

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)

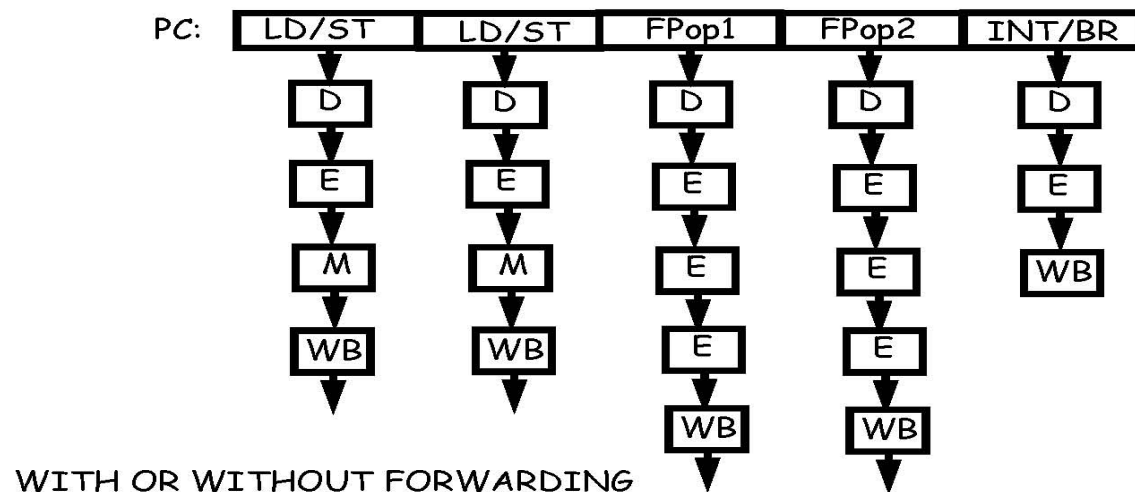
Static Scheduling: Revisiting Pipeline Design (Ch 3.5.2)

