Data Dependences

For example, consider the following MIPS code sequence that increments a vector of values in memory (starting at 0(R1) and with the last element at 8(R2)) by a scalar in register F2. (For simplicity, throughout this chapter, our examples ignore the effects of delayed branches.)

```
Loop: L.D F0,0(R1)        ;F0=array element
ADD.D F4,F0,F2            ;add scalar in F2
S.D F4,0(R1)              ;store result
DADDUI R1,R1,#-8          ;decrement pointer 8 bytes
BNE R1,R2,LOOP            ;branch R1!=R2
```

The data dependences in this code sequence involve both floating-point data:

```
Loop: L.D F0,0(R1)          ;F0=array element

ADD.D F4,F0,F2              ;add scalar in F2

S.D F4,0(R1)                ;store result

DADDIU R1,R1,#-8            ;decrement pointer;
                            8 bytes (per DW)
BNE R1,R2,Loop              ;branch R1!=R2
```

Control Dependences

For example, consider the following code fragment:

DADDU R1,R2,R3

BEQZ R4,L

DSUBU R1,R5,R6

L: ...

OR R7,R1,R8

In this example, the value of R1 used by the OR instruction depends on Whether the branch is taken or not. Data dependence alone is not sufficient to preserve correctness. The OR instruction is data dependent on both the DADDU and

DSUBU instructions, but preserving that order alone is insufficient for Correct execution.

Instead, when the instructions execute, the data flow must be preserved: If the branch is not taken, then the value of R1 computed by the DSUBU should be used by the OR, and, if the branch is taken, the value of R1 computed by the DADDU should be used by the OR. By preserving the control dependence of the OR on the branch, we prevent an illegal change to the data flow. For similar reasons,

the DSUBU instruction cannot be moved above the branch.

Speculation, which helps with the exception problem, will also allow us to lessen the impact of the control dependence while still maintaining the data flow

Sometimes we can determine that violating the control dependence cannot
affect either the exception behavior or the data flow. Consider the following code sequence:

DADDU R1,R2,R3
BEQZ R12,skip
DSUBU R4,R5,R6
DADDU R5,R4,R9
skip: OR R7,R8,R9

Suppose we knew that the register destination of the DSUBU instruction (R4) was unused after the instruction labeled skip. (The property of whether a value will be used by an upcoming instruction is called *liveness*.)

If R4 were unused, then changing the value of R4 just before the branch would not affect the data flow since R4 would be *dead (rather than live) in the code region after skip. Thus, if* R4 were dead and the existing DSUBU instruction could not generate an Exception (other than those from which the Processor resumes the same process), we could move the DSUBU instruction before the branch, since the data flow cannot be affected by this change.

If the branch is taken, the DSUBU instruction will execute and will be useless, but it will not affect the program results.

Basic Pipeline Scheduling and Loop Unrolling

Example:

for (i=999; i>=0; i=i–1)

x[i] = x[i] + s;

Latencies of FP operations used in this chapter.

Instruction producing result	Instruction using result	Latency in clock cycles
FP ALU op	Another FP ALU op	3
FP ALU op	Store double	2
Load double	FP ALU op	1
Load double	Store double	0

MIPS Code

The straightforward MIPS code, not scheduled for the pipeline, looks like this:

```
Loop: L.D F0,0(R1)        ;F0=array element
ADD.D F4,F0,F2            ;add scalar in F2
S.D F4,0(R1)              ;store result
DADDUI R1,R1,#-8          ;decrement pointer
                          ;8 bytes (per DW)
BNE R1,R2,Loop            ;branch R1!=R2
```

Example

Show how the loop would look on MIPS, both scheduled and unscheduled, including any stalls or idle clock cycles. Schedule for delays from floating-point operations, but remember that we are ignoring delayed branches.

