4.5 An overview of pipelining

> "Never waste time."
> *American proverb.*

Pipelining is an implementation technique in which multiple instructions are overlapped in execution. Today, **pipelining** is nearly universal.

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

This section relies heavily on one analogy to give an overview of the pipelining terms and issues. If you are interested in just the big picture, you should concentrate on this section and then skip to COD Sections 4.10 (Parallelism via instructions) and 4.11 (Real stuff: The ARM Cortex-A53 and Intel Core i7 pipelines) to see an introduction to the advanced pipelining techniques used in recent processors such as the Intel Core i7 and ARM Cortex-A53. If you are curious about exploring the anatomy of a pipelined computer, this section is a good introduction to COD Sections 4.6 (Pipelined datapath and control), 4.7 (Date hazards: Forwarding versus stalling), 4.8 (Control hazards), and 4.9 (Exceptions).

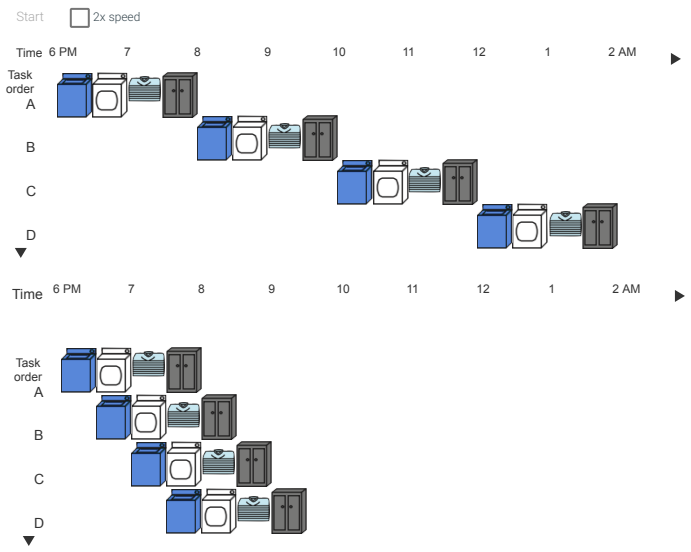
Anyone who has done a lot of laundry has intuitively used pipelining. The *non-pipelined* approach to laundry would be as follows:

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.

The *pipelined* approach takes much less time, as the animation below shows. 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.

PARTICIPATION ACTIVITY 4.5.1: The laundry analogy for pipelining (COD Figure 4.24).



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.

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 five times as long as one load, while 20 loads of sequential laundry takes 20 times as long as one load. It's only 2.3 times faster in the animation above, because we only show four loads. Notice that at the beginning and end of the workload in the pipelined version in the animation above, the pipeline is not completely full; this start-up and wind-down 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 four, then the stages will be full most of the time and the increase in throughput will be very close to four.

PARTICIPATION ACTIVITY 4.5.2: The laundry analogy for pipelining.

1) How many stages exist in the laundry analogy?

Check Show answer

2) If laundry is done sequentially, how many minutes do 60 loads take to wash?

minutes

Check Show answer

3) If laundry is done in a pipelined manner, execution is nearly 4 times faster than if done sequentially. Suppose doing 50 loads sequentially requires 6000 minutes. How long would those 50 loads take if done in a pipelined manner? Assume a 4 times speedup (ignore the fact that some stages are unused for the first few and last few loads).

minutes

Check Show answer

4) Each load of laundry takes 4 × 30 = 120 minutes to wash, dry, fold, and store (30 minutes each). How many minutes are required to complete one load of laundry when multiple loads of laundry are done in a pipelined manner?

minutes

Check Show answer

The same principles apply to processors where we pipeline instruction execution. LEGv8 instructions classically take five steps:

1. Fetch instruction from memory.
2. Read registers and decode the instruction.
3. Execute the operation or calculate an address.
4. Access an operand in data memory (if necessary).
5. Write the result into a register (if necessary).

Hence, the LEGv8 pipeline we explore in this chapter has five stages. The following example shows that pipelining speeds up instruction execution just as it speeds up the laundry.

Example 4.5.1: Single-cycle versus pipelined performance.

