problem).**

Problems with VLIW Processors

- 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.
- Incompatibility of binary cod
  - 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.

An Example

<u>Consider the following code in C</u>:

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

<u>Assumptions</u>: *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 ← x[i] ;(load double)
	ADF	F4,F0,F2	F4 ← F0 + F2 ;(floating pnt)
	STD	(R1),F4	x[i] ← F4 ;(store double)
	SBI	R1,R1,#8	R1 ← R1 - 8
	BGEZ	R1,Loop	

<u>Assumptions</u>: 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

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

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*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

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*6 = 2880 cycles

Loop Unrolling

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).

Loop Unrolling

<u>Let us unroll three iterations in our example</u>:

```
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

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*7 = 2240 cycles

Loop Unrolling

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;
}
```

Loop Unrolling

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*9 = 1080 cycles

Loop Unrolling

- 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.

- A good compiler has to find the optimal level of unrolling for each loop.

- Loop unrolling increases the memory space needed to store the program.

-

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

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

Example:

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

Trace Scheduling

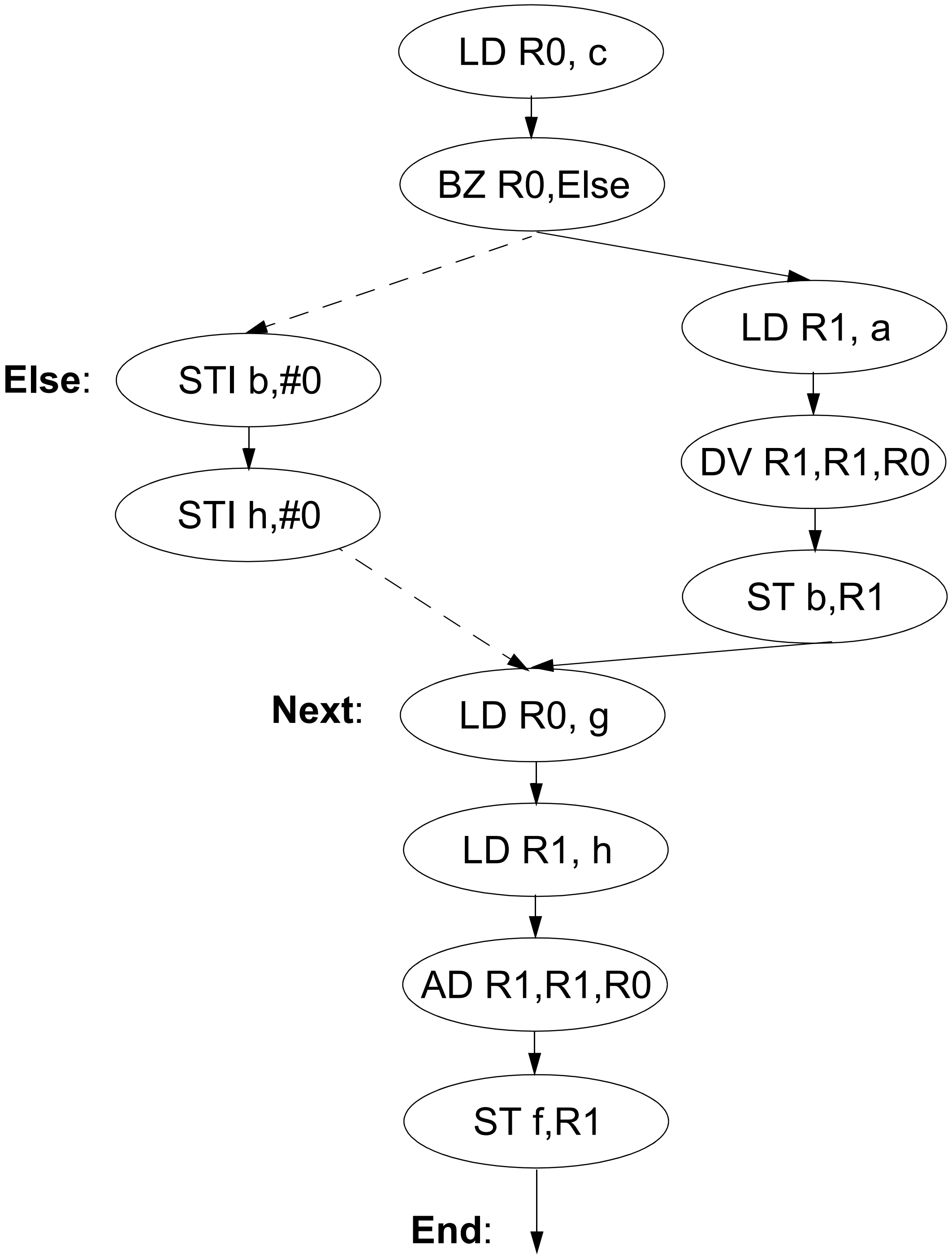
Example:

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

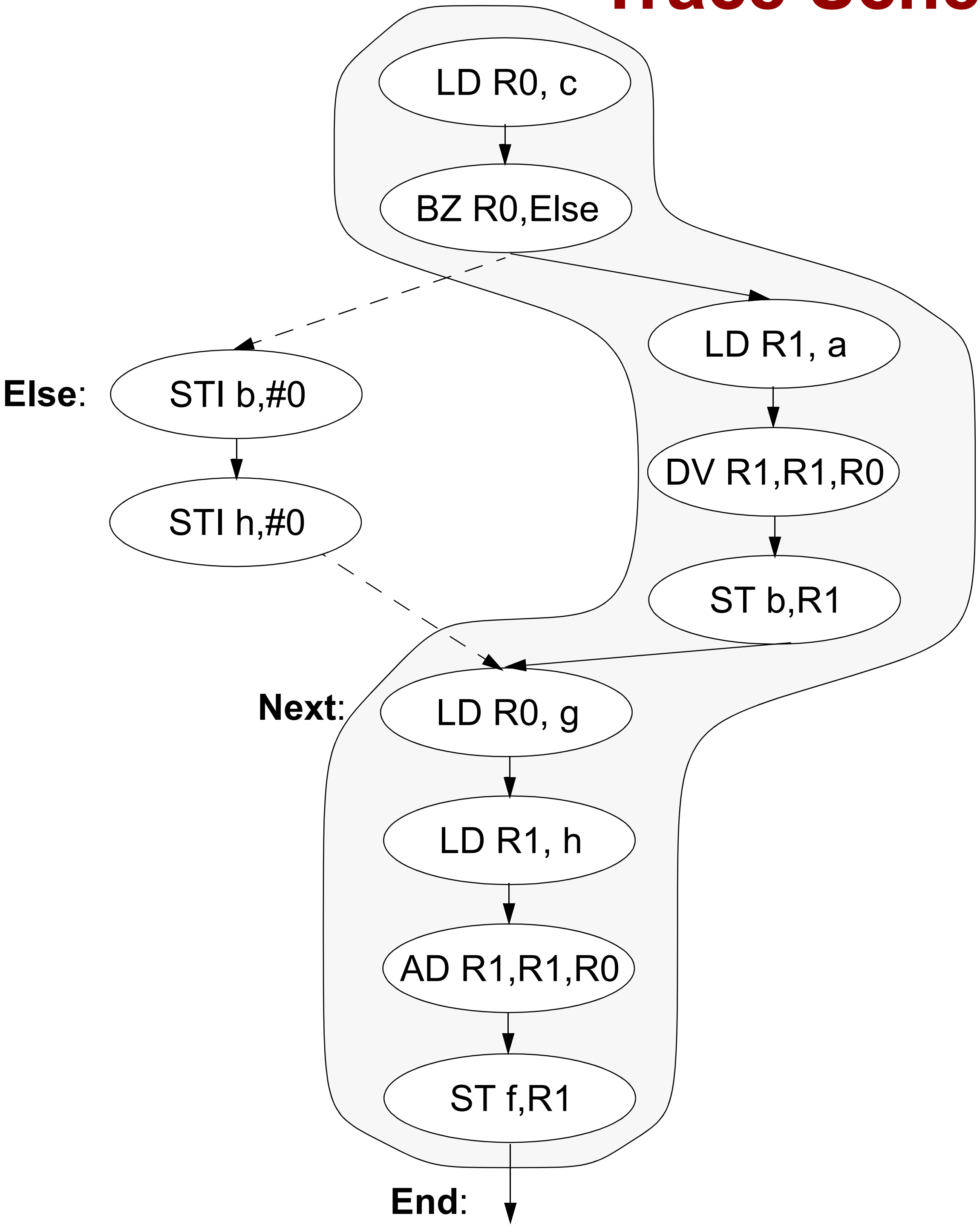
This (for an ordinary processor) would be compiled to:

```
        LD    R0, c         R0 ← c ;(load word)
        BZ    R0,Else
        LD    R1, a         R1 ← a ;(load integer)
        DV    R1,R1,R0      R1 ← R1 / R0 ;(divide 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 ;(add integer)
        ST    f,R1          f ← R1 ;(store word)
End:    - - - - - - - - - - - - - - -
```

