Topic:

Pipelining

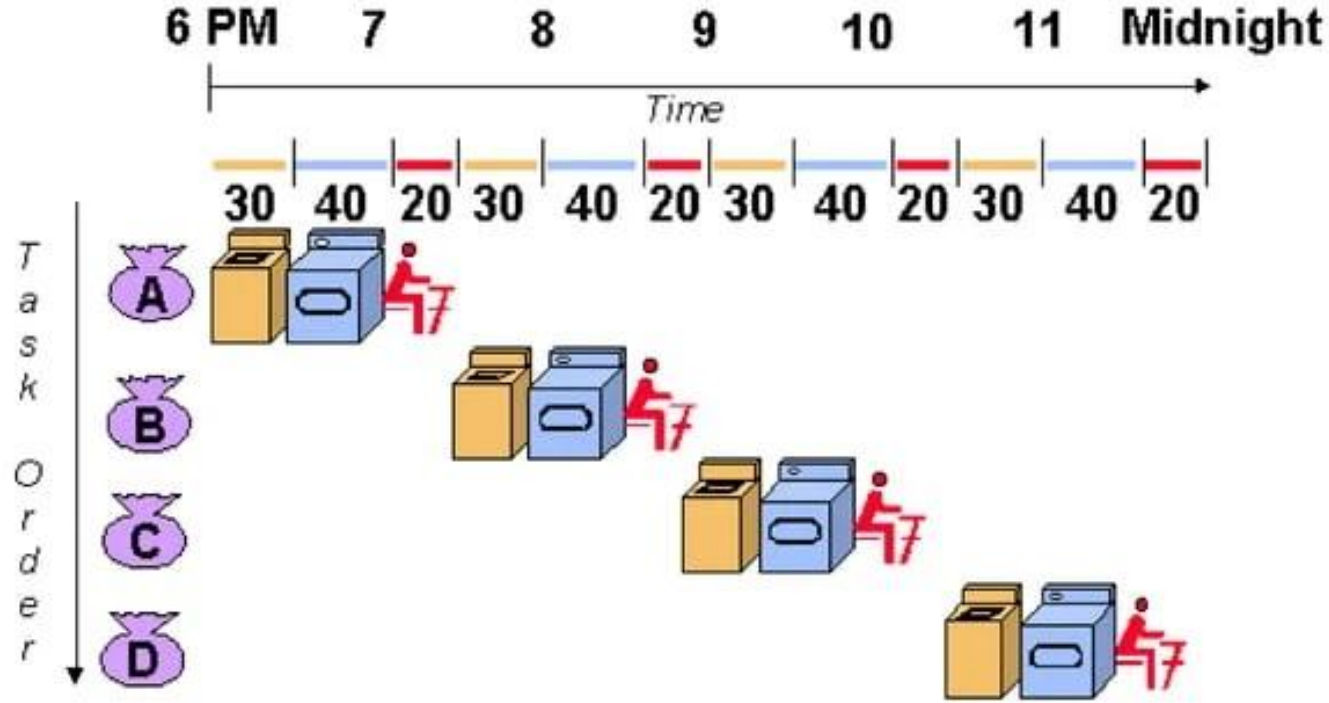
- A **Pipelining** is a series of stages, where some work is done at each stage in parallel.
- The stages are connected one to the next to form a pipe - instructions enter at one end, progress through the stages, and exit at the other end.

Pipelining Case: Laundry

- 4 loads of laundry that need to washed, dried, and folded.
  - 30 minutes to wash, 40 min. to dry, and 20 min. to fold.
  - We have 1 washer, 1 dryer, and 1 folding station.

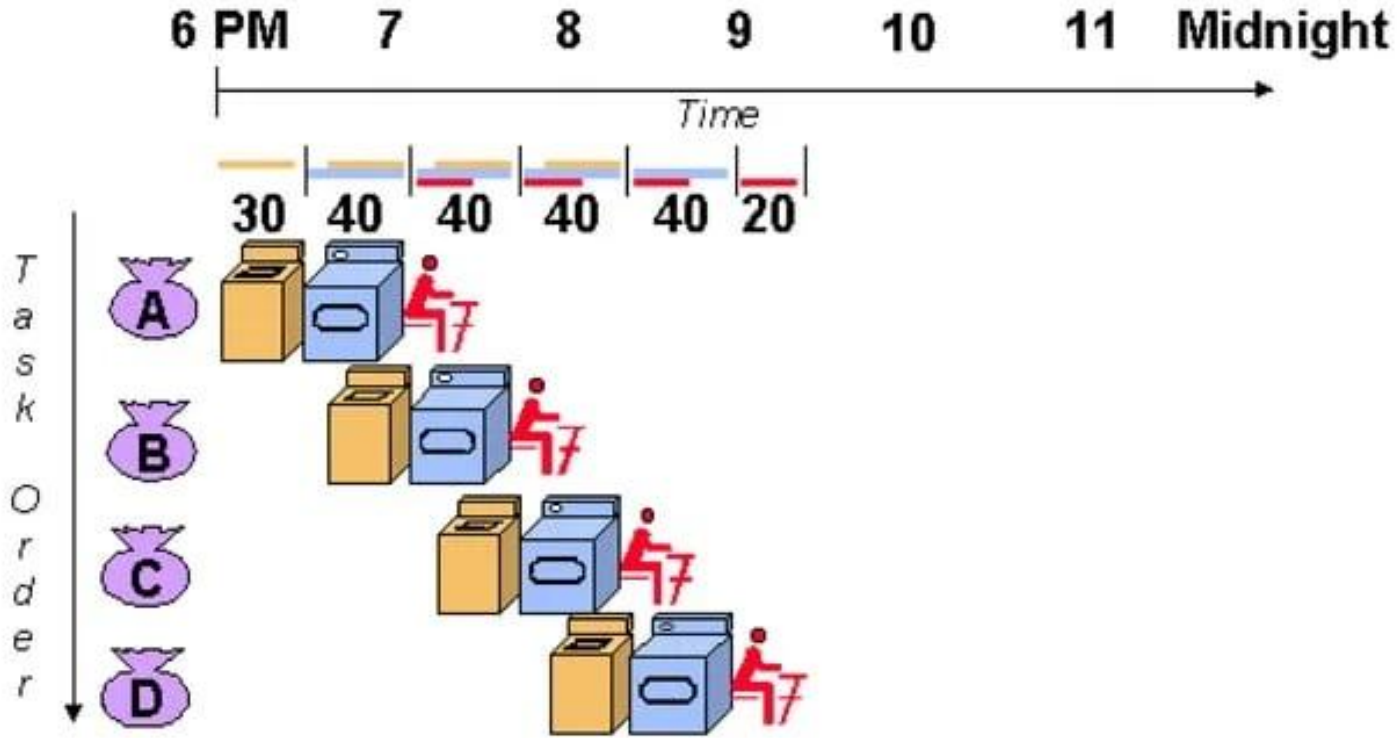
- What's the most efficient way to get the 4 loads of laundry done?