To make this discussion concrete, let's create a pipeline. In this example, and in the rest of this chapter, we limit our attention to seven instructions: load register (`LDUR`), store register (`STUR`), add (`ADD`), subtract (`SUB`), AND (`AND`), OR (`ORR`), and compare and branch on zero (`CBZ`).

Contrast the average time between instructions of a single-cycle implementation, in which all instructions take one clock cycle, to a pipelined implementation. Assume that the operation times for the major functional units in this example are 200 ps for memory access for instructions or data, 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.

Answer

The figure below shows the time required for each of the seven instructions. The single-cycle design must allow for the slowest instruction—in the figure below it is `LDUR`—so the time required for every instruction is 800 ps. Similarly to COD Figure 4.24 (The laundry analogy for pipelining), COD Figure 4.26 (Single-cycle, nonpipelined execution ...) compares nonpipelined and pipelined execution of three load register instructions. Thus, the time between the first and fourth instructions in the nonpipelined design is 3 x 800 ps or 2400 ps.

All the pipeline stages take a single clock cycle, so the clock cycle must be long enough to accommodate the slowest operation. Just as the single-cycle design must take the worst-case clock cycle of 800 ps, even though some instructions can be as fast as 500 ps, the pipelined execution clock cycle must have the worst-case clock cycle of 200 ps, even though some stages take only 100 ps. Pipelining still offers a fourfold performance improvement: the time between the first and fourth instructions is 3 x 200 ps or 600 ps.

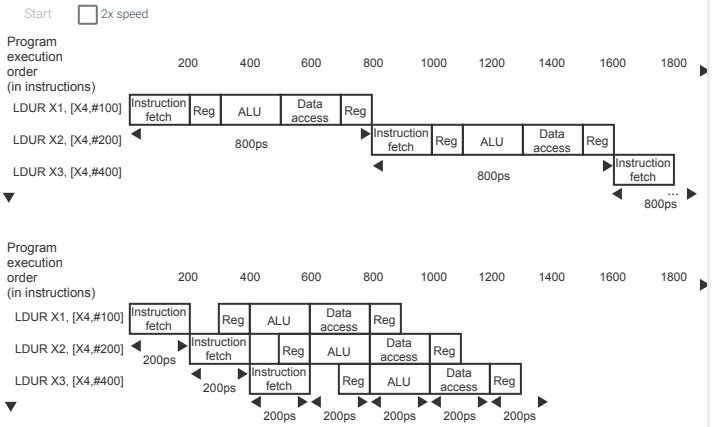
Figure 4.5.1: Total time for each instruction calculated from the time for each component (COD Figure 4.25).

This calculation assumes that the multiplexors, control unit, PC accesses, and sign extension unit have no delay.

Instruction class	Instruction fetch	Register read	ALU operation	Data access	Register write	Total time
Load register (LDUR)	200 ps	100 ps	200 ps	200 ps	100 ps	800 ps
Store register (STUR)	200 ps	100 ps	200 ps	200 ps		700 ps
R-format (ADD, SUB, AND, ORR)	200 ps	100 ps	200 ps		100 ps	600 ps
Branch (CBZ)	200 ps	100 ps	200 ps			500 ps

The animation below compares a non-pipelined vs. pipelined execution. Both use the same hardware components, whose time is listed in the figure above. In this case, we see a fourfold speed-up on average time between instructions, from 800 ps down to 200 ps. Compare this figure to COD Figure 4.24 (The laundry analogy for pipelining). For the laundry, we assumed all stages were equal. If the dryer were slowest, then the dryer stage would set the stage time. The pipeline stage times of a computer are also limited by the slowest resource, either the ALU operation or the memory access. We assume the write to the register file occurs in the first half of the clock cycle and the read from the register file occurs in the second half. We use this assumption throughout this chapter.

PARTICIPATION ACTIVITY 4.5.3: Single-cycle, nonpipelined execution in top versus pipelined execution in bottom (COD Figure 4.26).



We can turn the pipelining speed-up discussion above 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.

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, however, that the stages may be imperfectly balanced. Moreover, pipelining involves some overhead, the source of which will be clearer shortly. 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.

