Course on Advanced Computer Architectures

VLIW Code Scheduling

POLITECNICO
MILANO 1863

Prof. Cristina Silvano, email: cristina.silvano@polimi.it

POLITECNICO
MILANO 1863

Code Scheduling

- **Main goal:** Statically reordering instructions in object code so that they are executed in a minimum amount of time and semantically correct order.
  - Execute time-critical operations efficiently.
  - Try to increase the number of independent instructions fetched.

  $\Rightarrow$ Minimize execution time.

POLITECNICO
MILANO 1863

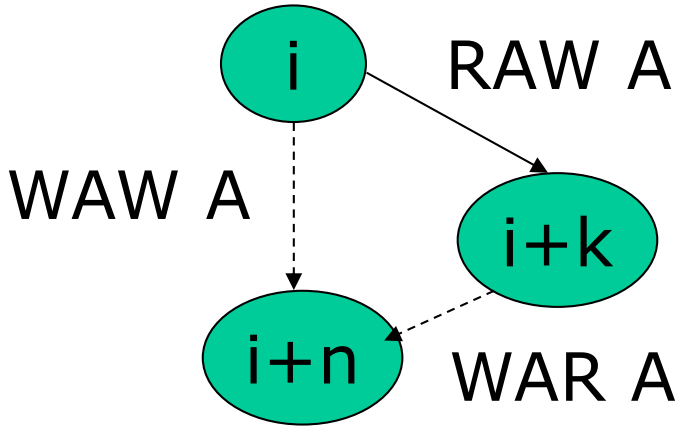
Scheduling Basics

- Decompose the code in **basic blocks.**
- A **basic block** is a code sequence with no branches in/out except to the entry/exit point.
- The code in a basic block has:
  - **One single entry point:** no code line within the basic block is the destination of a branch/jump anywhere in the program.
  - **One single exit point:** only the last instruction can cause the program to begin executing code in a different basic block.

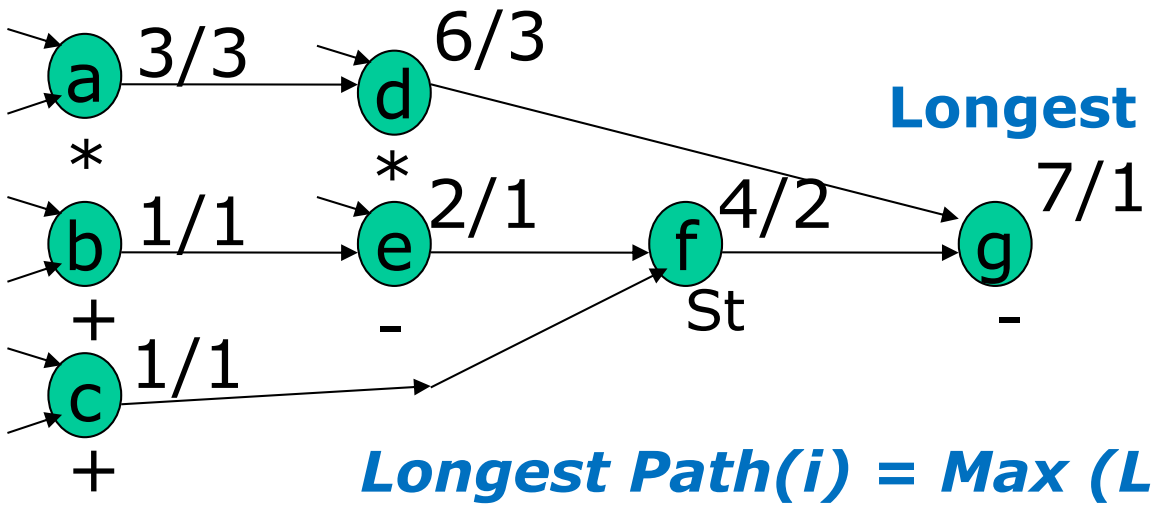
Dependence Graph

- A **dependence graph** captures true, anti and output dependencies between instructions.
- Anti and output dependencies are name dependences due to variables/registers reuse.

```
i)     A=B+1
.....
i+k) X=A+C
.....
i+n) A=E+C
```



Dependence Graph: an example



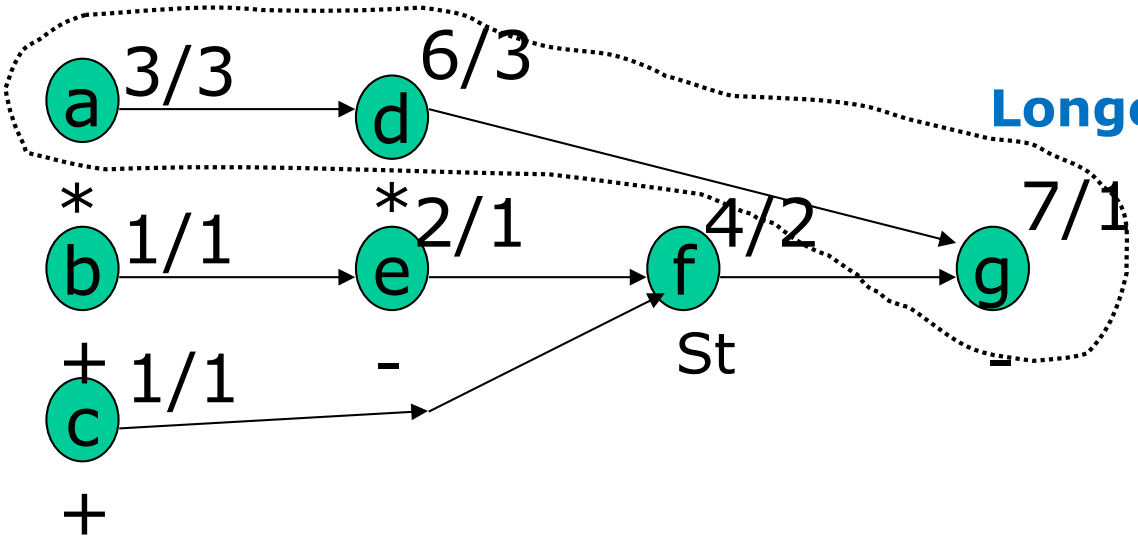
Longest Path/Latency

Longest Path(i) = Max (LP(Pred(i))) + Latency(i)

- Each node represents an **operation** in the basic block.
- There is an edge from a node *i* to another node *j* in the graph if the results of the operation *i* are used by the operation *j:*
  node i is a direct predecessor of node j.
- Each node is annotated with: the ***longest path*** to reach it / the node **latency**.

Longest Critical Path

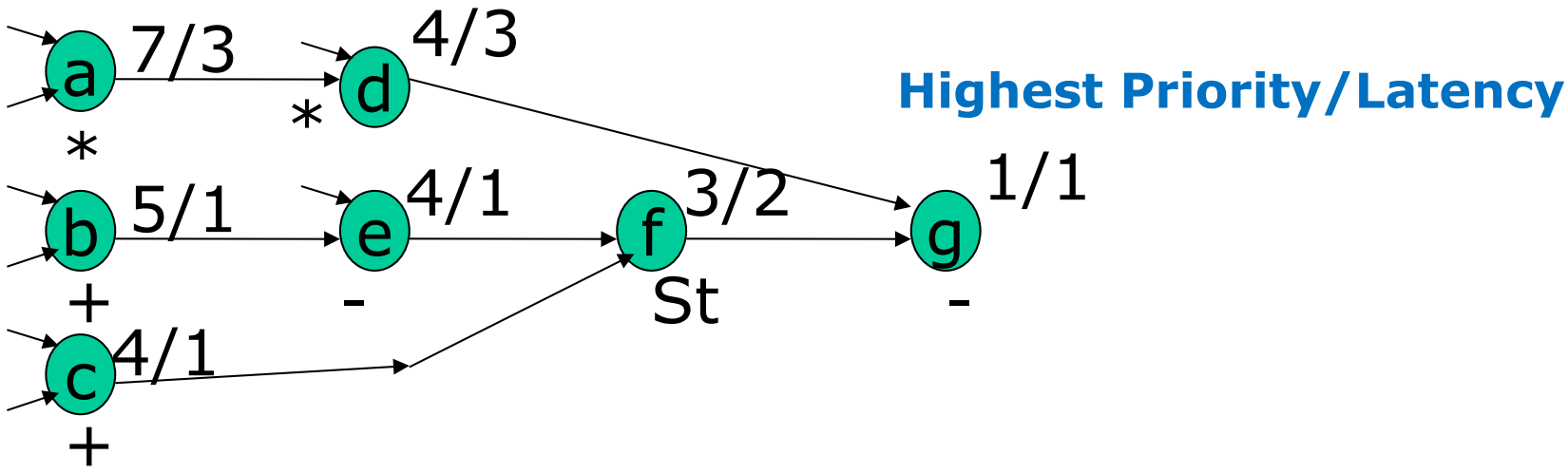
- It is the longest path in the dependence graph.
- It defines the minimum execution time of a basic block.



