

Pipeline : Basic and Intermediate Concepts (Week#12 and 13)

Hazards and Forwarding in Longer Latency Pipelines

If we assume that FP register file has one write port, sequences of FP operations and FP load together with FP operations cause conflicts for register write port as shown in Figure C.33.

Instruction	Clock cycle number										
	1	2	3	4	5	6	7	8	9	10	11
fmul.d f0,f4,f6	IF	ID	M1	M2	M3	M4	M5	M6	M7	MEM	WB
...		IF	ID	EX	MEM	WB					
...			IF	ID	EX	MEM	WB				
fadd.d f2,f4,f6				IF	ID	A1	A2	A3	A4	MEM	WB
...					IF	ID	EX	MEM	WB		
...						IF	ID	EX	MEM	WB	
fld f2,0(x2)							IF	ID	EX	MEM	WB

Figure C.33 Three instructions want to perform a write-back to the FP register file simultaneously, as shown in clock cycle 11. This is *not* the worst case, because an earlier divide in the FP unit could also finish on the same clock. Note that although the `fmul.d`, `fadd.d`, and `fld` are in the MEM stage in clock cycle 10, only the `fld` actually uses the memory, so no structural hazard exists for MEM.

Hazards and Forwarding in Longer Latency Pipelines

- ✓ In clock cycle 11, all three instructions will reach WB and want to write register file. With only a single register file write port, processor must serialize the instruction completion. Single register port represents a structural hazard.
- ✓ Increase the number of write ports can solve this, but additional write ports would be rarely used because maximum number of write ports needed is 1.

Implement interlock: Track the use of write port in ID stage and to stall an instruction before it issues.

- ✓ Tracking the use of write port can be done with a shift register that indicates when issued instructions will use register file.
- ✓ If instruction in ID needs to use register file at same time as an instruction already issued, instruction in ID is stalled for a cycle.
- ✓ On each clock reservation register is shifted 1 bit. This implementation has an advantage: It maintains property that all interlock detection and stall insertion occurs in ID stage. Cost is the addition of shift register.

Hazards and Forwarding in Longer Latency Pipelines

- ✓ An alternative scheme is to stall a conflicting instruction when it tries to enter either MEM or WB stage.
- ✓ If we wait to stall the conflicting instructions until they want to enter the MEM or WB stage, we can choose to stall either instruction.
- ✓ A simple heuristic is to give priority to the unit with the longest latency, because that is the one most likely to have caused another instruction to be stalled for a RAW hazard.
- ✓ Advantage of this scheme is that it does not require to detect the conflict until the entrance of the MEM or WB stage.
- ✓ Disadvantage is that it complicates pipeline control as stalls can now arise from two places.
- ✓ Stalling before entering MEM will cause the EX, A4, or M7 stage to be occupied, possibly forcing stall to trickle back in pipeline. Stalling before WB would cause MEM to back up.

Hazards and Forwarding in Longer Latency Pipelines

- ✓ Consider example in Fig C.33 If the fadd.d instruction were issued one cycle earlier and had a destination of f2, then it would create a WAW hazard, because it would write f2 one cycle earlier than fadd.d.
- ✓ This hazard only occurs when result of fadd.d is overwritten without any instruction ever using it. If there were a use of f2 between the fadd.d and fadd.d, the pipeline would need to be stalled for a RAW hazard, and the fadd.d would not issue until fadd.d was completed.

Instruction	Clock cycle number										
	1	2	3	4	5	6	7	8	9	10	11
fmul.d f0,f4,f6	IF	ID	M1	M2	M3	M4	M5	M6	M7	MEM	WB
...		IF	ID	EX	MEM	WB					
...			IF	ID	EX	MEM	WB				
fadd.d f2,f4,f6				IF	ID	A1	A2	A3	A4	MEM	WB
...					IF	ID	EX	MEM	WB		
...						IF	ID	EX	MEM	WB	
fld f2,0(x2)							IF	ID	EX	MEM	WB

Figure C.33 Three instructions want to perform a write-back to the FP register file simultaneously, as shown in clock cycle 11. This is *not* the worst case, because an earlier divide in the FP unit could also finish on the same clock. Note that although the fmul.d, fadd.d, and fld are in the MEM stage in clock cycle 10, only the fld actually uses the memory, so no structural hazard exists for MEM.

Hazards and Forwarding in Longer Latency Pipelines

- ✓ There are two possible ways to handle this WAW hazard.
- ✓ i) to delay the issue of load instruction until the fadd.d enters MEM.
- ✓ ii) to stamp out result of fadd.d by detecting hazard and changing control so that fadd.d does not write its result.
- ✓ WAW Hazard can be detected during ID when fadd.d is issuing and stalling fadd.d or making the fadd.d a no-op is easy.
- ✓ Difficult situation is to detect fadd.d might finish before the fadd.d, because that requires knowing the length of pipeline and current position of fadd.d. Code sequence (two writes with no intervening read) will be very rare, so a simple solution: if an instruction in ID wants to write same register, do not issue instruction to EX.

Instruction	Clock cycle number										
	1	2	3	4	5	6	7	8	9	10	11
fmul.d f0,f4,f6	IF	ID	M1	M2	M3	M4	M5	M6	M7	MEM	WB
...		IF	ID	EX	MEM	WB					
...			IF	ID	EX	MEM	WB				
fadd.d f2,f4,f6				IF	ID	A1	A2	A3	A4	MEM	WB
...					IF	ID	EX	MEM	WB		
...						IF	ID	EX	MEM	WB	
fld f2,0(x2)							IF	ID	EX	MEM	WB

Figure C.33 Three instructions want to perform a write-back to the FP register file simultaneously, as shown in clock cycle 11. This is *not* the worst case, because an earlier divide in the FP unit could also finish on the same clock. Note that although the fmul.d, fadd.d, and fld are in the MEM stage in clock cycle 10, only the fld actually uses the memory, so no structural hazard exists for MEM.

Hazards and Forwarding in Longer Latency Pipelines

- ✓ consider hazards among FP instructions and hazards between an FP instruction and an integer instruction.
- ✓ Except for FP loads-stores and FP-integer register moves, FP and integer registers are distinct. All integer instructions operate on integer registers, while FP operations operate only on their own registers.
- ✓ Consider FP loads-stores and FP register moves in detecting hazards between FP and integer instructions. Pipeline control is an additional advantage of having separate register files for integer and floating-point data.
- ✓ Main advantages are a doubling the number of registers and an increase in bandwidth without adding more ports.
- ✓ Main disadvantage beyond the need for an extra register file is the small cost of occasional moves needed between the two register sets.

Hazards and Forwarding in Longer Latency Pipelines

Assuming that pipeline does all hazard detection in ID, there are three checks that must be performed before an instruction can issue:

1. Check for structural hazards: Wait until the required functional unit is not busy (this is only needed for divides in this pipeline) and make sure register write port is available when it will be needed.

2. Check for a RAW data hazard: Wait until source registers are not listed as pending destinations in a pipeline register that will not be available when this instruction needs result.