Answer

Without any scheduling, the loop will execute as follows, taking nine cycles:

			Clock cycle issued
Loop:	L.D	F0,0(R1)	1
	Stall		2
	ADD.D	F4,F0,F2	3
	Stall		4
	Stall		5
	S.D	F4,0(R1)	6
	DADDUI	R1,R1,#-8	7
	Stall		8
			9

We can schedule the loop to obtain only two stalls and reduce the time to seven cycles:

			Clock cycle issued
Loop:	L.D	F0,0(R1)	1
	ADD.D	F4,F0,F2	2
	DADDUI	R1,R1,#-8	3
	Stall		4
	Stall		5
	S.D	F4,0(R1)	6
	BNE	R1,R2,Loop	7

The stalls after ADD.D are for use by the S.D.

Example

Show our loop unrolled so that there are four copies of the loop body, assuming R1 – R2 (that is, the size of the array) is initially a multiple of 32, which means that the number of loop iterations is a multiple of 4. Eliminate any obviously redundant computations and do not reuse any of the registers

Answer:

Loop:	L.D	F0,0(R1)	
	ADD.D	F4,F0,F2	
	S.D	F4,0(R1)	;drop DADDUI & BNE
	L.D	F6,-8(R1)	
	ADD.D	F8,F6,F2	
	S.D	F8,-8(R1)	;drop DADDUI & BNE
	L.D	F10,-16(R1)	
	ADD.D	F12,F10,F2	
	S.D	F12,-16(R1)	;drop DADDUI & BNE
	L.D	F14,-24(R1)	
	ADD.D	F16,F14,F2	
	S.D	F16,-24(R1)	
	DADDUI	R1,R1,#-32	
	BNE	R1,R2,Loop	

Explanation:

The result after merging the DADDUI instructions and dropping the unnecessary BNE operations that are duplicated during unrolling. Note that R2 must now be set so that 32(R2) is the Starting address of the last four elements.

We have eliminated three branches and three decrements of R1. The addresses on the loads and stores have been compensated to allow the DADDUI instructions on R1 to be merged. This optimization may seem trivial, but it is not; it requires symbolic substitution and simplification. Symbolic substitution and implification will rearrange expressions so as to allow constants to be collapsed, allowing an expression such as ((*i* + 1) + 1) *to be rewritten as (i + (1 + 1)) and then simplified* to (*i* + 2).

Explanation: (cont..)

Without scheduling, every operation in the unrolled loop is followed by a dependent operation and thus will cause a stall. This loop will run in 27 clock cycles each LD has 1 stall, each ADDD 2, the DADDUI 1, plus 14 instruction issue cycles—or 6.75 clock cycles for each of the four elements, but it can be scheduled to improve performance significantly. Loop unrolling is normally done early in the compilation process, so that redundant computations can be exposed and eliminated by the optimizer.

Example Show the unrolled loop in the previous example (refer slide no: 13) after it has been scheduled for the pipeline with the latencies from Figure 3.2.

Answer:

Loop:	L.D	F0,0(R1)
	L.D	F6,-8(R1)
	L.D	F10,-16(R1)
	L.D	F14,-24(R1)
	ADD.D	F4,F0,F2
	ADD.D	F8,F6,F2
	ADD.D	F12,F10,F2
	ADD.D	F16,F14,F2
	S.D	F4,0(R1)
	S.D	F8,-8(R1)
	DADDUI	R1,R1,#-32
	S.D	F12,16(R1)
	S.D	F16,8(R1)
	BNE	R1,R2,Loop

The execution time of the unrolled loop has dropped to a total of 14 clock cycles, or 3.5 clock cycles per element, compared with 9 cycles per element before any unrolling or scheduling and 7 cycles when scheduled but not unrolled.