Longest Path(i) = Max (LP(Pred(i))) + Latency(i)

Longest Critical Path = Max (LP(i)) = 7

Dependence Graph



Highest Priority/Latency

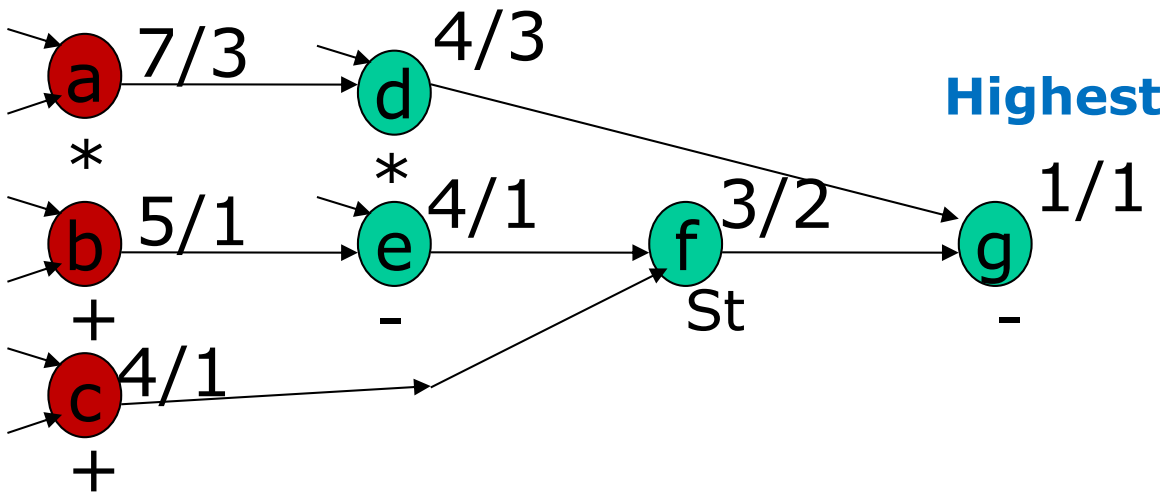
- Each node is annotated with its ***highest priority*** (longest path to the end or "sink" node of the graph) and the node **latency.**

POLITECNICO
MILANO 1863

Scheduling Basics

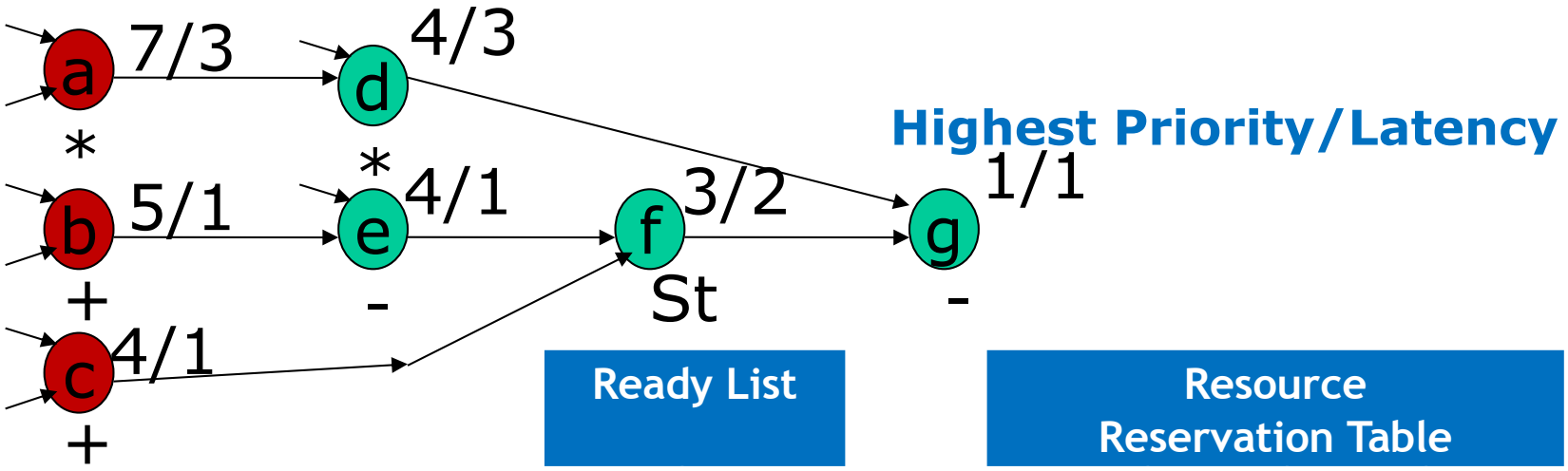
- The scheduling problem consists of assigning the cycle of execution to each operation in the graph.
- The simplest type of scheduling occurs when we want to execute the code with the minimum amount of time, given the latency of each node and we don't care about the number of resources required.
- **Ready List:** List of nodes whose predecessors are all scheduled and their operands are ready.
- **ASAP algorithm:** Simply assigning each node in the Ready List as soon as possible without any constraint on the resources.