Trace Scheduling



Trace Scheduling



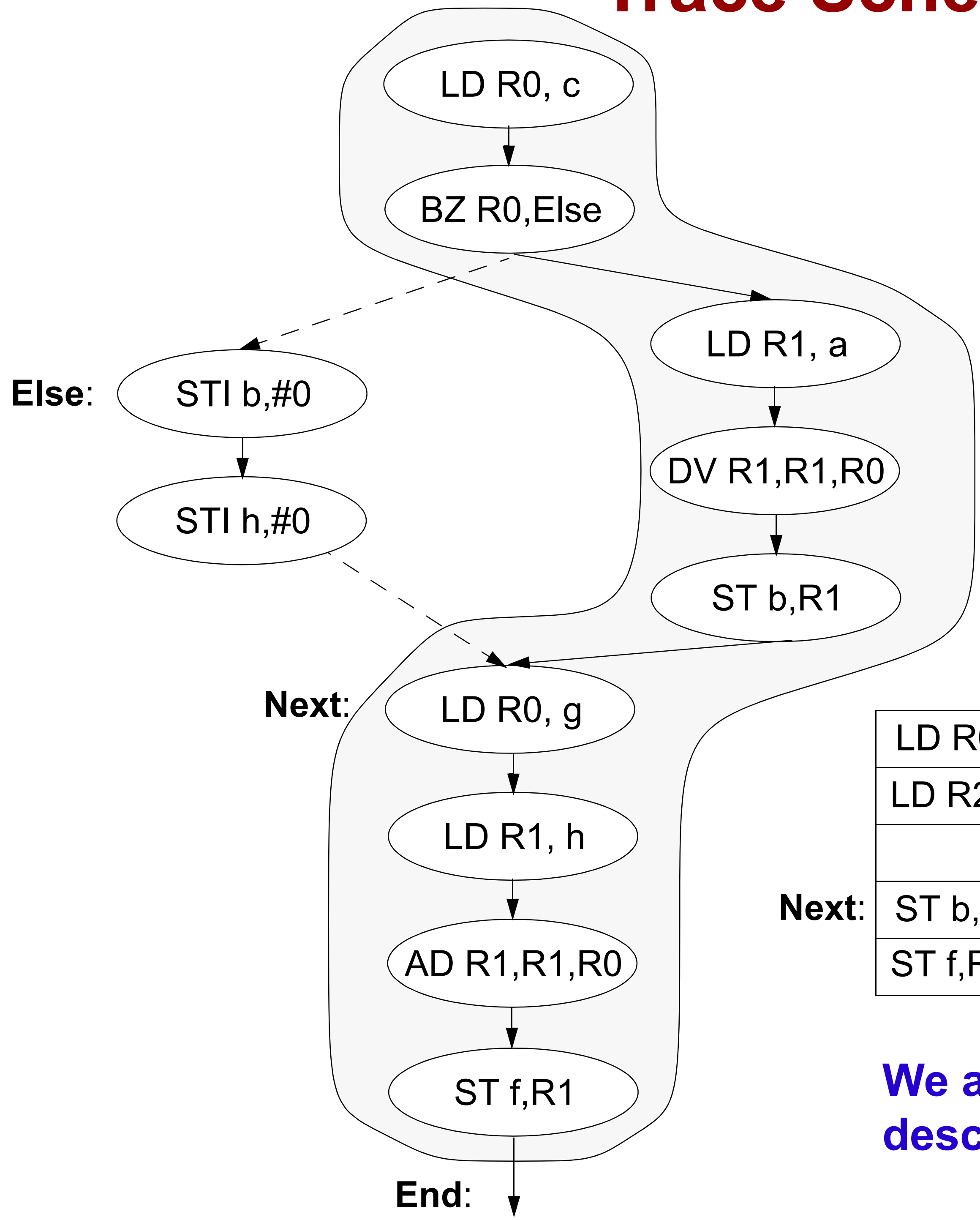
Trace selection:

- **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 prediction**
  - **The prediction can be based on profiling:**

    Execution of the program with several typical input sequences and collection of statistics concerning outcomes of conditional branches.

Trace Scheduling



<u>Instruction scheduling</u>:

- **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**

We assume the same processor as described earlier with Loop Unrolling.