- ✓ For example, if instruction in ID is an FP operation with source register f2, then f2 cannot be listed as a destination in ID/A1, A1/A2, or A2/A3, which correspond to FP add instructions that will not be finished when instruction in ID needs a result.
- ✓ Divide is more tricky, to allow last few cycles of a divide to be overlapped, need to handle when a divide is close to finishing.

Hazards and Forwarding in Longer Latency Pipelines

3. Check for a WAW data hazard: Determine if any instruction in A1,..., A4, D, M1,..., M7 has the same register destination as this instruction. If so, stall the issue of the instruction in ID.

- ✓ Hazard detection is more complex with the multicycle FP operations, same as RISC V integer pipeline and forwarding logic.
- ✓ Forwarding can be implemented by checking if destination register in any of EX/MEM, A4/MEM, M7/MEM, D/MEM, or MEM/WB registers is one of the source registers of a floating-point instruction.

Maintaining Precise Exceptions

Another problem caused by these long-running instructions can be illustrated with the following sequence of code:

```
fdiv.d    f0, f2, f4
fadd.d    f10, f10, f8
fsub.d    f12, f12, f14
```

- ✓ This code sequence has no dependences. A problem arises because an instruction issued early may complete after an instruction issued later. Expect fadd.d and fsub.d to complete before the fdiv.d completes. This is called out-of-order completion.
- ✓ Hazard detection will prevent any dependence among instructions from being violated, why is out-of-order completion a problem?
- ✓ Suppose fsub.d causes a floating-point arithmetic exception at a point where fadd.d has completed but fdiv.d has not. The result will be an imprecise exception.
- ✓ This could be handled by letting the floating-point pipeline drain, as we do for the integer pipeline. For example, if the fdiv.d decided to take a floating-point-arithmetic exception after the add completed, we could not have a precise exception at the hardware level. fadd.d destroys one of its operands, we could not restore state to it was before the fdiv.d.

Maintaining Precise Exceptions

Recent processors have solved this problem by introducing two modes of execution: a fast imprecise mode and a slower precise mode.

1. Slower precise mode: is implemented either with a mode switch or by insertion of explicit instructions that test for FP exceptions.

Amount of overlap and reordering permitted in FP pipeline is restricted so only one FP instruction is active at a time.

This solution was used in DEC Alpha 21064 and 21164 in IBM Power1 and Power2, and in MIPS R8000.

2. Buffer the results: of an operation until all the operations that were issued earlier are complete. Some processors use this solution but it becomes expensive when difference in running times among operations is large, because the number of results to buffer can become large.

Results from queue must be bypassed to continue issuing instructions while waiting for longer instruction. This requires a large number of comparators and a very large multiplexer.

Maintaining Precise Exceptions

3. Allow the exceptions: to become imprecise, but to keep enough information for trap-handling routines that can create a precise sequence for exception.

- ✓ After handling exception, software finishes any instructions that precede latest instruction completed and sequence can restart. Consider following worst-case code sequence:

Instruction₁—A long-running instruction that eventually interrupts execution.

Instruction₂, ..., Instruction_{n-1}—A series of instructions that are not completed.

Instruction_n—An instruction that is finished.

- ✓ PCs of all instructions in pipeline and exception return PC, software can find the state of instruction 1 and instruction n. If instruction n has completed, we want to restart execution at instruction n+1.
- ✓ After handling exception, software must simulate execution of instruction 1, ..., instruction n-1. Return from exception and restart at instruction n+1. Complexity of executing these instructions properly by handler is the major difficulty of this scheme.

Maintaining Precise Exceptions

- ✓ There is an important simplification for simple RISC V pipelines: If instruction 2, ... , instruction n are all integer instructions, if instruction n has completed then all of instruction 2, ... , instruction n-1 have also completed.
- ✓ Only FP operations need to be handled. Number of floating-point instructions that can be overlapped in execution can be limited.
- ✓ For example, if we only overlap two instructions then only interrupting instruction need be completed by software. This restriction may reduce the throughput if FP pipelines are deep or if there are a significant number of FP functional units.
- ✓ approach is used in SPARC implementations to allow overlap of floating-point and integer operations.

Maintaining Precise Exceptions

- 4. Hybrid Scheme:** allows instruction issue to continue only if it is certain that all instructions before the issuing instruction will complete without causing an exception.
- ✓ This guarantees that when an exception occurs, no instructions after interrupting one will be completed and all instructions before interrupting one can be completed.
 - ✓ This means stalling the processor to maintain precise exceptions.
 - ✓ To make this scheme work, floating-point functional units must determine if an exception is possible early in EX stage (in first 3 clock cycles in RISC V pipeline) to prevent further instructions from completing.
 - ✓ This scheme is used in MIPS R2000/3000, R4000, and Intel Pentium.

Performance of a Simple RISC V FP Pipeline

- ✓ Figure C.34 shows number of stall cycles for each type of floating-point operation on a per-instance basis (i.e., the first bar for each FP benchmark shows the number of FP result stalls for each FP add, subtract, or convert).
- ✓ Stall cycles per operation track the latency of FP operations, varying from 46% to 59% of the latency of the functional unit.

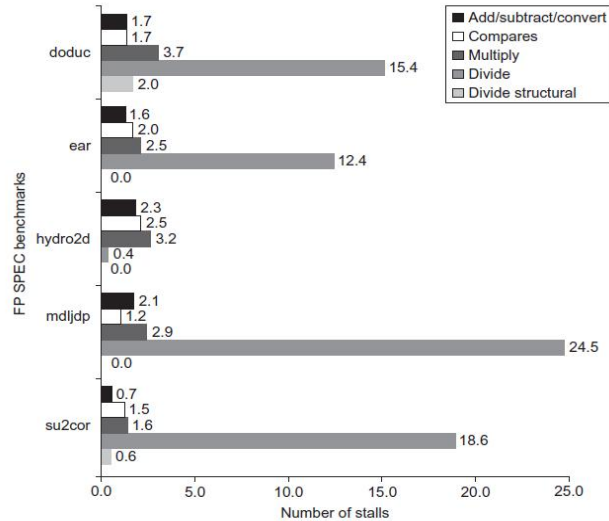


Figure C.34 Stalls per FP operation for each major type of FP operation for the SPEC89 FP benchmarks. Except for the divide structural hazards, these data do not depend on the frequency of an operation, only on its latency and the number of cycles before the result is used. The number of stalls from RAW hazards roughly tracks the latency of the FP unit. For example, the average number of stalls per FP add, subtract, or convert is 1.7 cycles, or 56% of the latency (three cycles). Likewise, the average number of stalls for multiplies and divides are 2.8 and 14.2, respectively, or 46% and 59% of the corresponding latency. Structural hazards for divides are rare, because the divide frequency is low.

Performance of a Simple RISC V FP Pipeline

- ✓ Figure C.35 gives complete breakdown of integer and FP stalls for five SPECfp benchmarks.
- ✓ There are four classes of stalls shown: FP result stalls, FP compare stalls, load and branch delays, and FP structural delays.
- ✓ Branch delay stalls, which would be small with a one cycle delay and a modest branch predictor, are not included.
- ✓ The total number of stalls per instruction varies from 0.65 to 1.21.

