An Overview of Pipelining

- Pipelining is an implementation technique in which multiple instructions are overlapped in execution.
- This section gives an overview of the pipelining terms and issues.

Pipelining: An implementation technique in which multiple instructions are overlapped in execution, much like an assembly line.

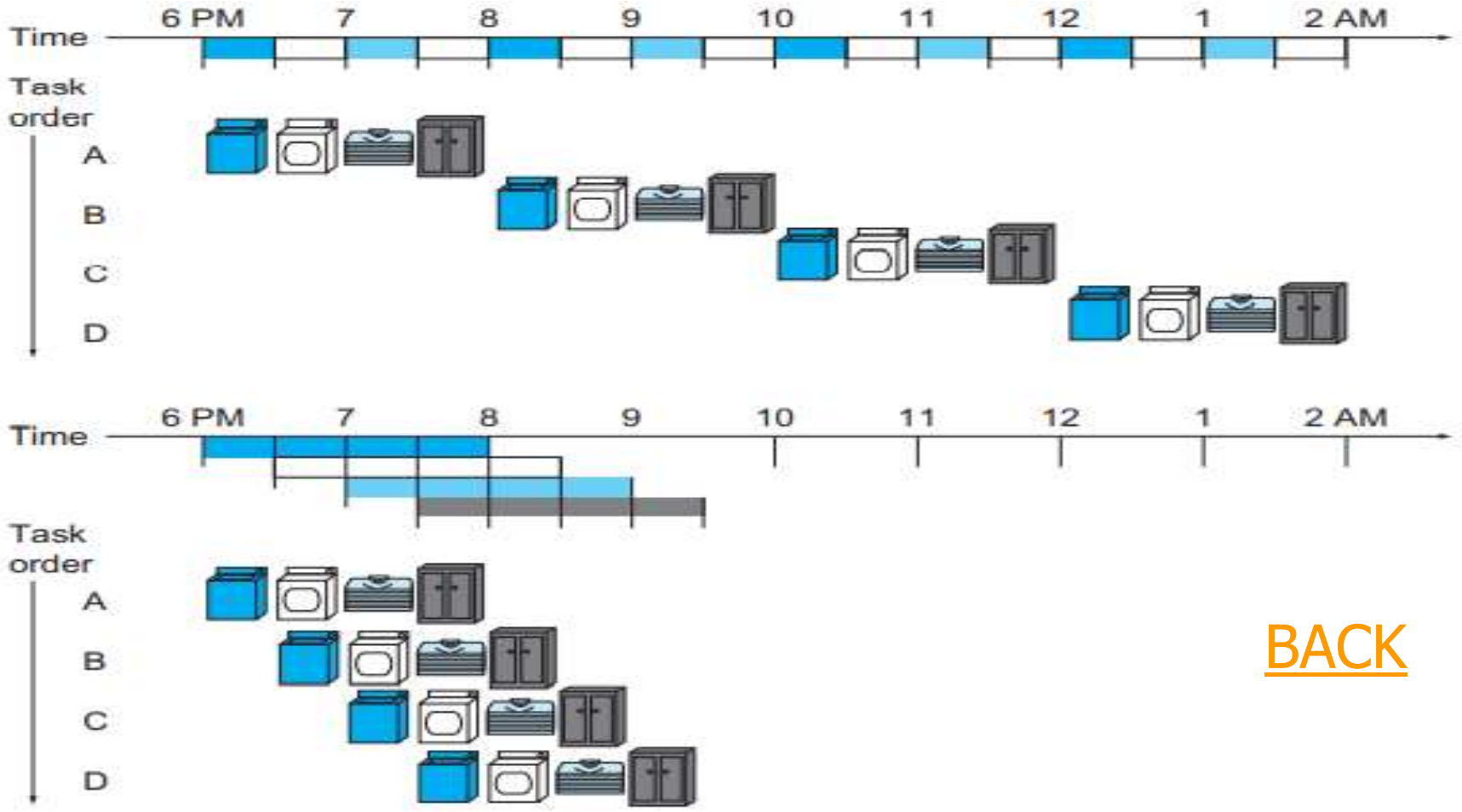
An Overview of Pipelining

- Anyone who has done a lot of laundry has intuitively used pipelining.
- The nonpipelined approach to laundry would be as follows:

  1. Place one dirty load of clothes in the washer.
  2. When the washer is finished, place the wet load in the dryer.
  3. When the dryer is finished, place the dry load on a table and fold.
  4. When folding is finished, ask your roommate to put the clothes away. When your roommate is done, start over with the next dirty load.

