Course on Advanced Computer Architectures

VLIW Code Scheduling

Code Scheduling

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

  $\Rightarrow$ Minimize Execution Time

Scheduling Basics

- Decompose the function in basic blocks.
- A basic block is a code sequence that does not contain a branch or a branch target within the sequence

Dependence Graph

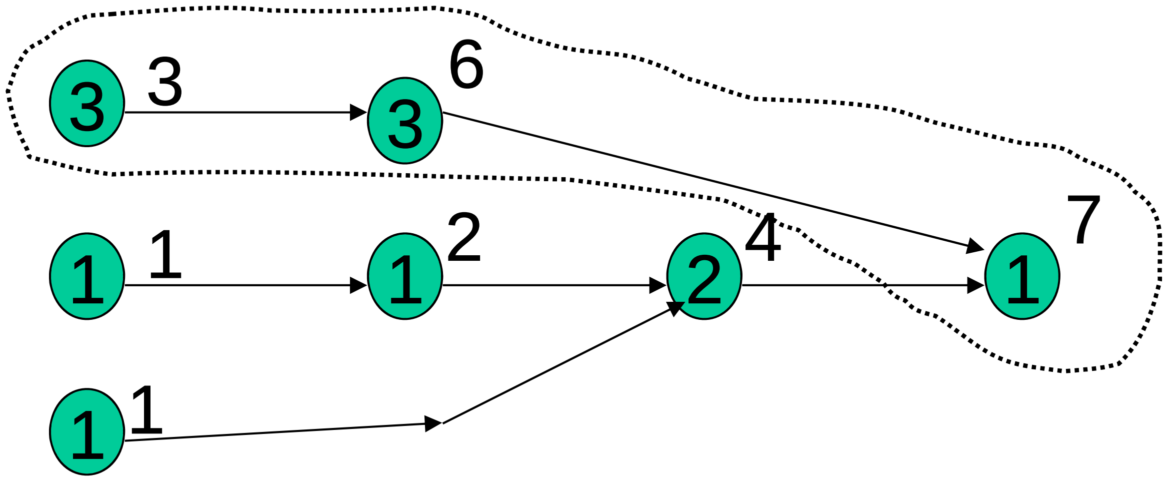
- A dependence graph captures true, anti and output dependencies. Anti and output dependencies are name dependences due to variables/registers reuse.

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

$$\begin{tikzcd}
i \arrow[dr, "\text{RAW}"] \arrow[dd, dashed, "\text{WAW}"'] & \\
& i+k \arrow[dl, dashed, "\text{WAR}"] \\
i+n &
\end{tikzcd}$$

Critical Path

- Is the longest path in the dependence graph
- Determines the minimum execution time of a basic block



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

LCP = Max (LP(i))

Scheduling Basics

- Scheduling selects the cycle of execution of each operation so that it executes in the minimum amount of time.
- On a processor with infinite resources, we could schedule all the ops of the CP and then the remaining ones by exploiting available slots.
- With finite resources, the execution time does not depend only on CP but also on how we schedule remaining instructions

Scheduling Basics

- An optimum scheduler must exhaustively search the space of optimal schedule.
- Space and time complexity is very big (NP)!
- We must use heuristics.