Example: Nodes without predecessors



Highest Priority/Latency

Example: ASAP Scheduling Algorithm

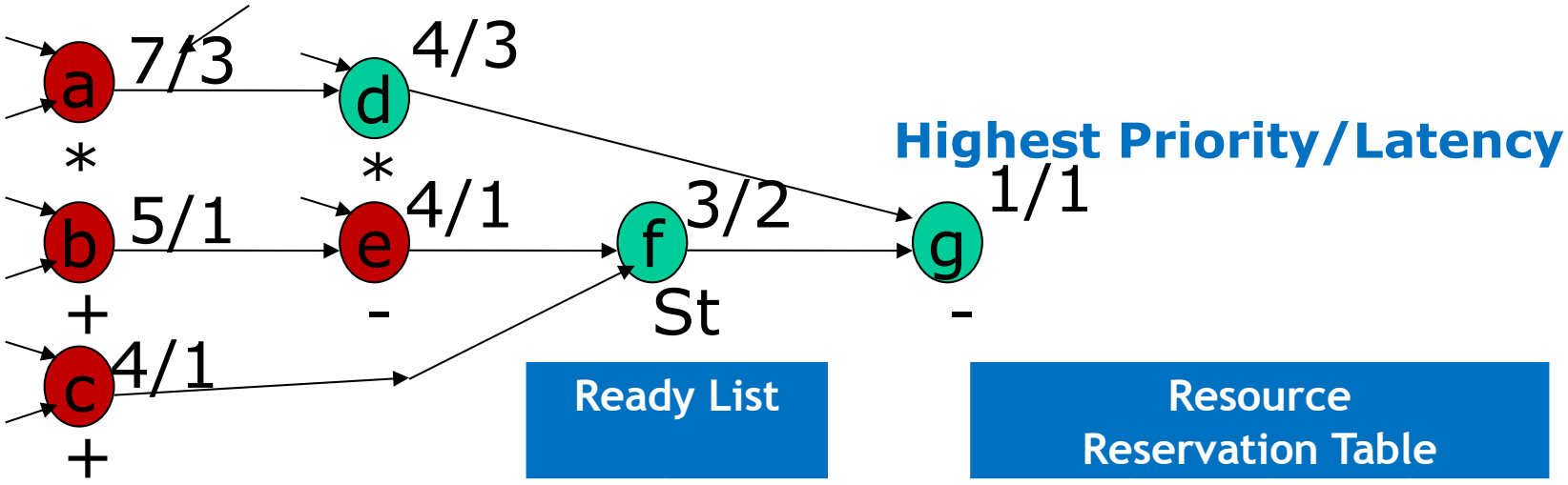


Highest Priority/Latency

Ready List	
1	a,b,c
2	
3	
4	
5	
6	
7	

Resource Reservation Table			
ALU1	ALU2	L/S	MUL
b	c		a
			a
			a

Example: ASAP Scheduling Algorithm

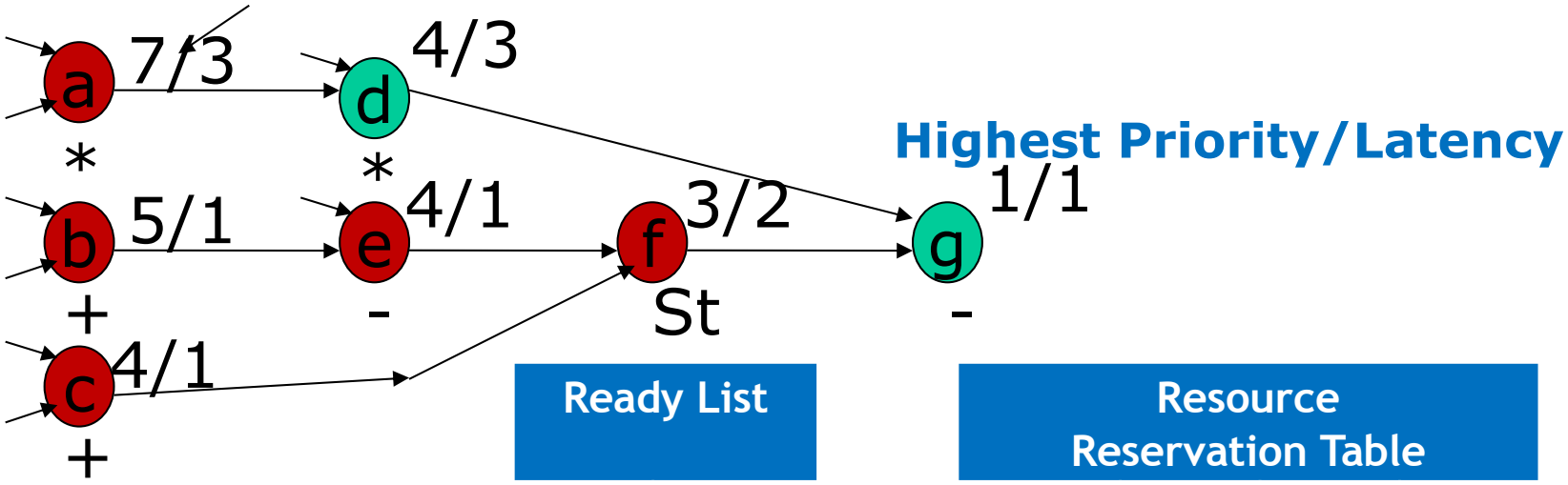


Highest Priority/Latency

Ready List	
1	a,b,c
2	e
3	
4	
5	
6	
7	

Resource Reservation Table			
ALU1	ALU2	L/S	MUL
b	c		a
e			a
			a

Example: ASAP Scheduling Algorithm

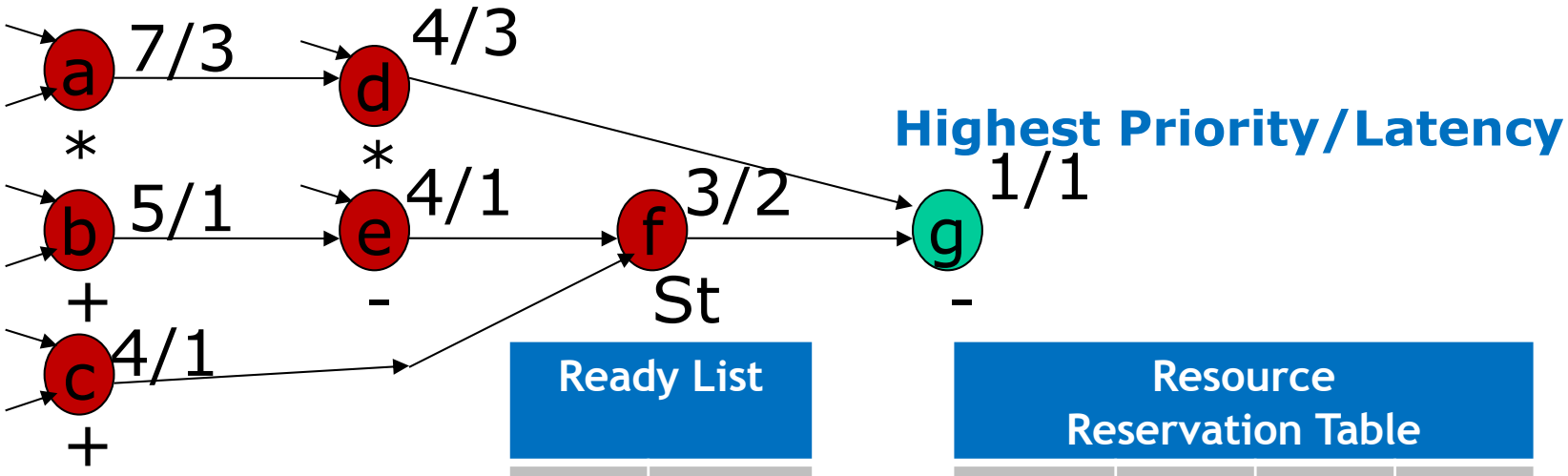


Highest Priority/Latency

Ready List	
1	a,b,c
2	e
3	f
4	
5	
6	
7	

Resource Reservation Table			
ALU1	ALU2	L/S	MUL
b	c		a
e			a
		f	a
		f	

Example: ASAP Scheduling Algorithm

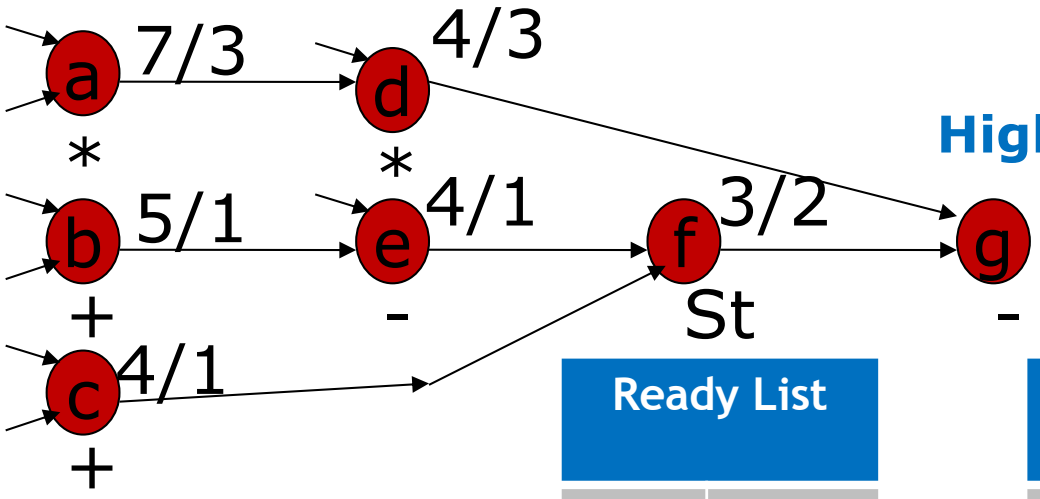


Highest Priority/Latency

Ready List	
1	a,b,c
2	e
3	f
4	d
5	
6	
7	

Resource Reservation Table			
ALU1	ALU2	L/S	MUL
b	c		a
e			a
		f	a
		f	d
			d
			d

Example: ASAP Scheduling Algorithm



Highest Priority/Latency

Ready List	
1	a,b,c
2	e
3	f
4	d
5	
6	
7	g

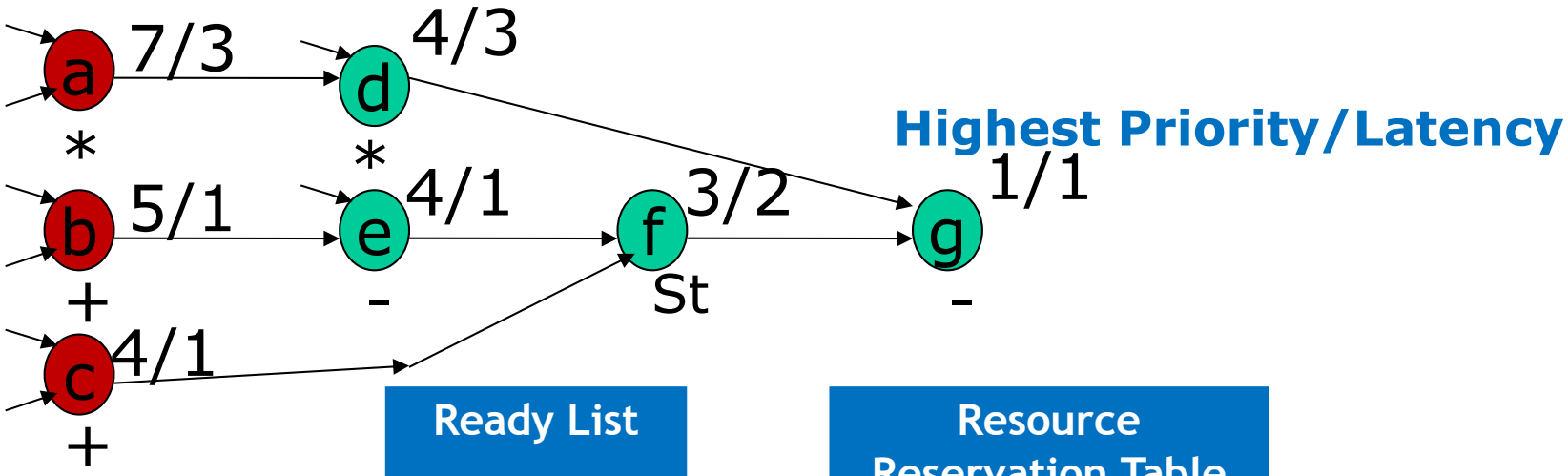
Resource Reservation Table			
ALU1	ALU2	L/S	MUL
b	c		a
e			a
		f	a
		f	d
			d
			d
g			

POLITECNICO
MILANO 1863

List-based Scheduling Algorithm

- This is a **resource-constrained** scheduling algorithm.
- Before scheduling begins, operations on top of the graph are inserted in the ***ready set.***
  - An *operation* is in the ***ready set*** if all of its predecessors have been scheduled and if the operands are ready.
- Starting from the first cycle, for each cycle try to schedule operations (nodes) from the ***ready set*** that can fill the available resource slots.
- When more nodes are in the ready set, select the node with the ***highest priority*** (longest path to the end of the graph).