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

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
```

Compiler scheduling

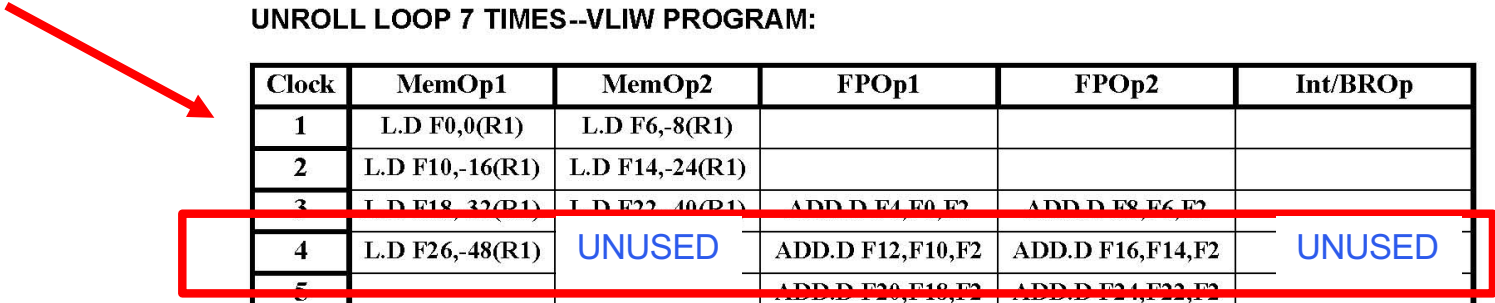
- **Local** (inside a basic block) or **global** (across basic blocks)
- **Cyclic** (loop unrolling or software pipelining) or **non-cyclic** (trace scheduling)

Loop Unrolling for VLIW

(Ch 3.5.3)

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

Quiz 7.1

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				

Which of the following statements are correct assuming that all instructions take a single cycle to execute

- a) IPC = 5
- b) IPC = 23/9
- c) The number of unused slots is 21

Software Pipelining for VLIW

(Ch 3.5.4)

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

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 (indicated by a red arrow pointing to the O1 in INST5, ITE5)

Kernel Code (indicated by a red circle around INST6, ITE1 to INST6, ITE6): O3, --, --, O2, --, O1

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 O(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 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

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 (indicated by a red arrow pointing to the diagonal from INST3 to INST6)

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 (indicated by a red arrow pointing to the O1 in INST5)

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 (indicated by a red arrow pointing to the O1 in INST5)

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 0(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 (indicated by a red arrow pointing to the O1 in INST5)

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 0(R1),F4	ADD.D F4,F0,F2	NOOP	LOOPBR R1,R2,CLK1

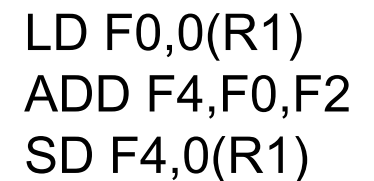
Data hazards

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

The diagram illustrates the iterative process of the RRB algorithm. It shows four iterations: iteration 0 (rrb=0), iteration 1 (rrb=1), iteration 2 (rrb=2), and iteration 3 (rrb=3). Each iteration consists of a 16x16 matrix of processors (P0 to P15) and a set of read requests (RR0 to RR6). Red boxes highlight the active read requests in each iteration. A feedback loop shows the RRB value being updated from iteration 3 back to iteration 0.

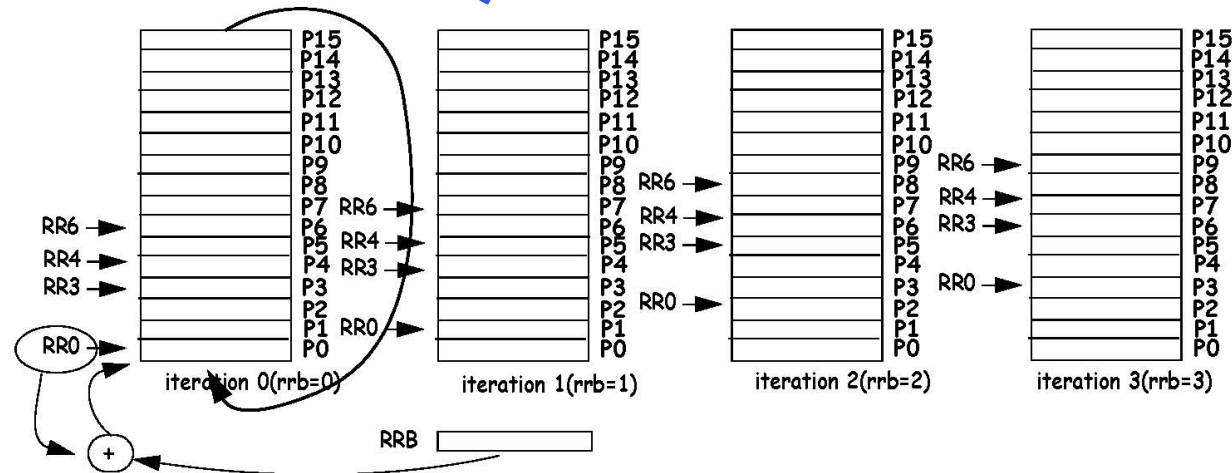
- Iteration 0 (rrb=0):** Active read requests are RR0, RR3, RR4, and RR6.
- Iteration 1 (rrb=1):** Active read requests are RR0, RR3, RR4, and RR6.
- Iteration 2 (rrb=2):** Active read requests are RR0, RR3, RR4, and RR6.
- Iteration 3 (rrb=3):** Active read requests are RR0, RR3, RR4, and RR6.

The RRB value is updated from iteration 3 back to iteration 0, as indicated by the feedback loop.



- CHALMERS**
Chalmers University of Technology

Quiz 7.2



Let RR6 point to physical register X in iteration Y. There are 18 physical registers. Which of the following statements are correct?

- a) RR0 points to X after Y+13 iterations
- b) RR5 points to X after Y+17 iterations
- c) RR0 points to X after Y+6 iterations
- d) RR5 points to X after Y+1 iterations

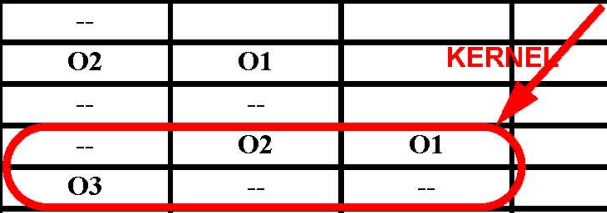
VLIW – Slot Conflicts

- 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	--	--



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 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

5

VLIW–Loop Carried Dependencies 1(3)

Loop carried dependency = dependency across loop iterations

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

Dependency spans two loop iterations:

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	

Dependency is through memory; rotating registers do not help

Let's look at the data dependency graph!

Loop Carried Dependencies

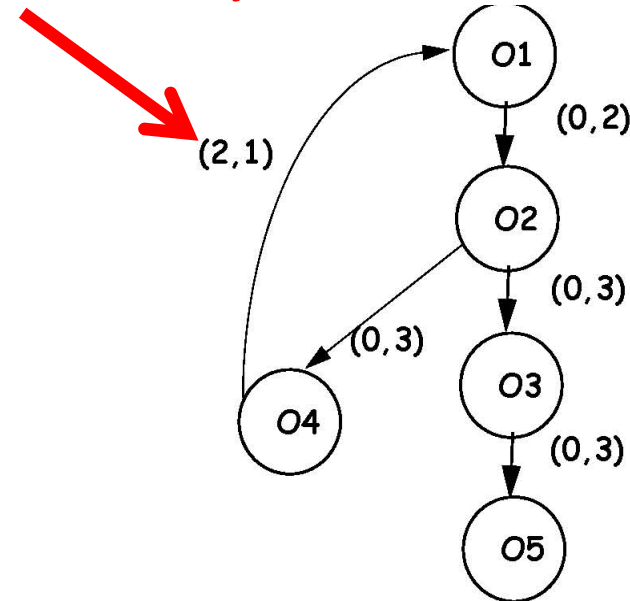
VLIW-Loop Carried Dependencies 2(3)

Loop:

```
L.D F0, 0(R2)
ADD.D F3, F0, F1
ADD.D F4, F3, F1
S.D F3, -16(R2)
S.D F4, 0(R3)
SUBI R2, R2, 8
SUBI R3, R3, 8
BNE R2, R4, Loop
```

O1
O2
O3
O4
O5

(iteration, min cycles to resolve RAW)



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

VLIW-Loop Carried Dependencies 3(3)

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

PROLOGUE

KERNEL

EPILOGUE

VLIW-Loop Carried Dependencies 3(3)

	ITE1	ITE2	ITE3	ITE4	ITE5
INST1	O1			PROLOGUE	
INST2	--				
INST3	O2				
INST4	--	O1			
INST5	--	--			
INST6	O3, O4	O2			
INST7	--	--	O1		
INST8	--	--	--		
INST9	O5	O3, O4	O2		
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

Quiz 7.3

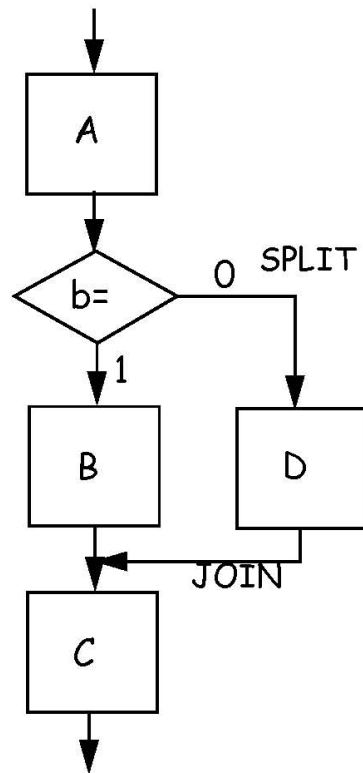
	ITE1	ITE2	ITE3	ITE4	ITE5
INST1	O1			PROLOGUE	
INST2	--				
INST3	O2				
INST4	--	O1			
INST5	--	--			
INST6	O3, O4	O2			
INST7	--	--	O1		
INST8	--	--	--		
INST9	O5	O3, O4	O2		
INST10	--		--	O1	
INST11		--	--	--	
INST12		O5	O3, O4	O2	
INST13		--		--	O1
INST14					
INST15			O5	O3, O4	O2

How many clocks does it take to execute all instructions in each of the original iterations?

- a) 3
- b) 6
- c) 9

Non-Cyclic Scheduling (Ch 3.5.5)

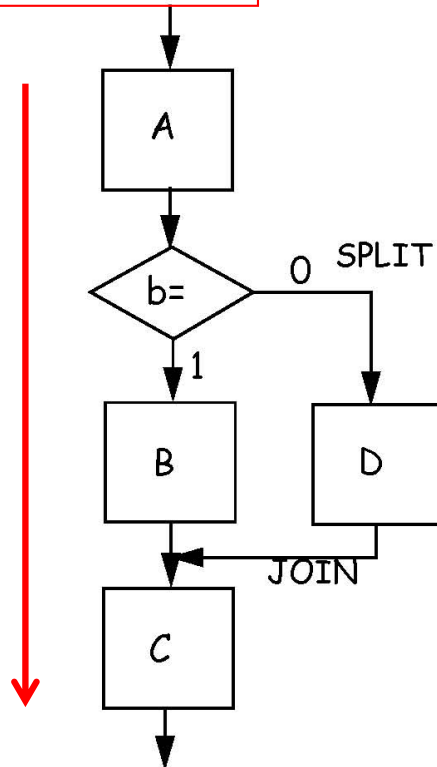
Non-Cyclic Scheduling



- 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
  