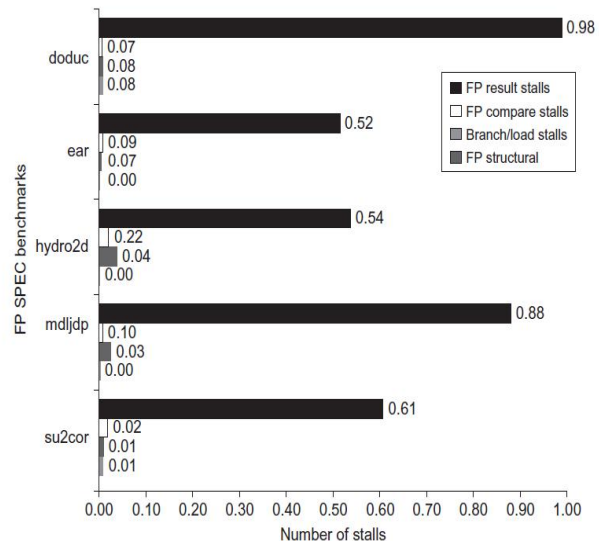


Figure C.35 The stalls occurring for the a simple RISC V FP pipeline for five of the SPEC89 FP benchmarks. The total number of stalls per instruction ranges from 0.65 for su2cor to 1.21 for doduc, with an average of 0.87. FP result stalls dominate in all cases, with an average of 0.71 stalls per instruction, or 82% of the stalled cycles. Compares generate an average of 0.1 stalls per instruction and are the second largest source. The divide structural hazard is only significant for doduc. Branch stalls are not accounted for, but would be small.

MIPS R4000 Pipeline

- ✓ MIPS architecture and RISC V are very similar, differing only in a few instructions, including a delayed branch in the MIPS ISA.
- ✓ R4000 implements MIPS64 but uses a deeper pipeline than five-stage design, both for integer and FP programs.

Superpipelining: deeper pipeline allows it to achieve higher clock rates by decomposing five-stage integer pipeline into eight stages. Cache access is time critical, extra pipeline stages come from decomposing memory access. This type of deeper pipelining is called superpipelining. Figure C.36 shows eight-stage pipeline structure

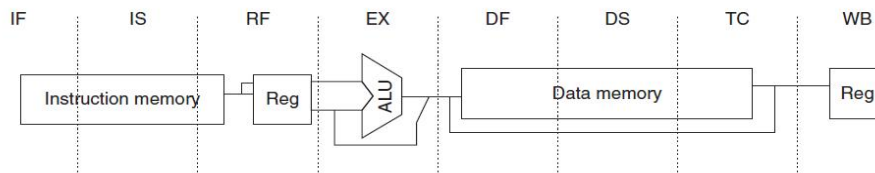


Figure C.36 The eight-stage pipeline structure of the R4000 uses pipelined instruction and data caches. The pipe stages are labeled and their detailed function is described in the text. The vertical dashed lines represent the stage boundaries as well as the location of pipeline latches. The instruction is actually available at the end of IS, but the tag check is done in RF, while the registers are fetched. Thus, we show the instruction memory as operating through RF. The TC stage is needed for data memory access, because we cannot write the data into the register until we know whether the cache access was a hit or not.

MIPS R4000 Pipeline

The function of each stage is as follows:

IF: First half of instruction fetch; PC selection actually happens here, together with initiation of instruction cache access.

IS: Second half of instruction fetch, complete instruction cache access.

RF: Instruction decode and register fetch, hazard checking, and instruction cache hit detection.

EX: Execution, which includes effective address calculation, ALU operation, and branch-target computation and condition evaluation.

DF: Data fetch, first half of data cache access.

DS: Second half of data fetch, completion of data cache access.

TC: Tag check, to determine whether the data cache access hit.

WB: Write-back for loads and register-register operations.

MIPS R4000 Pipeline

Figure C.37 shows overlap of successive instructions in pipeline. Instruction and data memory occupy multiple cycles, they are fully pipelined, so that a new instruction can start on every clock. Figure C.37 shows load delays are two cycles, because data value is available at the end of DS.

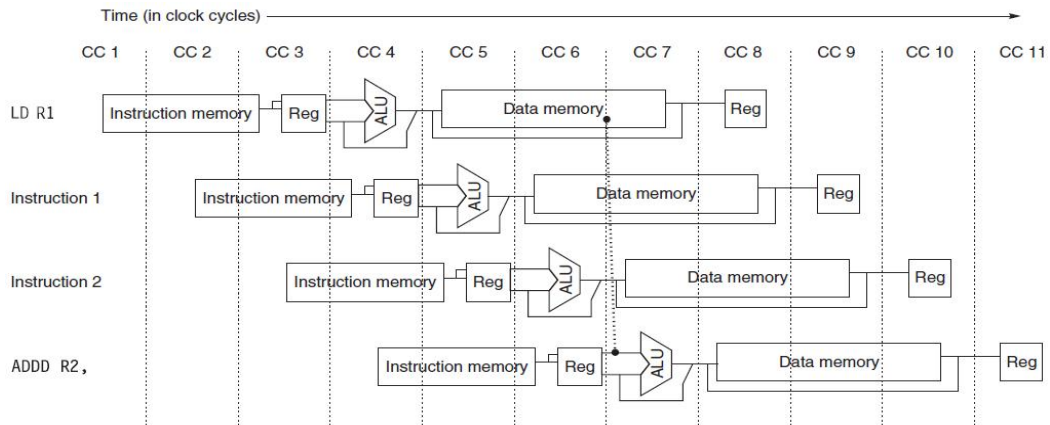


Figure C.37 The structure of the R4000 integer pipeline leads to a x1 load delay. A x1 delay is possible because the data value is available at the end of DS and can be bypassed. If the tag check in TC indicates a miss, the pipeline is backed up a cycle, when the correct data are available.

MIPS R4000 Pipeline

Figure C.38 shows pipeline schedule when a immediate use follows a load. It shows forwarding is required for result of a load instruction to a destination that is three or four cycles later.

Instruction number	Clock number								
	1	2	3	4	5	6	7	8	9
ld x1,...	IF	IS	RF	EX	DF	DS	TC	WB	
add x2,x1,...		IF	IS	RF	Stall	Stall	EX	DF	DS
sub x3,x1,...			IF	IS	Stall	Stall	RF	EX	DF
or x4,x1,...				IF	Stall	Stall	IS	RF	EX

Figure C.38 A load instruction followed by an immediate use results in a x1 stall. Normal forwarding paths can be used after two cycles, so the add and sub get the value by forwarding after the stall. The or instruction gets the value from the register file. Because the two instructions after the load could be independent and hence not stall, the bypass can be to instructions that are three or four cycles after the load.

MIPS R4000 Pipeline

Figure C.39 shows basic branch delay is three cycles because branch condition is computed during EX. MIPS architecture has a single-cycle delayed branch. R4000 uses a predicted-not-taken strategy for remaining two cycles of branch delay.