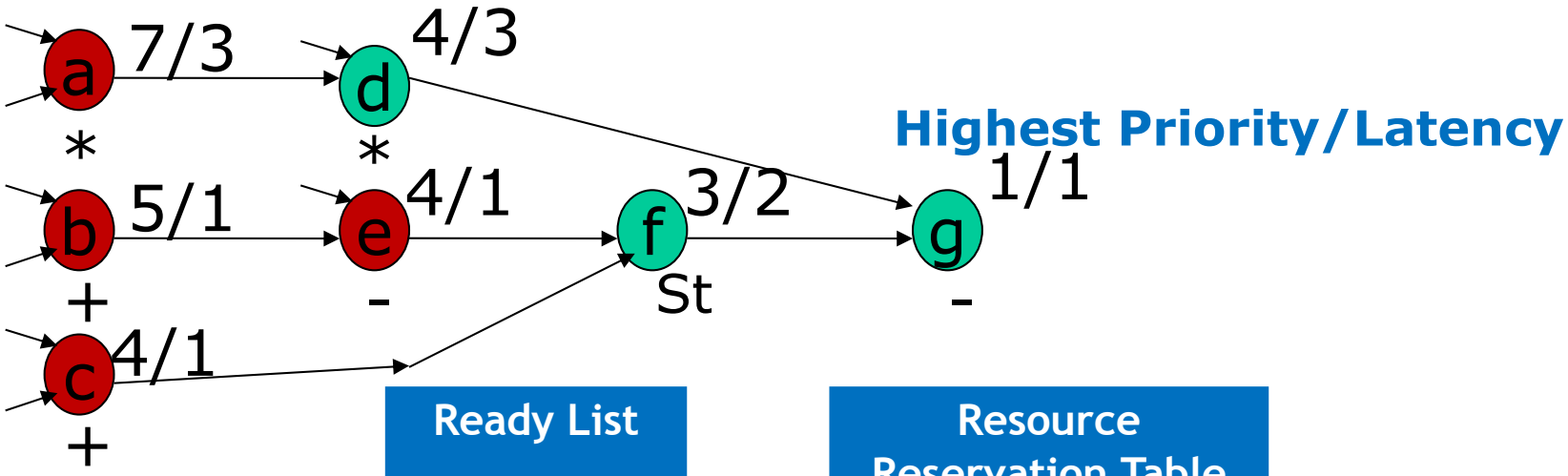
Example: List-based scheduling



Ready List	
1	a,b,c
2	
3	
4	
5	
6	
7	

Resource Reservation Table		
ALU1	L/S	MUL
b		a
		a
		a

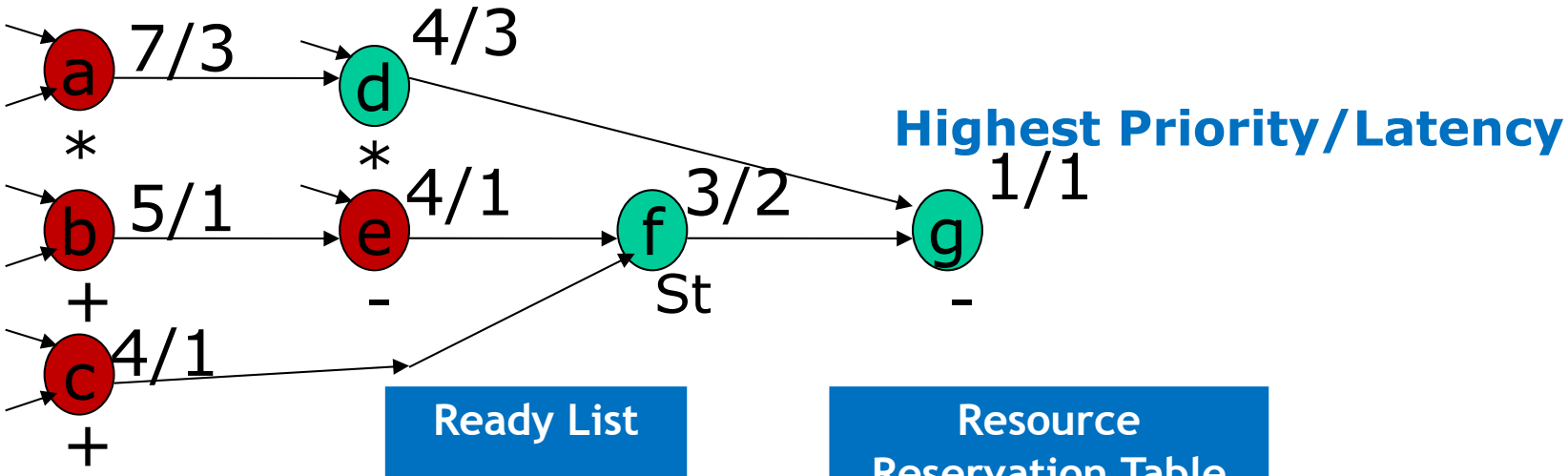
Example: List-based scheduling



Ready List	
1	a,b,c
2	c,e
3	
4	
5	
6	
7	

Resource Reservation Table		
ALU1	L/S	MUL
b		a
c		a
		a

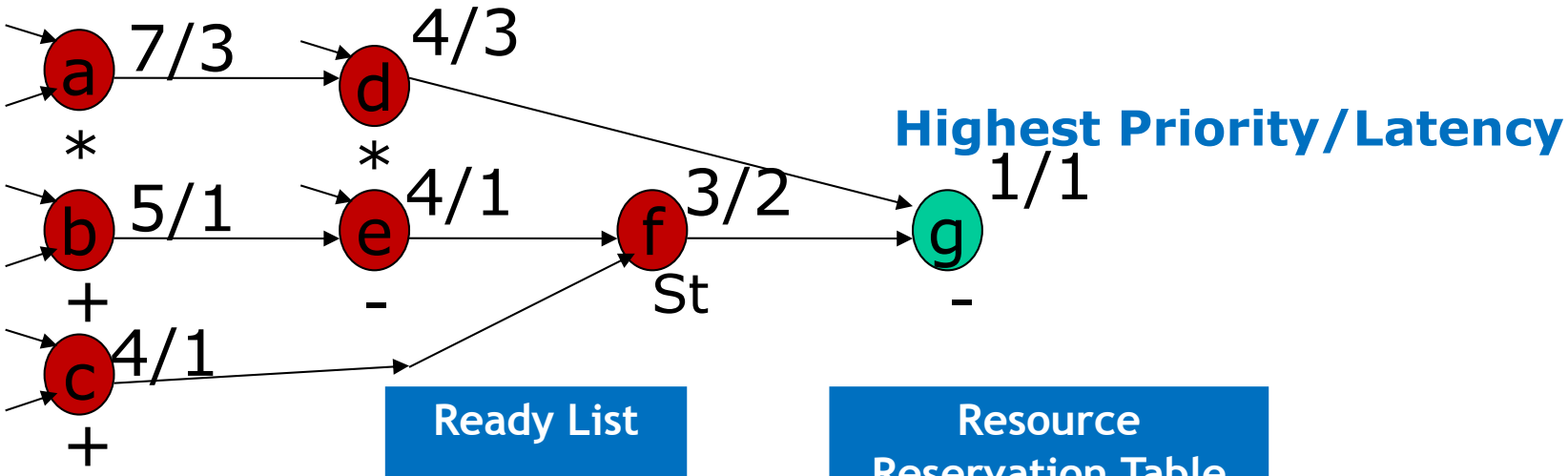
Example: List-based scheduling



Ready List	
1	a,b,c
2	c,e
3	e
4	
5	
6	
7	

Resource Reservation Table		
ALU1	L/S	MUL
b		a
c		a
e		a

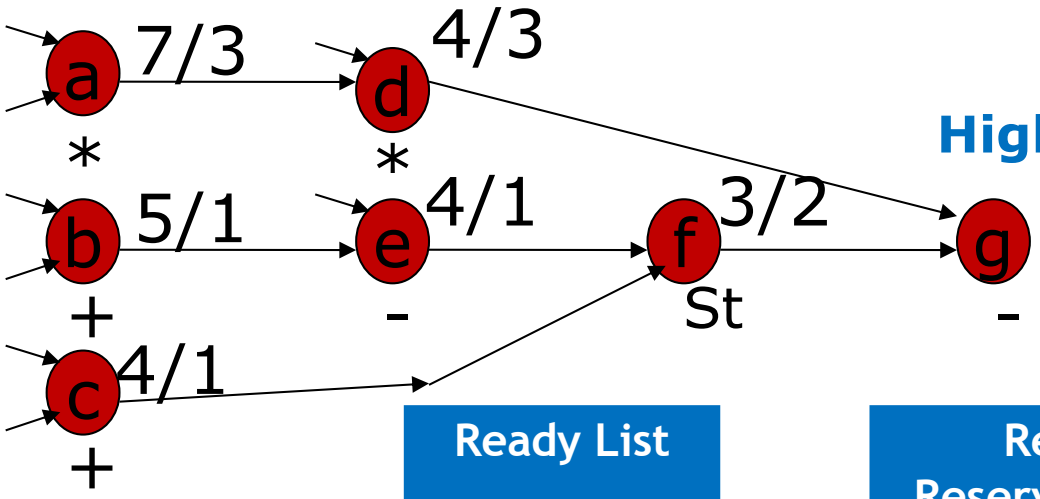
Example: List-based scheduling



Ready List	
1	a,b,c
2	c,e
3	e
4	d,f
5	
6	
7	

Resource Reservation Table		
ALU1	L/S	MUL
b		a
c		a
e		a
	f	d
	f	d
		d

Example: List-based scheduling



Highest Priority/Latency

Ready List	
1	a,b,c
2	c,e
3	e
4	d,f
5	
6	
7	g

Resource Reservation Table		
ALU1	L/S	MUL
b		a
c		a
e		a
	f	d
	f	d
		d
g		

VLIW Code		
ALU1	L/S	MUL
b	nop	a
c	nop	nop
e	nop	nop
nop	f	d
nop	nop	nop
nop	nop	nop
g	nop	nop

POLITECNICO
MILANO 1863

Local and Global Scheduling Techniques

POLITECNICO
MILANO 1863

Exploiting ILP: Local and Global Scheduling

- To exploit all the possible parallelism within each basic block, then the compiler must try to expand basic blocks and schedule instructions across basic blocks.
- **Local Scheduling** techniques operate within a single basic block:
  - Loop Unrolling
  - Software Pipelining
- **Global Scheduling** techniques operate across basic blocks:
  - Trace Scheduling
  - Superblock Scheduling

POLITECNICO
MILANO 1863

Local Scheduling Techniques: Loop Unrolling and Software Pipelining

POLITECNICO
MILANO 1863

Loop Unrolling

- The compiler must test if loop iterations are **independent** to each other.

- The compiler can increase the amount of available ILP by unrolling a loop: **the loop body is replicated multiple times** (depending on the **unrolling factor**), adjusting the loop termination code.

POLITECNICO MILANO 1863

Loop Unrolling

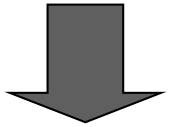
- **Pros:**
  - **Loop overhead** (number of counter increments and branches per loop) is minimized.
  - Loop unrolling extends the **length of the basic block** $\Rightarrow$ the loop exposes more instructions that can be effectively scheduled to minimize NOP insertions.
- **Cons:**
  - Loop unrolling increases the **register pressure** (number of allocated registers) due to the need of register renaming to avoid name dependencies.
  - Loop unrolling increases the **code size and instruction cache misses.**

POLITECNICO
MILANO 1863

Loop Unrolling Example

```
for (i=1000; i>0; i=i-1)
        x[i] = x[i]+s
```

Loop iterations are **independent** to each other



First, we consider a single iteration

```
Loop: LD     F0,0(R1)
      ADD    F4,F0,F2
      SD     F4,0(R1)
      SUBI   R1,R1,#8
      BNE    R1,R2,LOOP
```

Assembly code

(Not scheduled)

Loop overhead: 2 instructions per iteration

POLITECNICO MILANO 1863

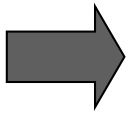
Loop Unrolling Example: Latencies

- Considering data and control dependences (branch solved in ID stage) and **fully pipelined FUs** with the following latencies:

Instruction producing result	**Instruction using result**	**Latency in cycles**
FP ALU op	Another FP ALU op	4
FP ALU op	Store double	3
Load double	FP ALU op	2
Load double	Store double	1
Integer op	Integer op	1
Integer op	Branch op	2

POLITECNICO
MILANO 1863

Loop Unrolling Example: Scalar Processor



```