Correlating Branch Predictors

Example:

Consider a small code fragment from the eqn tott benchmark, a member of early SPEC benchmark suites that displayed particularly bad branch prediction behavior:

```
if (aa==2)
aa=0;
if (bb==2)
bb=0;
if (aa!=bb)
 {
```

Here is the MIPS code that we would typically generate for this code fragment
assuming that aa and bb are assigned to registers R1 and R2:

```
        DADDIU    R3,R1,#–2
        BNEZ      R3,L1 ;          branch b1     (aa!=2)
        DADD      R1,R0,R0 ;                       aa=0
L1:     DADDIU    R3,R2,#–2
        BNEZ      R3,L2 ;          branch b2     (bb!=2)
        DADD      R2,R0,R0 ;                       bb=0
L2:     DSUBU     R3,R1,R2 ;                   R3=aa-bb
        BEQZ      R3,L3 ;          branch b3    (aa==bb)
```

Explanation:

Let’s label these branches b1, b2, and b3. The key observation is that the behavior of branch b3 is correlated with the behavior of branches b1 and b2. Clearly, if branches b1 and b2 are both not taken (i.e., if the conditions both evaluate to true and aa and bb are both assigned 0), then b3 will be taken, since aa and bb are clearly equal. A predictor that uses only the behavior of a single branch to predict the outcome of that branch can never capture this behavior

Example

How many bits are in the (0,2) branch predictor with 4K entries? How many entries are in a (2,2) predictor with the same number of bits?

Answer

The predictor with 4K entries has

The number of bits in an *(m,n)* predictor is

$2^m \times n \times$ *Number of prediction entries selected by the branch address*

$2^0 \times 2 \times 4K = 8K$ bits

How many branch-selected entries are in a (2,2) predictor that has a total of 8K bits in the prediction buffer?

We know that $2^2 \times 2 \times$ Number of prediction entries selected by the branch = 8K

Hence, the number of prediction entries selected by the branch =1K.

Dynamic Scheduling: Examples and the Algorithm

Example

Show what the information tables look like for the following code sequence when only the first load has completed and written its result:

1.	L.D	F6,32(R2)
2.	L.D	F2,44(R3)
3.	MUL.D	F0,F2,F4
4.	SUB.D	F8,F2,F6
5.	DIV.D	F10,F0,F6
6.	ADD.D	F6,F8,F2

Instruction status			
Instruction	**Issue**	**Execute**	**Write result**
L.D F6,32(R2)	√	√	√
L.D F2,44(R3)	√	√	
MUL.D F0,F2,F4	√		
SUB.D F8,F2,F6	√		
DIV.D F10,F0,F6	√		
ADD.D F6,F8,F2	√		

Reservation Stations							
Name	Busy	Op	Vj	Vk	Qj	Qk	A
Load1	No						
Load2	Yes	Load					44 + Regs[R3]
Add1	Yes	SUB		Mem[32 + Regs[R2]]	Load2		
Add2	Yes	ADD			Add1	Load2	
Add3	No						
Mult1	Yes	MUL		Regs [F4]	Load2		
Mult2	Yes	DIV		Mem [32 + Regs [R2]]	Mult1		

FP Register Status									
Field	F0	F2	F4	F6	F8	F10	F12	…..	F30
Qi	Mult1	Load2		Add2	Add1	Mult2			

Tomasulo's Algorithm

Example

Using code segment given below , show what the status tables look like when the MUL.D is ready to write its result.

1. L.D	F6,32(R2)
2. L.D	F2,44(R3)
3. MUL.D	F0,F2,F4
4. SUB.D	F8,F2,F6
5. DIV.D	F10,F0,F6
6. ADD.D	F6,F8,F2

Instruction status			
Instruction	**Issue**	**Execute**	**Write result**
L.D F6,32(R2)	√	√	√
L.D F2,44(R3)	√	√	√
MUL.D F0,F2,F4	√	√	
SUB.D F8,F2,F6	√	√	√
DIV.D F10,F0,F6	√		
ADD.D F6,F8,F2	√	√	√

