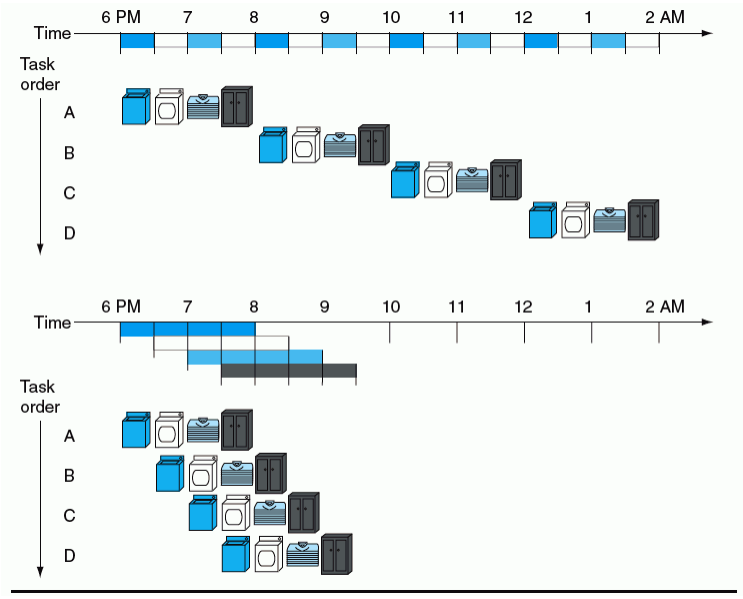
Pipelining in detail

Chapter-6 (TB2: Computer Organization & Design 3rd edition)

Overview of Pipelining (Section 6.1)

- Pipelining laundry example
- Non-pipelined version vs Pipelined version



- Non-pipelined version
  - Four students take 8 hours to wash all clothes
- Pipelined version
  - Four students take 3.5 hours to wash all clothes
- Ideally if there are infinite number of students, speed up gain will be 4 times.
  - Because four equal tasks are running in parallel.

- In above example, speed-up gain is only 8/3.5=2.3 times
  - Because, for first three half hours, not all are running in parallel.
  - For last three half an hours, not all are running in parallel.

MIPS Example – Single Cycle vs Pipelined Implementation

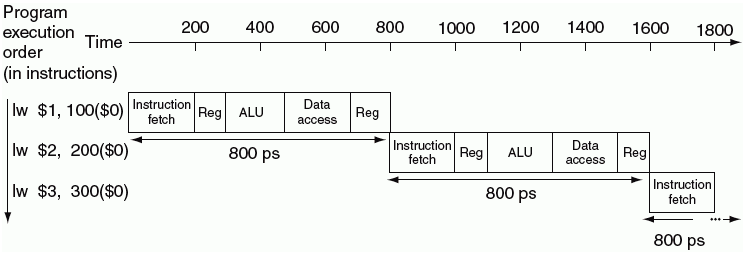
- Lets take a MIPS processor with the following eight instructions
  - Load Word (LW)
  - Store Word (SW)
  - Add (add)
  - Subtract (sub)
  - And (and)
  - Or (or)
  - Set-less-than (slt)
  - Branch-on-Equal (beq)

- **Single cycle implementation of MIPS**
  - In a single cycle implementation of MIPS, every instruction takes one clock cycle to complete an instruction.
  - 1-clock cycle stretched to accommodate all the instructions.
  - Table below shows the time taken by each functional unit
    - IF takes 200 ps
    - ID/Register read takes 100 ps
    - ALU takes 200 ps
    - Memory takes 200 ps
    - Register writeback takes 100ps
  - So total time taken by each instruction is computed.
    - LW takes 800ps
    - SW takes 700 ps
    - Add, Sub, and, or takes 600 ps
    - Branch takes 500 ps

