

Chapter3

Advanced compiler techniques for exploring ILP

❑ In Appendix H, 6th edition

Score board/ Tomasulo 中 WB、Issue 是并行执行的

- ❑ WB 中： 写 CDB，清空对应 ReservationStation，目的 RegS 置为可用
- ❑ Issue 中： 可以发射，修改 ReservationStation，
目的 RegS 置为对应 ReservationStation 编号
- ❑ 可以并行， 否则每一条指令 WB 周期后都多一个 cycle 清空周期
- ❑ 怎么实现： Issue 与 WB 要修改同一编号 ReservationStation（行）， 则做 Issue 修改， 而不做清空。
若 Issue 的目的 RegS 与 WB 的 RegS 相同， 则只做目的 RegS 置为对应 ReservationStation 编号， 而不再做 WB 的 “目的 RegS 置为可用”

Dynamic scheduling理论与是实现的区别

- ❑ Scoreboard中三张表、Tomasulo的Reservation中的值都是存在寄存器中，对应2个值：当前值、时钟上升沿之后的新值
- ❑ 理论上作业时，某一个cycle下表值或Reservation中的值填的一般是指在该cycle执行结束后得到的新值（TPA验收看到的值—还未写入寄存器）
- ❑ 但在硬件实现时，
该cycle执行结束后得到的新值 要在下个cycle上升沿后才会被写入寄存器，才会被看到（延后一拍才能看到对应寄存器中的值）

Tomasulo 算法实现注意点

□ WAW问题:

Reg的2nd Write 会覆盖Reg的1st write 但1st Write CDB的操作还是要预约，因为第一条指令的操作结果还要forwarding到需要该值的ResStation里（如下面例子中的Mul的ResStation）。

❑ LW F2, D1(Rx) F2的S会变成LoadB1

MUL F4, F2, ○ ○

❑ LW F2, D2(Ry) issue时, F2的S会变成LoadB2

Why software pipelining ?

```
1  LD      F0, 0(R1)
2  ADDDD   F4, F0, F2
3  SD      0(R1), F4
4  SUBI    R1, R1, #4
5  BNEZ    R1, LOOP
```

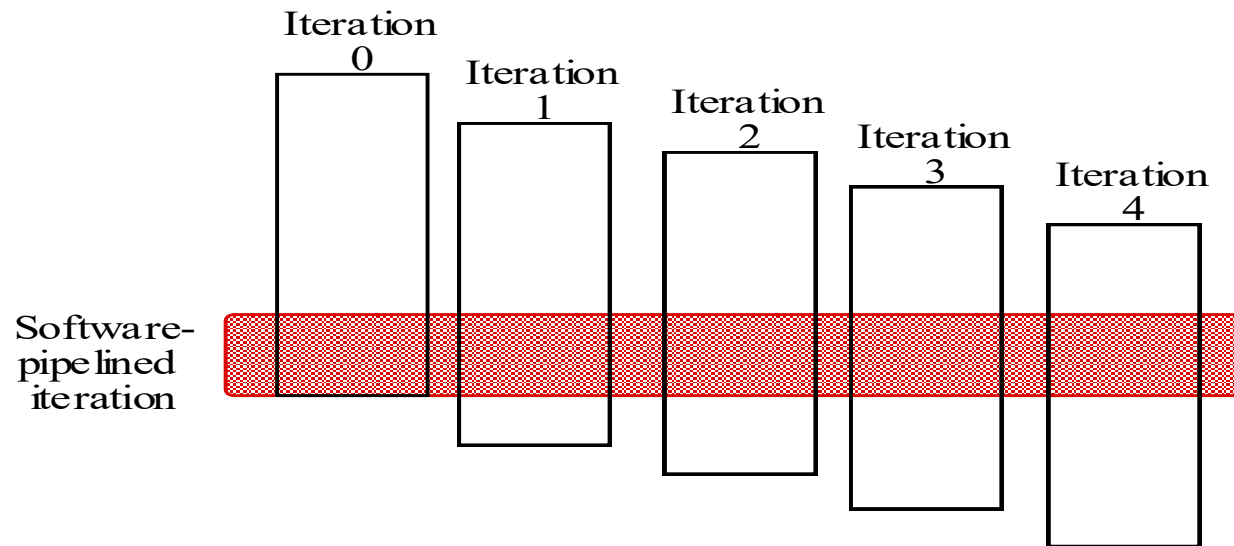
Loop Parallelism lost !

```
1  LD      F0, 0(R1)
2  stall
3  ADDDD   F4, F0, F2
4  stall
5  stall
6  SD      0(R1), F4
7  SUBI    R1, R1, #4
8  stall
9  BNEZ    R1, LOOP
10 stall
```



