PIPELINE HAZARDS

There are situations, called hazards, that prevent the next instruction in the instruction stream from executing during its designated clock cycle.

Hazards reduce the performance from the ideal speedup gained by pipelining.

REAL WORLD ANALOGY :: RESOURCES HAZARD

There are three different types of hazard relate to pipelining one of these hazards is explained using Car Manufacture pipeline example.

Assume that we have one employee that assembles and paints the car as well.

LHS table shows the ideal case, but there is a conflict of resource here, P and A cannot be done at same time.

This is called **resource hazard** with one object cannot proceed in pipeline because the resource need by it is busy with other object.

RHS table show that one solution can be that car 2 waits for A until P of car1 is complete (Car 3 waits for A till P of Car 2 is complete)

Another solution can be to add more resources, in this case hire another employee for paint job

Time line	1st hour	2nd hour	3rd hour	4th hour	5th hour
Car 1	A	P	**T**		
Car 2		A	**P**	T	
Car 3			**A**	P	T

Time line	1st hour	2nd hour	3rd hour	4th hour	5th hour	6th hour	7th hour
Car 1	A	P	**T**				
Car 2		idle	A	**P**	T		
Car 3				idle	**A**	P	T

PIPELINE HAZARD IN COMPUTER

A pipeline hazard occurs when the pipeline, or some portion of the pipeline, must stall because conditions do not permit continued execution.

Such a pipeline stall is also referred to as a pipeline bubble.

Three Hazard types:

- **Structural/Resource**
  - same resource is needed by multiple instruction in the same cycle
- **Data**
  - Instruction depends on result of prior instruction still in the pipeline data dependencies limit pipelining
- **Control**
  - Next executed instruction may not be the next specified instruction

PIPELINE HAZARD IN COMPUTER

Can always resolve hazards by stalling

More stall cycles = more CPU time = less performance

Increase performance = decrease stall cycles

STRUCTURAL/RESOURCE HAZARD

A resource hazard occurs when two (or more) instructions that are already in the pipeline need the same resource.

The result is that the instructions must be executed in serial fashion rather than parallel for a portion of the pipeline.

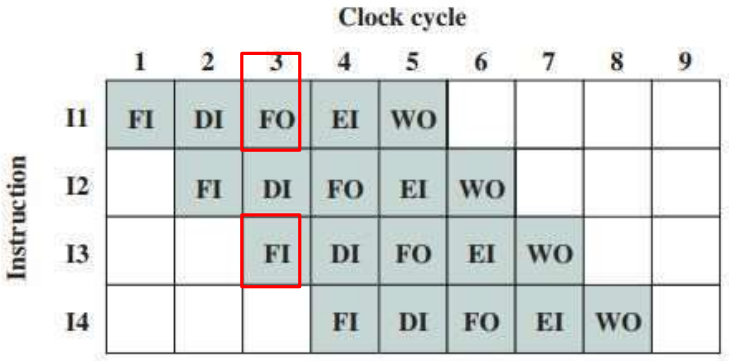
Examples:

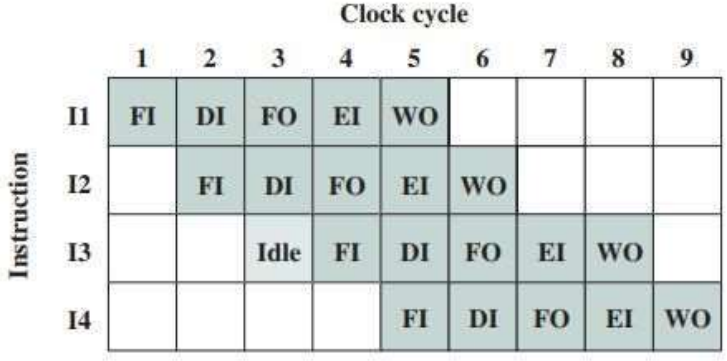
- Two accesses to a single ported memory
- Two operations need the same function unit at the same time
- Two operations need the same function unit in successive cycles, but the unit is not pipelined

Solutions:

- Stalling
- Add more hardware

STRUCTURAL/RESOURCE HAZARD EXAMPLE





Assume the there is only single port to memory

FO of I1 and FI of I3, both require reading from memory at cycle 3

Assume that for operand for rest of the Instructions are registers.

There resource conflict between I1 and I3

I3 must wait

DATA HAZARDS

Data hazards occur when the pipeline changes the order of read/write accesses to operands so that the order differs from the order seen by sequentially executing instructions on an unpipelined processor.