An Overview of Pipelining

- The pipelined approach takes much less time, as Figure 25 shows.



BACK

FIGURE 25 The laundry analogy for pipelining. Ann, Brian, Cathy, and Don each have dirty clothes to be washed, dried, folded, and put away. The washer, dryer, "folder," and "storer" each take 30 minutes for their task. Sequential laundry takes 8 hours for 4 loads of wash, while pipelined laundry takes just 3.5 hours. [J. Hennessy and D. Patterson, "Computer Organization and Design: The Hardware/Software Interface", 5th Edition.]

An Overview of Pipelining

Pipelined approach

- As soon as the washer is finished with the first load and placed in the dryer, you load the washer with the second dirty load.
- When the first load is dry, you place it on the table to start folding, move the wet load to the dryer, and put the next dirty load into the washer.
- Next you have your roommate put the first load away, you start folding the second load, the dryer has the third load, and you put the fourth load into the washer.
- At this point all steps—called stages in pipelining—are operating concurrently.
- As long as we have separate resources for each stage, we can pipeline the tasks.

An Overview of Pipelining

- The pipelining paradox is that the time from placing a single dirty sock in the washer until it is dried, folded, and put away is not shorter for pipelining; the reason pipelining is faster for many loads is that everything is working in parallel, so more loads are finished per hour.
- Pipelining improves throughput of our laundry system.
- Hence, pipelining would not decrease the time to complete one load of laundry, but when we have many loads of laundry to do, the improvement in throughput decreases the total time to complete the work.

An Overview of Pipelining

- If all the stages take about the same amount of time and there is enough work to do, then the speed-up due to pipelining is equal to the number of stages in the pipeline, in this case four: washing, drying, folding, and putting away.
- Therefore, pipelined laundry is potentially four times faster than nonpipelined: 20 loads would take about 5 times(20/4) as long as 1 load, while 20 loads of sequential laundry takes 20 times as long as 1 load.
- It's only 2.3 times (8/3.5) faster in Figure 25, because we only show 4 loads.

FIGURE

An Overview of Pipelining

- Notice that at the beginning and end of the workload in the pipelined version in Figure 25, the pipeline is not completely full; this start-up and winddown affects performance when the number of tasks is not large compared to the number of stages in the pipeline.
- If the number of loads is much larger than 4, then the stages will be full most of the time and the increase in throughput will be very close to 4.

An Overview of Pipelining

- The same principles apply to processors where we pipeline instruction-execution.

- MIPS instructions classically take five steps:
  1. Fetch instruction from memory.
  2. Read registers while decoding the instruction. The regular format of MIPS instructions allows reading and decoding to occur simultaneously.
  3. Execute the operation or calculate an address.
  4. Access an operand in data memory.
  5. Write the result into a register.

An Overview of Pipelining

EXAMPLE

Single-Cycle versus Pipelined Performance

- In this example we limit our attention to eight instructions: load word (lw), store word (sw), add (add), subtract (sub), AND (and), OR (or), set less than (slt), and branch on equal (beq).
- Compare the average time between instructions of a single-cycle implementation, in which all instructions take one clock cycle, to a pipelined implementation.
- The operation times for the major functional units in this example are 200 ps for memory access, 200 ps for ALU operation, and 100 ps for register file read or write.
- In the single-cycle model, every instruction takes exactly one clock cycle, so the clock cycle must be stretched to accommodate the slowest instruction.

