Q1: Answer the following questions briefly:

i. How does the technique of internal forwarding resolve data hazards in a pipelined processor? (4)

ANS: The problem posed in Figure C.4 can be solved with a simple hardware technique called forwarding (also called bypassing and sometimes short-circuiting).

The key insight in forwarding is that the result is not really needed by the sub until after the add actually produces it. If the result can be moved from the pipeline register where the add stores it to where the sub needs it, then the need for a stall can be avoided. Using this observation, forwarding works as follows:

1. The ALU result from both the EX/MEM and MEM/WB pipeline registers is always fed back to the ALU inputs.
2. If the forwarding hardware detects that the previous ALU operation has written the register corresponding to a source for the current ALU operation, control logic selects the forwarded result as the ALU input rather than the value read from the register file.

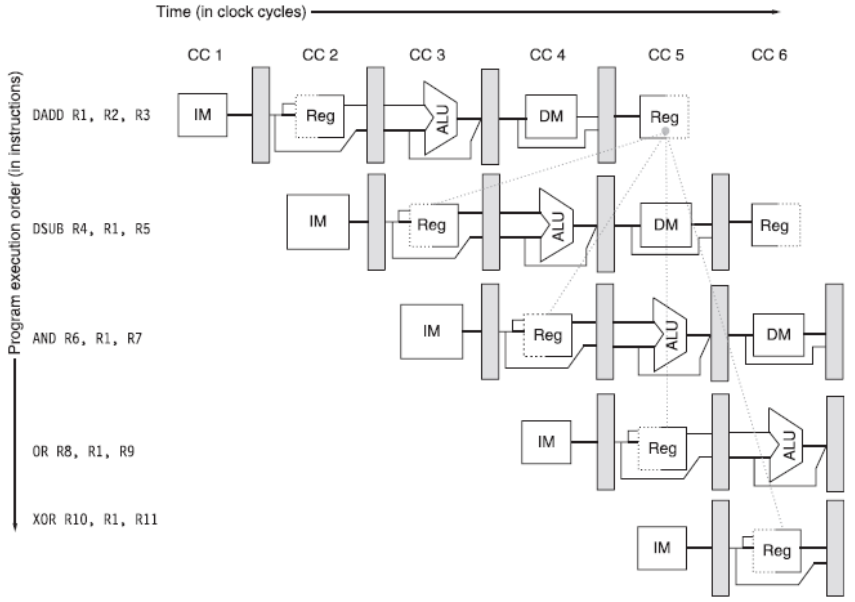


Figure C.4 The use of the result of the add instruction in the next three instructions causes a hazard, because the register is not written until after those instructions read it.

ii. Classify the following exceptions as synchronous or asynchronous: (4)

(a) Page fault (b) Printer error
(c) Stack overflow (d) Memory protection violation

ANS:

Synchronous exceptions: If the event occurs at the same place every time the program is executed with the same data and memory allocation, the event is synchronous.

Asynchronous exceptions: With the exception of hardware malfunctions, asynchronous events are caused by devices external to the processor and memory. Asynchronous events usually can be handled after the completion of the current instruction, which makes them easier to handle.

Synchronous exceptions	**Asynchronous exceptions**
Page fault	Printer error
Stack overflow	
Memory protection violation	

iii. Which exceptions are the most difficult to handle in pipelined processors and why? (4)

ANS: As in unpipelined implementations, the most difficult exceptions have two properties:(1) they occur within instructions (that is, in the middle of the instruction execution corresponding to EX or MEM pipe stages), and (2) they must be restartable. In our RISC V pipeline, for example, a virtual memory page fault resulting from a data fetch cannot occur until sometime in the MEM stage of the instruction. By the time that fault is seen, several other instructions will be in execution. A page fault must be restartable and requires the intervention of another process, such as the operating system. Thus, the pipeline must be safely shut down and the state saved so that the instruction can be restarted in the correct state.