Loop: LD     F0,0(R1)
      NOP
      ADD    F4,F0,F2
      NOP
      NOP
      SD     F4,0(R1)
      SUBI   R1,R1,#8
      NOP
      BNE    R1,R2,LOOP
      NOP  (br. delay slot)
```

Single-issue scheduled assembly code (no unrolling)

Exection time: 10 clock cycles per iteration

Loop overhead: 4 cycles per iteration

Code Efficiency i.e. Percentage of available slots that contains an operation is 5/10 => 50%

Loop Unrolling Example: 4x Unrolling

4 times loop unrolling (unrolling factor 4) - Not scheduled

```
Loop:  LD     F0,0(R1)
       ADD    F4,F0,F2
       SD     F4,0(R1)
       LD     F0,-8(R1)
       ADD    F4,F0,F2
       SD     F4,-8(R1)
       LD     F0,-16(R1)
       ADD    F4,F0,F2
       SD     F4,-16(R1)
       LD     F0,-24(R1)
       ADD    F4,F0,F2
       SD     F4,-24(R1)
       SUBI   R1,R1,#32
       BNE    R1,R2,LOOP
```

There are:

- True data dependences
- Name dependences

The compiler can apply **register renaming** to avoid name dependences

Loop overhead: 2 instructions per iteration

Loop Unrolling Example: Register Renaming

```
Loop:  LD     F0,0(R1)
       ADD    F4,F0,F2
       SD     F4,0(R1)
       LD     F6,-8(R1)
       ADD    F8,F6,F2
       SD     F8,-8(R1)
       LD     F10,-16(R1)
       ADD    F12,F10,F2
       SD     F12,-16(R1)
       LD     F14,-24(R1)
       ADD    F16,F14,F2
       SD     F16,-24(R1)
       SUBI   R1,R1,#32
       BNE    R1,R2,LOOP
```

After Register Renaming: There are **only true data dependences**

Next step: Apply list-based scheduling

Loop Unrolling: List-based Scheduling on Scalar Processor

```
Loop:  LD     F0, 0(R1)
       LD     F6,-8(R1)
       LD     F10,-16(R1)
       LD     F14,-24(R1)
       ADDD   F4,F0,F2
       ADDD   F8,F6,F2
       ADDD   F12,F10,F2
       ADDD   F16,F14,F2
       SD     F4, 0(R1)
       SD     F8,-8(R1)
       SD     F12,-16(R1)
       SD     F16,-24(R1)
       SUBI   R1,R1,#32
       NOP
       BNE    R1,R2,LOOP
       NOP  (br. delay slot)
```

Single-issue scheduled assembly code on a scalar processor => ***Performance improved***

Execution time: 16 cycles per 4 iterations => 4 cycles per iteration

Loop overhead: 4 cycles per 4 iterations => 1 cycle per iteration

Efficiency: 14/16 => 87.5%

Next step: Apply further scheduling optimizations

POLITECNICO MILANO 1863

Loop Unrolling: Further Scheduling Optimization on Scalar Processor

```
Loop:  LD     F0,0(R1)
       LD     F6,-8(R1)
       LD     F10,-16(R1)
       LD     F14,-24(R1)
       ADDD   F4,F0,F2
       ADDD   F8,F6,F2
       ADDD   F12,F10,F2
       ADDD   F16,F14,F2
       SD     F4,0(R1)
       SD     F8,-8(R1)
       SUBI   R1,R1,#32
       SD     F12,16(R1)
       BNE    R1,R2,LOOP
       SD     F16, 8(R1)
```

Further performance improvement

Execution time: 14 cycles per 4 iterations => 3.5 cycles per iteration

Loop overhead: 2 cycles per 4 iterations => 0.5 cycle per iteration

Efficiency: 14 / 14 => 100%

Next step: List-based scheduling on a 5-issue VLIW

Loop Unrolling: List-based Scheduling on 5-issue VLIW

Memory ref. 1	Memory ref. 2	FP op. 1	FP op.2	Integer op. /Branch
LD F0,0(R1)	LD F6,-8(R1)	NOP	NOP	NOP
LD F10,-16(R1)	LD F14,-24(R1)	NOP	NOP	NOP
NOP	NOP	ADDD F4,F0,F2	ADDD F8,F6,F23	NOP
NOP	NOP	ADDD F12,F10,F2	ADDD F16,F14,F2	NOP
NOP	NOP	NOP	NOP	NOP
SD F4, 0(R1)	SD F8,-8(R1)	NOP	NOP	NOP
SD F12,-16(R1)	SD F16,-24(R1)	NOP	NOP	SUBI R1,R1,#32
NOP	NOP	NOP	NOP	NOP
NOP	NOP	NOP	NOP	BNEZ R1,LOOP
NOP	NOP	NOP	NOP	NOP (br.del.slot)