An Overview of Pipelining

ANSWER

- Figure 26 shows the time required for each of the eight instructions.
- The single-cycle design must allow for the slowest instruction—in Figure 26 it is lw—so the time required for every instruction is 800 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 26 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.

[J. Hennessy and D. Patterson, "Computer Organization and Design: The Hardware/Software Interface", 5th Edition.]

An Overview of Pipelining

- We can turn the pipelining speed-up into a formula.
- If the stages are perfectly balanced, then the time between instructions on the pipelined processor — assuming ideal conditions — is equal to

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

- Under ideal conditions and with a large number of instructions, the speed-up from pipelining is approximately equal to the number of pipe stages; a five-stage pipeline is nearly five times faster.

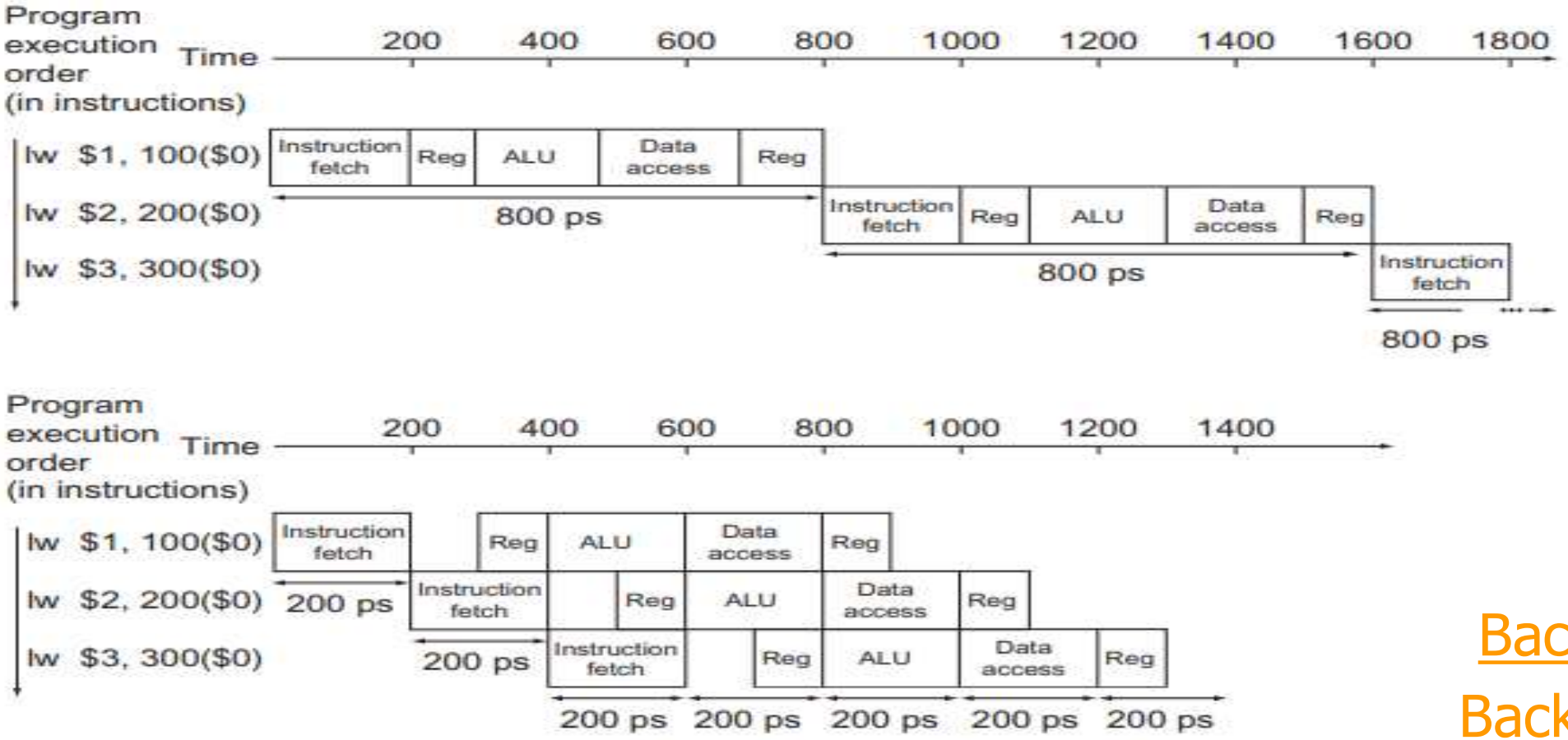
An Overview of Pipelining

- The formula suggests that a five-stage pipeline should offer nearly a fivefold improvement over the 800 ps nonpipelined time, or a 160 ps clock cycle.
- The example shows that the stages may be imperfectly balanced.
- Moreover, pipelining involves some overhead.
- Thus, the time per instruction in the pipelined processor will exceed the minimum possible, and speed-up will be less than the number of pipeline stages

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

An Overview of Pipelining

- Figure 27 compares nonpipelined and pipelined execution of three load word instructions.
- The time between the first and fourth instructions in the nonpipelined design is 3 × 800 ps or 2400 ps.



Back1

Back2

FIGURE 27 Single-cycle, nonpipelined execution in top versus pipelined execution in bottom.

[J. Hennessy and D. Patterson, "Computer Organization and Design: The Hardware/Software Interface", 5th Edition.]

An Overview of Pipelining

- Moreover, the improvement for our example is not reflected in the total execution time for the three instructions: it's 1400 ps versus 2400 ps(3x800 ps). (Figure 27)
- Of course, this is because the number of instructions is not large.

- What would happen if we increased the number of instructions?
- We could extend the previous figures to 1,000,003 instructions.
- We would add 1,000,000 instructions in the pipelined example; each instruction adds 200 ps to the total execution time.

An Overview of Pipelining

- The total execution time in the pipelined example would be 1,000,000 × 200 ps + 1400 ps, or 200,001,400 ps.