Reservation Stations							
Name	Busy	Op	Vj	Vk	Qj	Qk	A
Load1	No						
Load2	No						
Add1	No						
Add2	No						
Add3	No						
Mult1	Yes	MUL	Mem [44 + Regs [R3]]	Regs [F4]			
Mult2	Yes	DIV		Mem [32 + Regs [R2]]	Mult1		

FP Register Status									
Field	F0	F2	F4	F6	F8	F10	F12	…..	F30
Qi	Mult1					Mult2			

Tomasulo's Algorithm: A Loop-Based Example

To understand the full power of eliminating WAW and WAR hazards through dynamic renaming of registers, we must look at a loop. Consider the following simple sequence for multiplying the elements of an array by a scalar in F2:

```
Loop:   L.D         F0,0(R1)
        MUL.D       F4,F0,F2
        S.D         F4,0(R1)
        DADDIU      R1,R1,-8
        BNE         R1,R2,Loop;        branches if R1¦R2
```

Explanation:

If we predict that branches are taken, using reservation stations will allow multiple executions of this loop to proceed at once. This advantage is gained without changing the code—in effect, the loop is unrolled dynamically by the hardware using the reservation stations obtained by renaming to act as additional Registers.

Let's assume we have issued all the instructions in two successive iterations of the loop, but none of the floating-point load/stores or operations has completed. Figure 3.10 shows reservation stations, register status tables, and load and store buffers at this point. (The integer ALU operation is ignored, and it is assumed the branch was predicted as taken.) Once the system reaches this state, two copies of the loop could be sustained with a CPI close to 1.0, provided the multiplies could complete in four clock cycles. With a latency of six cycles, additional iterations will need to be processed before the steady state can be reached. This requires more reservation stations to hold instructions that are in execution

Instruction status				
Instruction	**From iteration**	**Issue**	**Execute**	**Write result**
L.D F0,0(R1)	1	√	√	
MUL.D F4,F0,F2	1	√		
S.D F4,0(R1)	1	√		
L.D F0,0(R1)	2	√	√	
MUL.D F4,F0,F2	2	√		
S.D F4,0(R1)	2	√		

Reservation Stations								
Name	Busy	Op	Vj	Vk	Qj	Qk	Dest	A
Load1	Yes	Load						Regs[R1] + 0
Load2	Yes	Load						Regs[R1] – 8
Add1	No							
Add2	No							
Add3	No							
Mult1	Yes	MUL		Regs [F2]	Load1		#3	
Mult2	Yes	MUL		Regs [F2]	Load2		#5	
Store1	Yes	Store	Regs [R1]			Mult1		
Store2	Yes	Store	Regs [R1] – 8			Mult2		

FP Register Status									
Field	F0	F2	F4	F6	F8	F10	F12	…..	F30
Qi	Load2		Mult2						

Example Assume the same latencies for the floating-point functional units as in earlier examples:
add is 2 clock cycles, multiply is 6 clock cycles, and divide is 12 clock cycles. Using the code segment below, the same one we used to generate Figure 3.8, show what the status tables look like when the MUL.D is ready to go to commit.

```
L.D      F6,32(R2)
L.D      F2,44(R3)
MUL.D    F0,F2,F4
SUB.D    F8,F2,F6
DIV.D    F10,F0,F6
ADD.D    F6,F8,F2
```

REORDER BUFFER					
Entry	Busy	Instruction	State	Destination	Value
1	No	L.D F6,32(R2)	Commit	F6	Mem[32 + Regs[R2]]
2	No	L.D F2,44(R3)	Commit	F2	Mem[44 + Regs[R3]]
3	Yes	MUL.D F0,F2,F4	Write Result	F0	#2 × Regs[F4]
4	Yes	SUB.D F8,F2,F6	Write Result	F8	#2 – #1
5	Yes	DIV.D F10,F0,F6	Execute	F10	
6	Yes	ADD.D F6,F8,F2	Write Result	F6	#4 + #2