Non Pipelined Laundry



- Takes a total of 6 hours; nothing is done in parallel

Pipelined Laundry



- Using this method, the laundry would be done at 9:30.

Pipelining is an speed up technique where multiple instructions are overlapped in execution on a processor.

Pipelining: Processors

- Computers, like laundry, typically perform the exact same steps for every instruction:
  - Fetch an instruction from memory
  - Decode the instruction
  - Execute the instruction
  - Read memory to get input
  - Write the result back to memory

INSTRUCTION PIPELINE

An instruction pipeline reads instruction from the memory while previous instructions are being executed in other segments of the pipeline. Thus we can execute multiple instructions simultaneously. The pipeline will be more efficient if the instruction cycle is divided into segments of equal duration.

Instruction Pipeline

- **Instruction execution process lends itself naturally to pipelining**
  - **overlap the subtasks of instruction fetch, decode and execute**

- **Instruction pipeline has six operations,**
  - Fetch instruction (FI)
  - Decode instruction (DI)
  - Calculate operands (CO)
  - Fetch operands (FO)
  - Execute instructions (EI)
  - Write result (WR)

  Overlap these operations

Instructions Fetch

- The IF stage is responsible for obtaining the requested instruction from memory. The instruction and the program counter are stored in the register as temporary storage.

Decode Instruction

- The DI stage is responsible for decoding the instruction and sending out the various control lines to the other parts of the processor.

Calculate Operands

- The CO stage is where any calculations are performed. The main component in this stage is the ALU. The ALU is made up of arithmetic, logic and capabilities.

Fetch Operands and Execute Instruction

- The FO and EI stages are responsible for storing and loading values to and from memory. They also responsible for input and output from the processor respectively.

Write Operands

- The WO stage is responsible for writing the result of a calculation, memory access or input into the register file.

Advantages

- Pipelining makes efficient use of resources.
- Quicker time of execution of large number of instructions
- The parallelism is invisible to the programmer.

Can Pipelining Get Us Into Trouble?

- Yes: Pipeline Hazards
  - structural hazards: attempt to use the same resource by two different instructions at the same time
  - data hazards: attempt to use data before it is ready
    - An instruction's source operand(s) are produced by a prior instruction still in the pipeline
  - control hazards: attempt to make a decision about program control flow before the condition has been evaluated and the new PC target address calculated
    - branch instructions

- Can always resolve hazards by waiting
  - pipeline control must detect the hazard
  - and take action to resolve hazards

Data Hazards occur when an instruction depends on the result of previous instruction and that result of instruction has not yet been computed. whenever two different instructions use the same storage. the location must appear as if it is executed in sequential order.
There are four types of data dependencies: Read after Write (RAW), Write after Read (WAR), Write after Write (WAW), and Read after Read (RAR). These are explained as follows below.
Read after Write (RAW) :
It is also known as True dependency or Flow dependency. It occurs when the value produced by an instruction is required by a subsequent instruction. For example,
ADD R1, --, --;
SUB --, R1, --;
Stalls are required to handle these hazards.

Write after Read (WAR) :
It is also known as anti-dependency. These hazards occur when the output register of an instruction is used right after read by a previous instruction. For example,
ADD --, R1, --;
SUB R1, --, --;
Write after Write (WAW) :
It is also known as output dependency. These hazards occur when the output register of an instruction is used for write after written by previous instruction. For example,
ADD R1, --, --;
SUB R1, --, --;

Read after Read (RAR) :
It occurs when the instruction both read from the same register. For example,
ADD --, R1, --;
SUB --, R1, --;
Since reading a register value does not change the register value, these Read after Read (RAR) hazards don’t cause a problem for the processor.

Handling Data Hazards :

These are various methods we use to handle hazards: Forwarding, Code recording, and Stall insertion.

These are explained as follows below.

Forwarding :

It adds special circuitry to the pipeline. This method works because it takes less time for the required values to travel through a wire than it does for a pipeline segment to compute its result.

Code reordering :

We need a special type of software to reorder code. We call this type of software a hardware-dependent compiler.

Stall Insertion :

it inserts one or more installs (no-op instructions) into the pipeline, which delays the execution of the current instruction until the required operand is written to the register file, but this method decreases pipeline efficiency and throughput.