Lecture 7

Very Large Instruction Word (VLIW)

- **VLIW – architectures and scheduling techniques (Ch. 3.5)**
  - ✓ VLIW architecture (3.5.2)
  - ✓ VLIW and loop unrolling (3.5.3)
  - ✓ VLIW and software pipelining (3.5.4)
  - ✓ Non-cyclic VLIW scheduling (3.5.5)
  - ✓ Predicated instructions (3.5.6)

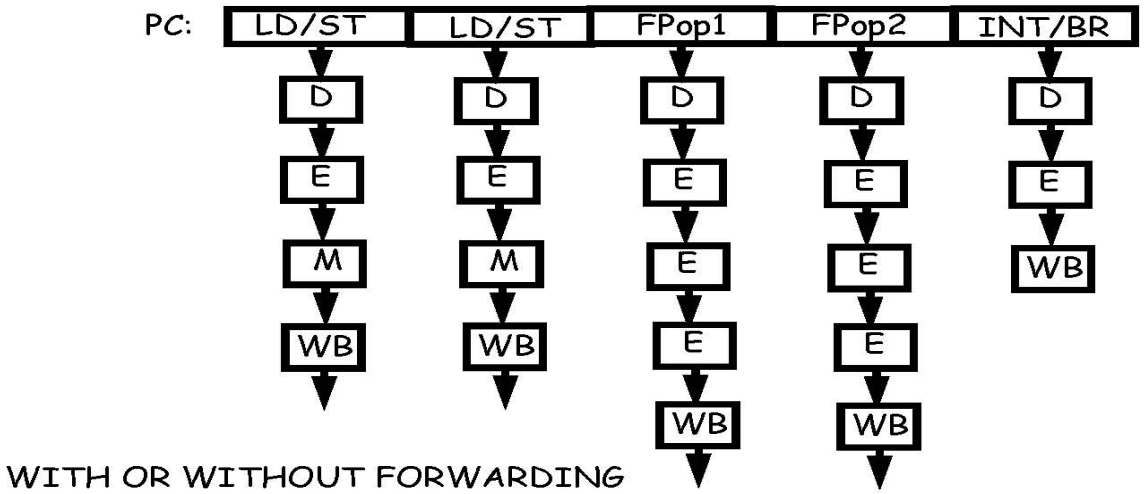
Static Scheduling: Revisiting Pipeline Design

Duality of Dynamic and Static Techniques

- **Instruction scheduling:** Compiler moves instructions. Same issues: data-flow and exception model
- **Software register renaming** for WAW and WAR hazards
- **Memory disambiguation** must be done by the compiler
- **Branch prediction** scheme: static prediction
- **Speculation:** speculate based on static branch prediction. Test dynamically and execute patch-up/recovery code if the speculation fails

Sometimes there is no need to speculate because the compiler knows the structure of the program (e.g. loops)

VLIW (Very-Long Instruction Word) Architectures



- Pipeline is simple with no hazard detection
- Compiler schedules instructions in “packets” or long instruction words (two memory, two floating-point and an integer operation in the example)
- Forwarding helps but is not needed

Program Example

PROGRAM EXAMPLE

INSTR. PRODUCING RESULT	INSTR. USING RESULT	LATENCY
FP ALU op	FP ALU op	2
LOAD DOUBLE	FP ALU op	1
STORE DOUBLE	LOAD DOUBLE	0
INT LOAD	INT ALU op/Branch	1
BRANCH DELAY SLOT	N/A	2

CONSIDER THE FOLLOWING PROGRAM:

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

which is compiled into:

```
Loop:   L.D     F0, 0(R1)
        ADD.D   F4, F0, F2
        S.D     F4, 0(R1)
        SUBI    R1,R1,#8
        BNE     R1,R2, Loop
        NOOP
```

Question:
How would each iteration be scheduled on the VLIW architecture?