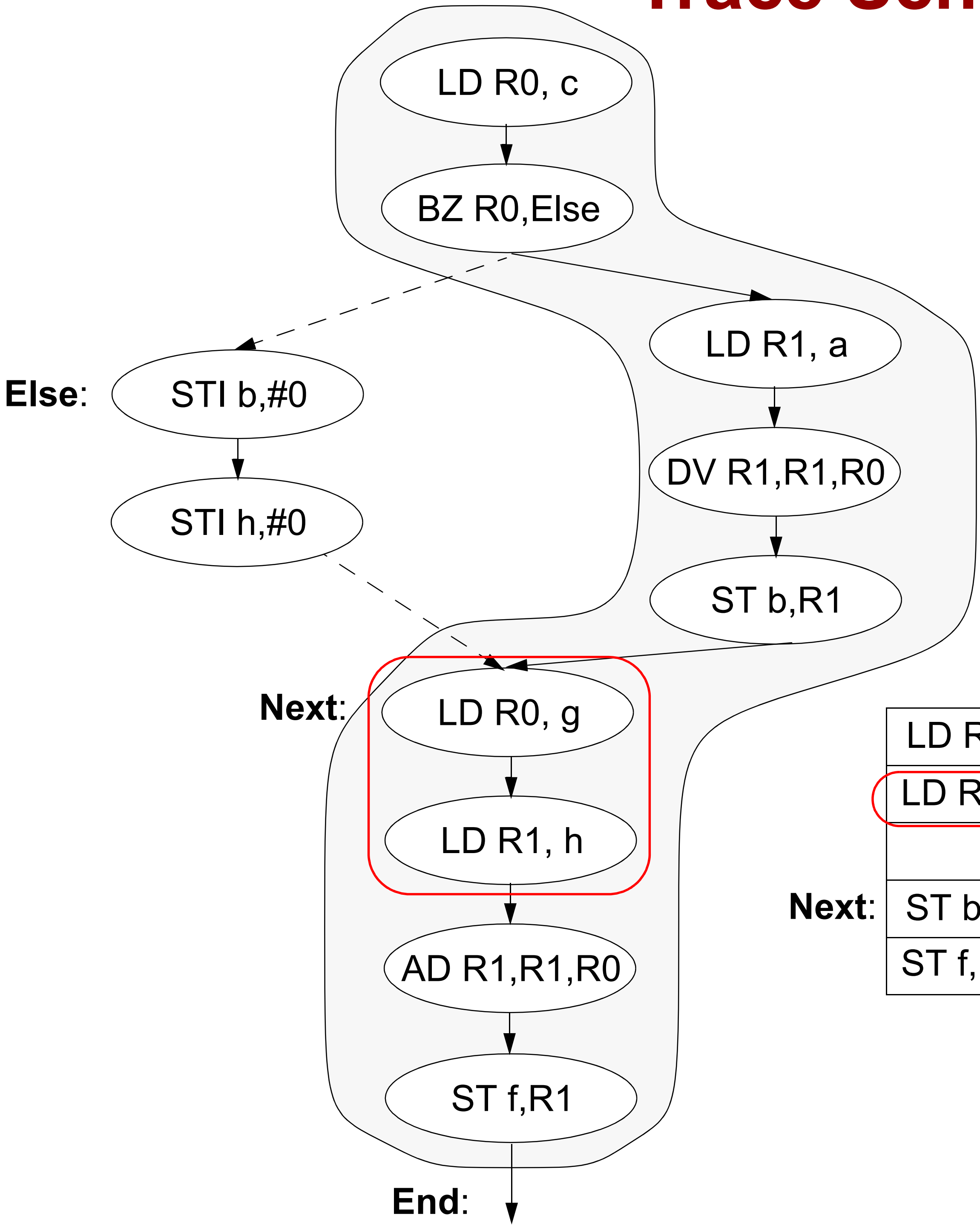
Trace Scheduling

Replacement and compensation:

- **The code for the entire sequence is produced by using the schedule generated for the selected trace.**
- **However: 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!

Trace Scheduling

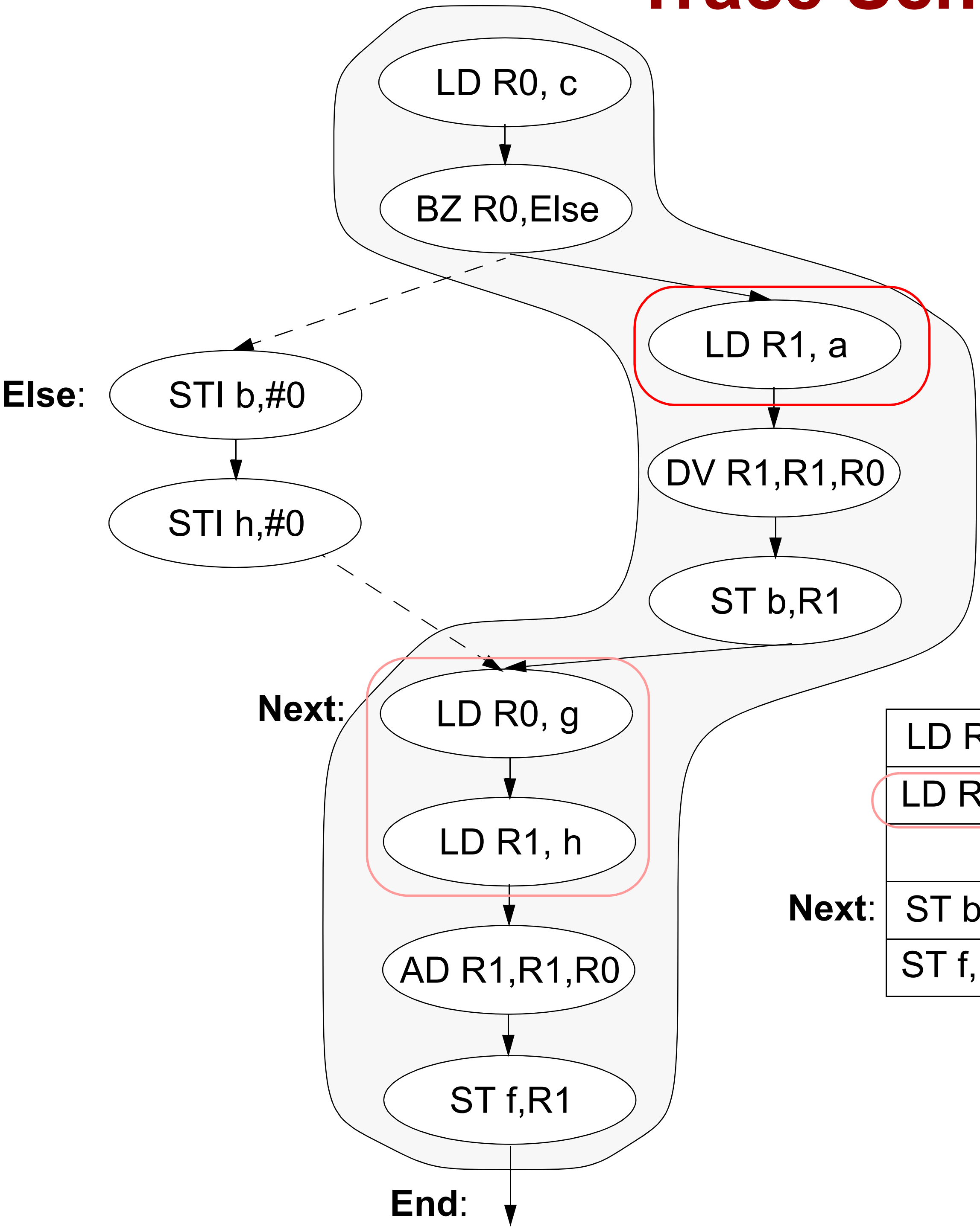


In the example:

- **the load of *g* and *h* is moved up, from the *next* sequence, before the conditional branch;**