Software Pipelining

- ❑ Observation: if iterations from loops are independent, then can get more ILP by taking instructions from different iterations
- ❑ **Software pipelining**: **reorganizes** loops so that each iteration is made from instructions chosen from different iterations of the original loop (Tomasulo in SW)



Software Pipelining Example

Before: Unrolled 3 times

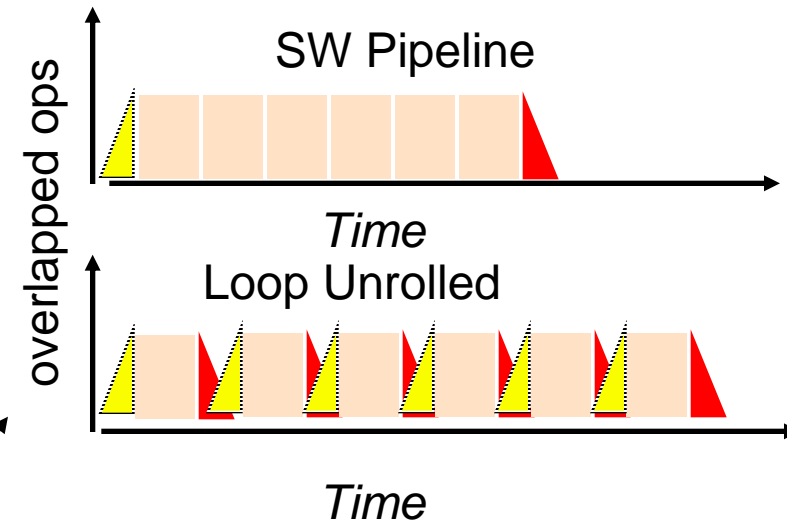
```
1  LD    F0,0(R1)
2  ADDD  F4,F0,F2
3  SD    0(R1),F4
4  LD    F6,-8(R1)
5  ADDD  F8,F6,F2
6  SD    -8(R1),F8
7  LD    F10,-16(R1)
8  ADDD  F12,F10,F2
9  SD    -16(R1),F12
10 SUBI  R1,R1,#24
11 BNEZ  R1,LOOP
```

After: Software Pipelined

```
1  SD    0(R1),F4 ; Stores M[i]
2  ADDD  F4,F0,F2 ; Adds to M[i-1]
3  LD    F0,-16(R1); Loads M[i-2]
4  SUBI  R1,R1,#8
5  BNEZ  R1,LOOP
```

- Symbolic Loop Unrolling

- Maximize result-use distance
- Less code space than unrolling
- Fill & drain pipe only once per loop
vs. once per each unrolled iteration in loop unrolling



5 cycles per iteration

Code after reorganized

```
1  LD      F0,0(R1)
1  ADDD    F4,F0,F2      start-up
2  LD      F0, -8(R1)
```

```
Loop: 1  SD      0(R1),F4 ; Stores M[i]
      2  ADDD    F4,F0,F2 ; Adds to M[i-1]
      3  LD      F0,-16(R1); Loads M[i-2]
      4  SUBI    R1,R1,#8
      5  BNEZ    R1,LOOP
```

```
2  SD      -8(R1), F4
1  ADDD    F4,F0,F2      clear-up
1  SD      -16(R1), F4
```


Software Pipelining with loop unrolling in VLIW

<i>Memory reference 1</i>	<i>Memory reference 2</i>	<i>FP operation 1</i>	<i>FP op. 2</i>	<i>Int. op/ branch</i>	<i>Clock</i>
LD F0,-48(R1)	ST 0(R1),F4	ADDD F4,F0,F2			1
LD F6,-56(R1)	ST -8(R1),F8	ADDD F8,F6,F2		SUBI R1,R1,#24	2
LD F10,-40(R1)	ST 8(R1),F12	ADDD F12,F10,F2		BNEZ R1,LOOP	3