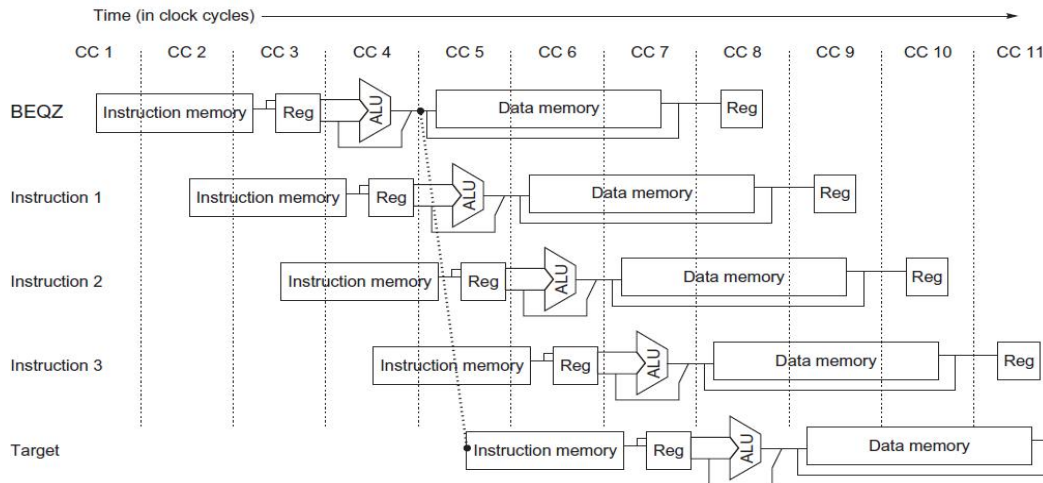


Figure C.39 The basic branch delay is three cycles, because the condition evaluation is performed during EX.

MIPS R4000 Pipeline

Figure C.40 shows, untaken branches are one-cycle delayed branches, while taken branches have a one-cycle delay slot followed by two idle cycles.

The instruction set provides a branch-likely instruction, which helps in filling the branch delay slot.

	Clock number								
Instruction number	1	2	3	4	5	6	7	8	9
Branch instruction	IF	IS	RF	EX	DF	DS	TC	WB	
Delay slot		IF	IS	RF	EX	DF	DS	TC	WB
Stall			Stall	Stall	Stall	Stall	Stall	Stall	Stall
Stall				Stall	Stall	Stall	Stall	Stall	Stall
Branch target					IF	IS	RF	EX	DF
Branch instruction	IF	IS	RF	EX	DF	DS	TC	WB	
Delay slot		IF	IS	RF	EX	DF	DS	TC	WB
Branch instruction + 2			IF	IS	RF	EX	DF	DS	TC
Branch instruction + 3				IF	IS	RF	EX	DF	DS

Figure C.40 A taken branch, shown in the top portion of the figure, has a one-cycle delay slot followed by a x1 stall, while an untaken branch, shown in the bottom portion, has simply a one-cycle delay slot. The branch instruction can be an ordinary delayed branch or a branch-likely, which cancels the effect of the instruction in the delay slot if the branch is untaken.

MIPS R4000 Pipeline

- ✓ Pipeline interlocks enforce both the x1 branch stall penalty on a taken branch and any data hazard stall that arises from use of a load result.
- ✓ All implementations of MIPS processor made use of dynamic branch prediction.
- ✓ In addition to increase in stalls for loads and branches, deeper pipeline increases the number of levels of forwarding for ALU operations.
- ✓ In RISC V five-stage pipeline, forwarding between two register-register ALU instructions could happen from ALU/MEM or MEM/WB registers.
- ✓ In R4000 pipeline, there are four possible sources for an ALU bypass: EX/DF, DF/DS, DS/TC, and TC/WB.

MIPS R4000 Pipeline

Floating Point Pipeline: R4000 floating-point unit consists of three functional units: a floating-point divider, a floating-point multiplier, and a floating-point adder.

Adder logic is used on final step of a multiply or divide. Double-precision FP operations can take from 2 cycles (for a negate) up to 112 cycles (for a square root).

FP functional unit can be thought of as having eight different stages shown in Figure C.41; these stages are combined in different orders to execute various FP operations. There is a single copy of each of these stages and various instructions may use a stage zero or more times and in different orders.

Stage	Functional unit	Description
A	FP adder	Mantissa add stage
D	FP divider	Divide pipeline stage
E	FP multiplier	Exception test stage
M	FP multiplier	First stage of multiplier
N	FP multiplier	Second stage of multiplier
R	FP adder	Rounding stage
S	FP adder	Operand shift stage
U		Unpack FP numbers

Figure C.41 The eight stages used in the R4000 floating-point pipelines.

MIPS R4000 Pipeline

Figure C.42 shows latency, initiation rate, and pipeline stages used by double-precision FP operations. Figure C.42 determine whether a sequence of different, independent FP operations can issue without stalling. If timing of sequence is such that a conflict occurs for a shared pipeline stage, then a stall will be needed

FP instruction	Latency	Initiation interval	Pipe stages
Add, subtract	4	3	U, S+A, A+R, R+S
Multiply	8	4	U, E+M, M, M, M, N, N+A, R
Divide	36	35	U, A, R, D ²⁸ , D+A, D+R, D+A, D+R, A, R
Square root	112	111	U, E, (A+R) ¹⁰⁸ , A, R
Negate	2	1	U, S
Absolute value	2	1	U, S
FP compare	3	2	U, A, R

Figure C.42 The latencies and initiation intervals for the FP operations initiation intervals for the FP operations both depend on the FP unit stages that a given operation must use. The latency values assume that the destination instruction is an FP operation; the latencies are one cycle less when the destination is a store. The pipe stages are shown in the order in which they are used for any operation. The notation S+A indicates a clock cycle in which both the S and A stages are used. The notation D²⁸ indicates that the D stage is used 28 times in a row.

MIPS R4000 Pipeline

Figures C.43 show two-instruction sequences: a multiply followed by an add.

Operation	Issue/stall	Clock cycle												
		0	1	2	3	4	5	6	7	8	9	10	11	12
Multiply	Issue	U	E+M	M	M	M	N	N+A	R					
Add	Issue		U	S+A	A+R	R+S								
	Issue			U	S+A	A+R	R+S							
	Issue				U	S+A	A+R	R+S						
	Stall					U	S+A	A+R	R+S					
	Stall						U	S+A	A+R	R+S				
	Issue							U	S+A	A+R	R+S			
	Issue								U	S+A	A+R	R+S		

Figure C.43 An FP multiply issued at clock 0 is followed by a single FP add issued between clocks 1 and 7. The second column indicates whether an instruction of the specified type stalls when it is issued n cycles later, where n is the clock cycle number in which the U stage of the second instruction occurs. The stage or stages that cause a stall are in bold. Note that this table deals with only the interaction between the multiply and one add issued between clocks 1 and 7. In this case, the add will stall if it is issued four or five cycles after the multiply; otherwise, it issues without stalling. Notice that the add will be stalled for two cycles if it issues in cycle 4 because on the next clock cycle it will still conflict with the multiply; if, however, the add issues in cycle 5, it will stall for only 1 clock cycle, because that will eliminate the conflicts.