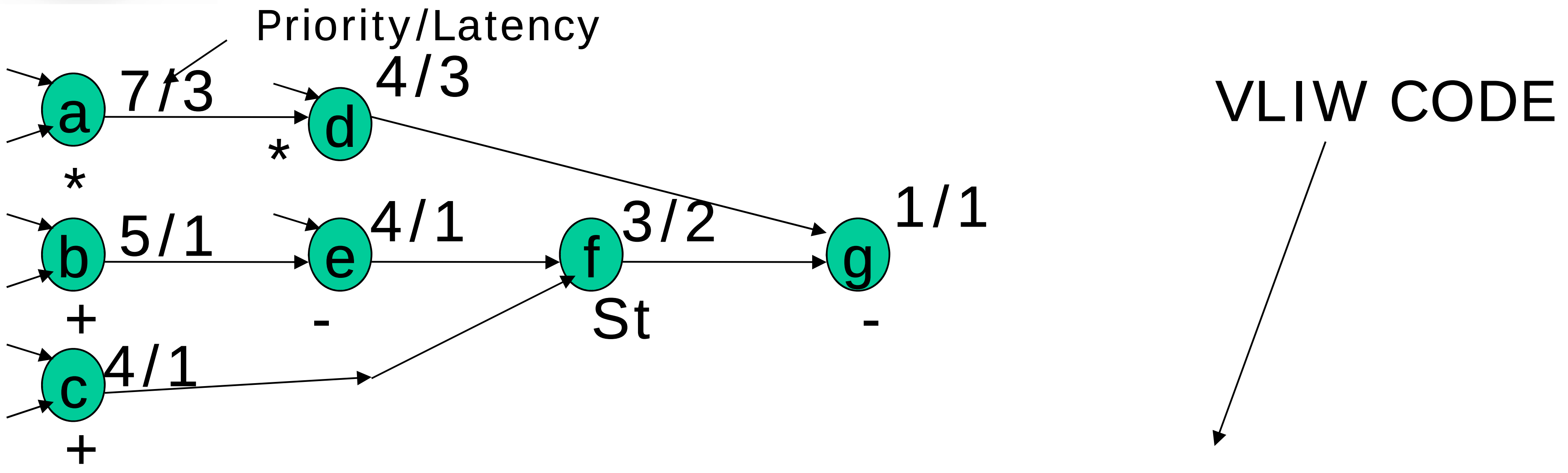
List-based Scheduling for a Basic Block

- Resource-constrained scheduling.
- Before scheduling begins, operations on top of the graph are inserted in the *ready set*.
- An *instruction* 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 instructions (nodes) from the *ready set* that can fill the available slots.
- When more nodes are in the ready set, select the op. with highest priority (longest path to the bottom of the graph).

Resource Reservation Table

- Keeps track of busy resources
- If the selected instruction has a busy unit, try with another operation in the ready set.

Example



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

RRT		
ALU1	L/S	MUL
b		a
c		a
e		a
	f	d
	f	d
		d
g		

VLIW		
ALU	L/S	MUL
b	n	a
c	n	n
e	n	n
n	f	d
n	n	n
n	n	n
g	n	n

Exploiting ILP: Local and Global Scheduling

- To exploit all the possible parallelism, the compiler must expand basic block or schedule instructions across basic blocks.
- Local Scheduling Techniques operate within a single basic block.
  - Example: Loop Unrolling, Software Pipelining
- Global Scheduling Techniques operate across basic blocks.
  - Example: Trace Scheduling, Superblock Scheduling

Local Scheduling Techniques: Loop Unrolling and Software Pipelining

Loop Unrolling

- The compiler can increase the amount of available ILP by unrolling loops: **the loop body is replicated multiple times** (depending on the unrolling factor), adjusting the loop termination code.
- The compiler must test if loop iterations are **independent.**
- **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 computation that can be effectively scheduled to minimize NOP insertions.
- Loop unrolling increases **register pressure** (number of required registers).
- Loop unrolling increases the **code size.**

Loop Unrolling: Example

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

Loop iterations are independent

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

Loop Unrolling: Example

- 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

Loop Unrolling: Example

```
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)
```

Scheduled
Assembly code
(no unrolling)

Exection time: 10 clock cycles per iteration

Loop overhead: 4 cycles per iteration

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

Loop Unrolling: Example

4 times loop unrolling (unrolling factor 4)

```
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:

- Data dependences
- Name dependences

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

Loop Unrolling: Example

```
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 remaning: Only true data dependences remain in each body and between the last 2 instructions

Next step: we can 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, 8(R1)
       SUBI   R1,R1,#32
       NOP
       BNE    R1,R2,LOOP
       NOP (br. delay slot)
```

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: further scheduling optimization

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: 16/16 => 100%

Next step: scheduling for 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 contained an operation is 14/50 => 28%

Average: 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 is 14/35 => 40%

Average: 14 / 7 => 2 ops per clock

Loop-carried dependences

- Loop-level analysis involves determining what data dependences exist among the operands in a loop across the iterations.
- 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)
{
  A[i]=B[i]+C[i];
}
```

- In this example an iteration of the loop does not have dependences with previous iteration (works on different array elements)
  $\Rightarrow$ NO LOOP CARRIED DEPENDENCIES

```
for(i=1;i<=97;i=i+4)
{
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];
}
```