❑ Software pipelined across 9 iterations of original loop

➤ In each iteration of above loop, we:

- Store to m,m-8,m-16 (iterations l-3,l-2,l-1)
- Compute for m-24,m-32,m-40 (iterations l,l+1,l+2)
- Load from m-48,m-56,m-64 (iterations l+3,l+4,l+5)

❑ 9 results in 9 cycles, or 1 clock per iteration

❑ Average: 3.3 ops per clock, 66% efficiency

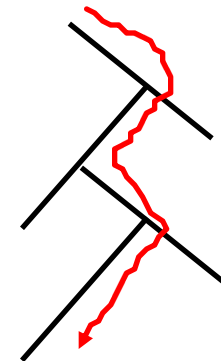
Note: Need less registers for software pipelining
(only using 7 registers here, was using 15)

Global code motion

- ❑ The loop unrolling and software pipelining work well with straight forward loop.
- ❑ What about a loop with internal control flow ?
- ❑ **Global code scheduling** : Effective scheduling of a loop body with internal control flow by moving instructions across branches.

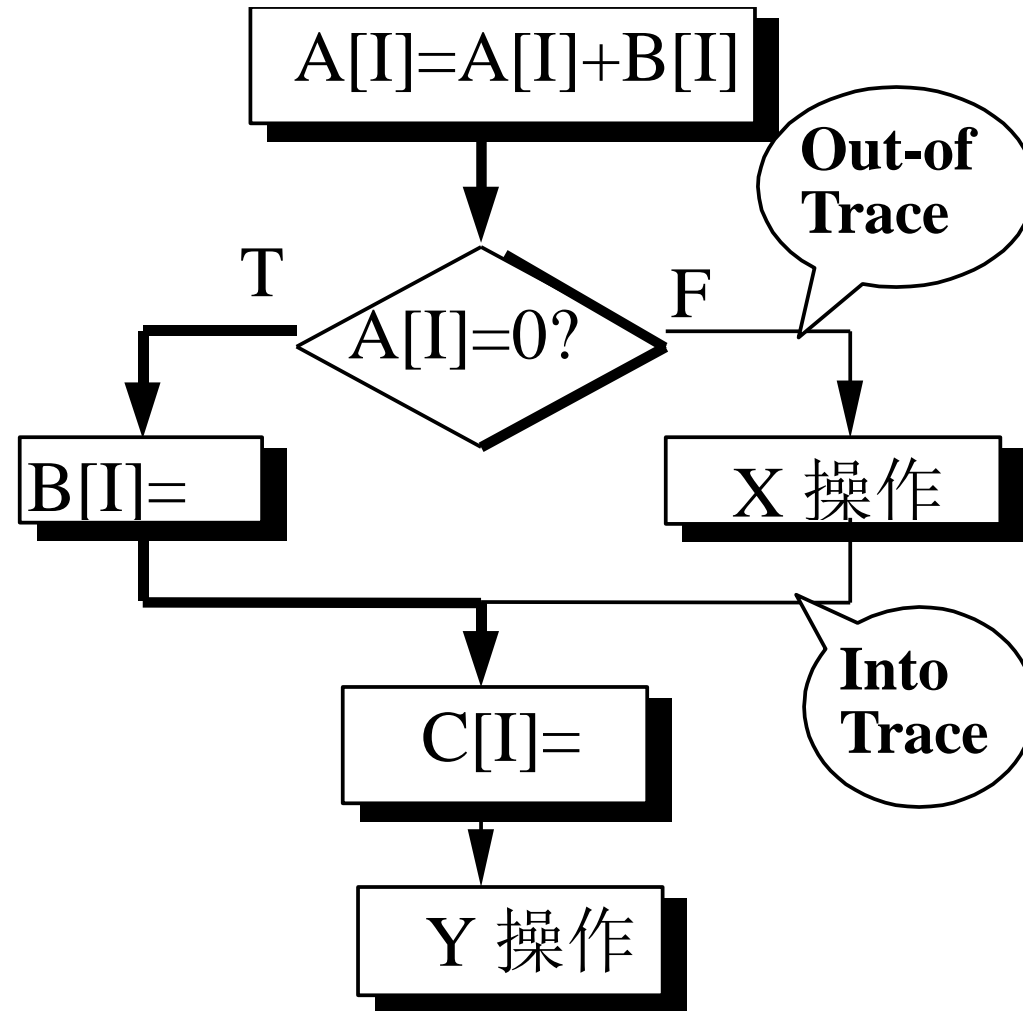
Trace Scheduling

- ❑ Parallelism across IF branches vs. LOOP branches
- ❑ 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 code in case prediction is wrong
- ❑ 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
- ❑ Subtle compiler bugs mean wrong answer vs. poorer performance; no hardware interlocks