However, even our claim of fourfold improvement for our example is not reflected in the total execution time for the three instructions: it's 1400 ps versus 2400 ps. 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. The total execution time would be 1,000,000 x 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 x 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:

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

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

PARTICIPATION ACTIVITY 4.5.4: Pipelining the LEGv8.

Refer to the above animation on nonpipelined and pipelined LEGv8 instruction execution, and the prior table listing the total time for each instruction.

1) The nonpipelined datapath implementation has how many stages?
   - ○ 1
   - ○ 5

2) The pipelined datapath implementation has how many stages?
   - ○ 1
   - ○ 5

3) The above figure shows the five stages as: Instruction fetch, Reg, ALU, Data access, and Reg. Are the two Regs doing the same thing?

- Yes
- No

4) Does every instruction require all 5 stages?
   - Yes
   - No

5) Suppose Instr1 is fetched in stage 1. Instr1 then proceeds to stage 2, Reg read. In a pipelined implementation, can Instruction2 be fetched simultaneously with that Reg read?
   - Yes
   - No

6) On computer X, a nonpipelined instruction execution would require 12 ns, and thus 12 ns exists between instructions. A pipelined implementation uses 6 equal-length stages of 2 ns each, resulting in _____ ns between instructions.
   - 1
   - 2

7) On computer X, a nonpipelined instruction execution would require 12 ns. A pipelined implementation uses 6 equal-length stages of 2 ns each. Assuming one million instructions execute and ignoring empty stages at the start/end, what is the speedup of the pipelined vs. non-pipelined implementation?
   - 2
   - 6

Designing instruction sets for pipelining

Even with this simple explanation of pipelining, we can get insight into the design of the LEGv8 instruction set, which was designed for pipelined execution.

First, all LEGv8 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. Modern implementations of the x86 architecture actually translate x86 instructions into simple operations that look like LEGv8 instructions and then pipeline the simple operations rather than the native x86 instructions! (See COD Section 4.10 (Parallelism via instructions).)

Second, LEGv8 has just a few instruction formats, with the first source register and destination register fields being located in the same place in each instruction.

Third, memory operands only appear in loads or stores in LEGv8. 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. We will shortly see the downside of longer pipelines.

Pipeline hazards

There are situations in pipelining when the next instruction cannot execute in the following clock cycle. These events are called *hazards*, and there are three different types.

Structural hazard

The first hazard is called a *structural hazard*. It means that the hardware cannot support the combination of instructions that we want to execute in the same clock cycle. A structural hazard in the laundry room would occur if we used a washer-dryer combination instead of a separate washer and dryer, or if our roommate was busy doing something else and wouldn't put clothes away. Our carefully scheduled pipeline plans would then be foiled.

Structural hazard: When a planned instruction cannot execute in the proper clock cycle because the hardware does not support the combination of instructions that are set to execute.

As we said above, the LEGv8 instruction set was designed to be pipelined, making it fairly easy for designers to avoid structural hazards when designing a pipeline. Suppose, however, that we had a single memory instead of two memories. If the pipeline in the above animation had a fourth instruction, we would see that in the same clock cycle the first instruction is accessing data from memory while the fourth instruction is fetching an instruction from that same memory. Without two memories, our pipeline could have a structural hazard.

PARTICIPATION ACTIVITY 4.5.5: Structural hazards.

1) In processor X's pipeline, an add instruction in stage 3 should use the ALU. A branch instruction in stage 4 also should use the ALU. Both instructions cannot simultaneously use the ALU. Such a situation is a structural hazard.
   - True
   -

False

2) LEGv8 implementations tend to have numerous structural hazards.
   - True
   - False

Data hazards

Data hazards occur when the pipeline must be stalled because one step must wait for another to complete. Suppose you found a sock at the folding station for which no match existed. One possible strategy is to run down to your room and search through your clothes bureau to see if you can find the match. Obviously, while you are doing the search, loads that have completed drying are ready to fold and those that have finished washing are ready to dry.

Data hazard: Also called a ***pipeline data hazard***. When a planned instruction cannot execute in the proper clock cycle because data that is needed to execute the instruction are not yet available.