Three types of data dependencies defined in terms of how succeeding instruction depends on preceding instruction

- RAW: Read after Write or Flow dependency
- WAR: Write after Read or anti-dependency
- WAW: Write after Write

RAW

Read After Write (RAW)

- $Instr_J$ tries to read operand before $Instr_I$ writes it
- I: ADD EAX, EBX
- J: SUB ECX, EAX

- EAX is 32 bit register

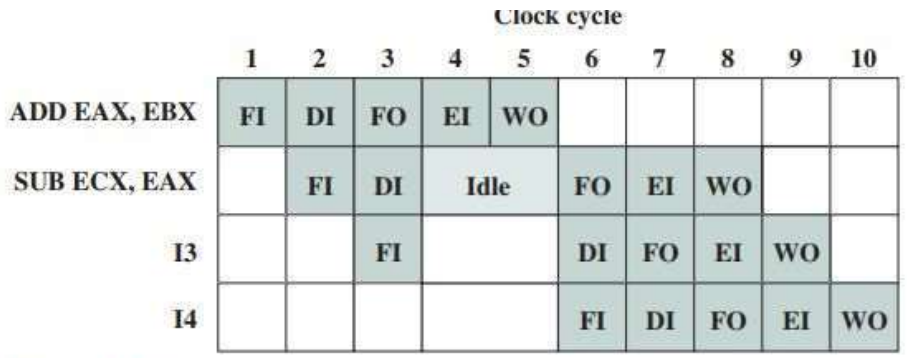


Figure 14.16 Example of Data Hazard

WRITE AFTER READ (WAR) OR ANTIDEPENDENCY

An instruction reads a register or memory location and a succeeding instruction writes to the location.

A hazard occurs if the write operation completes before the read operation takes place.

- I: read from location X
- J: write to location X

- Incorrect answer if J finishes before I

WRITE AFTER WRITE (WAW), OR OUTPUT DEPENDENCY

Two instructions both write to the same location.

A hazard occurs if the write operations take place in the reverse order of the intended sequence.

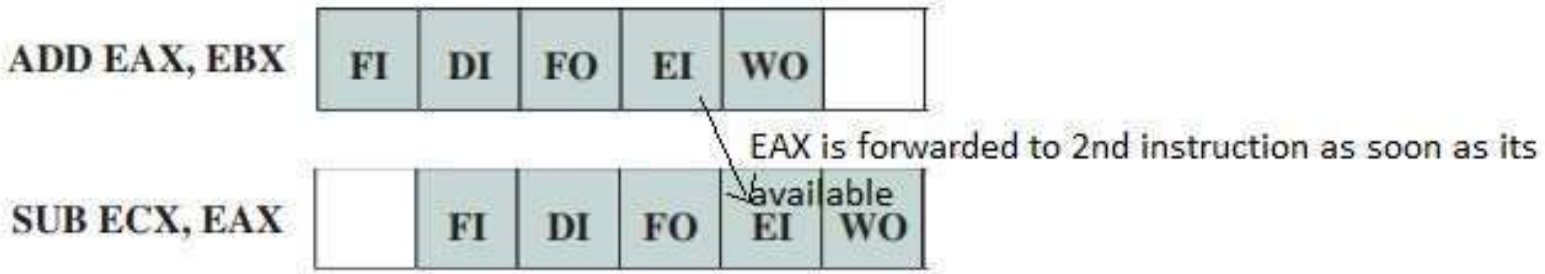
- I: write to location X
- J: write to location X

- Incorrect answer if J finishes before I

DATA HAZARD

Solution

- Stalling
- Logic Detection
- Data Forwarding
  - Used in RAW dependency,
  - Data is forwarded to next instruction as soon as its available so that next instruction doesn't have to wait for write operand
  - Needs changes in hardware



CONTROL HAZARD

A control hazard, also known as a branch hazard

Occurs when the pipeline makes the wrong decision on a branch prediction

Therefore brings instructions into the pipeline that must subsequently be discarded