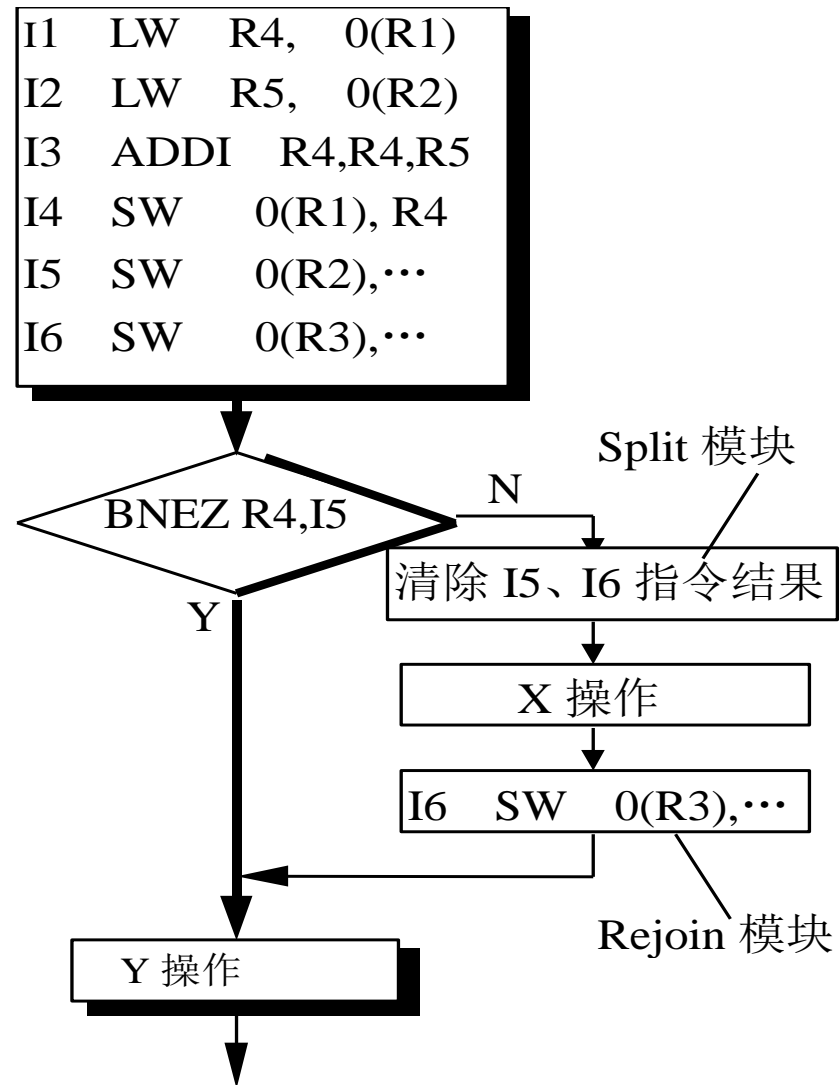
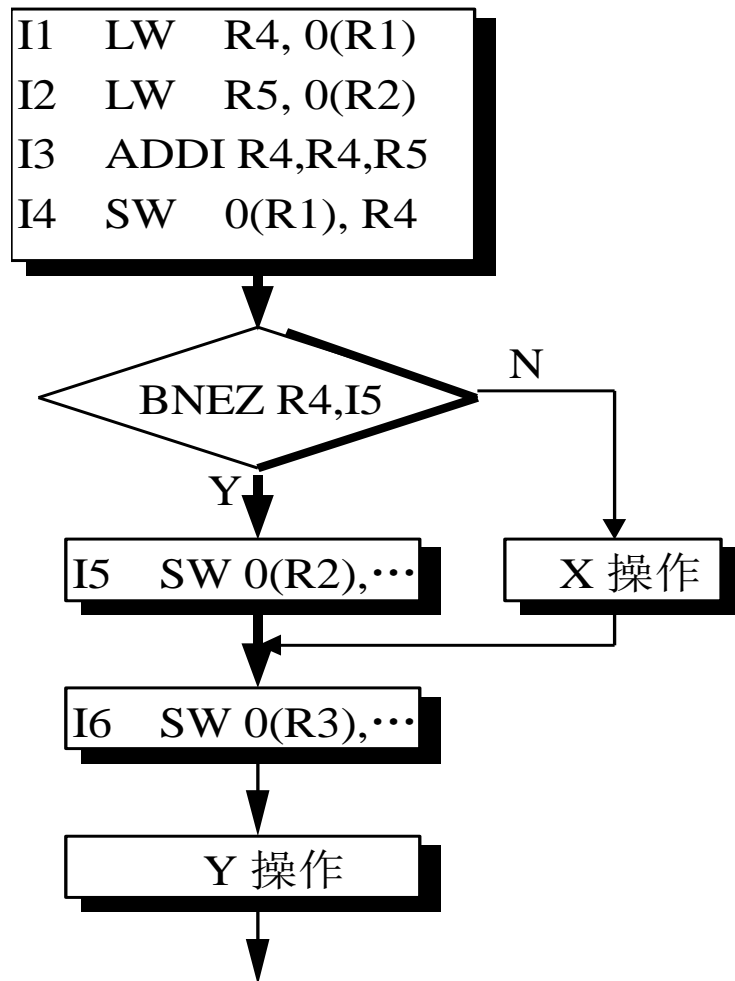


