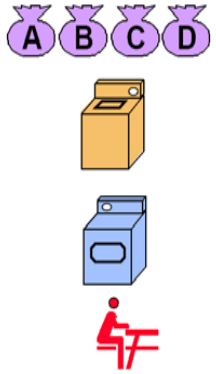
Pipelining
(Introduction)

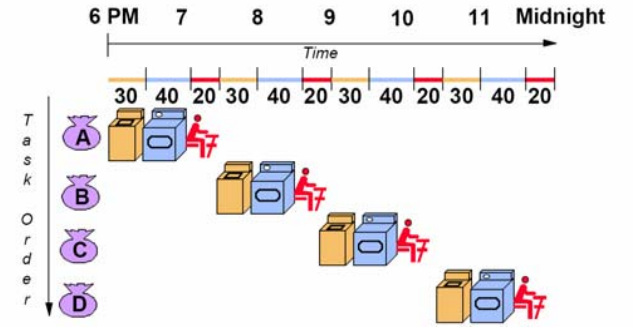
CIT 595
Spring 2008

Laundry Example

- Ann, Brian, Cathy and Dave each have one load of clothes to wash, dry and fold
- Washer takes 30 mins
- Dryer takes 40 mins
- "Folder" takes 20 mins

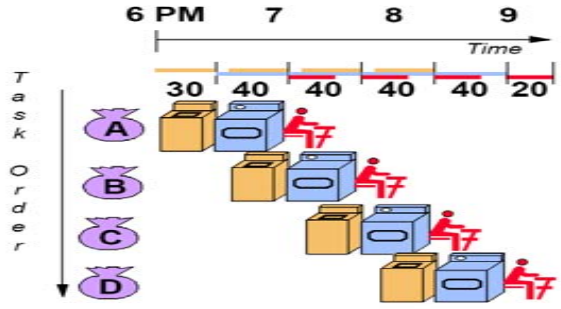


Sequential Laundry



- Entire workload takes 6 hours to complete

Pipelined Laundry



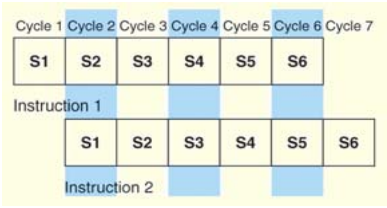
- Pipelined Laundry takes only 3.5 hours
- Speedup = 6/3.5 = 1.7
- Pipelining did not reduce completion time for one task but it helps the throughput of the entire workload in turn decreasing the completion time

Instruction Level Pipelining

- In instruction processing, each instruction goes through
  - F->D->EA->OP->EX->S cycle
- The instruction cycle is divided into stages
  - One stage could contain more than one phase of the instruction cycle or
  - one phase can be divided into two stages
- If an instruction is in a particular stage of the cycle, the rest of the stages are *idle*
  - We exploit this idleness to allow instructions to be executed in parallel
  - Such parallel execution is called ***instruction-level pipelining***

Instruction Level Pipelining: Big Picture

- Each stage of the Instruction Processing Cycle takes 1 clock cycle
  - 1 clock cycle = *x* time units per stage
- For each stage, one phase of instruction is carried out, and the stages are overlapped



S1. Fetch instruction
S2. Decode opcode
S3. Evaluate Address
S4. Fetch operands
S5. Execute
S6. Store result

Theoretical Speedup due to Pipelining

- The theoretical speedup offered by a pipeline can be determined as follows:
  - Let *k be total number of stages and* t_p be the time per stage
  - Let *n* be the total number of tasks
  - The first task (instruction) requires $k \times t_p$ time to complete in a *k*-stage pipeline.
  - The remaining (*n* - 1) tasks emerge from the pipeline one per cycle
    - So the total time to complete the remaining tasks is $(n - 1)t_p$
  - Thus, to complete *n* tasks using a *k*-stage pipeline requires:
    $(k \times t_p) + (n - 1)t_p = (k + n - 1)t_p$

Theoretical Speedup due to Pipelining

- If we take the time required to complete *n* tasks without a pipeline and divide it by the time it takes to complete *n* tasks using a pipeline, we find:

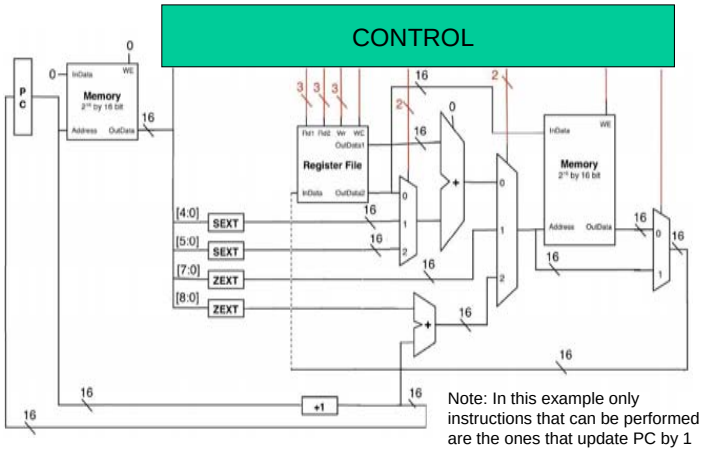
$$\text{Speedup } S = \frac{n t_n}{(k + n - 1)\, t_p}$$