In a computer pipeline, data hazards arise from the dependence of one instruction on an earlier one that is still in the pipeline (a relationship that does not really exist when doing laundry). For example, suppose we have an add instruction followed immediately by a subtract instruction that uses the sum (`X19`):

```
ADD   X19, X0, X1
SUB   X2, X19, X3
```

Without intervention, a data hazard could severely stall the pipeline. The add instruction doesn't write its result until the fifth stage, meaning that we would have to waste three clock cycles in the pipeline.

Although we could try to rely on compilers to remove all such hazards, the results would not be satisfactory. These dependences happen just too often and the delay is far too long to expect the compiler to rescue us from this dilemma.

The primary solution is based on the observation that we don't need to wait for the instruction to complete before trying to resolve the data hazard. For the code sequence above, as soon as the ALU creates the sum for the add, we can supply it as an input for the subtract. Adding extra hardware to retrieve the missing item early from the internal resources is called *forwarding* or *bypassing*.

Forwarding: Also called ***bypassing***. A method of resolving a data hazard by retrieving the missing data element from internal buffers rather than waiting for it to arrive from programmer-visible registers or memory.

Example 4.5.2: Forwarding with two instructions.

For the two instructions above, show what pipeline stages would be connected by forwarding. Use the drawing in the figure below to represent the datapath during the five stages of the pipeline. Align a copy of the datapath for each instruction, similar to the laundry pipeline in COD Figure 4.24 (The laundry analogy for pipelining).

Answer

COD Figure 4.28 (Graphical representation of forwarding) shows the connection to forward the value in `X1` after the execution stage of the `ADD` instruction as input to the execution stage of the `SUB` instruction.

Figure 4.5.2: Graphical representation of the instruction pipeline (COD Figure 4.27).

Here we use symbols representing the physical resources with the abbreviations for pipeline stages used throughout the chapter. The symbols for the five stages: *IF* for the instruction fetch stage, with the box representing instruction memory; *ID* for the instruction decode/ register file read stage, with the drawing showing the register file being read; *EX* for the execution stage, with the drawing representing the ALU; *MEM* for the memory access stage, with the box representing data memory; and *WB* for the write-back stage, with the drawing showing the register file being written. The shading indicates the element is used by the instruction. Hence, MEM has a white background because `ADD` does not access the data memory. Shading on the right half of the register file or memory means the element is read in that stage, and shading of the left half means it is written in that stage. Hence the right half of ID is shaded in the second stage because the register file is read, and the left half of WB is shaded in the fifth stage because the register file is written.

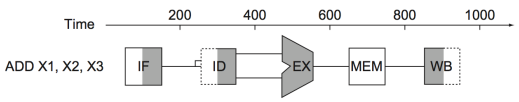
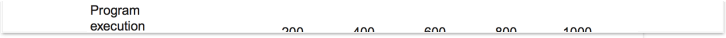


Figure 4.5.3: Graphical representation of forwarding (COD Figure 4.28).

The connection shows the forwarding path from the output of the EX stage of `ADD` to the input of the EX stage for `SUB`, replacing the value from register `X1` read in the second stage of `SUB`.



In this graphical representation of events, forwarding paths are valid only if the destination stage is later in time than the source stage. For example, there cannot be a valid forwarding path from the output of the memory access stage in the first instruction to the input of the execution stage of the following, since that would mean going backward in time.

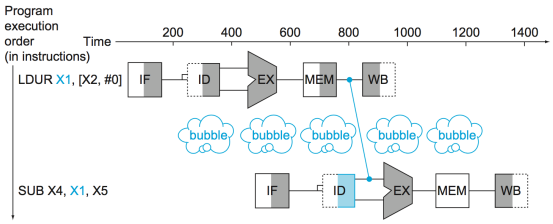
Forwarding works very well and is described in detail in COD Section 4.7 (Data hazards: Forwarding versus stalling). It cannot prevent all pipeline stalls, however. For example, suppose the first instruction was a load of `X1` instead of an add. As we can imagine from looking at the figure above, the desired data would be available only after the fourth stage of the first instruction in the dependence, which is too late for the input of the third stage of `SUB`. Hence, even with forwarding, we would have to stall one stage for a *load-use data hazard*, as COD Figure 4.29 (We need a stall even with forwarding …) shows. This figure shows an important pipeline concept, officially called a *pipeline stall*, but often given the nickname *bubble*. We shall see stalls elsewhere in the pipeline. COD Section 4.7 (Data hazards: Forwarding versus stalling) shows how we can handle hard cases like these, using either hardware detection and stalls or software that reorders code to try to avoid load-use pipeline stalls, as this example illustrates.