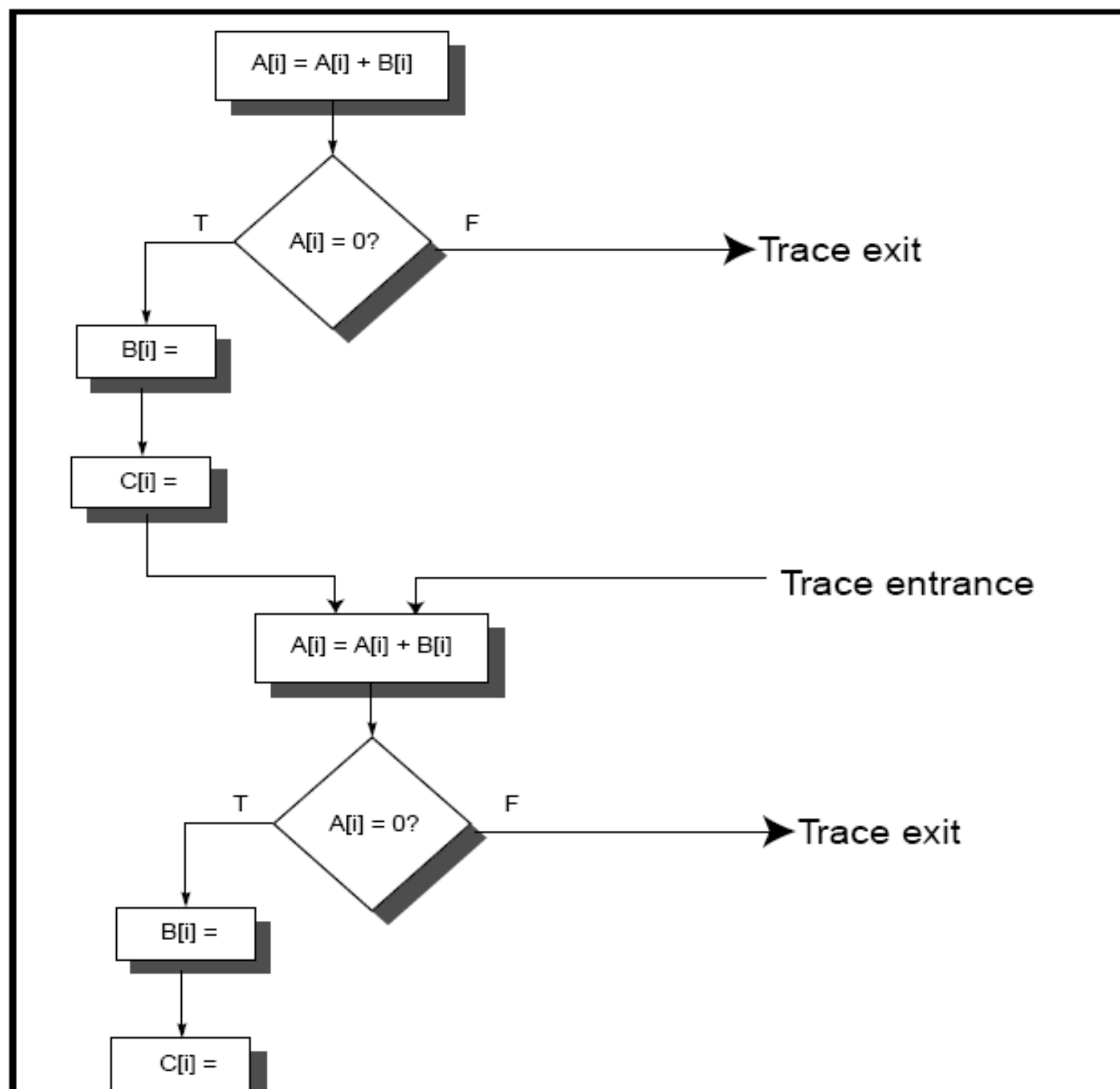
Example of Trace Scheduling

```
A[I]=A[I]+B[I];  
IF A[I]=0  
    B[I]= ... ;  
ELSE{  
    .....  
    X 操作;  
    .....  
}  
C[I]= ... ;  
    Y 操作;  
    .....
```