$\leftarrow$ 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)
{
   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*) so we can unroll the loop up to 5:

```
for(i=6; i<=96; i=i+5)
{
   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)

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 */
}
```

- Two different dependences:
  - S1 uses a value computed by S1 in an earlier iteration for array A; the same for S2 and array B
    $\Rightarrow$ *LOOP CARRIED DEPENDENCES FOR A[]*
  - S2 uses the value a[i+1] computed by S1 in the same iteration
    $\Rightarrow$ *NO LOOP CARRIED DEPENDENCES FOR A[]*

Loop Peeling and Fusion

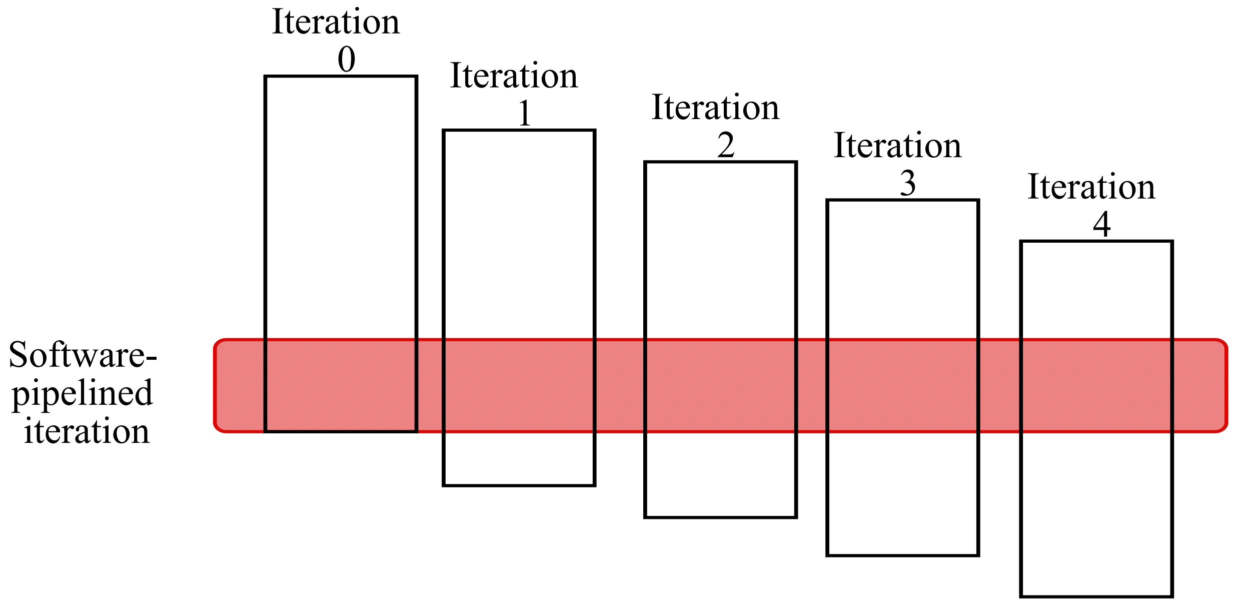
```
for(i=0;i<102;i++) b[i]=b[i-2]+c; //Loop A
for(j=0;j<100;j++) a[j]=a[j]*2;   //Loop B
```