Compiler scheduling

- Local (inside a basic block) or global (across basic blocks)
- Cyclic (loop unrolling or software pipelining) or non-cyclic (trace scheduling)

Loop Unrolling for VLIW

Unroll Loop Seven Times

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

Question:
How would the unrolled loop be scheduled on the VLIW architecture?

VLIW – Loop Unrolling

UNROLL LOOP 7 TIMES--VLIW PROGRAM:

Clock	MemOp1	MemOp2	FPOp1	FPOp2	Int/BROp
1	L.D F0,0(R1)	L.D F6,-8(R1)			
2	L.D F10,-16(R1)	L.D F14,-24(R1)			
3	L.D F18,-32(R1)	L.D F22,-40(R1)	ADD.D F4,F0,F2	ADD.D F8,F6,F2	
4	L.D F26,-48(R1)		ADD.D F12,F10,F2	ADD.D F16,F14,F2	
5			ADD.D F20,F18,F2	ADD.D F24,F22,F2	
6	S.D 0(R1),F4	S.D -8(R1),F8	ADD.D F28,F26,F2		SUBI R1,R1,#56
7	S.D 40(R1),F12	S.D 32(R1),F16			BNE R1,R2,Clock1
8	S.D 24(R1),F20	S.D 16(R1),F24			
9	S.D 8(R1),F28				

Issues with loop unrolling

- Code size
- Empty slots
- Register pressure
- Binary compatibility
- Limited scope for ILP exploitation

VLIW – Loop Unrolling

UNROLL LOOP 7 TIMES--VLIW PROGRAM:

Clock	MemOp1	MemOp2	FPOp1	FPOp2	Int/BROp
1	L.D F0,0(R1)	L.D F6,-8(R1)			
2	L.D F10,-16(R1)	L.D F14,-24(R1)			
3	L.D F18,-32(R1)	L.D F22,-40(R1)	ADD.D F4,F0,F2	ADD.D F8,F6,F2	
4	L.D F26,-48(R1)	UNUSED	ADD.D F12,F10,F2	ADD.D F16,F14,F2	UNUSED
5			ADD.D F20,F18,F2	ADD.D F24,F22,F2	
6	S.D 0(R1),F4	S.D -8(R1),F8	ADD.D F28,F26,F2		SUBI R1,R1,#56
7	S.D 40(R1),F12	S.D 32(R1),F16			BNE R1,R2,Clock1
8	S.D 24(R1),F20	S.D 16(R1),F24			
9	S.D 8(R1),F28				

Issues with loop unrolling

- Code size
- Empty slots
- Register pressure
- Binary compatibility
- Limited scope for ILP exploitation

Software Pipelining for VLIW

VLIW – Software Pipelining

```
Loop:   L.D     F0,0(R1)        O1
        ADD.D   F4,F0,F2        O2
        S.D     0(R1),F4        O3
```

1 LOOP ITERATION PER CLOCK

VLIW – Software Pipelining

```
Loop:   L.D     F0,0(R1)        O1
        ADD.D   F4,F0,F2        O2
        S.D     0(R1),F4        O3
```

1 LOOP ITERATION PER CLOCK

	ITE1	ITE2	ITE3	ITE4	ITE5	ITE6
INST1	O1					
INST2	--	O1				
INST3	O2	--	O1			
INST4	--	O2	--	O1		
INST5	--	--	O2	--	O1	
INST6	O3	--	--	O2	--	O1
INST7		O3	--	--	O2	--
INST8			O3	--	--	O2

KERNEL

Time

KERNEL CODE:(WE HAVE 5 ITERATIONS BETWEEN LD AND SD

VLIW – Software Pipelining

```
Loop:   L.D     F0,0(R1)        O1
        ADD.D   F4,F0,F2        O2
        S.D     0(R1),F4        O3
```

1 LOOP ITERATION PER CLOCK

	ITE1	ITE2	ITE3	ITE4	ITE5	ITE6
INST1	O1					
INST2	--	O1				
INST3	O2	--	O1			
INST4	--	O2	--	O1		
INST5	--	--	O2	--	O1	
INST6	O3	--	--	O2	--	O1
INST7		O3	--	--	O2	--
INST8			O3	--	--	O2