MIPS R4000 Pipeline

Figures C.44 show two-instruction sequences: an add followed by a multiply.

		Clock cycle												
Operation	Issue/stall	0	1	2	3	4	5	6	7	8	9	10	11	12
Add	Issue	U	S+A	A+R	R+S									
Multiply	Issue		U	E+M	M	M	M	N	N+A	R				
	Issue			U	M	M	M	M	N	N+A	R			

Figure C.44 A multiply issuing after an add can always proceed without stalling, because the shorter instruction clears the shared pipeline stages before the longer instruction reaches them.

MIPS R4000 Pipeline

Figures C.45 show two-instruction sequences: a divide followed by an add.

Operation	Issue/stall	Clock cycle											
		25	26	27	28	29	30	31	32	33	34	35	36
Divide	Issued in cycle 0...	D	D	D	D	D	D+A	D+R	D+A	D+R	A	R	
Add	Issue		U	S+A	A+R	R+S							
	Issue			U	S+A	A+R	R+S						
	Stall				U	S+A	A+R	R+S					
	Stall					U	S+A	A+R	R+S				
	Stall						U	S+A	A+R	R+S			
	Stall							U	S+A	A+R	R+S		
	Stall								U	S+A	A+R	R+S	
	Stall									U	S+A	A+R	R+S
	Issue										U	S+A	A+R
	Issue											U	S+A
	Issue												U

Figure C.45 An FP divide can cause a stall for an add that starts near the end of the divide. The divide starts at cycle 0 and completes at cycle 35; the last 10 cycles of the divide are shown. Because the divide makes heavy use of the rounding hardware needed by the add, it stalls an add that starts in any of cycles 28–33. Notice that the add starting in cycle 28 will be stalled until cycle 36. If the add started right after the divide, it would not conflict, because the add could complete before the divide needed the shared stages, just as we saw in [Figure C.44](#) for a multiply and add. As in the earlier figure, this example assumes *exactly* one add that reaches the U stage between clock cycles 26 and 35.

MIPS R4000 Pipeline

Figures C.46 show two-instruction sequences: an add followed by a divide

		Clock cycle												
Operation	Issue/stall	0	1	2	3	4	5	6	7	8	9	10	11	12
Add	Issue	U	S+A	A+R	R+S									
Divide	Stall		U	A	R	D	D	D	D	D	D	D	D	D
	Issue			U	A	R	D	D	D	D	D	D	D	D
	Issue				U	A	R	D	D	D	D	D	D	D

Figure C.46 A double-precision add is followed by a double-precision divide. If the divide starts one cycle after the add, the divide stalls, but after that there is no conflict.

Performance of the R4000 Pipeline

Stalls that occur for the SPEC92 benchmarks when running on the R4000 pipeline structure. There are four major causes of pipeline stalls or losses:

- 1. Load stalls:** Delays arising from the use of a load result one or two cycles after the load
- 2. Branch stalls:** Two-cycle stalls on every taken branch plus unfilled or canceled branch delay slots. MIPS instruction set implemented in R4000 supports instructions that predict a branch at compile time and cause instruction in branch delay slot to be canceled when branch behavior differs from prediction. This makes it easier to fill branch delay slots.
- 3. FP result stalls:** Stalls because of RAW hazards for an FP operand
- 4. FP structural stalls:** Delays because of issue restrictions arising from conflicts for functional units in the FP pipeline

Performance of the R4000 Pipeline

Figure C.47 shows pipeline CPI breakdown for R4000 pipeline for SPEC92 benchmarks.

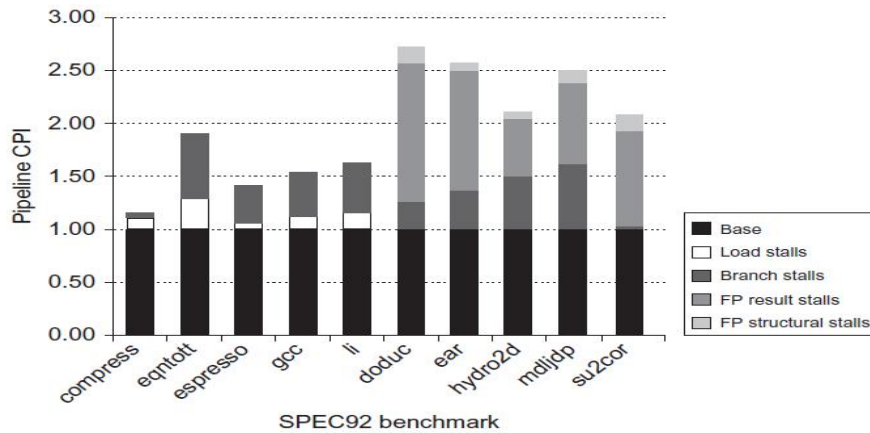


Figure C.47 The pipeline CPI for 10 of the SPEC92 benchmarks, assuming a perfect cache. The pipeline CPI varies from 1.2 to 2.8. The left-most five programs are integer programs, and branch delays are the major CPI contributor for these. The right-most five programs are FP, and FP result stalls are the major contributor for these. [Figure C.48](#) shows the numbers used to construct this plot.

Performance of the R4000 Pipeline

Figure C.48 shows the same data but in tabular form.

Benchmark	Pipeline CPI	Load stalls	Branch stalls	FP result stalls	FP structural stalls
Compress	1.20	0.14	0.06	0.00	0.00
Eqntott	1.88	0.27	0.61	0.00	0.00
Espresso	1.42	0.07	0.35	0.00	0.00
Gcc	1.56	0.13	0.43	0.00	0.00
Li	1.64	0.18	0.46	0.00	0.00
Integer average	1.54	0.16	0.38	0.00	0.00
Doduc	2.84	0.01	0.22	1.39	0.22
Mdljdp2	2.66	0.01	0.31	1.20	0.15
Ear	2.17	0.00	0.46	0.59	0.12
Hydro2d	2.53	0.00	0.62	0.75	0.17
Su2cor	2.18	0.02	0.07	0.84	0.26
FP average	2.48	0.01	0.33	0.95	0.18
Overall average	2.00	0.10	0.36	0.46	0.09

Figure C.48 The total pipeline CPI and the contributions of the four major sources of stalls are shown. The major contributors are FP result stalls (both for branches and for FP inputs) and branch stalls, with loads and FP structural stalls adding less.

Performance of the R4000 Pipeline

- ✓ Figures C.47 and C.48 shows penalty of deeper pipelining. R4000's pipeline has much longer branch delays than five-stage pipeline.
- ✓ Longer branch delay increases cycles spent on branches, especially for the integer programs with a higher branch frequency.
- ✓ All subsequent processors with moderate to deep pipelines (8–16 stages) employ dynamic branch predictors.
- ✓ Latency of FP functional units leads to more result stalls than the structural hazards, which arise both from the initiation interval limitations and from conflicts for functional units from different FP instructions.
- ✓ Reducing latency of FP operations should be first target, rather than more pipelining or replication of functional units.
- ✓ Reducing the latency increase the structural stalls, because many potential structural stalls are hidden behind data hazards.