```

Quiz 7.4

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
```

Which of the following statements are correct?

- a) The “non-taken” trace consists of 1 more instruction in the original code compared to the optimized trace
- b) The “non-taken” trace consists of the same number of instructions in the original code and the optimized trace
- c) The “taken” optimized trace executes two more instructions than in the original code

Predicated Execution

(Ch 3.5.6)

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

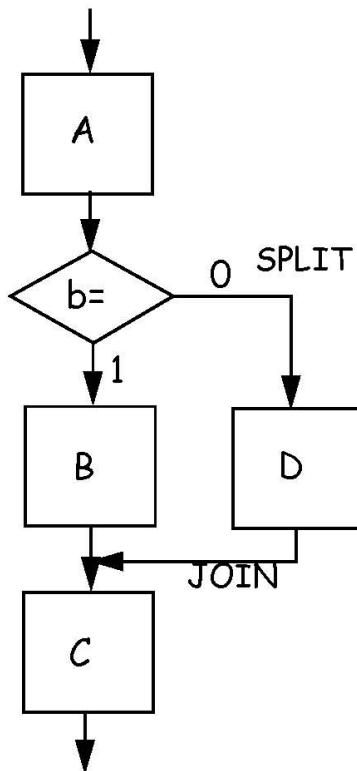
Example – Predicated Execution

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
CLWNZ R6,0(R2),R3
SW R6,0(R1)
```



What you should know by now

- **VLIW architectures**
 - Parallel simple pipelines
 - No support for dynamic scheduling
 - Assumes compiler does static scheduling
- **VLIW and loop unrolling**
 - Challenge is to fill operation slots
- **VLIW and software pipelining**
 - Renaming using rotating registers
 - Impact of slot conflicts
 - Impact of loop-carried dependencies
- **Trace scheduling**
- **Predicated (conditional) instructions**