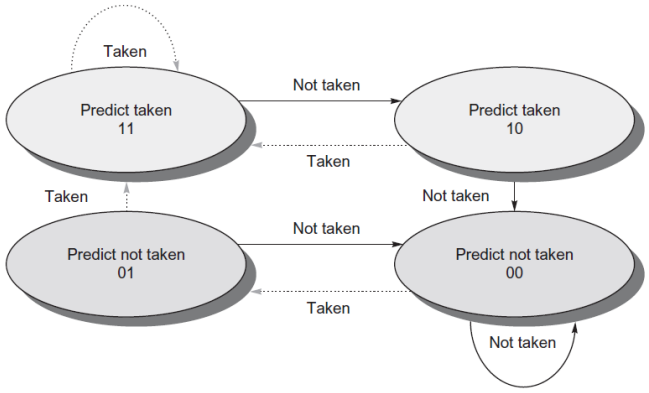
iv. What do you understand by the latency and initiation interval of a pipelined functional unit? (4)

ANS: We define latency the same way we defined it earlier: the number of intervening cycles between an instruction that produces a result and an instruction that uses the result. The initiation or repeat interval is the number of cycles that must elapse between issuing two operations of a given type.

v. How does the 2-bit dynamic branch predictor based on Branch History Table work? (4)

The simplest dynamic branch-prediction scheme is a branch-prediction buffer or branch history table. A branch-prediction buffer is a small memory indexed by the lower portion of the address of the branch instruction. The memory contains a bit that says whether the branch was recently taken or not. This scheme is the simplest sort of buffer; it has no tags and is useful only to reduce the branch delay when it is longer than the time to compute the possible target PCs.

In a 2-bit scheme, a prediction must miss twice before it is changed.



vi. What is a basic block? Suggest a compiler-based technique to improve the size of the basic block. (4)

a basic block—a straight-line code sequence with no branches in except to the entry and no branches out except at the exit—is quite small. The simplest and most common way to increase the ILP is to exploit parallelism among iterations of a loop. This type of parallelism is often called loop-level parallelism. We will examine a number of techniques for converting such **loop-level parallelism** into instruction-level parallelism. Basically, such techniques work by unrolling the loop either statically by the compiler

vii. What are the advantages of dynamic scheduling over static scheduling of pipelined processors? (4)

ANS: dynamic scheduling, where the hardware rearranges the instruction execution to reduce the stalls. Dynamic scheduling offers several advantages:

- It enables handling some cases when dependencies are unknown at compile time (e.g., because they may involve a memory reference);
- It simplifies the compiler;
- It allows code that was compiled with one pipeline in mind to run efficiently on a different pipeline.

These advantages are gained at a cost of a significant increase in hardware complexity.

Q2(a) A processor comprises of five steps of execution having the following delays :S 1=32nSec•, S2 = 25nSec•, S3 = 20nSec; S4 = 25nSec; S5= 20nSec. If the processor is pipelined, what should be the clock frequency if the pipeline latch delay is ln Sec? Give the speedup of the pipelined processor over its non-pipelined counterpart. (6)

Given-

Five stage pipeline is used

Delay of stages = 32, 25, 20,25 and 20 ns

Latch delay or delay due to each register =1 ns

Pipeline Cycle Time-

Cycle time = Maximum delay due to any stage + Delay due to its register

= Max {32, 25, 20,25, 20 } + 1 ns

= 32 ns + 1 ns

= 33 ns

$$\text{Clock frequency} = \frac{1}{Clock\ cycle} \qquad \text{Clock frequency} = \frac{1}{33 \times 10^{-9}} = 30.303 \times 10^{6}$$

$$\boldsymbol{Clock\ frequency = 30.303\ MHZ}$$

Non-Pipeline Execution Time-

Non-pipeline execution time for one instruction

= (32+25+ 20+25+ 20) ns **= 122 ns**

Speed Up Ratio-

Speed up = Non-pipeline execution time / Pipeline execution time

= 122 ns / 33 ns **Speed up = 3.697**

Q2(b) What is loop unrolling and how does it improve the performance of pipelined processors? What other techniques used to augment loop unrolling, in order to reduce pipeline stalls? (6)

ANS: Loop unrolling is a compiler optimization applied to certain kinds of loops to reduce the frequency of branches and loop maintenance instructions. It is easily applied to sequential array processing loops where the number of iterations is known prior to execution of the loop.