KERNEL

Time

KERNEL CODE:(WE HAVE 5 ITERATIONS BETWEEN LD AND SD

VLIW – Software Pipelining

```
Loop:   L.D     F0,0(R1)        O1
        ADD.D   F4,F0,F2        O2
        S.D     0(R1),F4        O3
```

1 LOOP ITERATION PER CLOCK

	ITE1	ITE2	ITE3	ITE4	ITE5	ITE6
INST1	O1					
INST2	--	O1				
INST3	O2	--	O1			
INST4	--	O2	--	O1		
INST5	--	--	O2	--	O1	
INST6	O3	--	--	O2	--	O1
INST7		O3	--	--	O2	--
INST8			O3	--	--	O2

KERNEL

Prolog

Time

KERNEL CODE:(WE HAVE 5 ITERATIONS BETWEEN LD AND SD

Clock	MemOp1	MemOp2	FPOp1	FPOp2	Int/BROp
1	L.D F0,0(R1)	S.D 40(R1),F4	ADD.D F4,F0,F2	NOOP	LOOPBR R1,R2,CLK1

Question:
What instructions are contained in the epilog?

VLIW – Software Pipelining

```
Loop:   L.D     F0,0(R1)        O1
        ADD.D   F4,F0,F2        O2
        S.D     0(R1),F4        O3
```

1 LOOP ITERATION PER CLOCK

Time

	ITE1	ITE2	ITE3	ITE4	ITE5	ITE6
INST1	O1					
INST2	--	O1				
INST3	O2	--	O1			
INST4	--	O2	--	O1		
INST5	--	--	O2	--	O1	
INST6	O3	--	--	O2	--	O1
INST7		O3	--	--	O2	--
INST8			O3	--	--	O2

KERNEL

KERNEL CODE:(WE HAVE 5 ITERATIONS BETWEEN LD AND SD

Clock	MemOp1	MemOp2	FPOp1	FPOp2	Int/BROp
1	L.D F0,0(R1)	S.D 40(R1),F4	ADD.D F4,F0,F2	NOOP	LOOPBR R1,R2,CLK1

VLIW – Software Pipelining

```
Loop:   L.D     F0,0(R1)        O1
        ADD.D   F4,F0,F2        O2
        S.D     0(R1),F4        O3
```

1 LOOP ITERATION PER CLOCK

Time

	ITE1	ITE2	ITE3	ITE4	ITE5	ITE6
INST1	O1					
INST2	--	O1				
INST3	O2	--	O1			
INST4	--	O2	--	O1		
INST5	--	--	O2	--	O1	
INST6	O3	--	--	O2	--	O1
INST7		O3	--	--	O2	--
INST8			O3	--	--	O2

KERNEL

KERNEL CODE:(WE HAVE 5 ITERATIONS BETWEEN LD AND SD

Clock	MemOp1	MemOp2	FPOp1	FPOp2	Int/BROp
1	L.D F0,0(R1)	S.D 40(R1),F4	ADD.D F4,F0,F2	NOOP	LOOPBR R1,R2,CLK1

VLIW – Software Pipelining

```
Loop:   L.D     F0,0(R1)        O1
        ADD.D   F4,F0,F2        O2
        S.D     0(R1),F4        O3
```

1 LOOP ITERATION PER CLOCK

Time

	ITE1	ITE2	ITE3	ITE4	ITE5	ITE6
INST1	O1					
INST2	--	O1				
INST3	O2	--	O1			
INST4	--	O2	--	O1		
INST5	--	--	O2	--	O1	
INST6	O3	--	--	O2	--	O1
INST7		O3	--	--	O2	--
INST8			O3	--	--	O2