Load-use data hazard: A specific form of data hazard in which the data being loaded by a load instruction has not yet become available when it is needed by another instruction.

Pipeline stall: Also called ***bubble***. A stall initiated in order to resolve a hazard.

Figure 4.5.4: We need a stall even with forwarding when an R-format instruction following a load tries to use the data (COD Figure 4.29).

Without the stall, the path from memory access stage output to execution stage input would be going backward in time, which is impossible. This figure is actually a simplification, since we cannot know until after the subtract instruction is fetched and decoded whether or not a stall will be necessary. COD Section 4.7 (Data hazards: Forwarding versus stalling) shows the details of what really happens in the case of a hazard.



Example 4.5.3: Reordering code to avoid pipeline stalls.

Consider the following code segment in C:

```
a = b + e;
c = b + f;
```

Here is the generated LEGv8 code for this segment, assuming all variables are in memory and are addressable as offsets from `X0`:

```
LDUR   X1, [X0,#0]   // Load b
LDUR   X2, [X0,#8]   // Load e
ADD    X3, X1, X2    // b + e
STUR   X3, [X0,#24]  // Store a
LDUR   X4, [X0,#16]  // Load f
ADD    X5, X1, X4    // b + f
STUR   X5, [X0,#32]  // Store c
```

Find the hazards in the preceding code segment and reorder the instructions to avoid any pipeline stalls.

Answer

Both `ADD` instructions have a hazard because of their respective dependence on the previous `LDUR` instruction. Notice that forwarding eliminates several other potential hazards, including the dependence of the first `ADD` on the first `LDUR` and any hazards for store instructions. Moving up the third `LDUR` instruction to become the third instruction eliminates both hazards:

```
LDUR   X1, [X0,#0]
LDUR   X2, [X0,#8]
LDUR   X4, [X0,#16] // Instruction moved up
ADD    X3, X1, X2
STUR   X3, [X0,#24]
ADD    X5, X1, X4
STUR   X5, [X0,#32]
```

On a pipelined processor with forwarding, the reordered sequence will complete in two fewer cycles than the original version.

Forwarding yields another insight into the LEGv8 architecture. Each LEGv8 instruction writes at most one result and does this in the last stage of the pipeline. Forwarding is harder if there are multiple results to forward per instruction or if there is a need to write a result early on in instruction execution.

Elaboration

The name "forwarding" comes from the idea that the result is passed forward from an earlier instruction to a later instruction. "Bypassing" comes from passing the result around the register file to the desired unit.

PARTICIPATION ACTIVITY 4.5.6: Data hazards.

1) Does the following cause a data hazard for the 5-stage LEGv8 pipeline?

i1: `ADD X0, X1, X2`
i2: `ADD X3, X0, X4`

- ○ Yes
- ○ No

2) Does the following cause a data hazard for the 5-stage LEGv8 pipeline?

i1: `ADD X0, X1, X2`
i2: `ADD X3, X1, X4`

- ○ Yes
- ○ No

3) Does the following cause a data hazard for the 5-stage LEGv8 pipeline?

i1: `ADD X3, X3, X4`

- ○ Yes
- ○ No

4) The following causes a data hazard for the 5-stage LEGv8 pipeline

i1: `ADD  X0, X10, X11`
i2: `SUB  X12, X0, X13`

"Forwarding" can resolve the hazard by providing the ALU's output (for i1's stage 3) directly to the ALU's input (for i2's stage 3).

- ○ Yes
- ○ No

5) If forwarding cannot resolve a data hazard, the pipeline can be _____.

- ○ stalled
- ○ bypassed

6) Can these instructions be reordered to avoid a pipeline stall?

i1: `ADD X0, X1, X2`
i2: `ADD X3, X4, X5`
i3: `LDUR X11, [X10,#0]`
i4: `ADD X13, X11, X14`

- ○ Yes
- ○ No

7) Forwarding resolves some data hazards. Reordering resolves some others. If neither can resolve a data hazard, a stall may become necessary.