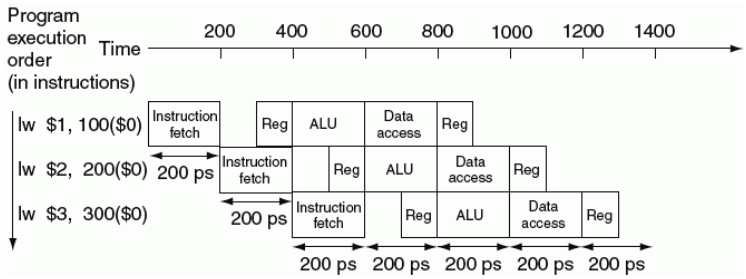
Instruction class	Instruction fetch	Register read	ALU operation	Data access	Register write	Total time
Load word (`lw`)	200 ps	100 ps	200 ps	200 ps	100 ps	800 ps
Store word (`sw`)	200 ps	100 ps	200 ps	200 ps		700 ps
R-format (`add, sub, and, or, slt`)	200 ps	100 ps	200 ps		100 ps	600 ps
Branch (`beq`)	200 ps	100 ps	200 ps			500 ps

FIGURE 6.2 Total time for each instruction calculated from the time for each component. This calculation assumes that the multiplexors, control unit, PC accesses, and sign extension unit have no delay.

  - Single cycle implementation has to decide one clock
    - The clock must accommodate all the instructions, so it must be selected by seeing the slowest instruction.
    - Clock time period selected min 800 ps, not less.
  - Each instruction takes 800 ps to complete.
  - Execution of three single-cycle, non-pipelined MIPS instruction takes 2400ps as shown in figure below



- **Pipelined implementation of MIPS**
  - All pipeline stages takes a single cycle to execute.
  - So each clock cycle must be large enough to accommodate the slowest stage.
  - Pipelined clock should minimum have the time-period of 200ps.
  - Execution of three pipelined MIPS instruction takes 1400ps as shown in figure below
  - Once pipelined in filled, each new instruction is completed every 200ps. (i.e. 4 times speedup gain)



- **Speedup formula**

  If all stages are perfectly balanced (i.e. having same delay), then the speedup formula will be

$$\text{Time between instructions}_{\text{pipelined}} = \frac{\text{Time between instructions}_{\text{nonpipelin}}}{\text{Number of pipe stages}}$$

Time between Insts (Pipelined) = 800ps/5
= 160ps

- However, in our case, when we have infinite number of instruction, Time between Insts is 200ps, because we have imbalanced delay of FUs.
- Moreover, there are few other delays as well in the pipelined.
- In short, in pipelining, we never achieve the maximum delay. We only tend to be closer to the ideal delay.
- In our case, 2400ps vs 1400 ps not close to ideal because we have less number of instructions.

- **Increase instructions**
- Instead of 3 instructions, now try 1000003 instructions.
- In non-pipelined version, total time taken will be 800 x 1000003 = 800002400 ps
- In pipelined version, total time taken will be 800 ps (to fill first four stages) + 1000003 * 200ps = 200001400
- Total speedup = 800002400/200001400 = 3.99998

Pipeline improves performance by improving throughput not delay.

Designing Instruction Set for Pipelining

- *Instruction length* should be same, easier to fetch instruction. In MIPS all instructions are 32-bit wide.
  - In IA-32, instruction length vary from 1 to 17 bytes.
- *Instruction length* should be same, Its easy to decode same length instructions.
- *Few instruction formats*: Easy to decode, easy to read source operands.
- *Memory operations only in load/store*: Address can be computed in execute phase. If memory operands were allowed in arithmetic instructions, we required two execute phase. (one for computing the address then memory was fetcher, later on another execute was required to compute the arithmetic operation)
- *Memory must be aligned*: We don't worry if one byte or four bytes are read from memory, they take same time.

Pipeline Hazards

- ***Pipeline hazard is a situation in which next instruction cannot execute in the following clock cycle.***
- There are three types of hazards
  - Structural hazards
  - Data hazards
  - Control hazards

Structural Hazards

- Cannot execute an instruction combination due to unavailability of a resource.
- IF MIPS had only one memory. The same memory was used to
  - Fetch instruction
  - Load/store data
- We have had a structural hazard with four instructions
  - First instruction wants to get data from memory
  - Fourth instruction wants to fetch inst from mem.

1.	IF	ID	Exe	Mem	Wb		
2.		IF	ID	Exe	Mem	Wb	
3.			IF	ID	Exe	Mem	Wb
4.				IF	ID	Exe	Mem	WB

- Without two memories, we will experience a structural hazard.