Execution time: 10 cycles per 4 iterations => 2.5 cycles per iteration

Loop overhead: 3 cycles per 4 iterations => 0.75 cycles per iteration

Efficiency i.e. Percentage of available slots that contains an operation: 14/50 => 28%

Average ops per clock: 14 / 10 => 1.4 ops per clock

Loop Unrolling: Further Scheduling Optimization on 5-issue VLIW

Memory ref. 1	*Memory ref. 2*	*FP Op. 1*	*FP Op. 2*	*Integer op. /Branch*
LD F0,0(R1)	LD F6,-8(R1)	NOP	NOP	NOP
LD F10,-16(R1)	LD F14,-24(R1)	NOP	NOP	SUBI R1,R1,#32
NOP	NOP	ADDD F4,F0,F2	ADDD F8,F6,F23	NOP
NOP	NOP	ADDD F12,F10,F2	ADDD F16,F14,F2	NOP
NOP	NOP	NOP	NOP	NOP
SD F4, 32(R1)	SD F8,24(R1)	NOP	NOP	BNEZ R1,LOOP
SD F12,16(R1)	SD F16,8(R1)	NOP	NOP	NOP

Execution time: 7 cycles per 4 iterations => 1.75 cycles per iteration

No loop overhead

Efficiency i.e. Percentage of available slots that contained an operation: 14/35 => 40%

Average ops per clock: 14 / 7 => 2 ops per clock

POLITECNICO
MILANO 1863

Loop-level Analysis and Software Pipelining

POLITECNICO
MILANO 1863

Loop-carried dependences

- **Loop-level analysis** involves determining what data dependences exist among the operands across the iterations of a loop.

- **Loop-carried dependence:** Whether data accesses in later iterations are dependent on data values produced in earlier iterations.

Loop-carried dependences

```
for(i=1;i<=100;i=i+1)   /* 100 iterations */
{
  A[i] = B[i]+C[i];
}
```

- In this example an iteration of the loop does not have any dependence with previous iteration (works on different array elements) ⇒ ***NO LOOP-CARRIED DEPENDENCIES*** => we can apply ***loop unrolling:***

```
for(i=1;i<=97;i=i+4)      /* 25 iterations */
{
A[i]   = B[i]+C[i];
A[i+1] = B[i+1]+C[i+1];
A[i+2] = B[i+1]+C[i+2];
A[i+3] = B[i+3]+C[i+3];
}
```

Larger basic block
with extended parallelism
(the unrolling factor could
also be greater than 4)

Loop-carried dependences

```
for(i=6; i<=100; i=i+1).  /* 95 iterations */
{
   Y[i] = Y[i-5]+Y[i]
}
```

- Each iteration **`i`** depends on the value of the iteration ***`i-5`***.
  Iterations **`i, i+1,i+2,i+3, i+4`** are independent!
  (***`i+5`*** is dependent on ***`i`***) => ***we can unroll the loop up to 5:***

```
for(i=6; i<=96; i=i+5)   /* 19 iterations */
{
   Y[i]   = Y[i-5]+Y[i]
   Y[i+1]= Y[i-4]+Y[i+1]
   Y[i+2]= Y[i-3]+Y[i+2]
   Y[i+3]= Y[i-2]+Y[i+3]
   Y[i+4]= Y[i-1]+Y[i+4]
}
```

Larger basic block
with extended parallelism
(the unrolling factor could
not be greater than 5)

POLITECNICO MILANO 1863

Loop-carried dependences

```
for(i=1;i<=100;i=i+1)
{
   A[i+1]=A[i]+C[i];    /* S1 */
   B[i+1]=B[i]+A[i+1]   /* S2 */
}
```

- **Analysis of dependences:**
  - S1 uses a value computed by S1 in an earlier iteration for array A;
    ⇒ ***LOOP CARRIED DEPENDENCE***
  - S2 uses a value computed by S2 in an earlier iteration for array B;
    ⇒ ***LOOP CARRIED DEPENDENCE***
  - S1 uses the value of array C[] that is never modified in the loop;
    ⇒ **NO *LOOP CARRIED DEPENDENCE***
  - S2 uses the value a[i+1] computed by S1 at the same iteration
    ⇒ ***NO LOOP CARRIED DEPENDENCE***

POLITECNICO
MILANO 1863

Loop Peeling and Fusion

```
for(i=0;i<102;i++) b[i]=b[i-2]+c; /* Loop A: 102 iterations */
for(j=0;j<100;j++) a[j]=a[j]*2;   /* Loop B: 100 iterations */
```

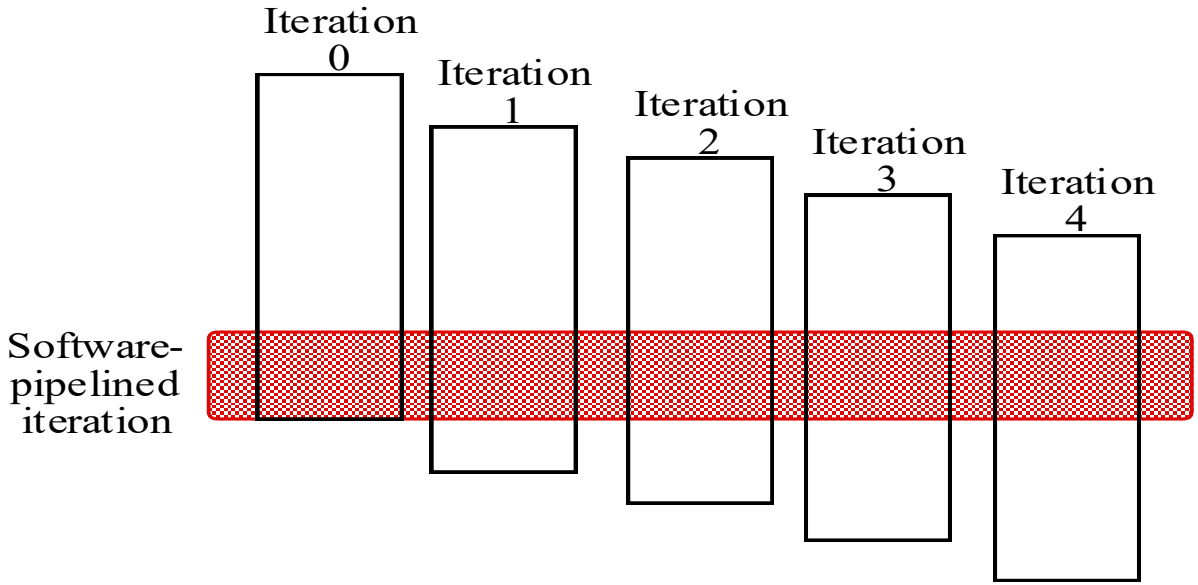
- We ***"peel"*** loop A from the last two iterations (making it a 100 iteration loop).
- Then we fuse the two 100 iteration loops to make a larger basic block..

```
for(i=0;i<100;i++){
   b[i]=b[i-2]+c;          / * Fused loops: 100 iterations */
   a[i]=a[i]*2; }

   b[100]=b[98]+c;         /* Peeled instructions from loop A */
   b[101]=b[99]+c;
```

Software Pipelining

- Suppose that the following loop presents independent instructions in different iterations (evidenced in red).



- **We can reorganize the loop in a new loop so that each new iteration ("cycle") executes instructions ("stages") chosen from different iterations of the original loop.**

POLITECNICO
MILANO 1863

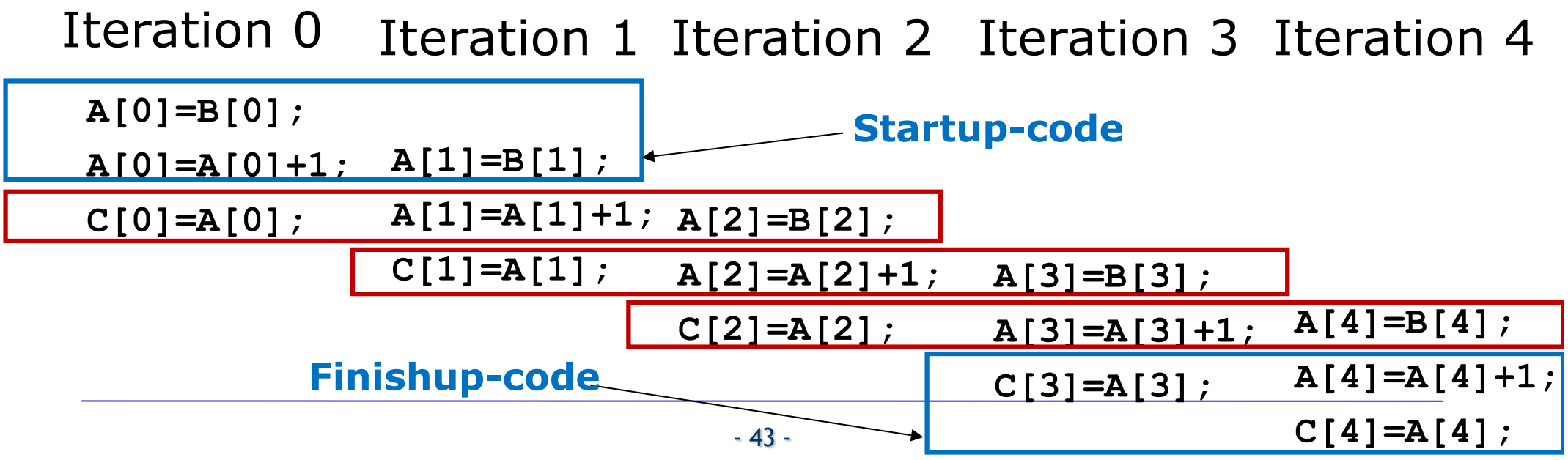
Software Pipelining

- Technique for reorganizing loops such that each iteration of the software-pipelined code is made from instructions chosen from different iterations of the original loop.
- Software pipelining interleaves instructions from different iterations without unrolling the loop.
- Software pipelining can be thought as a sort of Symbolic Loop Unrolling