	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**

Trace Scheduling

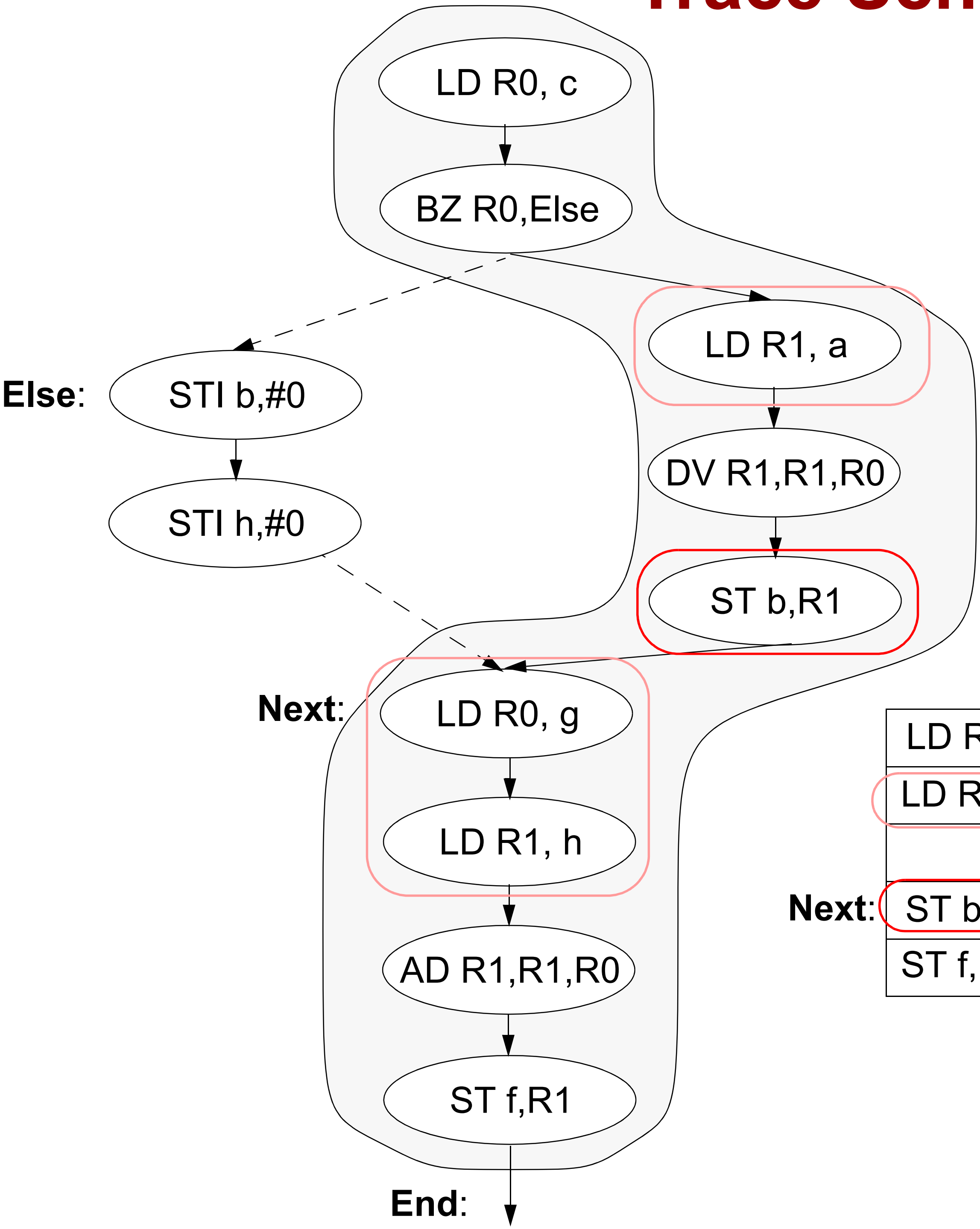


In the example:

- **the load of *g* and *h* is moved up, from the *next* sequence, before the conditional branch;**
- **the load of *a* is moved before the conditional branch;**

	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**

Trace Scheduling

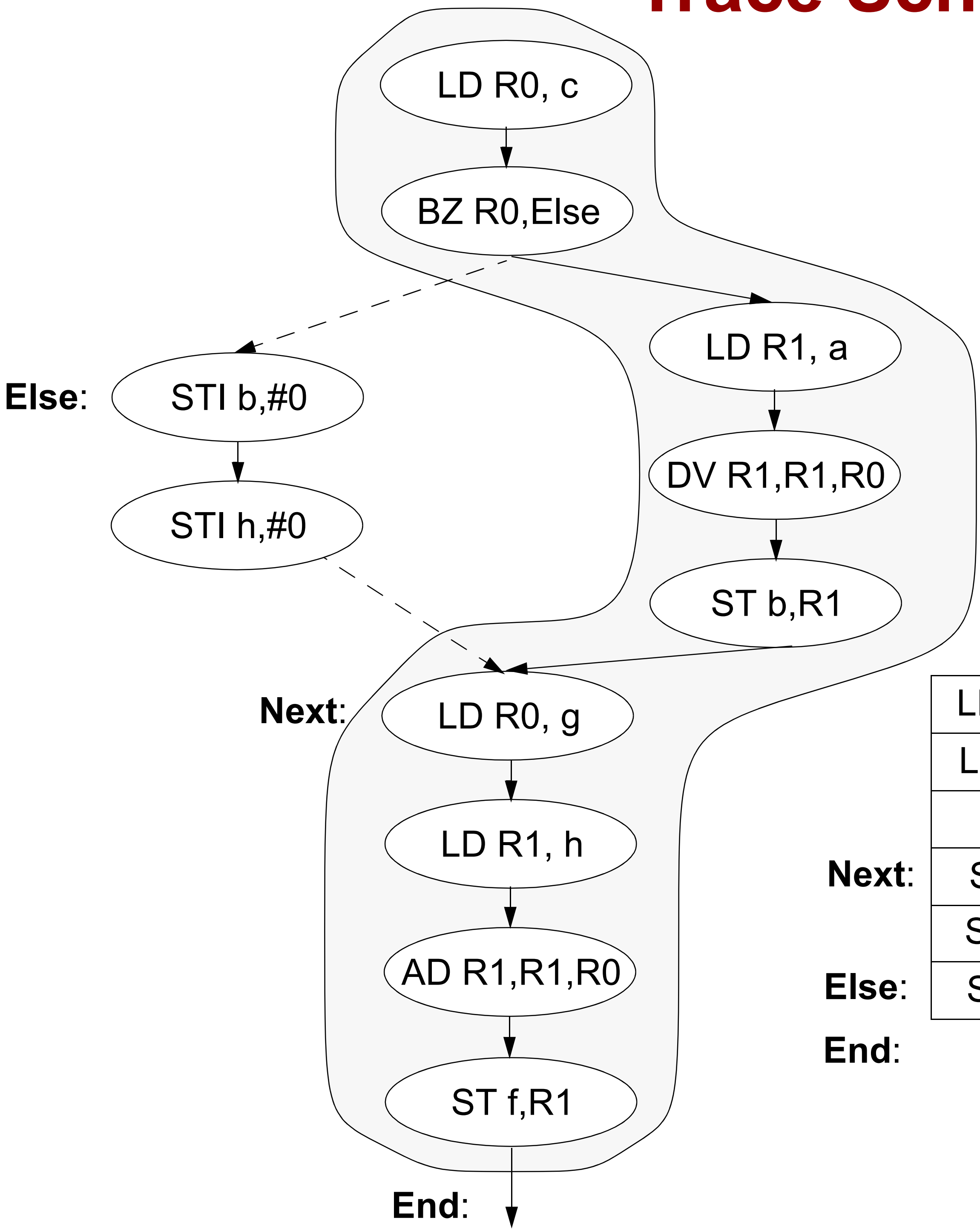


In the example:

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

Trace Scheduling

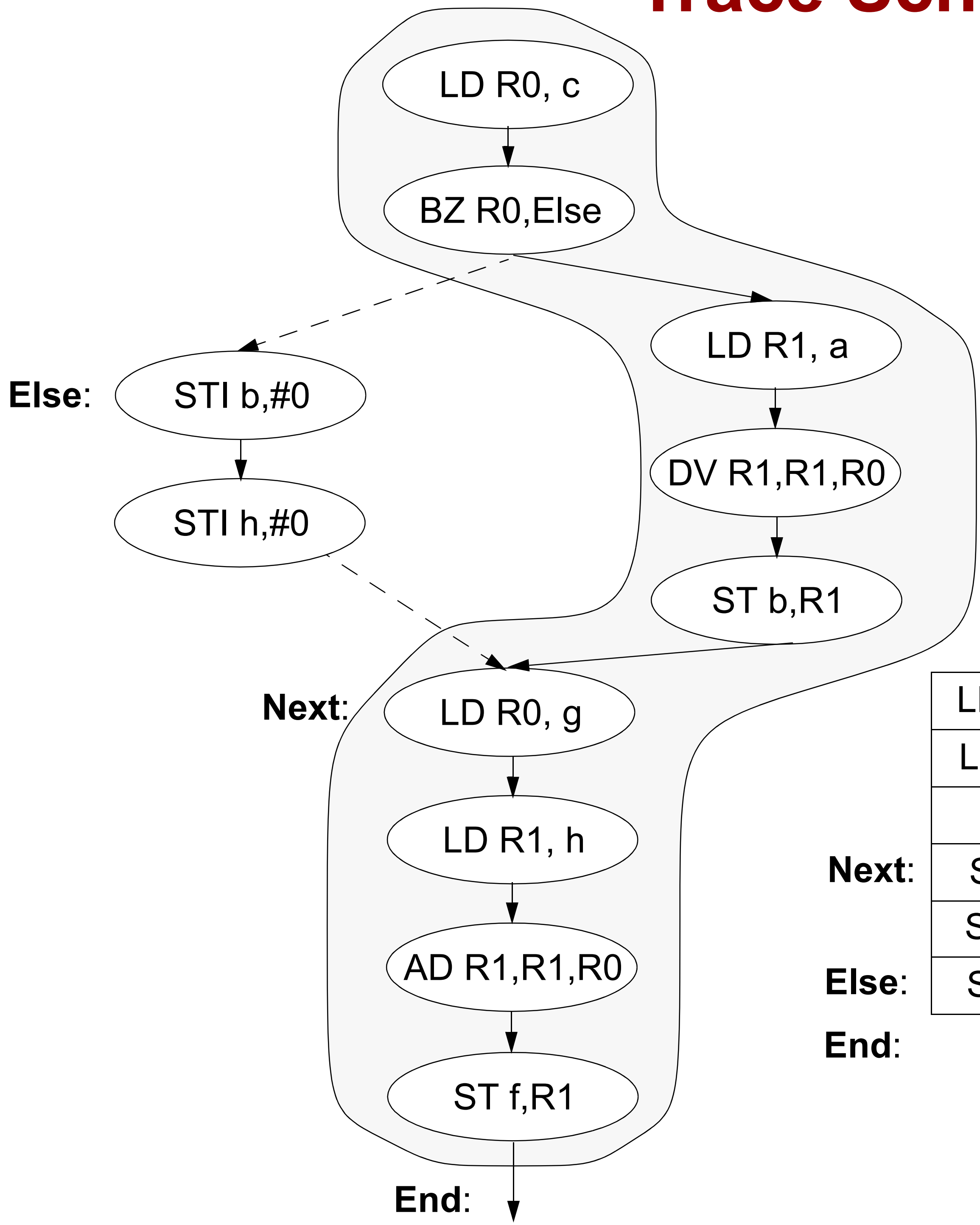


Simply merging the code for the two sequences does not work!

	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:

Trace Scheduling

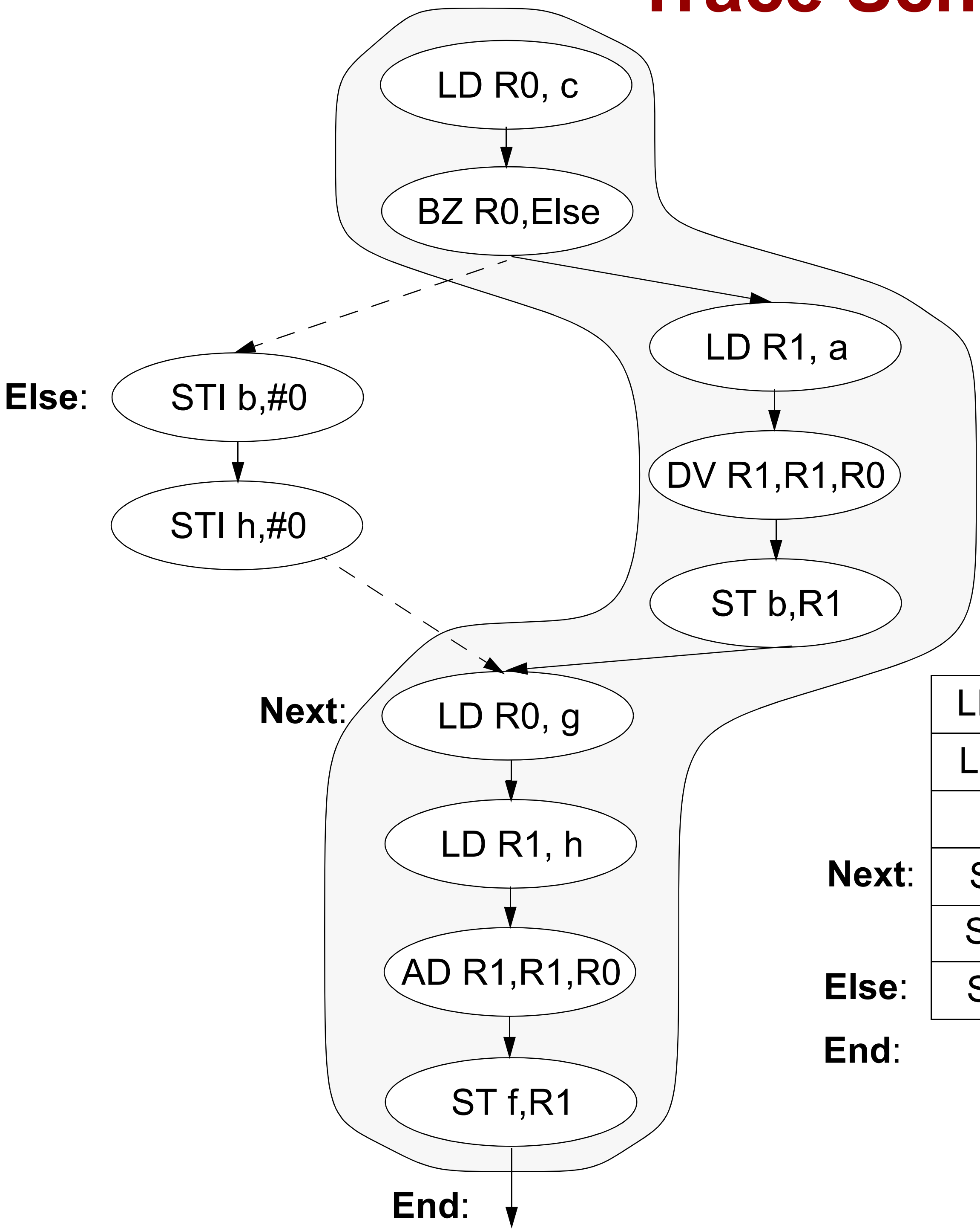


Simply merging the code for the two sequences does not work!

store **in the** ***next*** **sequence over-writes** ***STI*** **in** ***else*** **sequence (*****store*** **of** ***b*** **is moved down into the** ***next*** **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 b,#0	STI h,#0			BR **Next**
End:					

Trace Scheduling



Simply merging the code for the two sequences does not work!