Data Hazard

- An occurrence in which a planned instruction cannot execute in proper clock cycle because data that is needed to execute the instruction is not yet available.
- Data hazard occurs due to data dependency
    - Data dependency on an instruction which is still in pipeline.
- Example
    - ADD S0, T0, T1
    - SUB T2, S0, T3

- For the above example,
    - 'Add' instruction writes value of S0 in clock 5
    - However 'sub' instruction reads it in clock 3

1	2	3	4	5		
IF	ID	EXE	MEM	WB		
	IF	ID	EXE	MEM	WB	
		IF	ID	EXE	MEM	WB

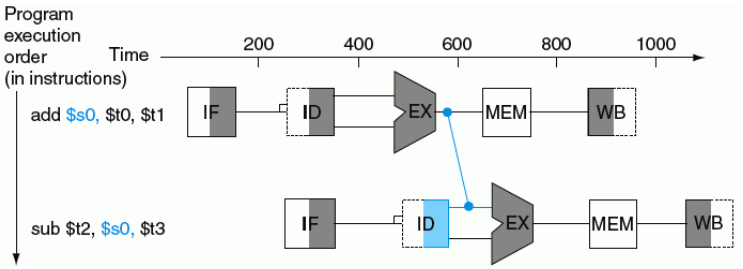
- **Solution -1**
    - Insert nops, or bubbles.
    - No instruction fetched during nop.
    - Three nops inserted.
    - Not a good solution. Waste of time.
    - Modified pipeline becomes as follows

1	2	3	4	5			
IF	ID	EXE	MEM	WB			
	Nop	nop	nop	nop	nop		
		Nop	nop	nop	nop	nop	
			Nop	nop	nop	nop	nop
				IF	ID	EXE	MEM

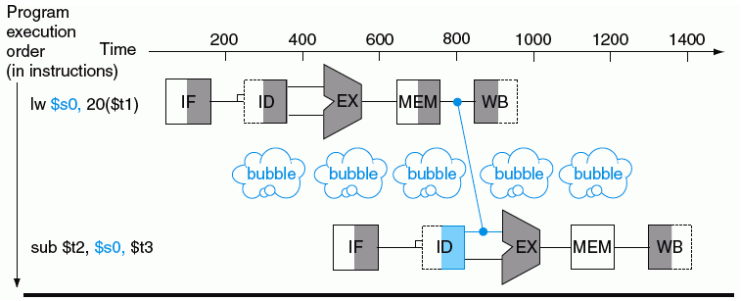
- **Solution -2**
    - Rely on optimizing compilers.
    - Can be used, but not reliable (as dependency can be data dependency as well)

- **Solution-3**
    - Implement forwarding or bypassing. It is a method to resolve data hazard by retrieving the missing data element from internal buffers, rather than waiting for it to arrive from programmer-visible register

- Graphical representation of forwarding is as follows



- Forwarding cannot resolve all stalls, like this one.



- Stall or bubble is required in the above case. Hardware must detect these cases, and implement these stalls. We cannot rely on software (compiler) to resolve these issues.

Reordering Code to avoid Pipeline stalls

- Even if stalls are properly detected by hardware (i.e. interlocking is implemented), we need to minimize these stalls to get more throughput.
- Code can be reordered to remove these stalls.
- Consider the following C code
    - A = B + E
    - C = B + F
- The MIPS code for the above C code will be like

```
lw      $t1, 0($t0)
lw      $t2, 4($t0)
add     $t3, $t1,$t2
sw      $t3, 12($t0)
lw      $t4, 8($01)
add     $t5, $t1,$t4
sw      $t5, 16($t0)
```

- **Explanation of above code**
- Load B, E
- Add B and E
- Store the added result in A

- Load F

- Data hazard in both add operations.
- These hazards can be removed by rearranging the code as follows
    - I.e. in a pipelined processor with forwarding, this re-ordered code will take 2 less cycles.

```
lw      $t1, 0($t0)
lw      $t2, 4($t0)
lw      $t4, 8($t0)
add     $t3, $t1,$t2
sw      $t3, 12($t0)
add     $t5, $t1,$t4
sw      $t5, 16($t0)
```