Cross Cutting Issues

RISC Instruction Sets and Efficiency of Pipelining: Advantages of instruction set simplicity in building pipelines:

- ✓ Make it easier to schedule code to achieve efficiency of execution in a pipeline. Example, add two values in memory and store the result back to memory. It will take two or three.
- ✓ RISC architecture require four instructions (two loads, an add, and a store). These instructions cannot be scheduled sequentially in most pipelines without intervening stalls.
- ✓ With a RISC instruction set, individual operations are separate instructions and may be individually scheduled either by compiler or using dynamic hardware scheduling techniques.
- ✓ Intel processors and ARM processors use these approach for more complex instructions.

Cross Cutting Issues

Dynamically Scheduled Pipelines:

- ✓ Pipelines fetch an instruction and issue it, unless there is a data dependence between an instruction already in the pipeline and the fetched instruction that cannot be hidden with bypassing or forwarding.
- ✓ Forwarding logic reduces the effective pipeline latency so that certain dependences do not result in hazards. If there is an unavoidable hazard, then hazard detection hardware stalls the pipeline.
- ✓ **Static Scheduling:** No new instructions are fetched or issued until the dependence is cleared. To overcome these performance losses, the compiler schedule instructions to avoid the hazard; this approach is called compiler or static scheduling.
- ✓ Several early processors used another approach, called dynamic scheduling, whereby the hardware rearranges the instruction execution to reduce the stalls.

Cross Cutting Issues

- ✓ All the techniques discussed so far use in-order instruction issue.
- ✓ With in-order issue, if two instructions have a hazard between them, pipeline will stall even if there are later instructions that are independent and would not stall.
- ✓ In the RISC V pipeline, both structural and data hazards were checked during instruction decode (ID)
- ✓ To allow an instruction to begin execution as soon as its operands are available, even if a predecessor is stalled.
- ✓ Separate the issue process into two parts: checking the structural hazards and waiting for the absence of a data hazard.
- ✓ decode and issue instructions in order, we want the instructions to begin execution as soon as their data operands are available. pipeline will do out-of-order pipe execution. To implement out-of-order execution, we must split the ID pipe stage into two stages:
 1. **Issue:** Decode instructions to check for structural hazards.
 2. **Read operands:** Wait until no data hazards then read operands.

Cross Cutting Issues

- ✓ IF stage proceeds the issue stage and EX stage follows read operands stage as in RISC V pipeline.
- ✓ RISC V floating- point pipeline, execution may take multiple cycles, depending on the operation.
- ✓ To distinguish when an instruction begins execution and when it completes execution; between the two times, the instruction is in execution.
- ✓ This allows multiple instructions to be in execution at the same time.
- ✓ Change the functional unit design by varying the number of units, latency of operations, and functional unit pipelining.

Cross Cutting Issues

Dynamic Scheduling With a Scoreboard: In a dynamically scheduled pipeline, all instructions pass through the issue stage in order.

They can be stalled or bypass each other in second stage (read operands) and enter execution out of order.

Scoreboarding: is a technique for allowing instructions to execute out of order when there are sufficient resources and no data dependences; it is named after CDC 6600 scoreboard, which developed this capability.

- ✓ scoreboarding could be used in the RISC V pipeline, it is important to observe that WAR hazards, which did not exist in the RISC V floating-point or integer pipelines, may arise when instructions execute out of order.
- ✓ For example, consider following code sequence:

```
fdiv.d      f0, f2, f4
fadd.d      f10, f0, f8
fsub.d      f8, f8, f14
```

Cross Cutting Issues

- ✓ WAR hazard between fadd.d and fsub.d: If the pipeline executes the fsub.d before the fadd.d, it will violate yield incorrect execution.
- ✓ Pipeline must avoid WAW hazards. Both hazards are avoided in a scoreboard by stalling the later instruction involved in the hazard.
- ✓ Goal of a scoreboard is to maintain an execution rate of one instruction per clock cycle (when there are no structural hazards) by executing an instruction as early as possible.
- ✓ When next instruction to execute is stalled, other instructions can be issued and executed if they do not depend on any active or stalled instruction.
- ✓ Scoreboard takes full responsibility for instruction issue and execution, including all hazard detection.
- ✓ Advantage of out-of-order execution requires multiple instructions to be in their EX stage simultaneously.
- ✓ This can be achieved with multiple functional units, with pipelined functional units, or with both.

```
fdiv.d    f0,f2,f4
fadd.d    f10,f0,f8
fsub.d    f8,f8,f14
```

Cross Cutting Issues

- ✓ CDC6600 had 16 separate functional units, including 4 floating-point units, 5 units for memory references, and 7 units for integer operations.
- ✓ On a processor for RISC V architecture, scoreboards make sense. There are two multipliers, one adder, one divide unit, and a single integer unit for all memory references, branches, and integer operations.
- ✓ Both RISC V and the CDC6600 are load-store architectures, the techniques are nearly identical for two processors. Figure C.49 shows the processor looks like

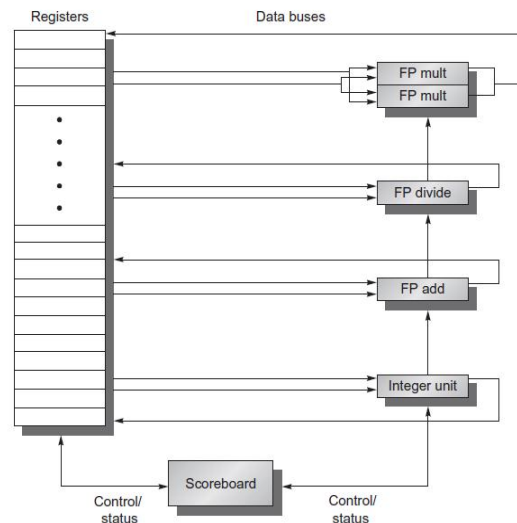


Figure C.49 The basic structure of a RISC V processor with a scoreboard. The scoreboard's function is to control instruction execution (vertical control lines). All of the data flow between the register file and the functional units over the buses (the horizontal lines, called *trunks* in the CDC 6600). There are two FP multipliers, an FP divider, an FP adder, and an integer unit. One set of buses (two inputs and one output) serves a group of functional units. We will explore scoreboarding and its extensions in more detail in Chapter 3.

