Pipelining

Pipelining is one architectural technique for improving speed of programs. Multiple stages of different instructions are executed in different sections of the processor i.e. overlapped execution of instructions. *note that it improves throughput and not latency*
An easier way to think of it is an assembly line of a car, with a new car rolling out once every few seconds but each car taking days.

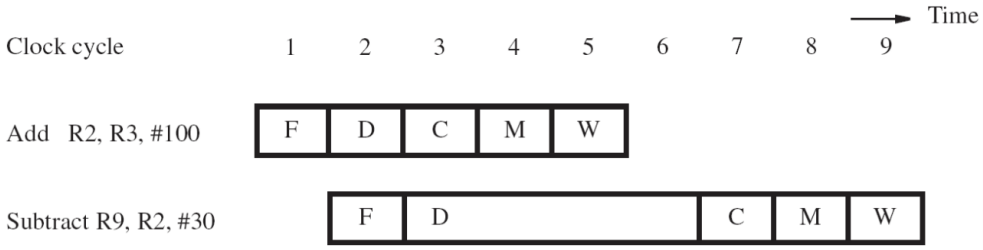
0.0.1 Hazards

Although the throughput is improved generally, there are often cases where the pipeline must stall.

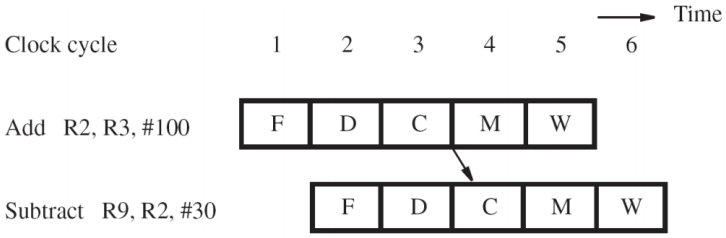
- Data Dependency: The computation of Instruction i, depends on the computation result of an instruction further ahead in the pipeline [*data hazard* specifically Read-after Write [*RAW*]]
- Memory Delay: Memory takes too long to read or write (i.e. MFC takes a while to assert)
- Branch conditions: We realize the condition is a branch, so must discard any computation of the instructions right after the branch

Hazard: Any condition that causes the pipeline to stall is called a hazard How to get rid of data hazards:

1. Stall: Stall the pipeline, by forcing a NOP [no operation *bubble*] on each of the instruction ahead in the pipeline



2. Operand Forwarding: The result of the computation is actually determined in the compute stage, and can be simply forwarded by connecting a wire between RZ and RA/RB.
   This allows to handle the hazard without stalling the pipeline



   Although connecting wires is easy, this also expands the logic of our multiplexers at teh register stage, where they now have to choose between the register file and the new RZ [among other for RB]

Memory Stalls still exist, even when there is a cache hit, and must be accounted for. Cache hit resolves in one cycle, i.e you must wait one cycle at the least. If there is a cache miss then you must wait longer.
To solve this:

1. Stall until MFC [have to regardless of cache hit/miss]
2. Forward the result from RY to ALU inputs

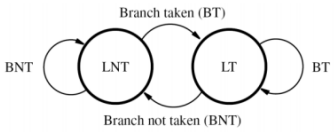
0.0.2 Branch Conditions

- **Unconditional Branch Delays**:
  - Branch instruction identified at Decode stage, and target address determined at Compute Stage.
  - This causes 2 instructions to be fetched, and one of them to be decoded. They must be discarded i.e. 2 cycle penalty
  - *Fast Branch:* To solve this, just insert another adder inside the decoder to compute the target address i.e. you are now at a one cycle penalty
- **Conditional Branches**: Need to evaluate a condition as well now
  - Adder still in decode stage for target address
  - But comparison still in Execute Stage i.e. still the 2 cycle penalty
  - Add comparator to decode stage i.e. return to 1 cycle penalty

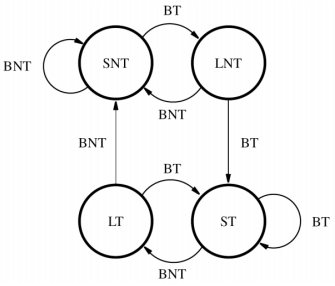
Branch Delay Slot: The instruction right after the branch is the one that is discarded if the branch is taken. This location is called the branch delay slot. We can make effective use of this slot to avoid discarding it. This can be done by re-ordering the instructions to get a non-dependent instruction below the branch the condition from above. This way the instruction fetched after the branch is decoded is always correct, regardless of the branch decision. This can only be done by the compiler, and is not always possible and is called *delayed branching*. If a useful instruction is inserted then the delay is zero, otherwise a NOP causes a delay of 1.

9.0.2.1 Branch Prediction

1. Static Branch Prediction: Assume that branch will always not be taken [penalty discovered in decode stage, and predicted accuracy should be 50%] Accuracy is actually lower than 50 this way, because useful programs have loops, and we can improve accuracy by imposing a condition that if it's a backward branch it'll generally be taken, and a forward branch wont be [the sign of the offset is enough].
2. Dynamic Branch prediction
   (a) **2-state algorithm**: likely taken (LT) or likely not taken (LNT). Basically assume that the branch will go the same way it went last time, and update state if it doesn't. This prediction, might be changing it's predictions too fast. (2 nested loops, one longer 100, one just 3)



   (b) **4-state algorithm**: : strongly taken (ST), likely taken (LT), strongly not taken (SNT), likely not taken (LNT)



9.0.2.1 Branch Target Buffer

A Branch Target Buffer (BTB) is a buffer that stores a mapping from PC of branch instructions to their computed target address based on earlier calculations, also stores the branch prediction algorithm bits.
If we now move predictor and BTB to the fetch stage, and comparator to decode stage then:

1. Cycle 1: use branch instruction address lookup, and then use the bits for prediction
2. Cycle 2: Fetch next instruction using the prediction
3. Cycle 3: If incorrect prediction, correct mistake in Decode stage and fetch the correct next instruction

0.1 Performance Evaluations

- T: execution time (seconds/program)
- N: dynamic instruction count (instructions/program)
- S(CPI): average number of clock cycles to fetch and execute one instruction (cycles/instruction)
- R: clock rate (cycles/second)

For Non-pipelined:

$$T = N \times S \times \frac{1}{R}$$

Throughput(P):

$$P_{np} = \frac{R}{S} \quad [S = 5 \text{ for non-pipelined; } S = 1 \text{ for pipelined, ideal case i.e. no stalls}]$$

If we account for stalls:

$$P_p = \frac{R}{1 + \delta_{stall}} = \frac{R}{1 + \delta_{load} + \delta_{branch}}$$