$t_n = k \times t_p$

- If we take the limit as *n* approaches infinity, (*k* + *n* - 1) approaches *n*, which results in a theoretical speedup of:

$$\text{Speedup } S = \frac{k t_p}{t_p} = k$$

LC3 Hardwired Control (modified)



Note: In this example only instructions that can be performed are the ones that update PC by 1

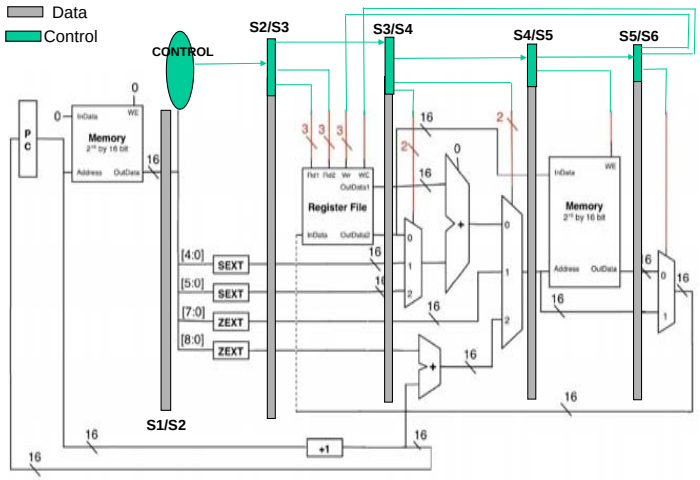
How Pipelining actually Implemented?

- Since we are overlapping stages
- All the control information plus data must be remembered per instruction and must be carried through each stage
- This is achieved by placing a n-bit register that can hold the control/data information in between each stage

LC3 Pipelined Implementation

- Evaluate Address and Execute are combined as they both use ALU
- Operand Fetch is separated into Register Fetch and Memory Access
- Store consists of only register writes
- Memory Write is part of Memory Access
- Thus we have a total of 6 stages

LC3 Pipelined Implementation

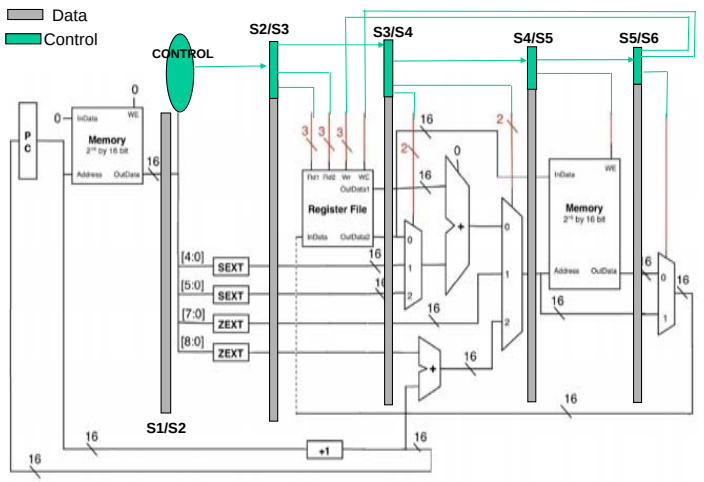


Impact on Clock Cycle due to Pipelining

- The *clock* is sequencing the stages (instructions move in lock step fashion)
- Make sure that all work done in one stage gets done on time before it moves to next stage
- Hence, the clock cycle time should be as long as time it takes through the longest pipe stage
  - This also includes the time for capturing data into registers in between stages

Clock cycle time = $\max(t_1,t_2,t_3,t_4,t_5,t_6)$

LC3 Pipelined Implementation



Impact on Clock cycle time due to Pipelining

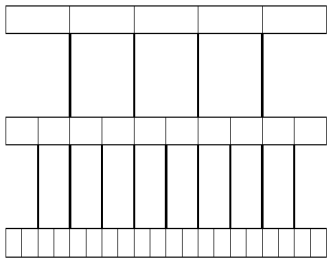
- Recall

$$\text{CPU Time} = \frac{\text{time}}{\text{program}} = \frac{\text{time}}{\text{cycle}} \times \frac{\text{cycles}}{\text{instruction}} \times \frac{\text{instructions}}{\text{program}}$$

$\frac{\text{time}}{\text{cycle}}$: **Clock Cycle time**; $\frac{\text{cycles}}{\text{instruction}}$: CPI; $\frac{\text{instructions}}{\text{program}}$: Instruction Count

- If we lower the time per cycle, this will lower the program execution time and hence improve performance
- This implies that we if we shorten the time per pipeline stages, we will lower clock cycle time
  - This can be achieved by adding more pipe stages of shorter duration

Impact on Clock cycle time due to Pipelining



5 stage

10 stage – clock cycle time reduced by one half

20 stage – clock cycle time reduced by one fourth

Cycles Per Instruction (CPI) with Pipelining

- In pipelining, one instruction is in each stage
- Since one instruction will be fetched (or finish) each cycle, the average CPI will equal 1 (obviously we are ignoring the very first instruction – cold start)

- However, CPI = 1 is barely achieved

Why CPI is not always 1?

- We assume that the pipeline can be kept filled at all times
- However, this is not always the case
- The situations that cause pipeline not to filled at all times arises due to what is known as *Pipeline Hazards*