A simple scheme for increasing the number of instructions relative to the branch and overhead instructions is loop unrolling. Unrolling simply replicates the loop body multiple times, adjusting the loop termination code. Loop unrolling can also be used to improve scheduling. Because it eliminates the branch, it allows instructions from different iterations to be scheduled together.

Q3(a) The problems observed in Scoreboard technique of dynamic pipeline scheduling include the lack of internal forwarding and name dependences. Does Tomasulo's approach resolve these issues? How? (6)

ANS: The primary difference is that Tomasulo's algorithm handles antidependences and output dependences by effectively renaming the registers dynamically.

In Tomasulo's scheme, register renaming is provided by reservation stations, which buffer the operands of instructions waiting to issue and are associated with the functional units. The basic idea is that a reservation station fetches and buffers an operand as soon as it is available, eliminating the need to get the operand from a register.

This bypassing is done with a common result bus that allows all units waiting for an operand to be loaded simultaneously (on the 360/91, this is called the common data bus, or CDB). In pipelines that issue multiple instructions per clock and also have multiple execution units, more than one result bus will be needed.

Q3(b) The following code sequence is executed in a processor using Tomasulo's approach of dynamic scheduling. Give the entries of Reservation stations, Load and Store buffers and Register status for the given status of the processor. The processor comprises of the following resources: (14)

-Three reservation stations for Add/Sub functional unit

-Two reservation stations for multiply/divide functional unit

-Four load buffers

-Four are store buffers

			Issue	Execute	write result
1	fld	f0, 16(x1)	✓	✓	✓
2	fld	f2, 32(x2)	✓	✓	
3	fadd.d	f4,f2,f0	✓		
4	fsub.d	f6,f4,f8	✓		
5	fsd	f6,100(x3)	*		
6	fld	f0, 24(x1)	*		
7	fld	f2, 40(x2)	*		
8	fmul.d	f10,f0,f2	*		
9	fadd.d	f8,f10,f4	*		
10	fsd	f8,93(x3)	*		

Note: Determine whether the instruction with "*" in the issue filed are issued or not.

Reservation stations

Name	Busy	Op	Vj	Vk	Qj	Qk	A
Load1	No						
Load2	Yes	Load					32 + Regs[x2]
Add1	Yes	ADD		Mem[16 + Regs[x1]]	Load2		
Add2	yes	SUB		Regs[f8]	Add1		
Store1	yes	Store	100+ Regs[x3]			add2	
Load3	Yes	Load					24 + Regs[x1]
Load4	Yes	Load					40 + Regs[x2]
Mult1	Yes	MUL			Load3	Load4	
Add3	Yes	ADD			Mult1	add1	
Store2	Yes	Store	93+ Regs[x3]			add3	

Register status

Field	f0	f2	f4	f6	f8	f10	f12
Qi	Load3	Load2	Add1	Add2	Add3	Mult1	…

			Issue	Execute	write result
1	fld	f0, 16(x1)	✓	✓	✓
2	fld	f2, 32(x2)	✓	✓	
3	fadd.d	f4,f2,f0	✓		
4	fsub.d	f6,f4,f8	✓		
5	fsd	f6,100(x3)	✓		
6	fld	f0, 24(x1)	✓		
7	fld	f2, 40(x2)	✓		
8	fmul.d	f10,f0,f2	✓		
9	fadd.d	f8,f10,f4	✓		
10	fsd	f8,93(x3)	✓		

All instructions are issued as we have reservation stations available for all instructions.