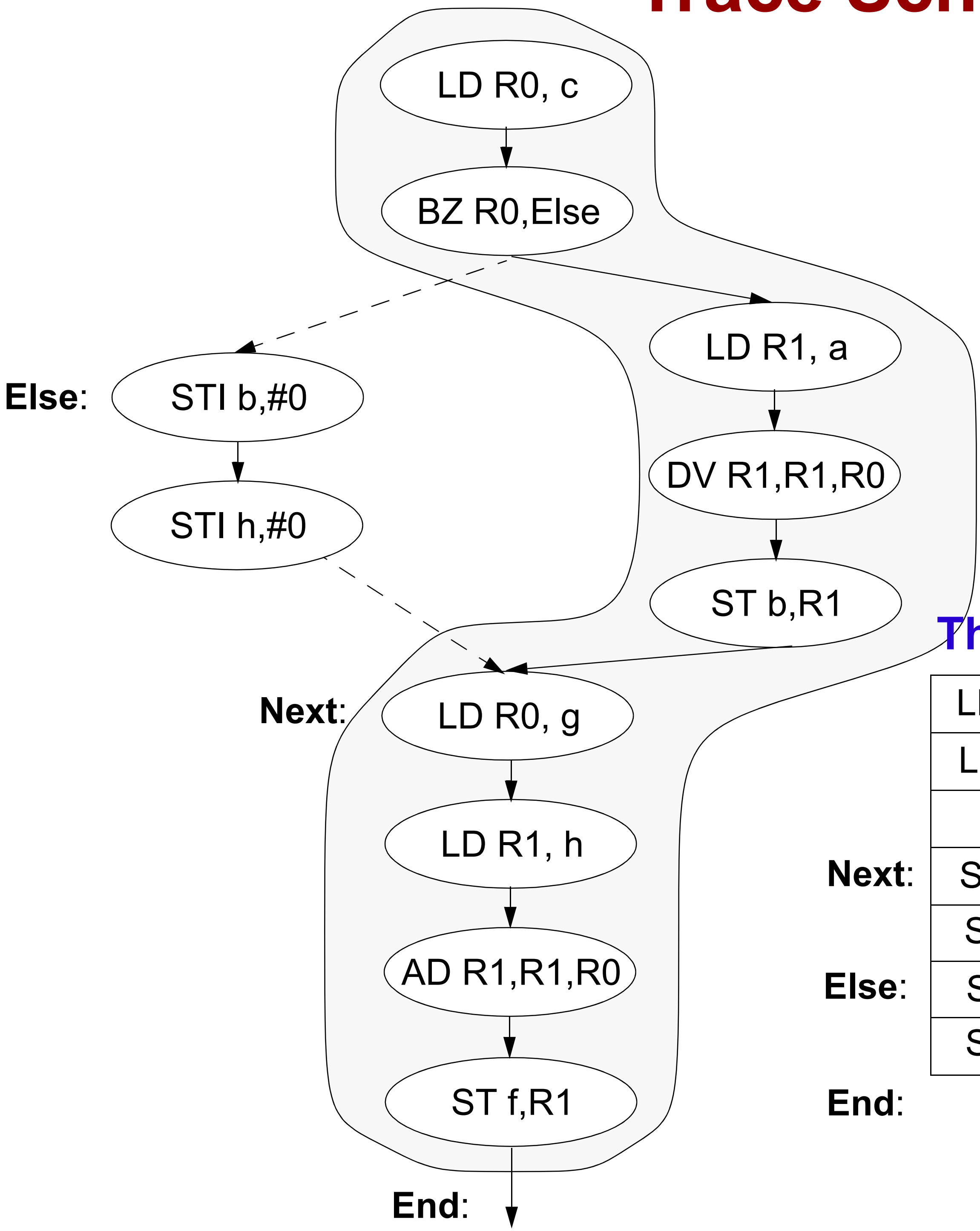
store **in the** ***next*** **sequence over-writes** ***STI*** **in** ***else*** **sequence (*****store*** **of** ***b*** **is moved down into the** ***next*** **sequence!).**

Value assigned to ***h*** **in the** ***else*** **se-quence is ignored for the addition (*****load*** **of** ***h*** **is moved up from the** ***next*** **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 b,#0	STI h,#0			BR **Next**
End:					

Compensation is needed!

Trace Scheduling

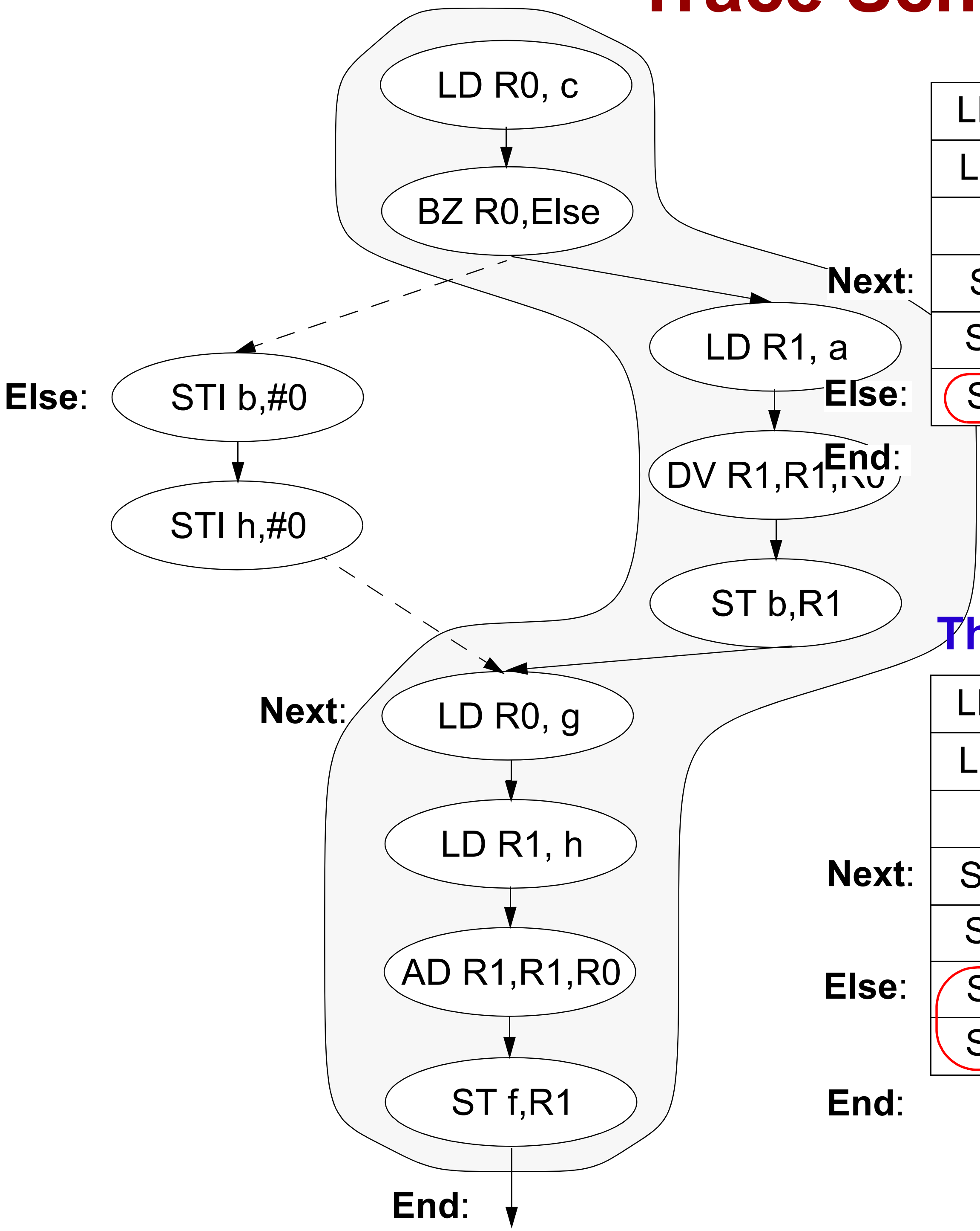


That's the correct code:

	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:

Trace Scheduling



	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:					

Compensation!

That's the correct code:

	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:					

Trace Scheduling

- ***Trace scheduling* is different from *speculative execution***:
  - **This is a *compiler optimization* (and not a run time technique!) and tries to optimize the code so that the path which is most likely to be taken, is executed as fast as possible.**

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

- **At program execution always the correct path will be taken (of course!); however, if this is not the one predicted *by the compiler*, execution will be slower because of the compensation code.**

- **Independently of trace scheduling, *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

<u>Examples of successful VLIW processors</u>:

- 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 a processor for servers and workstations.
    - The first product of the family was the *Itanium* processor.