KERNEL

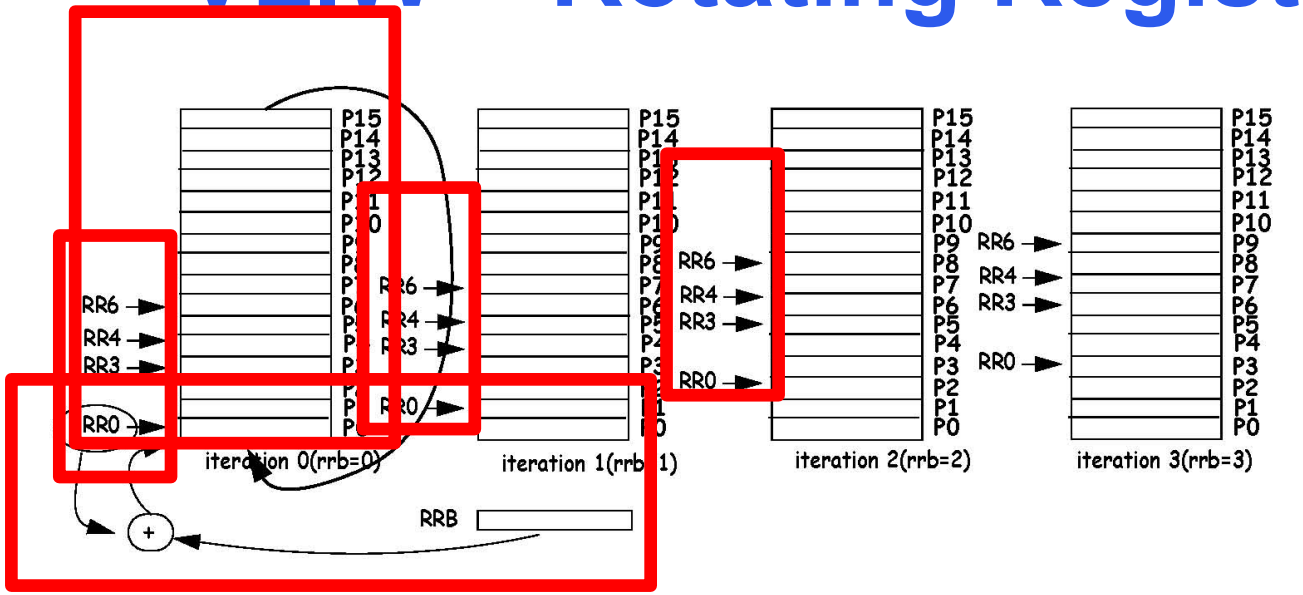
KERNEL CODE:(WE HAVE 5 ITERATIONS BETWEEN LD AND SD

Clock	MemOp1	MemOp2	FPOp1	FPOp2	Int/BROp
1	L.D F0,0(R1)	S.D 40(R1),F4	ADD.D F4,F0,F2	NOOP	LOOPBR R1,R2,CLK1

Data hazards

- RAW hazards are handled correctly
- For WAR hazards, use rotating registers (register renaming technique)

VLIW – Rotating Registers




```
LD F0,0(R1)
ADD F4,F0,F2
SD F4,0(R1)
```

- **Two iterations between LD and ADD**; RR6 and RR4 point to same physical register
- **Three iterations between ADD and SD**; RR3 and RR0 point to same register

VLIW – Slot Conflicts

- Restrict the number of slots: 1 LD/ST, 1 FP, 1 BR/INT
- Two LD/ST per iterations so two instructions for kernel

VLIW – Slot Conflicts

- Restrict the number of slots: 1 LD/ST, 1 FP, 1 BR/INT
- Two LD/ST per iterations so two instructions for kernel

	ITE1	ITE2	ITE3	ITE4
INST1	O1			
INST2	--			
INST3	O2	O1		
INST4	--	--		
INST5	--	O2	O1	
INST6	O3	--	--	
INST7		--	O2	O1
INST8		O3	--	--