Example





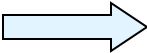
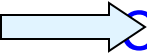
Superblock

- ❑ A form of extended basic block that are restricted to have a single entry point but allow multiple exits.
- ❑ Easier to compact a superblock than a trace.

Hardware Support for Exploiting ILP at compile time

❑ Conditional Instruction (predicated instruction)

- A conditional instruction refers to a condition which is evaluated as part of the instruction execution,
- Example:

If (A==0) {S=T}  BNEZ R1, L  CMOV R2,R3, R1
 ADDU R2, R3, R0
 L:

- the CPU always executes the instruction but writes the result only if the condition is met.
- A conditional branch changes a **control** dependence into a **data** dependence.

Example: improving scheduloing performance using Conditional Inst.

1st inst.	2nd inst
LW R1, 40(R2)	ADD R3, R4, R5
	ADD R6,R3, R7
BEQZ R10, L	
LW R8, 0(R10)	
LW R9, 0(R8)	

Using conditional instruction

□ LMC Rx, D(Ry), Rz

➤ Load Rx, D(Ry) if Rz != 0

➤ Nop if Rz == 0

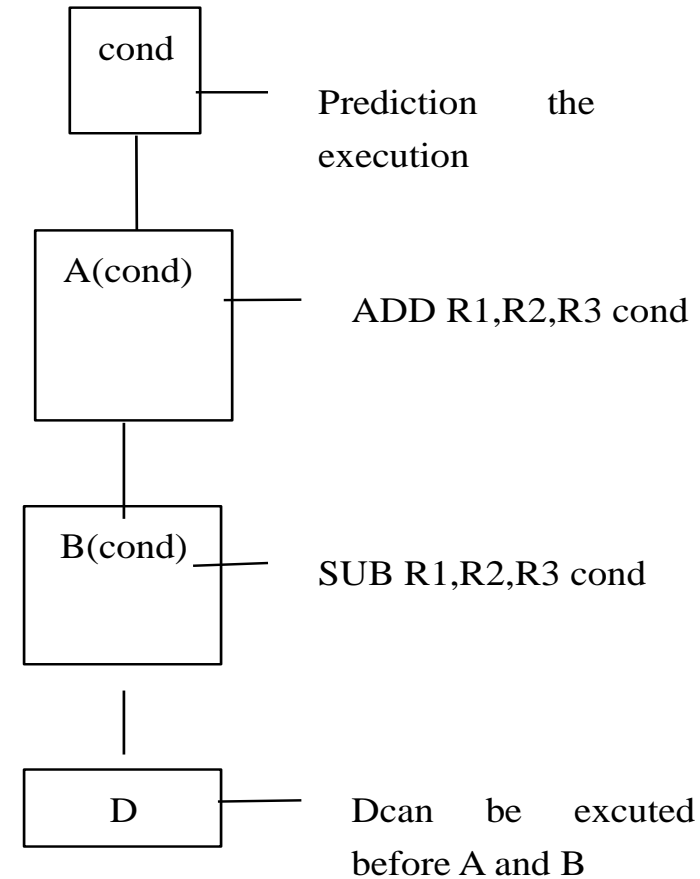
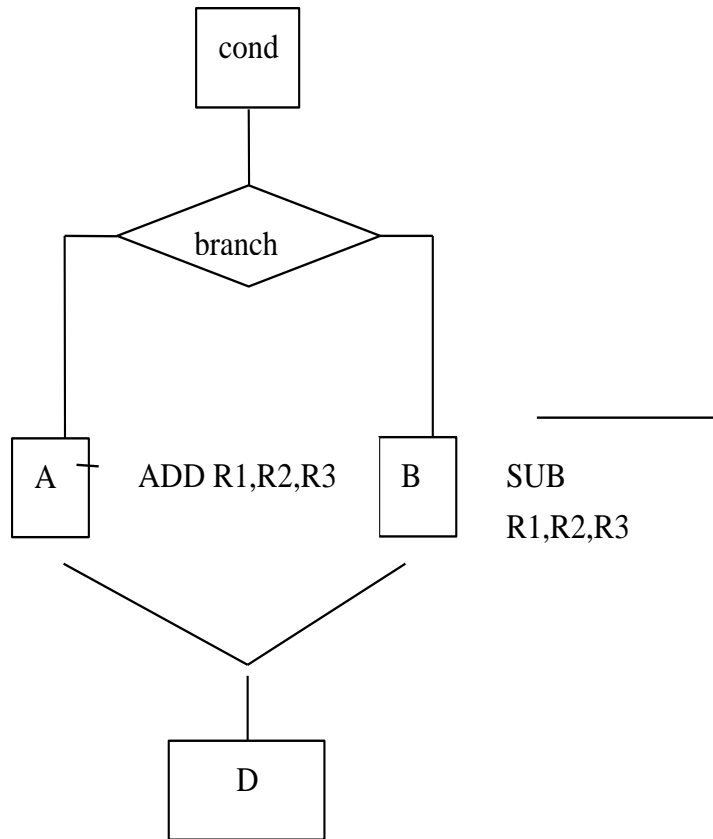
1 st instruction	2 nd instruction
LW R1, 40(R2)	ADD R3, R4, R5
LWC R8, 0(R10),R10	ADD R6,R3, R7
BEQZ R10, L	
LW R9, 0(R8)	

Conditional instructions in real computer

Alpha	HP PA	MIPS	SPARC
Conditiaonal move	All r-r inst. can turn the following instruction into nop if condition is false.	Conditiaonal move	Conditiaonal move

hyper block:

--super block + prediction



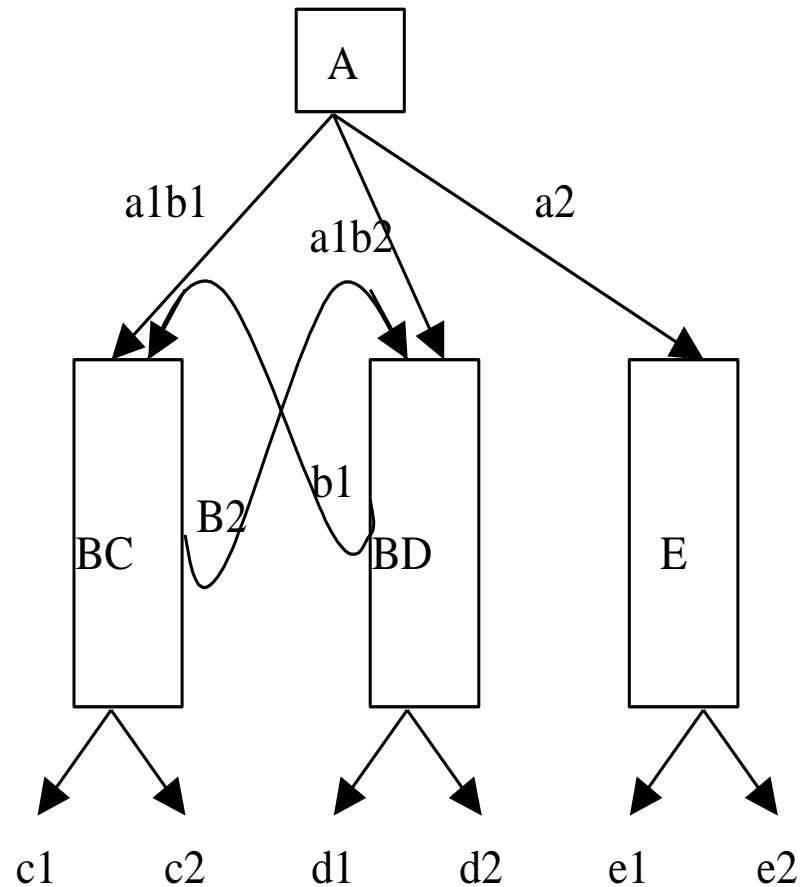
No branch ,No prediction No throw away ,But an extra source code field.