- We “peel” loop A of the last two iterations (making it a 100 iteration loop. Then we fuse the two 100 iteration loop to make a larger basic block. Then other techniques can be applied.

```
for(i=0;i<100;i++){
  b[i]=b[i-2]+c;              // fused loops
  a[i]=a[i]*2; }
  b[100]=b[98]+c;             // peeled 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 iteration of the original loop.

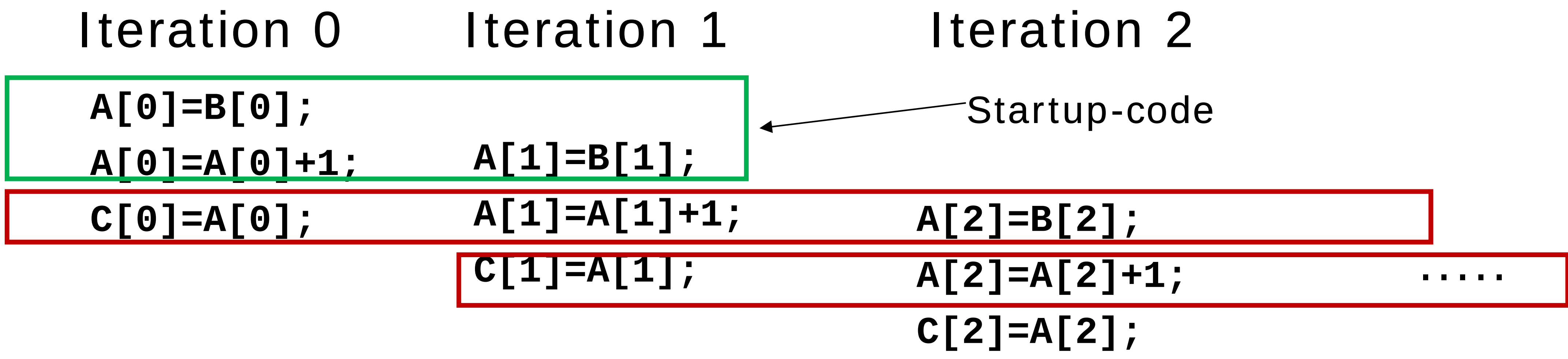
Software Pipelining

- 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 of as Symbolic Loop Unrolling

Software Pipelining: Example

```
for(i=0; i<100; i++)
{
   A[i]=B[i];    // stage X
   A[i]=A[i]+1; // stage Y
   C[i]=A[i];    // stage Z
}
```

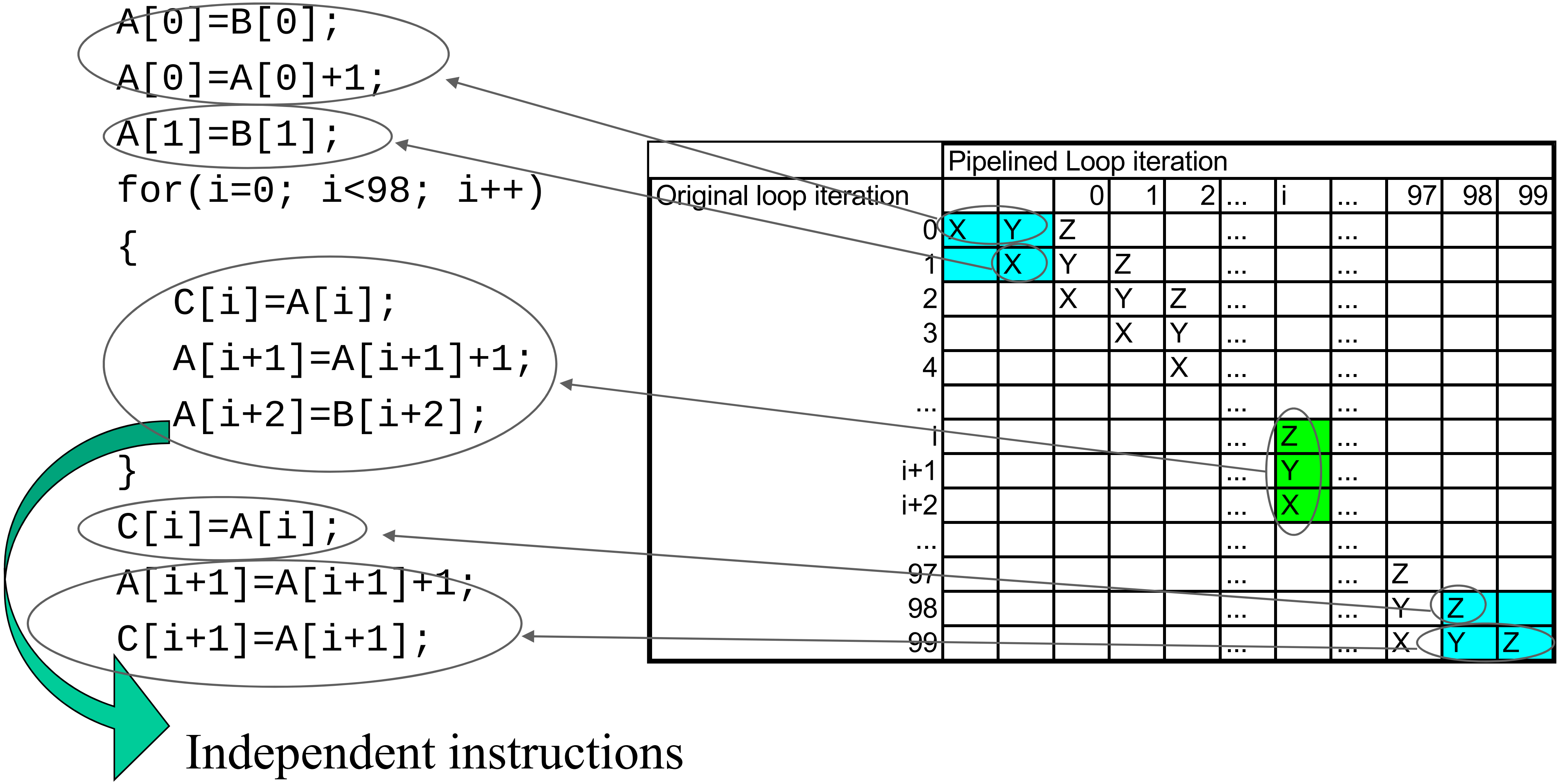
- No loop carried dep.
- Intra-Body depen. present