KERNEL

VLIW – Slot Conflicts

- Restrict the number of slots: 1 LD/ST, 1 FP, 1 BR/INT
- Two LD/ST per iterations so two instructions for kernel

	ITE1	ITE2	ITE3	ITE4
INST1	O1			
INST2	--			
INST3	O2	O1		
INST4	--	--		
INST5	--	O2	O1	
INST6	O3	--	--	
INST7		--	O2	O1
INST8		O3	--	--

KERNEL

KERNEL CODE: STORE IS TWO ITERATIONS BEHIND THE LOAD; SUB MOVED UP

Clock	MemOp1	FPOp1	Int/BROp
1	L.D F0,0(R1)	ADD.D F4,F0,F2	SUB R1
2	S.D 24(R1),F4	NOOP	LOOPBR

VLIW–Loop Carried Dependencies 1(3)

Loop carried dependency = dependency across loop iterations

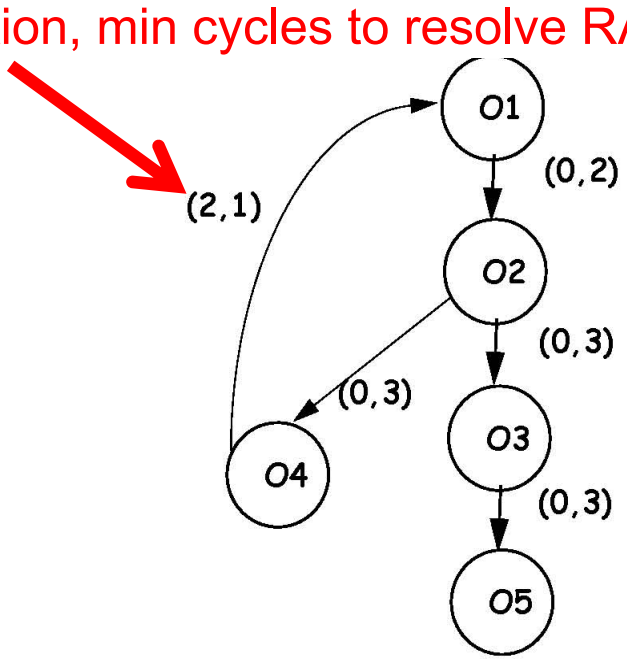
Dependency spans two loop iterations:

Dependency is through memory; rotating registers do not help

Let’s look at the data dependency graph!

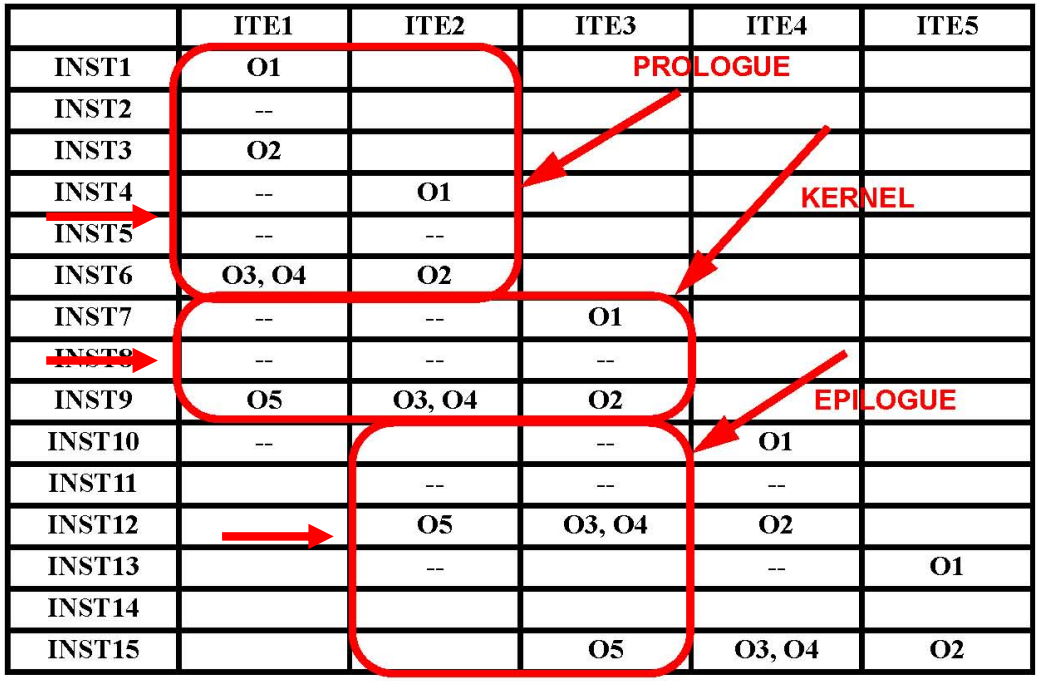
VLIW–Loop Carried Dependencies 2(3)

```
Loop:   L.D F0, 0(R2)        O1
        ADD.D F3, F0, F1     O2
        ADD.D F4, F3, F1     O3
        S.D F3, -16(R2)      O4
        S.D F4, 0(R3)        O5
        SUBI R2, R2, 8
        SUBI R3, R3, 8
        BNE R2, R4, Loop
```