The Itanium Architecture

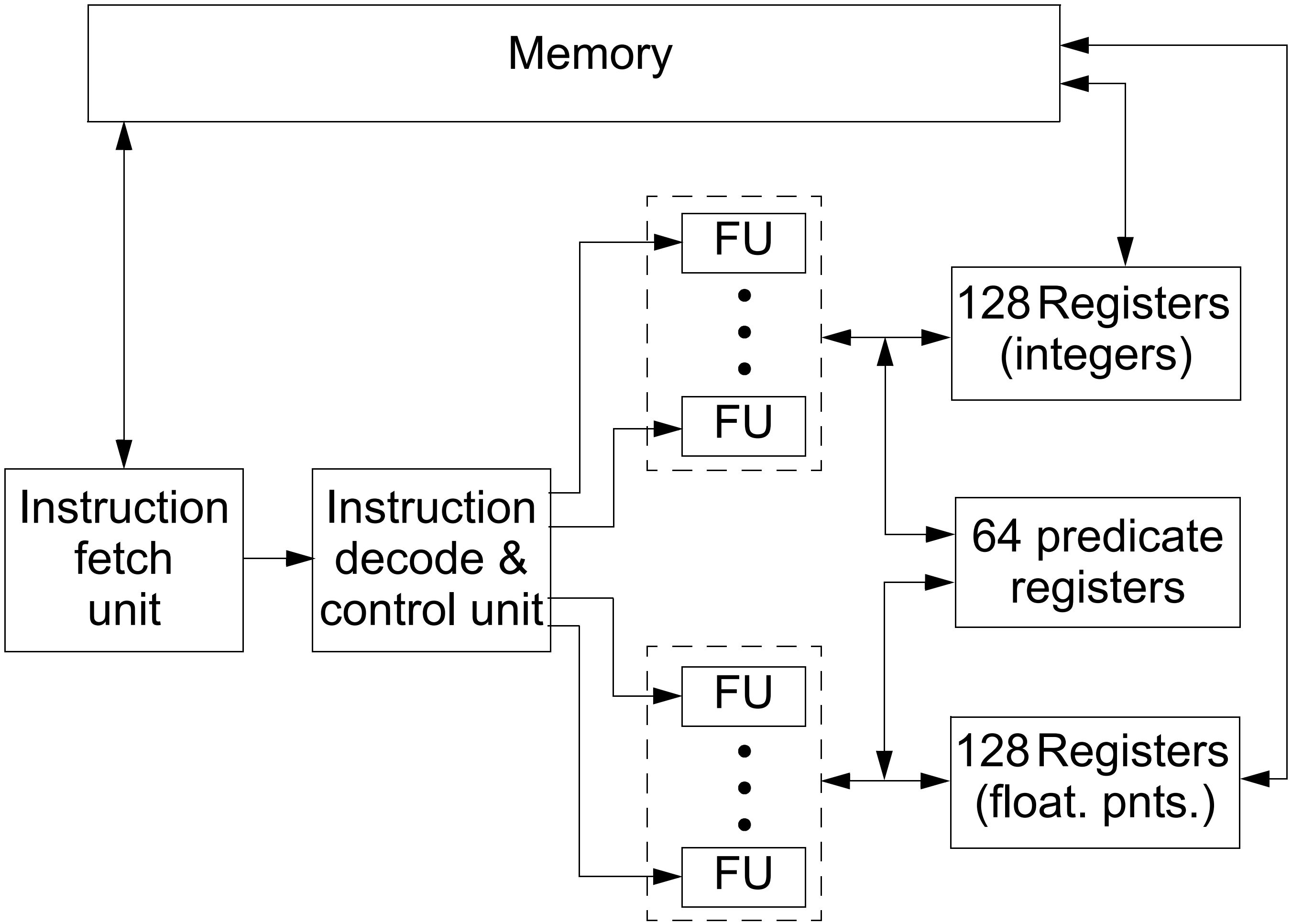
The Itanium is not a pure VLIW architecture, but many of its features are typical for VLIW processors.

<u>Particular features with Itanium</u>:

- **These are typical VLIW features:**
  - **Instruction-level parallelism fixed at compile-time.**
  - **(Very) long instruction word.**
- **Other interesting concepts:**
  - **Branch predication.**

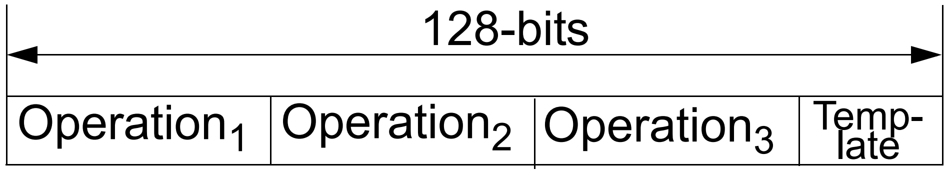
***Intel* calls the Itanium an *EPIC* (explicitly parallel instruction computing) processor: the parallelism of operations is explicit in the instruction word.**

General Organization



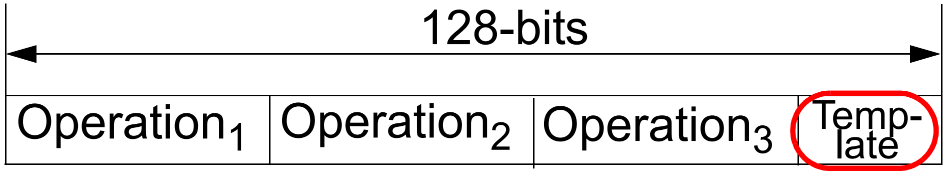
- Registers (both integer and floating point) are 64-bit.
- Predicate registers are 1-bit.
- 8 or more functional units.

Instruction Format



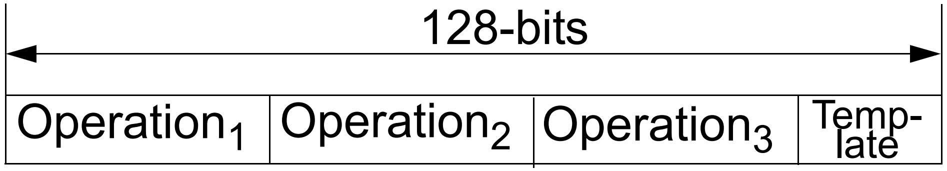
- **3 operations/instruction word (40 bits/operation)**
  - **This does not mean that max. 3 operations can be executed in parallel!**
  - **The three operations in the instruction are not necessarily parallel!**