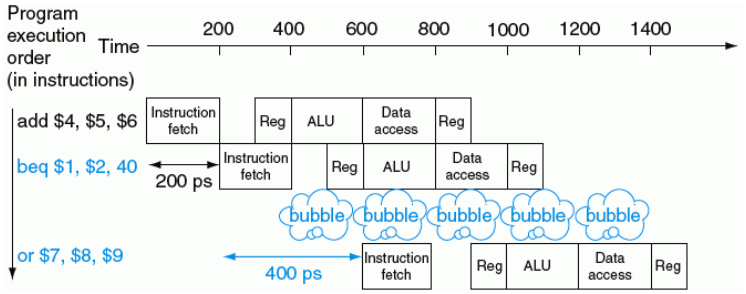
Control hazards

- Operation cannot execute in proper clock cycle because the instruction fetched is not the instruction to be executed.
- When a branch instruction is fetched, how quickly can we calculate the new PC address.
    - Generally after the execution of EXE phase.
- So when we have executed the EXE phase, our 2nd instruction has completed ID phase, our 3rd instruction has been fetched.
- If branch is taken, it means that we have to remove 2nd and 3rd instruction.

1	2	3	4	5		
IF	ID	EXE	MEM	WB		
	IF	ID	EXE	MEM	WB	
		IF	ID	EXE	MEM	WB

Solution 1

- Update the ID phase in such a manner that perform the following tasks in ID phase
    - Test register
    - Calculate branch address
    - Update PC
- Even with this solution, we need one stall as shown below
- Should we always add this delay?

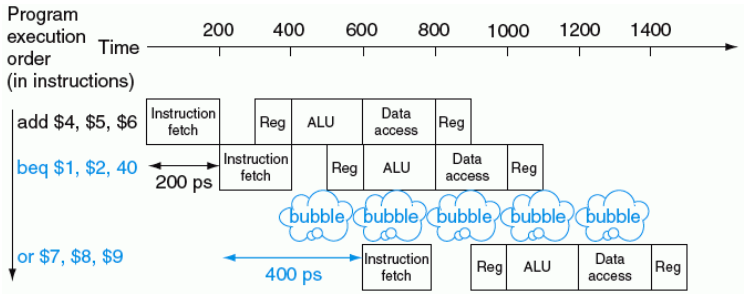


Solution 2 Branch prediction

- Calculating branch address and updating PC in ID phase is too costly.
- Solution 2 is to predict about branch.
- Simple solution is to always predict that branch will be untaken. Continue loading the same instruction.
    - When prediction is correct, continue as it is
    - When prediction is wrong, insert stall.

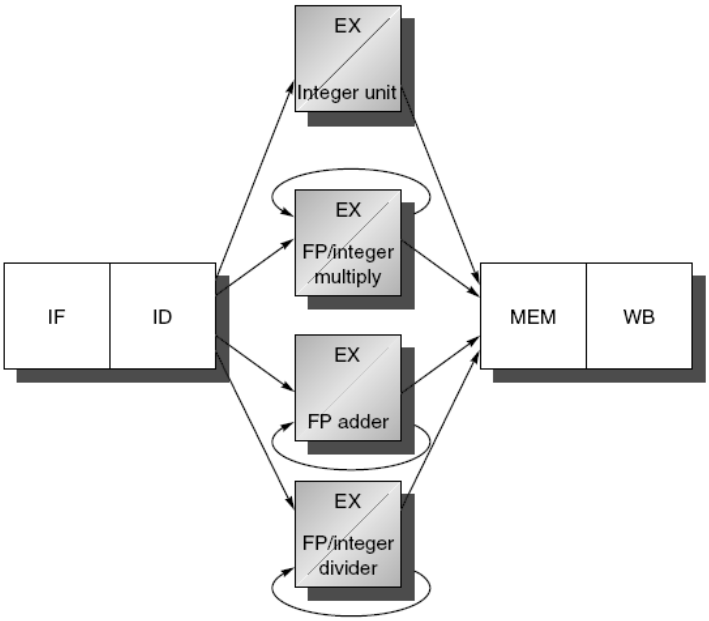
Soution 3 Delayed branch decision

- IN the below given figure, assembler can move the add after the branch, and will always execute it.



A.5 Extending the MIPS Pipeline to Handle Multi-cycle operations

- Floating point operations, or multiplication division cannot execute in one clock cycle.
- Lets assume we have four separate FUs
  - Integer ALU operation + branches
  - FP and integer multiplier
  - FP adder
  - FP and integer divider
- Assume that all these above blocks are non-pipelined.
- How should we handle them.



- FP/Integer Multiplier/Divider will simply loop back
- Only one instruction issued every cycle.
- Since each block is non-pipelined, no instruction can be issued, until previous instruction has completed execution.
- With pipelined components we will be having MIPS pipeline as follows.