Cross Cutting Issues

- ✓ scoreboard determines when the instruction can read its operands and begin execution.
- ✓ If the scoreboard decides the instruction cannot execute immediately, it monitors every change in the hardware and decides when the instruction can execute.
- ✓ scoreboard also controls when an instruction can write its result into the destination register.
- ✓ all hazard detection and resolution are centralized in the scoreboard

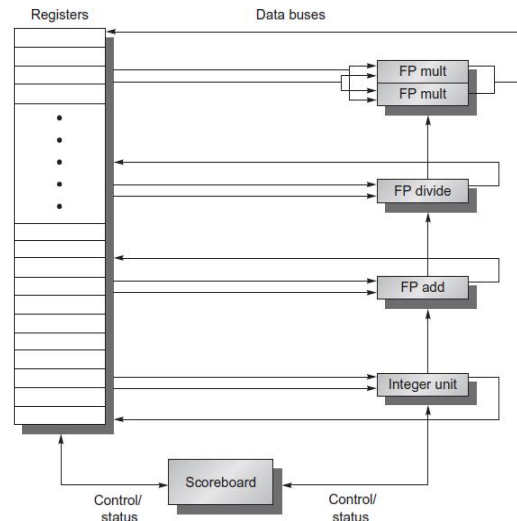


Figure C.49 The basic structure of a RISC V processor with a scoreboard. The scoreboard's function is to control instruction execution (vertical control lines). All of the data flow between the register file and the functional units over the buses (the horizontal lines, called *trunks* in the CDC 6600). There are two FP multipliers, an FP divider, an FP adder, and an integer unit. One set of buses (two inputs and one output) serves a group of functional units. We will explore scoreboarding and its extensions in more detail in Chapter 3.

Cross Cutting Issues

RISC V Pipeline with Scoreboard:

- 1. Issue:** If a functional unit for instruction is free and no other active instruction has same destination register, **scoreboard issues the instruction to the functional unit and updates its internal data structure**. This step replaces ID step in RISC V pipeline. Ensuring no other active functional unit wants to write its result into destination register.
- 2. Read operands:** Scoreboard monitors the availability of the source operands. A source operand is available if no earlier issued active instruction is going to write it. When source operands are available, scoreboard tells the functional unit to proceed to read the operands from registers and begin execution.
- 3. Execution:** Functional unit begins execution upon receiving operands. When the result is ready, it notifies scoreboard that it has completed execution. This step replaces the EX step in RISC V pipeline and takes multiple cycles in RISC V FP pipeline.
- 4. Write result:** Once scoreboard is aware that functional unit has completed execution, scoreboard checks for WAR hazards and stalls completing instruction, if necessary.

Cross Cutting Issues

- ✓ Scoreboard controls instruction progression from one step to the next by communicating with functional units.
- ✓ There are only a limited number of source operand buses and result buses to register file, which represents a structural hazard.
- ✓ Scoreboard must guarantee that number of functional units allowed to proceed into steps 2 and 4 does not exceed the number of buses available.
- ✓ CDC 6600 solved this problem by grouping 16 functional units together into four groups and supplying a set of buses, called data trunks, for each group.
- ✓ Only one unit in a group could read its operands or write its result during a clock.

Instruction Level Parallelism: Concepts and Challenges

Introduction

- ✓ Both programs and processors that limit the amount of parallelism that can be exploited among instructions whether a program property will actually limit performance and under what circumstances.
- ✓ The value of the CPI (cycles per instruction) for a pipelined processor is the sum of the base CPI and all contributions from stalls:

Pipeline CPI = Ideal pipeline CPI + Structural stalls + Data hazard stalls + Control stalls

Instruction Level Parallelism

Instruction-level parallelism (ILP): is the parallel or simultaneous execution of a sequence of instructions in a computer program.

- ✓ To increase ILP is to exploit parallelism among iterations of a loop. This type of parallelism is called loop-level parallelism. Example of a loop that adds two 1000-element arrays and is completely parallel:

```
for (i=0; i<=999; i=i+1)
  x[i] = x[i] + y[i];
```
- ✓ Every iteration of the loop can overlap with any other iteration
- ✓ Exploiting loop-level parallelism is the use of SIMD in both vector processors and graphics processing units (GPUs)
- ✓ SIMD instruction exploits data-level parallelism by operating on a small to moderate number of data items in parallel (two to eight).
- ✓ A vector instruction exploits data-level parallelism by operating on many data items in parallel using both parallel execution units and a deep pipeline.

Data Dependences and Hazards

Data Dependences: There are three different types of dependences: data dependences (also called true data dependences), name dependences, and control dependences.

An instruction j is data-dependent on instruction i if either of the following holds:

- i. Instruction i produces a result that may be used by instruction j .
- ii. Instruction j is data-dependent on instruction k , and instruction k is data dependent on instruction i .

Second condition states that one instruction is dependent on another if there exists a chain of dependences of the first type between the two instructions.

This dependence chain can be as long as the entire program. Dependence within a single instruction (such as `add x1,x1,x1`) is not considered a dependence.

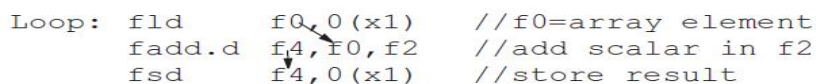
Data Dependences and Hazards

For example, consider following RISC-V code sequence that increments a vector of values in memory (starting at `0(x1)` ending with the last element at `0(x2)`) by a scalar in register `f2`.

```
Loop: fld      f0,0(x1)      //f0=array element
      fadd.d   f4,f0,f2      //add scalar in f2
      fsd      f4,0(x1)      //store result
      addi     x1,x1,-8       //decrement pointer 8 bytes
      bne      x1,x2,Loop    //branch x1≠x2
```

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

```
Loop: fld      f0,0(x1)      //f0=array element
      fadd.d   f4,f0,f2      //add scalar in f2
      fsd      f4,0(x1)      //store result
```



and integer data:

```
addi     x1,x1,-8           //decrement pointer
                                   //8 bytes (per DW)
bne      x1,x2,Loop         //branch x1≠x2
```



If two instructions are data-dependent, they must execute in order and cannot execute simultaneously or completely overlapped. Dependence implies that there would be a chain of one or more data hazards between the two instructions.

Data Dependences and Hazards

- ✓ Executing instructions simultaneously will cause a processor with pipeline interlocks to detect a hazard and stall.
- ✓ In a processor without interlocks that relies on compiler scheduling, compiler cannot schedule dependent instructions in such a way that they completely overlap.
- ✓ Data dependence in an instruction sequence reflects a data dependence in source code from which instruction sequence was generated.
- ✓ This difference is critical to understanding how instruction-level parallelism can be exploited. A data dependence conveys three things:
 - (1) the possibility of a hazard,
 - (2) the order in which results must be calculated,
 - (3) an upper bound on how much parallelism can possibly be exploited.

Data Dependences and Hazards

- ✓ Data dependence can limit the amount of instruction-level parallelism.
- ✓ A dependence can be overcome in two different ways:
 - (1) maintaining the dependence but avoiding a hazard
 - (2) eliminating a dependence by transforming the code.
- ✓ A data value may flow between instructions either through registers or through memory locations.
- ✓ When the data flow occurs through a register, detecting the dependence is straight forward because register names are fixed in instructions, although it gets more complicated when branches intervene and correctness concerns force a compiler or hardware to be conservative.
- ✓ Dependences that flow through memory locations are more difficult to detect because two addresses may refer to same location but look different: For example, 100(x4) and 20(x6) may be identical memory addresses.
- ✓ Effective address of a load or store may change from one execution of instruction to another (so that 20(x4) and 20(x4) may be different), further complicating detection of a dependence.

Name Dependences

Name dependence: occurs when two instructions use same register or memory location, called a name.

There are two types of name dependences between an instruction i that precedes instruction j in program order:

1. Antidependence: between instruction i and instruction j occurs when instruction j writes a register or memory location that instruction i reads. Original ordering must be preserved to ensure that i reads the correct value.