Instruction Format



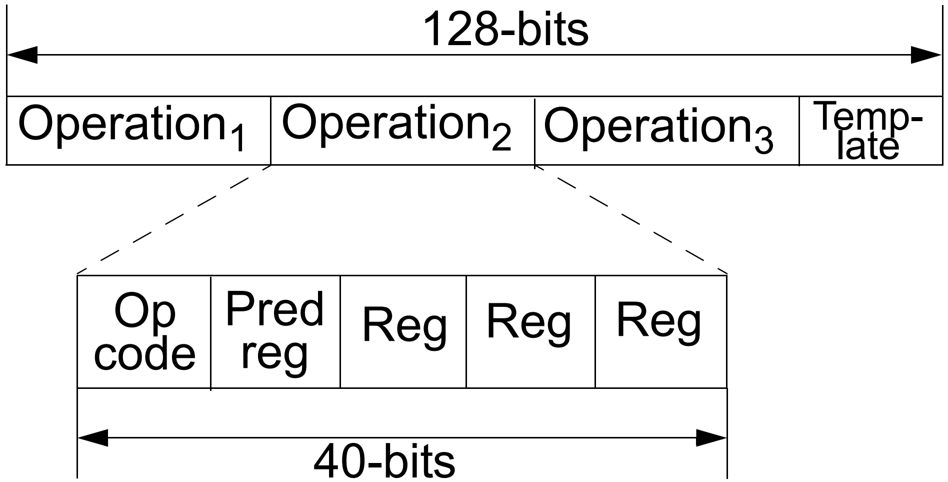
- **3 operations/instruction word (40 bits/operation)**
  - **This does not mean that max. 3 operations can be executed in parallel!**
  - **The three operations in the instruction are not necessarily parallel!**
- **The *template* (8bits) indicates what can be executed in parallel.**
  - **The encoding in the template shows which of the operations in the instruction can be executed in parallel.**
  - **The template connects also to neighbouring instructions $\Rightarrow$ operations from different instructions can be executed in parallel.**

Instruction Format

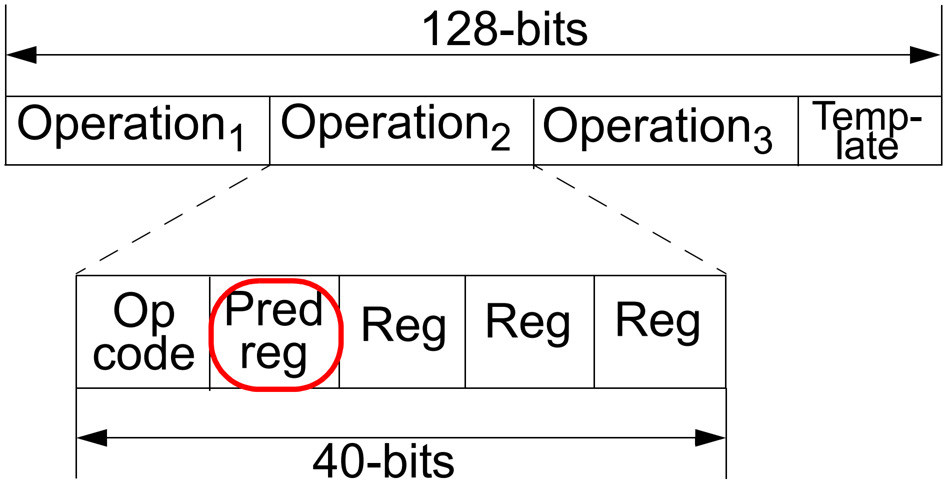


- 3 operations/instruction word (40 bits/operation)
  - This does not mean that max. 3 operations can be executed in parallel!
  - The three operations in the instruction are not necessarily parallel!
- The *template* (8bits) indicates what can be executed in parallel.
  - The encoding in the template shows which of the operations in the instruction can be executed in parallel.
  - The template connects also to neighbouring instructions $\Rightarrow$ operations from different instructions can be executed in parallel.
- 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 $\Rightarrow$ no places have to be left empty when no parallel operation is available.
  - The number of parallel operations is not restricted by the instruction size $\Rightarrow$ processor generations have different number of functional units without changing instruction format $\Rightarrow$ binary compatibility.
  - If, according to the template, there are more parallel operations than functional units available $\Rightarrow$ processor takes them sequentially.

Predicated Execution



Predicated Execution



- **Any operation can refer to a predicate register**

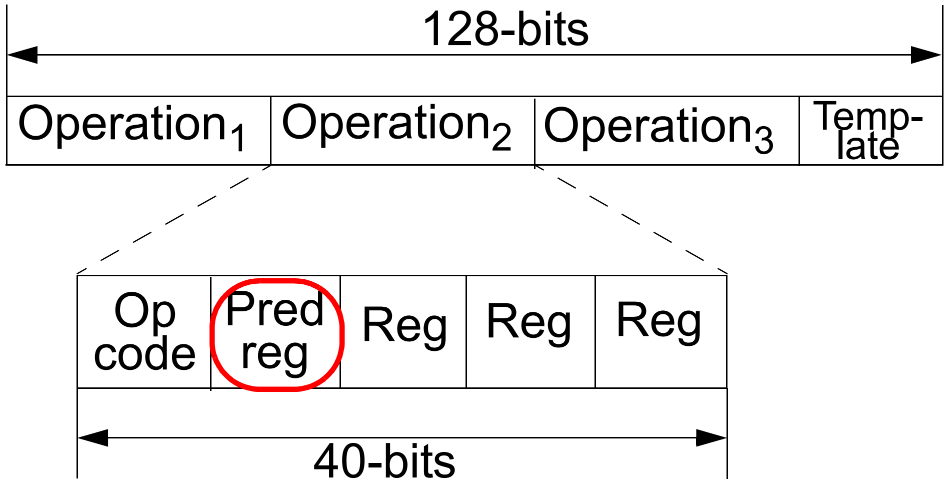
  <*Pi*> *operation* **i is number of a predicate register (between 0 and 63)**

  - **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.

<P3> ADI R2, R2,#1

Predicated Execution



- **Any operation can refer to a predicate register**

  <Pi> ***operation*** ***i*** **is 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

Predicate assignment

Pj, Pk = relation ***j*** **and** ***k*** **indicate predicate registers (between 0 and 63).**

- **Sets the value of predicate register** ***Pj*** **to** ***true*** **and that of predicate register** ***Pk*** **to** ***false*** **if the relation is evaluated to** ***true*****;** ***Pj*** **will be set to** ***false*** **and** ***Pk*** **to** ***true*** **if the relation evaluates to** ***false*****.**

P1, P2 = EQ(R0, #0)

Predicated predicate assignment

<Pi> Pj, Pk = relation ***i*****,** ***j*** **and** ***k*** **indicate predicate registers.**

- **Predicate registers** ***Pj*** **and** ***Pk*** **will be updated if and only if predicate register** ***Pi*** **is true.**

<P2> P1, P3 = EQ(R1, #0)

Branch Predication

- ***Branch predication*** **is a very aggressive *compilation technique* for generation of code with instruction level parallelism (code with parallel operations).**

- **Branch predication lets *operations from both branches of a conditional branch to be executed in parallel*.**

- **Branch predication is *based on the available hardware support*: instructions for *predicated execution* provided by the Itanium 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 instructions are committed which correspond to the right branch.

Branch Predication

Branch *predication* is not branch *prediction*:

- **Branch prediction:**

  Guess which branch is taken **and then go along that one; if the guess was wrong, 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, all others are discarded.**

There is no lost time with failed predictions.

Branch Predication

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

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.**

Let us read it in this way:

if not(a == 0) and not(b == 0)	ADI	R2, R2,#1
if not(not(a == 0) and not(b == 0)) and not(c == 0)	ADI	R4, R4,#1
if not(not(a == 0) and not(b == 0)) and not(not(c == 0))	SBI	R4, R4,#1
if not(not(a == 0) and not(b == 0))	MPI	R5, R4,#5
	ADI	R6, R6,#1

Branch Predication

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.

With predicated execution:

```
(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
```

Let us read it in this way:

if not(a == 0) and not(b == 0)	ADI	R2, R2,#1
if not(not(a == 0) and not(b == 0)) and not(c == 0)	ADI	R4, R4,#1
if not(not(a == 0) and not(b == 0)) and not(not(c == 0))	SBI	R4, R4,#1
if not(not(a == 0) and not(b == 0))	MPI	R5, R4,#5
	ADI	R6, R6,#1

Branch Predication

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.

With predicated execution:

```
(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) with (7) and (6) with (7) 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.