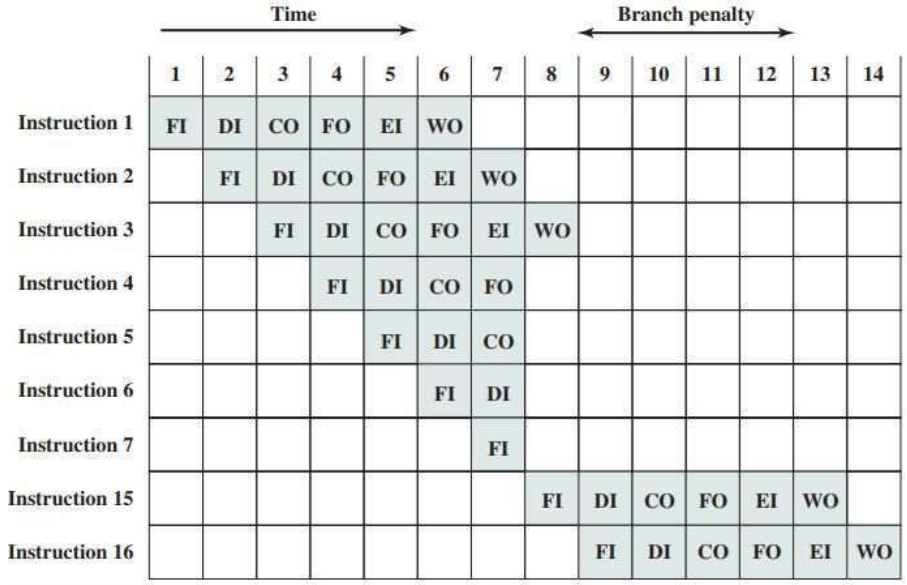


Figure 14.11 The Effect of a Conditional Branch on Instruction Pipeline Operation

If instruction 3 was conditional branching to instruction 15

Until the instruction is actually executed, it is impossible to determine whether the branch will be taken or not.

If I3 takes jump to I15, all the work done for i4-i7 is wasted

DEALING WITH BRANCHES

A variety of approaches have been taken for dealing with conditional branches

- Multiple streams
- Pre-fetch branch target
- Loop buffer
- Branch prediction (will see this in detail)
- Delayed branch

BRANCH PREDICTION

- Predict never taken
- Predict always taken
- Predict by opcode
- Taken/not taken switch
- Branch history table

PREDICT NEVER TAKEN / PREDICT ALWAYS TAKEN

Simplest approaches

Either always assume that the branch will not be taken and continue to fetch instructions in sequence, or they always assume that the branch will be taken and always fetch from the branch target.

The predict-never-taken approach is the most popular of all the branch prediction methods. (Because its simple)

Studies analyzing program behavior have shown that conditional branches are taken more than 50% of the time [LILJ88]

So if the cost of prefetching from either path is the same, then predict always taken works better

PREDICT BY OPCODE

Makes the decision based on the opcode of the branch instruction

The processor assumes that the branch will be taken for certain branch opcodes and not for others.

[LILJ88] reports success rates of greater than 75% with this strategy.

TAKEN/NOT TAKEN SWITCH OR BRANCH HISTORY TABLE

Based on history

One or more bit required to keep a track of decision in last branch

Figure shows that if last two branches were not taken, next prediction will be not taken and vise versa.

Similarly there can be many variants to optimally use and keep branch history

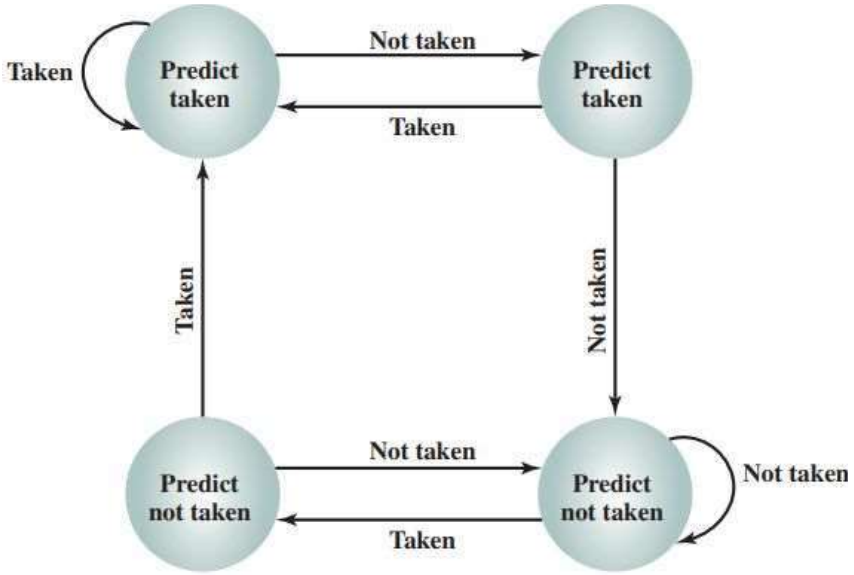


Figure 14.19 Branch Prediction State Diagram