- In the nonpipelined example, we would add 1,000,000 instructions, each taking 800 ps, so total execution time would be 1,000,000 × 800 ps + 2400 ps, or 800,002,400 ps.
- Under these conditions, the ratio of total execution times for real programs on nonpipelined to pipelined processors is close to the ratio of times between instructions.

$$\frac{800,002,400\,ps}{200,001,400\,ps} \simeq \frac{800ps}{200ps} \simeq 4.00$$

An Overview of Pipelining

- Pipelining improves performance by increasing instruction throughput, as opposed to decreasing the execution time of an individual instruction, but instruction throughput is the important metric because real programs execute billions of instructions.

Designing Instruction Sets for Pipelining

- Even with this explanation of pipelining, we can get insight into the design of the MIPS instruction set, which was designed for pipelined execution.
- First, all MIPS instructions are the same length.
- This restriction makes it much easier to fetch instructions in the first pipeline stage and to decode them in the second stage.
- In an instruction set like the x86, where instructions vary from 1 byte to 15 bytes, pipelining is considerably more challenging.
- Recent implementations of the x86 architecture actually translate x86 instructions into simple operations that look like MIPS instructions and then pipeline the simple operations rather than the native x86 instructions!

Designing Instruction Sets for Pipelining

- Second, MIPS has only a few instruction formats, with the source register fields being located in the same place in each instruction.
- This symmetry means that the second stage can begin reading the register file at the same time that the hardware is determining what type of instruction was fetched.
- If MIPS instruction formats were not symmetric, we would need to split stage 2, resulting in six pipeline stages.
- We will shortly see the downside of longer pipelines.

Designing Instruction Sets for Pipelining

- Third, memory operands only appear in loads or stores in MIPS.
- This restriction means we can use the execute stage to calculate the memory address and then access memory in the following stage.
- If we could operate on the operands in memory, as in the x86, stages 3 and 4 would expand to an address stage, memory stage, and then execute stage.
- Fourth, operands must be aligned in memory.
- Hence, we need not worry about a single data transfer instruction requiring two data memory accesses; the requested data can be transferred between processor and memory in a single pipeline stage