- ○ True
- ○ False

Control Hazards

The third type of hazard is called a *control hazard*, arising from the need to make a decision based on the results of one instruction while others are executing.

> ***Control hazard***: Also called ***branch hazard***. When the proper instruction cannot execute in the proper pipeline clock cycle because the instruction that was fetched is not the one that is needed; that is, the flow of instruction addresses is not what the pipeline expected.

Suppose our laundry crew was given the happy task of cleaning the uniforms of a football team. Given how filthy the laundry is, we need to determine whether the detergent and water temperature setting we select is strong enough to get the uniforms clean but not so strong that the uniforms wear out sooner. In our laundry pipeline, we have to wait until after the second stage to examine the dry uniform to see if we need to change the washer setup or not. What to do?

Here is the first of two solutions to control hazards in the laundry room and its computer equivalent.

Stall: Just operate sequentially until the first batch is dry and then repeat until you have the right formula.

This conservative option certainly works, but it is slow.

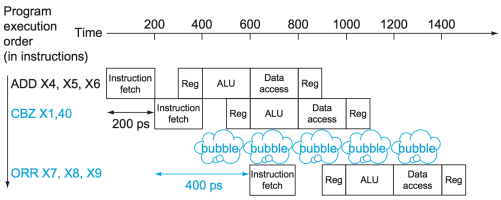
The equivalent decision task in a computer is the conditional branch instruction. Notice that we must begin fetching the instruction following the branch on the following clock cycle. Nevertheless, the pipeline cannot possibly know what the next instruction should be,

since it *only just received* the branch instruction from memory! Just as with laundry, one possible solution is to stall immediately after we fetch a branch, waiting until the pipeline determines the outcome of the branch and knows what instruction address to fetch from.

Let's assume that we put in enough extra hardware so that we can test a register, calculate the branch address, and update the PC during the second stage of the pipeline (see COD Section 4.8 (Control hazards) for details). Even with this added hardware, the pipeline involving conditional branches would look like the figure below. The instruction to be executed if the branch fails is stalled one extra 200 ps clock cycle before starting.

Figure 4.5.5: Pipeline showing stalling on every conditional branch as solution to control hazards (COD Figure 4.30).

This example assumes the conditional branch is taken, and the instruction at the destination of the branch is the `ORR` instruction. There is a one-stage pipeline stall, or bubble, after the branch. In reality, the process of creating a stall is slightly more complicated, as we will see in COD Section 4.8 (Control hazards). The effect on performance, however, is the same as would occur if a bubble were inserted.



Example 4.5.4: Performance of "stall on branch".

Estimate the impact on the *clock cycles per instruction* (CPI) of stalling on branches. Assume all other instructions have a CPI of 1.

Answer

COD Figure 3.28 (The frequency of the LEGv8 instructions ...) in COD Chapter 3 (Arithmetic for Computers) shows that conditional branches are 17% of the instructions executed in SPECint2006. Since the other instructions run have a CPI of 1, and conditional branches took one extra clock cycle for the stall, then we would see a CPI of 1.17 and hence a slowdown of 1.17 versus the ideal case.

If we cannot resolve the branch in the second stage, as is often the case for longer pipelines, then we'd see an even larger slowdown if we stall on conditional branches. The cost of this option is too high for most computers to use and motivates a second solution to the control hazard using one of our great ideas from COD Chapter 1 (Computer Abstractions and Technology):

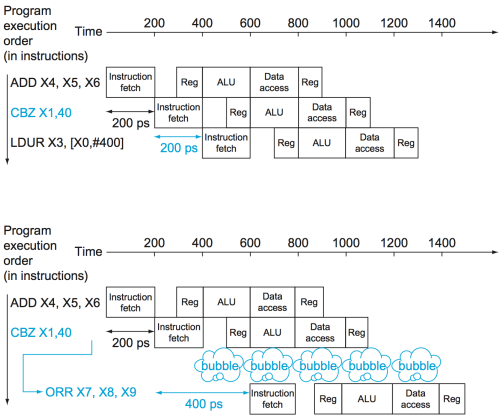
Predict: If you're sure you have the right formula to wash uniforms, then just *predict* that it will work and wash the second load while waiting for the first load to dry.