```
Iteration 0           Iteration 1              Iteration 2

 A[0]=B[0];
 A[0]=A[0]+1;         A[1]=B[1];
 C[0]=A[0];           A[1]=A[1]+1;             A[2]=B[2];
                      C[1]=A[1];               A[2]=A[2]+1;           .....
                                               C[2]=A[2];
```

Software Pipelining



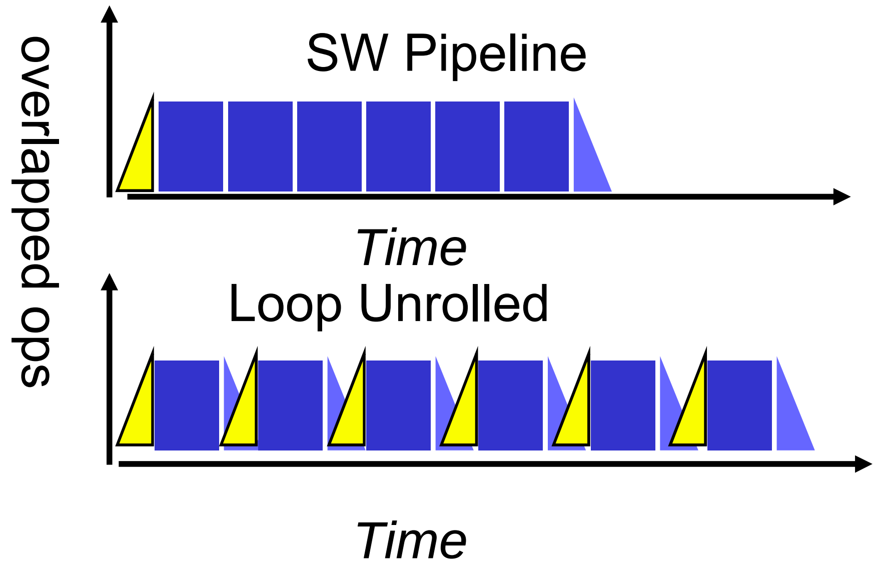
```
A[0]=B[0];
A[0]=A[0]+1;
A[1]=B[1];
for(i=0; i<98; i++)
{
    C[i]=A[i];
    A[i+1]=A[i+1]+1;
    A[i+2]=B[i+2];
}
C[i]=A[i];
A[i+1]=A[i+1]+1;
C[i+1]=A[i+1];
```

	Pipelined Loop iteration										
Original loop iteration			0	1	2	...	i	...	97	98	99
0	X	Y	Z			...		...			
1		X	Y	Z		...		...			
2			X	Y	Z	...		...			
3				X	Y	...		...			
4					X	...		...			
...						...		...			
i						...	Z	...			
i+1						...	Y	...			
i+2						...	X	...			
...						...		...			
97						...		...	Z		
98						...		...	Y	Z	
99						...		...	X	Y	Z

Independent instructions

Advantages of Software Pipelining

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