- The cycle in the graph takes 6 cycles and spans two iterations; 3 cycles at least per iteration

VLIW–Loop Carried Dependencies 3(3)



VLIW–Loop Carried Dependencies 3(3)

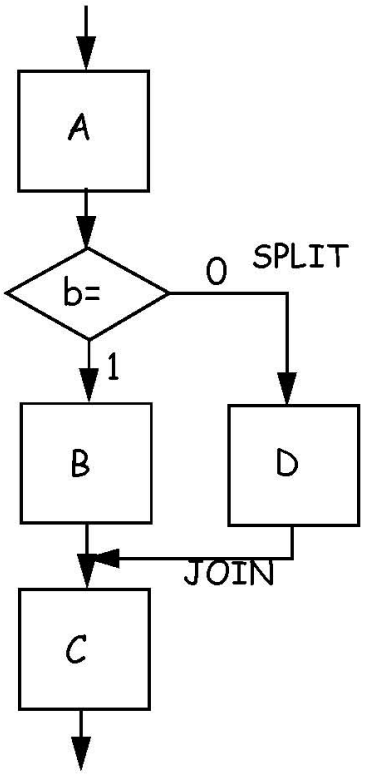
	ITE1	ITE2	ITE3	ITE4	ITE5
INST1	O1		PROLOGUE		
INST2	--				
INST3	O2				
INST4	--	O1		KERNEL	
INST5	--	--			
INST6	O3, O4	O2			
INST7	--	--	O1		
INST8	--	--	--		
INST9	O5	O3, O4	O2	EPILOGUE	
INST10	--		--	O1	
INST11		--	--	--	
INST12		O5	O3, O4	O2	
INST13		--		--	O1
INST14					
INST15			O5	O3, O4	O2

KERNEL (USING 2 LD/SD, 2 FP AND 1 BR/INT SLOTS)

Clock	MemOp1	MemOp2	FPOp1	FPOp2	Int/BROp
1	NOOP	L.D F0,0(R2)	NOOP	NOOP	SUB R2
2	NOOP	NOOP	NOOP	NOOP	SUB R3
3	S.D F3,-16(R2)	S.D F4,0(R3)	ADD.D F3,F0,F1	ADD.D F4,F3,F1	LOOPBR R2,R4,CLK1