This option does not slow down the pipeline when you are correct. When you are wrong, however, you need to redo the load that was washed while guessing the decision.

Computers do indeed use **prediction** to handle conditional branches. One simple approach is to predict always that conditional branches will be untaken. When you're right, the pipeline proceeds at full speed. Only when conditional branches are taken does the pipeline stall. The figure below shows such an example.

Figure 4.5.6: Predicting that branches are not taken as a solution to control hazard (COD Figure 4.31).

The top drawing shows the pipeline when the branch is not taken. The bottom drawing shows the pipeline when the branch is taken. As we noted in in the above figure, the insertion of a bubble in this fashion simplifies what actually happens, at least during the first clock cycle immediately following the branch. COD Section 4.8 (Control hazards) will reveal the details.



A more sophisticated version of *branch prediction* would have some conditional branches predicted as taken and some as untaken. In our analogy, the dark or home uniforms might take one formula while the light or road uniforms might take another. In the case of programming, at the bottom of loops are conditional branches that branch back to the top of the loop. Since they are likely to be taken and they branch backward, we could always predict taken for conditional branches that branch to an earlier address.

Branch prediction: A method of resolving a branch hazard that assumes a given outcome for the conditional branch and proceeds from that assumption rather than waiting to ascertain the actual outcome.

Such rigid approaches to branch prediction rely on stereotypical behavior and don't account for the individuality of a specific branch instruction. *Dynamic* hardware predictors, in stark contrast, make their guesses depending on the behavior of each conditional branch and may change predictions for a conditional branch over the life of a program. Following our analogy, in dynamic prediction a person would look at how dirty the uniform was and guess at the formula, adjusting the next **prediction** depending on the success of recent guesses.

One popular approach to dynamic prediction of conditional branches is keeping a history for each conditional branch as taken or untaken, and then using the recent past behavior to predict the future. As we will see later, the amount and type of history kept have become extensive, with the result being that dynamic branch predictors can correctly predict conditional branches with more than 90% accuracy (see COD Section 4.8 (Control hazards)). When the guess is wrong, the pipeline control must ensure that the instructions following the wrongly guessed conditional branch have no effect and must restart the pipeline from the proper branch address. In our laundry analogy, we must stop taking new loads so that we can restart the load that we incorrectly predicted.

As in the case of all other solutions to control hazards, longer pipelines exacerbate the problem, in this case by raising the cost of misprediction. Solutions to control hazards are described in more detail in COD Section 4.8 (Control hazards).

Elaboration

There is a third approach to the control hazard, called delayed decision. In our analogy, whenever you are going to make such a decision about laundry, just place a load of non-football clothes in the washer while waiting for football uniforms to dry. As long as you have enough dirty clothes that are not affected by the test, this solution works fine.

Called the delayed branch in computers, this is the solution actually used by the LEGv8 architecture. The delayed branch always executes the next sequential instruction, with the branch taking place after that one instruction delay. It is hidden from the LEGv8 assembly language programmer because the assembler can automatically arrange the instructions to get the branch behavior desired by the programmer. LEGv8 software will place an instruction immediately after the delayed branch instruction that is not affected by the branch, and a taken branch changes the address of the instruction that follows this safe instruction. In our example, the `ADD` *instruction before the branch in COD Figure 4.30 (Pipeline showing stalling on every conditional branch as solution to control hazards) does not affect the branch and can be moved after the branch to hide the branch delay fully. Since delayed branches are useful when the branches are short, it is rare to see a processor with a delayed branch of more than one cycle. For longer branch delays, hardware-based branch prediction is usually used.*

PARTICIPATION ACTIVITY 4.5.7: Control (branch) hazards.

1) The following code has a control hazard:

```
        CBZ X1, L1
L1: ADD X0, X1, X2
L2: SUB X3, X4, X5
```

   - ○ True
   - ○ False

2) A control hazard can be resolved via a stall.
   - ○ True
   - ○ False

3) To reduce stalls due to branches, branch prediction involves executing a next instruction even if the processor is not sure that instruction should be next.
   - ○ True
   - ○ False

4) A smart branch predictor assumes the branch is not taken.
   - ○ True
   - ○ False

Pipeline overview summary

Pipelining is a technique that exploits **parallelism** between the instructions in a sequential instruction stream. It has the substantial advantage that, unlike programming a multiprocessor (see COD Chapter 6 (Parallel Processors from Client to Cloud)), it is fundamentally invisible to the programmer.