Reservation Stations								
Name	Busy	Op	Vj	Vk	Qj	Qk	Dest	A
Load1	No							
Load2	No							
Add1	No							
Add2	No							
Add3	No							
Mult1	No	MUL.D	Mem [44 + Regs [R3]]	Regs [F4]			#3	
Mult2	Yes	DIV.D		Mem [32 + Regs [R2]]	#3		#5	

FP Register Status										
Field	F0	F1	F2	F3	F4	F5	F6	F7	F8	F10
Reorder	3				7				4	5
Busy	Yes	No	No	No	No	No	Yes	-	Yes	Yes

Example

Consider the code example used earlier for Tomasulo's algorithm

```
Loop: L.D F0,0(R1)          ;F0=array element
ADD.D F4,F0,F2              ;add scalar in F2
S.D F4,0(R1)                ;store result
DADDUI R1,R1,#-8            ;decrement pointer
                            ;8 bytes (per DW)
BNE R1,R2,Loop              ;branch R1!=R2
```

Assume that we have issued all the instructions in the loop twice. Let's also assume that the L.D and MUL.D from the first iteration have committed and All other instructions have completed execution. Normally, the store Would wait in the ROB for both the effective address operand (R1 in this example) and the value (F4 in this example). Since we are only considering the floating-point pipeline, assume the effective address for the store is computed by the time the instruction is issued.

REORDER BUFFER					
Entry	Busy	Instruction	State	Destination	Value
1	No	L.D F0,0(R1)	Commit	F0	Mem[0 +Regs[R1]]
2	No	MUL.D F4,F0,F2	Commit	F4	#1 × Regs[F2]
3	Yes	S.D F4,0(R1)	Write Result	0 + Regs [R1]	#2
4	Yes	DADDIU R1,R1,#-8	Write Result	R1	Regs[R1] – 8
5	Yes	BNE R1,R2,Loop	Write Result		
6	Yes	L.D F0,0(R1)	Write Result	F0	Mem[#4]
7	Yes	MUL.D F4,F0,F2	Write Result	F4	#6 × Regs[F2]
8	Yes	S.D F4,0(R1)	Write Result	0 + #4	#7
9	Yes	DADDIU R1,R1,#-8	Write Result	R1	#4 – 8
10	Yes	BNE R1,R2,Loop	Write Result		

FP Register Status									
Field	F0	F1	F2	F3	F4	F5	F6	F7	F8
Reorder	6				7				
Busy	Yes	No	No	No	Yes	No	No	-	No

Executing ILP using multiple issue and Static Scheduling

Basic VLIW Approach

Example

Suppose we have a VLIW that could issue two memory references, two FP operations, and one integer operation or branch in every clock cycle. Show an unrolled version of the loop x[i] = x[i] + s for such a processor. Unroll as many times as necessary to eliminate any stalls. Ignore delayed branches.

MIPS CODE:

```
Loop: L.D F0,0(R1)          ;F0=array element
ADD.D F4,F0,F2              ;add scalar in F2
S.D F4,0(R1)                ;store result
DADDUI R1,R1,#-8            ;decrement pointer
                            ;8 bytes (per DW)
BNE R1,R2,Loop              ;branch R1!=R2
```

Memory reference 1	Memory reference 2	FP operation 1	FP operation 2	Integer operation/branch
L.D F0,0(R1)	L.D F6,-8(R1)			
L.D F10,-16(R1)	L.D F14,-24(R1)			
L.D F18,-32(R1)	L.D F22,-40(R1)	ADD.D F4,F0,F2	ADD.D F8,F6,F2	
L.D F26,-48(R1)		ADD.D F12,F10,F2	ADD.D F16,F14,F2	
		ADD.D F20,F18,F2	ADD.D F24,F22,F2	
S.D F4,0(R1)	S.D F8,-8(R1)	ADD.D F28,F26,F2		
S.D F12,-16(R1)	S.D F16,-24(R1)			
S.D F20,24(R1)	S.D F24,16(R1)			
S.D F28,8(R1)				DADDUI R1,R1,#-56
				BNE R1,R2,Loop