Non-Cyclic Scheduling

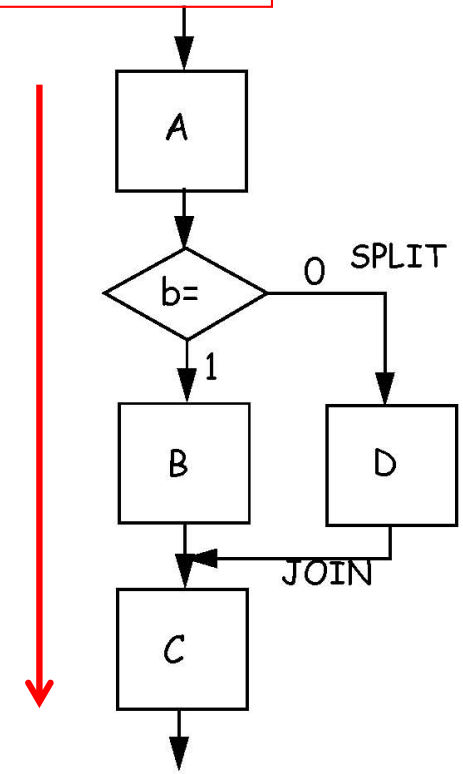
Non-Cyclic Scheduling



- Most likely path (A, B, C in the example) is established through ***profiling***
- This path (A, B, C), called ***trace,*** forms a larger basic block of code for instruction scheduling
- Instruction scheduling respects RAW dependencies but can ignore control dependencies
- Must fix the execution on branch misspeculation so that misspeculated trace (A, D, C) is correctly executed.

Example

Most likely trace



Original code

```
  LW R4,0(R1)
  ADDI R6,R4,#1
/* block A*/
  BEQ R5,R4,LAB
  LW R6,0(R2)
/*block B*/
/*block D* empty/
  LAB: SW R6,0(R1)
```

Trace schedule

```
  LW R4,0(R1)
  ADDI R6,R4,#1
  LW R6,0(R2)
  BEQ R5,R4,LAB1
LAB2: SW R6,0(R1)
/* jump to second trace if
prediction is wrong */
….
LAB1: ADDI R6,R4,#1
      J LAB2
```

Optimized trace

```
  LW R4,0(R1)
  LW R6,0(R2)
  BEQ R5,R4,LAB1
LAB2: SW R6,0(R1)
/* jump to second trace if
prediction is wrong */
….
LAB1: ADDI R6,R4,#1
      J LAB2
```

Predicated Execution

Predicated Instructions

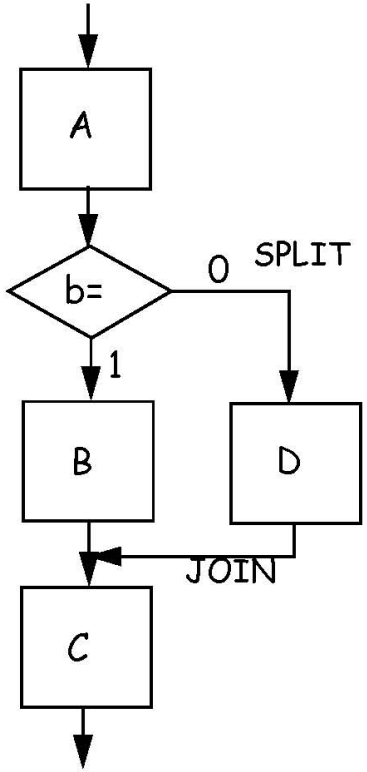
- Trace scheduling works well if branches are highly biased (one trace is considerably more likely than another)

Predicated instruction = conditionally executed instruction

Example 1) CLWZ R1,0(R2),R3; if (R3)==0 then LW R1,0(R2)

- Only executed if condition is met; other No Operation
- Predication can be applied to any instruction
- Needs an additional operand – a **predicate** register
- Longer instruction not a problem in VLIW

Example – Predicated Execution



Original code

```
    LW R4,0(R1)
    ADDI R6,R4,#1
    BEQ R5,R4,LAB
    LW R6,0(R2)
LAB: SW R6,0(R1)
```

Predicated code

```
    LW R4,0(R1)
    ADDI R6,R4,#1
    SUB R3,R5,R4
    CLWZ R6,0(R2),R3
    SW R6,0(R1)
```