In the next few sections of this chapter, we cover the concept of pipelining using the LEGv8 instruction subset from the single-cycle implementation in COD Section 4.4 (A simple implementation scheme) and show a simplified version of its pipeline. We then look at the problems that **pipelining** introduces and the performance attainable under typical situations.

If you wish to focus more on the software and the performance implications of pipelining, you now have sufficient background to skip to COD Section 4.10 (Parallelism via instructions). COD Section 4.10 (Parallelism via instructions) introduces advanced pipelining concepts, such as superscalar and dynamic scheduling, and COD Section 4.11 (Real stuff: The ARM Cortex-A53 and Intel Core i7 pipelines) examines the pipelines of recent microprocessors.

Alternatively, if you are interested in understanding how pipelining is implemented and the challenges of dealing with hazards, you can proceed to examine the design of a pipelined datapath and the basic control, explained in COD Section 4.6 (Pipelined datapath and control). You can then use this understanding to explore the implementation of forwarding and stalls in COD Section 4.7 (Data

hazards: Forwarding versus stalling). You can next read COD Section 4.8 (Control hazards) to learn more about solutions to branch hazards, and finally see how exceptions are handled in COD Section 4.9 (Exceptions).

PARTICIPATION ACTIVITY 4.5.8: Check yourself: Stalls and forwarding.

For each code sequence below, state whether the code sequence must stall, can avoid stalls using only forwarding, or can execute without stalling or forwarding.

1)
```
i1: LDUR  X0, [X0, #0]
i2: ADD   X1, X0, X0
```
- must stall
- can avoid stalls using only forwarding
- execute without stalling or forwarding

2)
```
i1: ADD   X1, X0, X0
i2: ADDI  X2, X0, #5
i3: ADDI  X4, X1, #5
```
- must stall
- can avoid stalls using only forwarding
- execute without stalling or forwarding

3)
```
i1: ADDI  X1, X0, #1
i2: ADDI  X2, X0, #2
i3: ADDI  X3, X0, #3
i4: ADDI  X4, X0, #4
i5: ADDI  X5, X0, #5
```
- must stall
- can avoid stalls using only forwarding
- execute without stalling or forwarding

Understanding program performance

Outside the memory system, the effective operation of the pipeline is usually the most important factor in determining the CPI of the processor and hence its performance. As we will see in COD Section 4.10 (Parallelism via instructions), understanding the performance of a modern multiple-issue pipelined processor is complex and requires understanding more than just the issues that arise in a simple pipelined processor. Nonetheless, structural, data, and control hazards remain important in both simple pipelines and more sophisticated ones.

For modern pipelines, structural hazards usually revolve around the floating-point unit, which may not be fully pipelined, while control hazards are usually more of a problem in integer programs, which tend to have higher conditional branch frequencies as well as less predictable branches. Data hazards can be performance bottlenecks in both integer and floating-point programs. Often it is easier to deal with data hazards in floating-point programs because the lower conditional branch frequency and more regular memory access patterns allow the compiler to try to schedule instructions to avoid hazards. It is more difficult to perform such optimizations in integer programs that have less regular memory access, involving more use of pointers. As we will see in COD Section 4.10 (Parallelism via instructions), there are more ambitious compiler and hardware techniques for reducing data dependences through scheduling.

The Big Picture

Pipelining increases the number of simultaneously executing instructions and the rate at which instructions are started and completed. Pipelining does not reduce the time it takes to complete an individual instruction, also called the **latency**. For example, the five-stage pipeline still takes 5 clock cycles for the instruction to complete. In the terms used in COD Chapter 1 (Computer Abstractions and Technology), pipelining improves instruction throughput rather than individual instruction execution time or latency.

Instruction sets can either make life harder or simpler for pipeline designers, who must already cope with structural, control, and data hazards. Branch **prediction** and forwarding help make a computer fast while still getting the right answers.

Latency (***pipeline***): The number of stages in a pipeline or the number of stages between two instructions during execution.

Provide feedback on this section