- Can be associated with loop unrolling to provide better performance

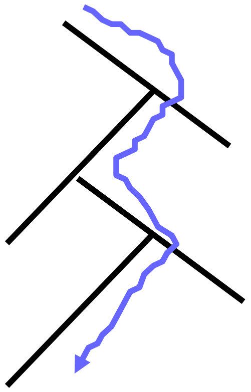
Global Scheduling Techniques:
Trace Scheduling and Superblock Scheduling

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 likely sequence of basic blocks (*trace*) of (statically predicted or profile predicted) long sequence of straight-line code
  - *Trace Compaction*
    - Squeeze trace into few VLIW instructions
    - Need bookkeeping (compensation) code in case prediction is wrong



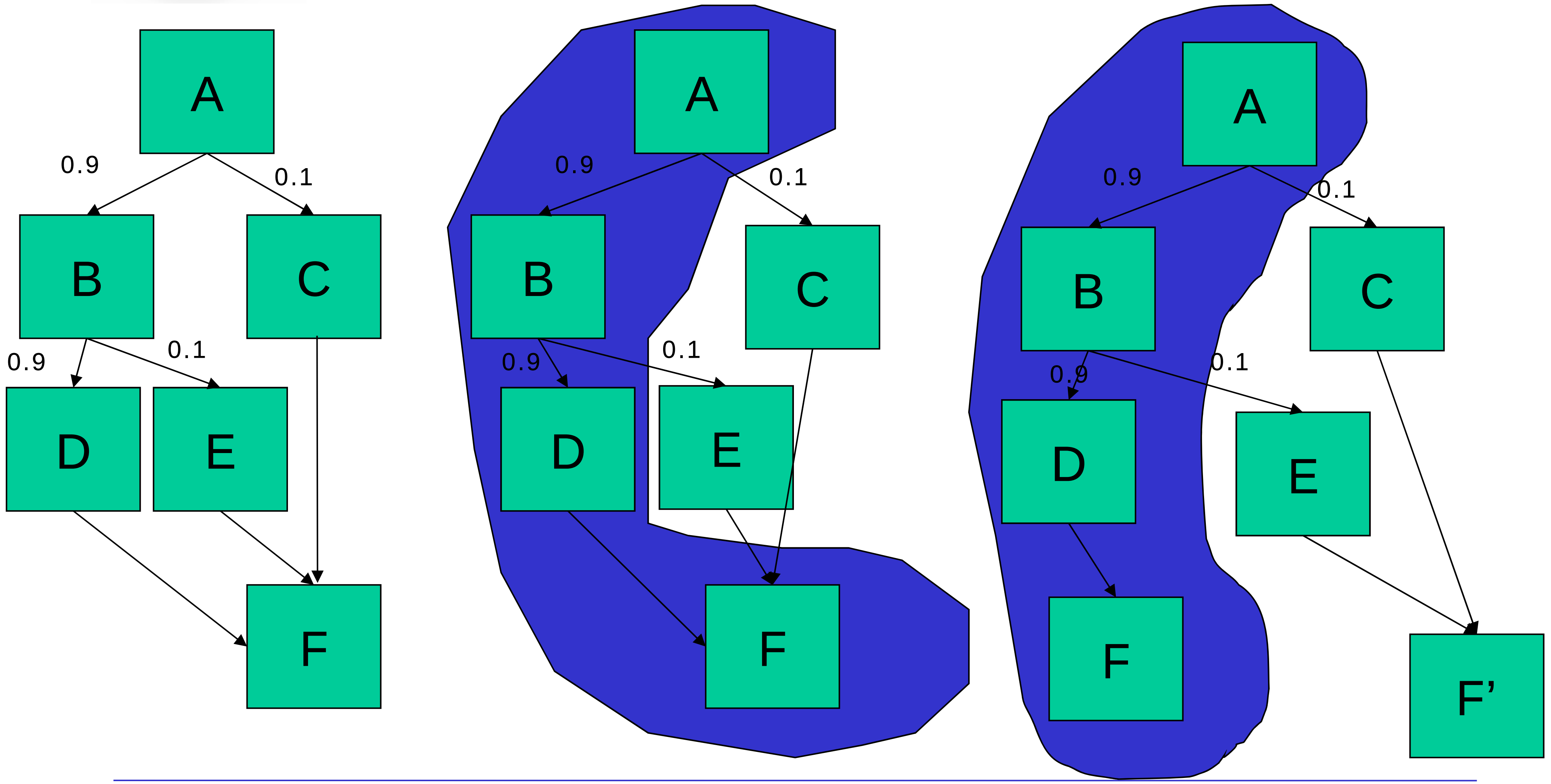
Trace Scheduling

- This is a form of compiler-generated speculation
  - Compiler must generate “fixup” code to handle cases in which trace is not the taken branch
  - Needs extra registers: undoes bad guess by discarding

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 exits and not for entrance.

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.

Conditional (or Predicated) Execution

- Predicated Instruction:

  ```
          (p)  op Rd,R1,R2
  ```

  `p` is a boolean predicate register.

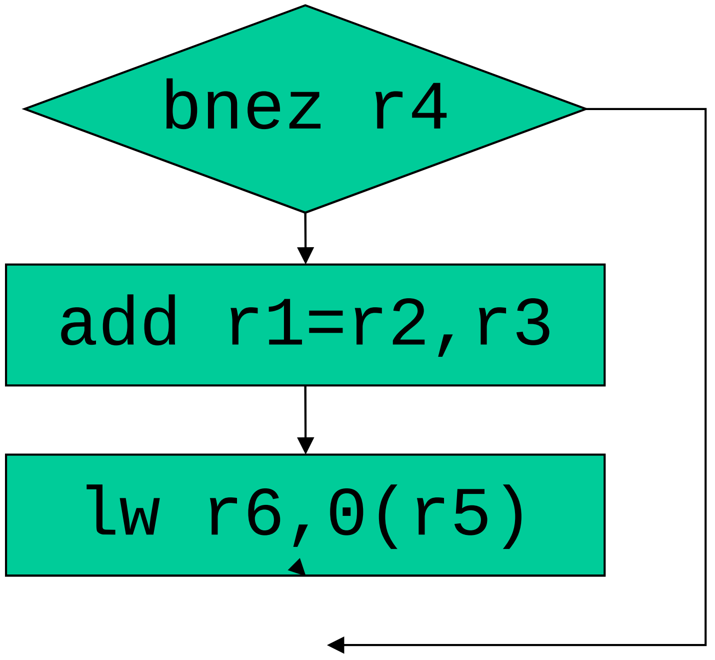
  `op Rd,R1,R2` is a normal triadic operation.

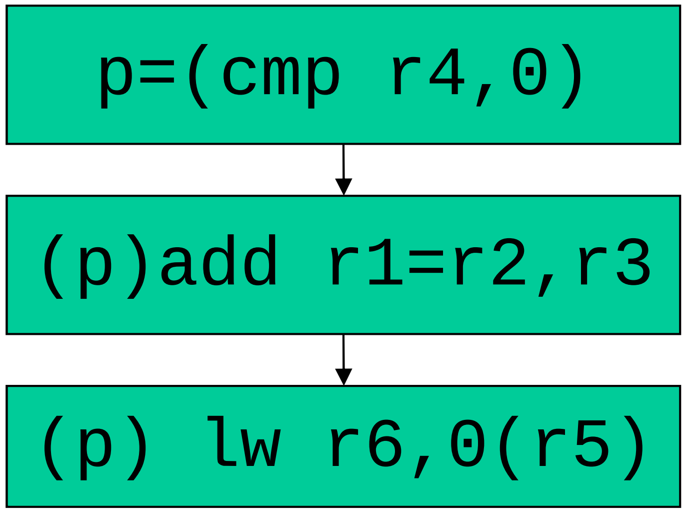
  `op` is committed only if **`p`** is **`True`**

- The execution of the operation is controlled by the predicate: When the predicate is false $\Rightarrow$ the operation becomes a **nop**.

If-conversion

- If-conversion is the process that converts a conditional branch into a sequence of predicated instructions:





Predicated Execution

- Control dependencies are transformed in data-dependencies $\Rightarrow$ Branches are eliminated.
- Advantages:
  - Data-dependency based movements can then be applied.
  - Branch misprediction is eliminated. We do not need to flush the pipeline!
  - Enlarge basic block to improve scheduling.

Predicated Execution

- Effective if:
  - Misprediction rate and penalty are considerable. Otherwise a branch to the most likely code would result in better performance.
  - Branches are unbalanced. The longest path is executed more frequently. Otherwise we must avoid to lenghten the execution of the little and more frequent path.