Software Pipelining: Example

```
for(i=0; i<5; i++)
{
   A[i]=B[i];   // stage X
   A[i]=A[i]+1; // stage Y
   C[i]=A[i];   // stage Z
}
```

- *No loop carried dep.*
- *Intra-body data dependences*



POLITECNICO MILANO 1863

Software Pipelining

```
for(i=0; i<3; i++)
{
   C[i]=A[i];
   A[i+1]=A[i+1]+1;
   A[i+2]=B[i+2];
}
```

Independent instructions

Software Pipelining

```
A[0]=B[0];
A[0]=A[0]+1;
A[1]=B[1];
for(i=0; i<3; i++)
{
   C[i]=A[i];
   A[i+1]=A[i+1]+1;
   A[i+2]=B[i+2];
}
C[3]=A[3];
A[4]=A[4]+1;
C[4]=A[4];
```

Startup-code

Independent instructions

Finishup-code

POLITECNICO
MILANO 1863

Software Pipelining: Example in assembly code

```
LOOP: LD $F0, 0($R1)
      ADDD $F4, $F0, $F2
      SD $F4, 0($R1)
      ADDUI $R1, $R1, 8
      BNE $R1, $R2, LOOP
```

- *No loop carried dep.*
- *Intra-body data dependences*

Show a software-pipelined version of this loop by omitting the start-up and finish up code and loop indexes adjustment.

POLITECNICO
MILANO 1863

Software Pipelining: Example in assembly code

```
LOOP: LD $F0, 0($R1)
      ADDD $F4, $F0, $F2
      SD $F4, 0($R1)
      ADDUI $R1, $R1, 8
      BNE $R1, $R2, LOOP
```

- *No loop carried dep.*
- *Intra-body data dependences*

LD $F0, 0($R1)			
ADDD $F4, $F0, $F2	LD $F0, 8($R1)		
SD $F4, 0($R1)	ADDD $F4, $F0, $F2	LD $F0, 16($R1)	
	SD $F4, 8($R1)	ADDD $F4, $F0, $F2	....
		SD $F4, 16($R1)	....
			....

POLITECNICO
MILANO 1863

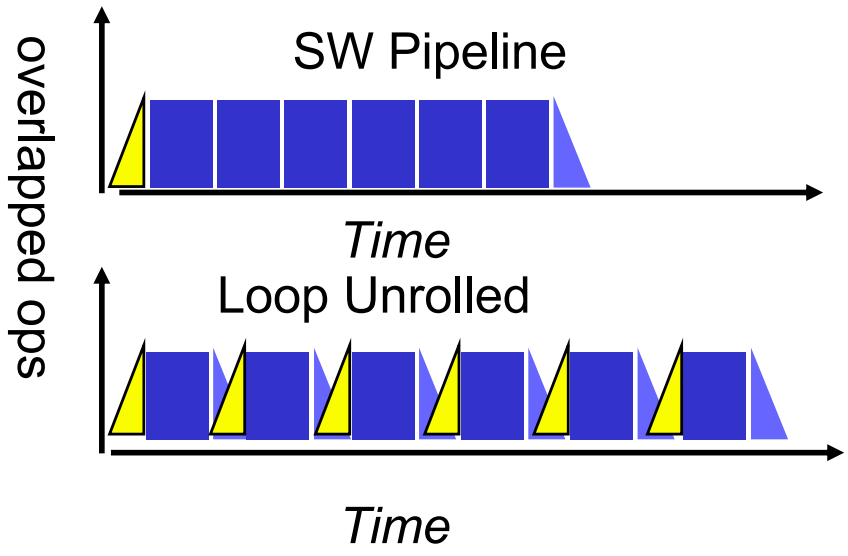
Software Pipelining: Example in assembly code

```
SLOOP: SD $F4, 0 ($R1)      /* SD from iter.[i] corr. to 0($R1)*/
       ADDD $F4, $F0, $F2   /* ADDD from iter.[i+1] corr. to 8($R1)*/
       LD $F0, 16 ($R1)     /* LD from iter.[i+2] corr. to 16($R1)*/
       ADDUI $R1, $R1, 8
       BNE $R1, $R3, SLOOP
```

LD $F0, 0($R1)			
ADDD $F4, $F0, $F2	LD $F0, 8($R1)		
SD $F4, 0($R1)	ADDD $F4, $F0, $F2	LD $F0, 16($R1)	
	SD $F4, 8($R1)	ADDD $F4, $F0, $F2	....
		SD $F4, 16($R1)	....
			....

Advantages of SW Pipelining vs Loop Unrolling

- Consumes less space (no need to duplicate body-code) than loop unrolling.
- **Fill and drain pipeline only once per loop** vs. once per each unrolled iteration in loop unrolling



- Can be associated with loop unrolling to provide better performance

POLITECNICO
MILANO 1863

Global Scheduling Techniques: Trace Scheduling and Superblock Scheduling

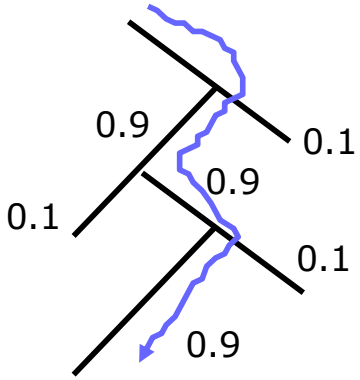
POLITECNICO
MILANO 1863

Global Code Scheduling

- Software pipelining works well when the loop body is a single basic block.
- When the loop body contains **internal control flow,** effective scheduling requires moving instructions across branches => **Global Code Scheduling**
- **Global Code Scheduling** aims at compacting a code fragment with **internal control structures** into the shortest possible sequence preserving data and control dependences

Trace Scheduling

- **Tries to find parallelism across conditional branches** (global code scheduling).
- Composed of two steps:
  - *Trace Selection*
    - Find most likely sequence of basic blocks (*trace*) of long sequence of straight-line code (statically-predicted or profile-predicted).
  - *Trace Compaction*
    - Squeeze the trace into few VLIW instructions
    - Need book-keeping (compensation) code in case the prediction is wrong.



POLITECNICO MILANO 1863

Trace Scheduling

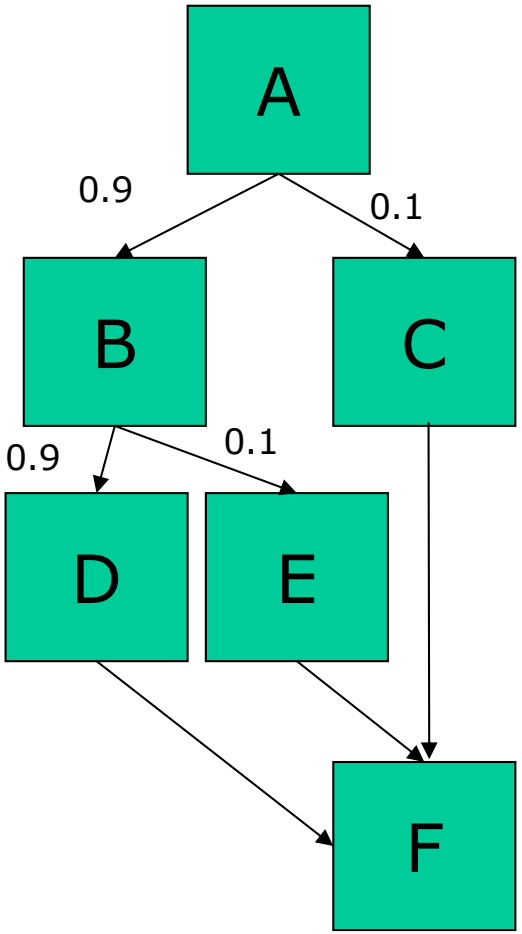
- This is a form of compiler-generated speculation
  - Compiler must generate "*fixup*" code to handle cases in which the trace is wrong (misprediction).
  - Needs extra registers and time to undo badly predicted instructions.