Exploiting ILP Using Dynamic Scheduling, Multiple Issue, and Speculation

Example

Consider the execution of the following loop, which increments each element of an integer array, on a two-issue processor, once without speculation and once with speculation:

```
Loop:   LD       R2,0(R1)       ;R2=array element
        DADDIU   R2,R2,#1       ;increment R2
        SD       R2,0(R1)       ;store result
        DADDIU   R1,R1,#8       ;increment pointer
        BNE      R2,R3,LOOP     ;branch if not last element
```

Assume that there are separate integer functional units for effective address calculation, for ALU operations, and for branch condition evaluation. Create a table for the first three iterations of this loop for both processors. Assume that Up to two instructions of any type can commit per clock

WITHOUT SPECULATION						
Iteration number	Instructions	Issues at clock cycle number	Executes at clock cycle number	Memory access at clock cycle number	Write CDB at clock cycle number	Comment
1	LD R2,0(R1)	1	2	3	4	First issue
1	DADDIU R2,R2,#1	1	5		6	Wait for LW
1	SD R2,0(R1)	2	3	7		Wait for DADDIU
1	DADDIU R1,R1,#8	2	3		4	Execute directly
1	BNE R2,R3,LOOP	3	7			Wait for DADDIU
2	LD R2,0(R1)	4	8	9	10	Wait for BNE
2	DADDIU R2,R2,#1	4	11		12	Wait for LW
2	SD R2,0(R1)	5	9	13		Wait for DADDIU
2	DADDIU R1,R1,#8	5	8		9	Wait for BNE
2	BNE R2,R3,LOOP	6	13			Wait for DADDIU
3	LD R2,0(R1)	7	14	15	16	Wait for BNE
3	DADDIU R2,R2,#1	7	17		18	Wait for LW
3	SD R2,0(R1)	8	15	19		Wait for DADDIU
3	DADDIU R1,R1,#8	8	14		15	Wait for BNE
3	BNE R2,R3,LOOP	9	19			Wait for DADDIU

WITH SPECULATION

Iteration number	Instructions	Issues at clock cycle number	Executes at clock cycle number	Memory access at clock cycle number	Write CDB at clock cycle number	Commits at clock number	Comment
1	LD R2,0(R1)	1	2	3	4	5	First issue
1	DADDIU R2,R2,#1	1	5		6	7	Wait for LW
1	SD R2,0(R1)	2	3			7	Wait for DADDIU
1	DADDIU R1,R1,#8	2	3		4	8	Execute directly
1	BNE R2,R3,LOOP	3	7			8	Wait for DADDIU
2	LD R2,0(R1)	4	5	6	7	9	Wait for BNE
2	DADDIU R2,R2,#1	4	8		9	10	Wait for LW
2	SD R2,0(R1)	5	6			10	Wait for DADDIU
2	DADDIU R1,R1,#8	5	6		7	11	Wait for BNE
2	BNE R2,R3,LOOP	6	10			11	Wait for DADDIU
3	LD R2,0(R1)	7	8	9	10	12	Wait for BNE
3	DADDIU R2,R2,#1	7	11		12	13	Wait for LW
3	SD R2,0(R1)	8	9			13	Wait for DADDIU
3	DADDIU R1,R1,#8	8	9		10	14	Wait for BNE
3	BNE R2,R3,LOOP	9	13			14	Wait for DADDIU

Example

Determine the total branch penalty for a branch-target buffer assuming the Penalty cycles for individual mispredictions from the below Figure Make the Following assumptions about the prediction accuracy and hit rate:

- Prediction accuracy is 90% (for instructions in the buffer).
- Hit rate in the buffer is 90% (for branches predicted taken).