2. Output dependence: occurs when instruction i and instruction j write the same register or memory location. Ordering between the instructions must be preserved to ensure value finally written corresponds to instruction j .

Name Dependences

- ✓ Both antidependences and output dependences are name dependences, as opposed to true data dependences, because there is no value being transmitted between the instructions.

Register Renaming:

- ✓ Name dependence is not a true dependence, instructions involved in a name dependence can execute simultaneously or be reordered.
- ✓ if the name (register number or memory location) used in the instructions is changed so the instructions do not conflict. This renaming can be more easily done for register operands, where it is called register renaming.
- ✓ Register renaming can be done either statically by a compiler or dynamically by the hardware.

Control Dependences

Control dependence determines the ordering of an instruction i with respect to a branch instruction so that instruction i is executed.

Every instruction except for those in the first basic block of the program is control dependent on some set of branches these control dependences must be preserved to preserve program order.

Control dependence is dependence of statements. For example, in code segment

```
if p1 {
  S1;
};
if p2 {
  S2;
}
```

$S1$ is control-dependent on $p1$, and $S2$ is control-dependent on $p2$ but not on $p1$.

Control Dependences

Constraints of Control Dependences: two constraints are imposed by control dependences:

1. An instruction that is control-dependent on a branch cannot be moved before branch so that its execution is no longer controlled by branch. For example, we cannot take an instruction from the then portion of an if statement and move it before the if statement.
2. An instruction that is not control-dependent on a branch cannot be moved after the branch so that its execution is controlled by the branch. For example, we cannot take a statement before the if statement and move it into the then portion.

Two properties critical to program correctness by maintaining both data and control dependences are exception behavior and data flow

Control Dependences

- ✓ Preserving the exception behavior means that any changes in the ordering of instruction execution must not change how exceptions are raised in the program.
- ✓ Reordering of instruction execution must not cause any new exceptions in program. Consider this code sequence:

```
add x2,x3,x4
```

```
beq x2,x0,L1
```

```
ld x1,0(x2)
```

```
L1:
```

- ✓ if we do not maintain data dependence involving x2, we can change the result of program. if we ignore the control dependence and move the load instruction before branch, load instruction may cause a memory protection exception.
- ✓ no data dependence prevents us from interchanging the beq and ld; it is only the control dependence.

Control Dependences

- ✓ Data flow is the actual flow of data values among instructions that produce results and consume them.
- ✓ Branches make the data flow dynamic because they allow the source of data for a given instruction to come from many points.
- ✓ It is insufficient to just maintain data dependences because an instruction may be data-dependent on more than one predecessor.
- ✓ Program order is what determines which predecessor will actually deliver a data value to an instruction.
- ✓ Program order is ensured by maintaining control dependences. For example, consider following code fragment:

```
add x1,x2,x3
```

```
beq x4,x0,L
```

```
sub x1,x5,x6
```

```
L: ...
```

```
or x7,x1,x8
```

OR instruction is data-dependent on add and sub instructions, but preserving that order alone is insufficient for correct execution.

Control Dependences

```
add x1,x2,x3
beq x4,x0,L
sub x1,x5,x6
L: ...
or x7,x1,x8
```

- ✓ When instructions execute the data flow must be preserved: If the branch is not taken, then the value of x1 computed by the sub should be used by the OR, and if the branch is taken, the value of x1 computed by the add should be used by the OR.
- ✓ By preserving control dependence of OR on the branch, prevent an illegal change to data flow.
- ✓ SUB instruction cannot be moved above the branch.
- ✓ Speculation helps with the exception problem, which allow us to decrease the impact of control dependence while maintaining the data flow.

Software speculation: *is a technique where the compiler or software anticipates the outcome of certain instructions and executes them before they are definitely needed.*

Control Dependences

Consider the following code sequence:

```
add x1,x2,x3
beq x12,x0,skip
sub x4,x5,x6
add x5,x4,x9
skip: or x7,x8,x9
```

- ✓ Register destination of sub instruction (x4) was unused after the instruction labeled skip.
- ✓ **The property of whether a value will be used by an upcoming instruction is called liveness.**
- ✓ If x4 were unused, then changing the value of x4 before the branch would not affect the data flow because x4 would be dead after skip.
- ✓ If x4 were dead and the existing sub instruction could not generate an exception, move sub instruction before the branch because the data flow could not be affected by this change.
- ✓ If the branch is taken, sub instruction will execute and will be useless, but it will not affect the program results.
- ✓ This type of code scheduling is called software speculation because the compiler is betting on the branch outcome; and the branch is usually not taken.

Basic Compiler Techniques for Exposing ILP

Loop Unrolling: is a compiler optimization technique where a loop's body is replicated multiple times, reducing the loop control overhead and potentially improving performance.

This is achieved by replacing the loop's iterative structure with a sequence of statements, essentially expanding the code. While this can increase code size, the reduction in loop overhead can lead to faster execution, especially for loops with a small number.

Figure 3.2 shows the FP unit latencies.

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

Figure 3.2 Latencies of FP operations used in this chapter. The last column is the number of intervening clock cycles needed to avoid a stall. These numbers are similar to the average latencies we would see on an FP unit. The latency of a floating-point load to a store is 0 because the result of the load can be bypassed without stalling the store. We will continue to assume an integer load latency of 1 and an integer ALU operation latency of 0 (which includes ALU operation to branch).

Basic Compiler Techniques for Exposing ILP

We will rely on the following code segment, which adds a scalar to a vector:

```
for (i=999; i>=0; i=i-1)
```

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

x1 is initially the address of the element in the array with the highest address and f2 contains the scalar values.

Register x2 is precomputed so that Regs[x2]+8 is the address of the last element to operate on.

The straightforward RISC-V code, not scheduled for the pipeline, looks like this:

```

Loop: fld      f0,0(x1)      //f0=array element
      fadd.d   f4,f0,f2      //add scalar in f2
      fsd      f4,0(x1)      //store result
      addi     x1,x1,-8       //decrement pointer
                               //8 bytes (per DW)
      bne      x1,x2,Loop     //branch x1≠x2
```

Basic Compiler Techniques for Exposing ILP

Example Show how the loop would look on RISC-V, both scheduled and unscheduled, including any stalls or idle clock cycles. Schedule for delays from floating-point operations.

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

		<u>Clock cycle issued</u>
Loop:	<code>fld f0,0(x1)</code>	1
	<code>stall</code>	2
	<code>fadd.d f4,f0,f2</code>	3
	<code>stall</code>	4
	<code>stall</code>	5
	<code>fsd f4,0(x1)</code>	6
	<code>addi x1,x1,-8</code>	7
	<code>bne x1,x2,Loop</code>	8

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

```

Loop: fld    f0,0(x1)
      addi   x1,x1,-8
      fadd.d f4,f0,f2
      stall
      stall
      fsd    f4,8(x1)
      bne    x1,x2,Loop
  
```

The stalls after `fadd.d` are for use by the `fsd`, and repositioning the `addi` prevents the stall after the `fld`.