POLITECNICO
MILANO 1863

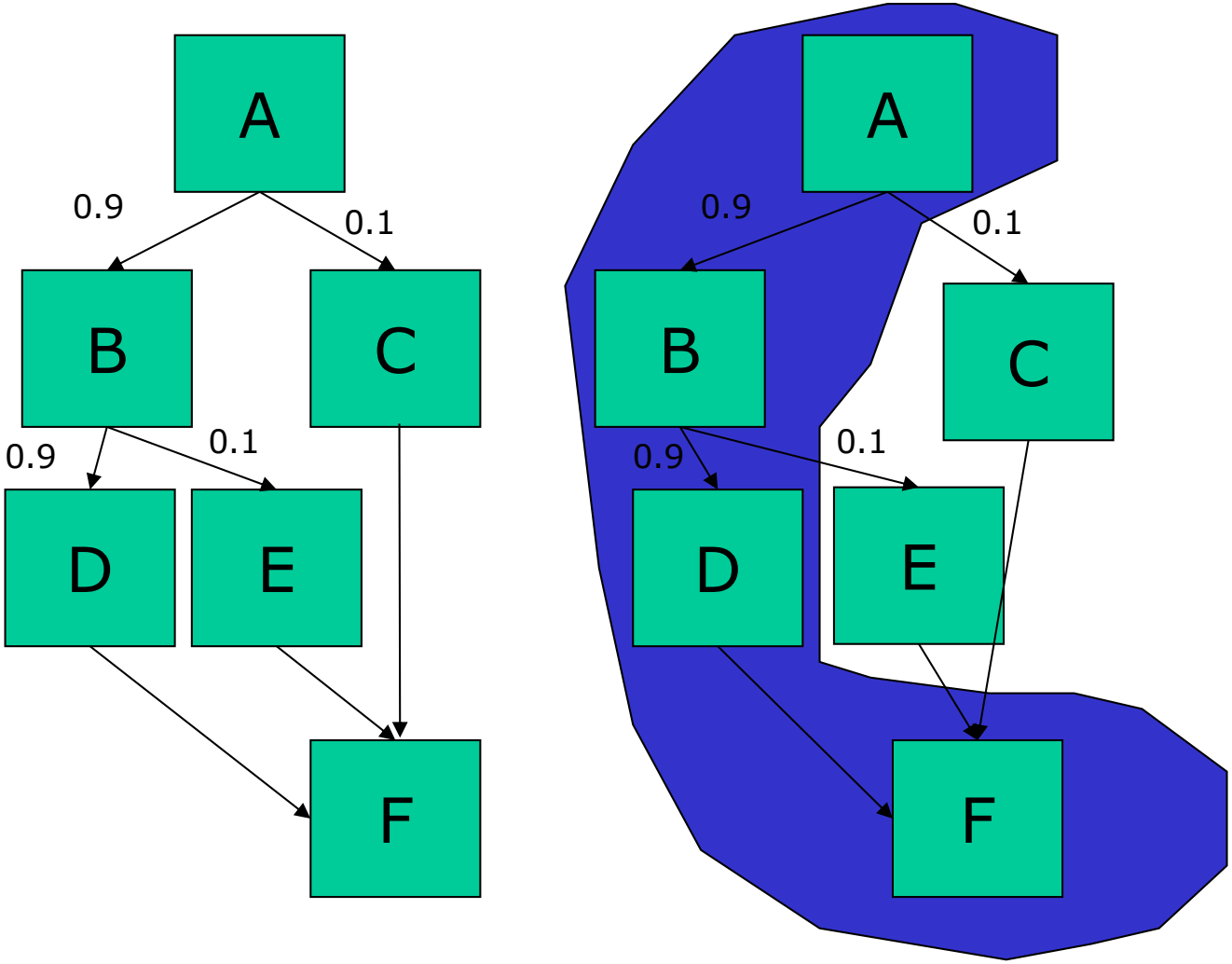
Superblock Scheduling

- Extension/Optimization of Trace Scheduling
- A **Superblock** is a group of basic blocks with a single entrance and multiple control exits.
- Superblocks are constructed by profiling the application and by duplicating tails (blocks after an entrance in the trace).
- Advantages:
  - Optimization simpler because there are no side entrances.
  - We need to create compensation code only for the exits and not for the entrance.

Superblock Formation

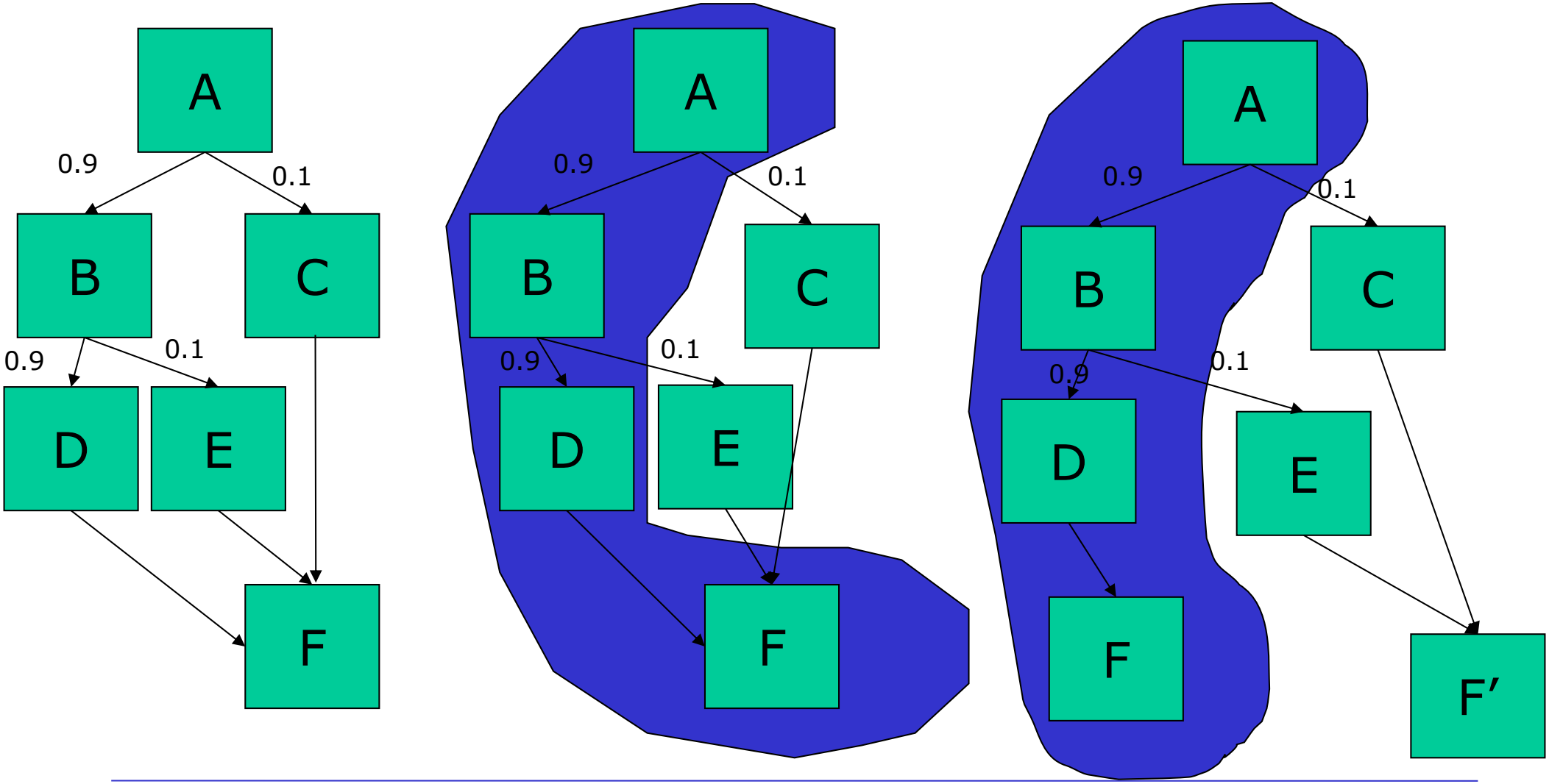


Superblock Formation



POLITECNICO
MILANO 1863

Superblock Formation



Hardware Support for Exploiting More ILP at Compile Time

- Techniques such as loop unrolling, software pipelining, and trace scheduling can be used to increase ILP when the branch behavior is fairly predictable at compile time.
- Otherwise control dependences may limit the parallelism that can be exploited.
- To overcome such limitation, we can:
  - Extend the instruction set to include Conditional or Predicated Instructions
  - Use Compiler Speculation with hardware support to enable the compiler to speculatively move code over branches, while preserving exception behavior.