Instruction in buffer	Prediction	Actual branch	Penalty cycles
Yes	Taken	Taken	0
Yes	Taken	Not Taken	2
No		Taken	2
No		Not Taken	0

Answer

We compute the penalty by looking at the probability of two events: the branch is predicted taken but ends up being not taken, and the branch is taken but is not found in the buffer. Both carry a penalty of two cycles.

Probability (branch in buffer, but actually not taken) Percent buffer hit rate = × Percent incorrect predictions = 90% × 10% = 0.09

Probability (branch not in buffer, but actually taken) = 10%

Branch penalty = (0.09 + 0.10) × 2

Branch penalty = 0.38

Example

Consider the following three hypothetical, but not atypical, processors, which we run with the SPEC gcc benchmark:

1. A simple MIPS two-issue static pipe running at a clock rate of 4 GHz and achieving a pipeline CPI of 0.8. This processor has a cache system that Yields 0.005 misses per instruction.

2. A deeply pipelined version of a two-issue MIPS processor with slightly smaller caches and a 5 GHz clock rate. The pipeline CPI of the processor is 1.0, and the smaller caches yield 0.0055 misses per instruction on average

3. A speculative superscalar with a 64-entry window. It achieves one-half of The ideal issue rate measured for this window size. (Use the data in Figure 3.27.) This processor has the smallest caches, which lead to 0.01 misses per instruction, but it hides 25% of the miss penalty on every miss by dynamic scheduling. This processor has a 2.5 GHz clock

Assume that the main memory time (which sets the miss penalty) is 50 ns. Determine the relative performance of these three processors.

Answer

First, we use the miss penalty and miss rate information to compute the Contribution to CPI from cache misses for each configuration. We do this with the following formula:

We need to compute the miss penalties for each system:

Cache CPI = Misses per instruction × Miss penalty

$$\text{Miss penalty} = \frac{\text{Memory access time}}{\text{Clock cycle}}$$

The clock cycle times for the processors are 250 ps, 200 ps, and 400 ps, respectively.

Hence, the miss penalties are

$$\text{Miss penalty 1} = \frac{50\text{ ns}}{250\text{ ps}} = 200\text{ cycles}$$

$$\text{Miss penalty 2} = \frac{50\text{ ns}}{200\text{ ps}} = 250\text{ cycles}$$

$$\text{Miss penalty 3} = \frac{0.75 \times 50\text{ ns}}{400\text{ ps}} = 94\text{ cycles}$$

Applying this for each cache:

Cache $CPI_1 = 0.005 \times 200 = 1.0$

Cache $CPI_2 = 0.0055 \times 250 = 1.4$

Cache $CPI_3 = 0.01 \times 94 = 0.94$

We know the pipeline CPI contribution for everything but processor 3; its Pipeline CPI is given by:

$$\text{Pipeline } CPI_3 = \frac{1}{\text{Issue rate}} = \frac{1}{9 \times 0.5} = 0.22$$

Now we can find the CPI for each processor by adding the pipeline and cache CPI contributions:

$CPI_1 = 0.8 + 1.0 = 1.8$

$CPI_2 = 1.0 + 1.4 = 2.4$

$CPI_3 = 0.22 + 0.94 = 1.16$

Since this is the same architecture, we can compare instruction execution rates in millions of instructions per second (MIPS) to determine relative performance:

$$\text{Instruction execution rate} = \frac{CR}{CPI}$$

$$\text{Instruction execution rate}_1 = \frac{4000\ \text{MHz}}{1.8} = 2222\ \text{MIPS}$$

$$\text{Instruction execution rate}_2 = \frac{4000\ \text{MHz}}{2.4} = 2083\ \text{MIPS}$$

$$\text{Instruction execution rate}_3 = \frac{2500\ \text{MHz}}{1.16} = 2155\ \text{MIPS}$$