Block-Structure

□ Basic idea:

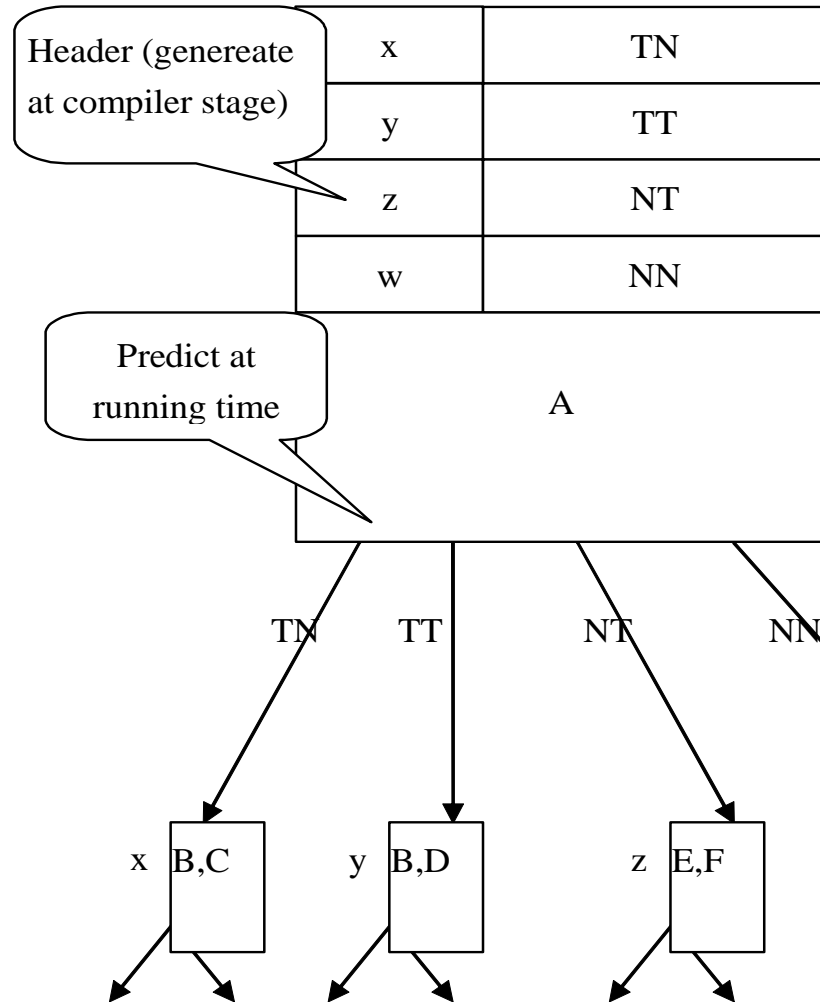
- **Block** substitute instruction as **atomic unit** running in computer.
 - One block is always executed in whole or entirely, but not just half or part of it.
- **Additional mechanism** need to solve the **exception** that happens in the middle of the basic block.
- If the intermediate result of the basic block is not used by other blocks, then we can use linkage within the block but not the register (that is used by the software) to connect the producer and consumer, which can save space and power.

Block-Structure



- ❑ Block is produced by a compiler as trace cache segment.
- ❑ If an exception happens at the middle of the block, then all the block work have done will be throw away and go to another block.
- ❑ The hardware can find more concurrency in one block.

Block-structure: Example



Header include information about:

- where to go when two branch predictor predicts(x,y,z,w)
- where to go when screw up

