ECSE324 : Computer Organization

Processor Pipeline
Chapter 6

Christophe Dubach

Fall 2023

Revision history:

Warren Gross – 2017

Christophe Dubach – W2020, F2020, F2021, F2022, F2023

Brett H. Meyer – W2021, W2022, W2023

Some material from Hamacher, Vranesic, Zaky, and Manjikian, *Computer Organization and Embedded Systems, 6 th ed*, 2012, McGraw Hill, and “Introduction to the ARM Processor using Altera Toolchain.”

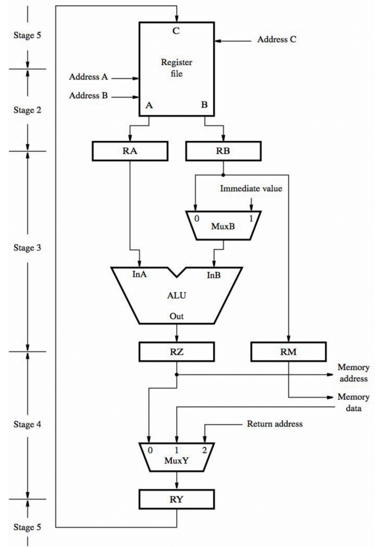
Timestamp: 2023/11/20 09:33:00

Disclaimer

It is possible (and even likely) that I will (sometimes) make mistakes and give incorrect information during the live lectures. If you have any doubts, please check the textbook, or ask for clarification online.

Reacap about the Datapath: Stages 2–5

- Inter-stage registers RA, RA, RZ, RM, and RY are used to carry data from one stage to the next
- Register file: used in stages 2 and 5; first, to read operands
- ALU: used in stage 3
- Memory: used in stage 4
- Write-back: the final stage is used to write the result to the register file

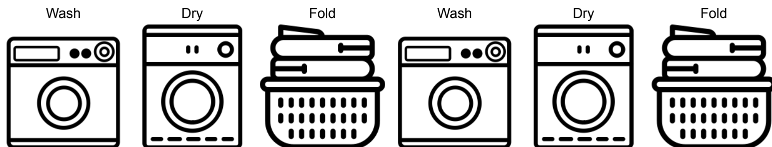


Pipelining

Textbook§6.1-6.3

Example

Consider doing laundry. If each operation requires one hour, the latency per load is three hours.

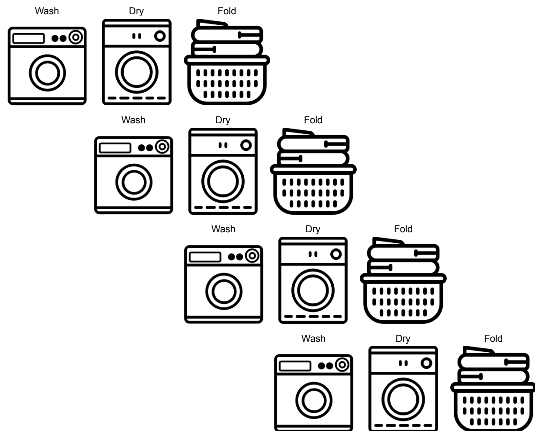


Two loads? Six hours total.

This is inefficient when there's a lot of laundry: when the dryer is working, the washer is idle!

Example

What happens if we make use of washer and dryer simultaneously on different loads?



Six hours, with pipelining? *Four* loads, instead of two.

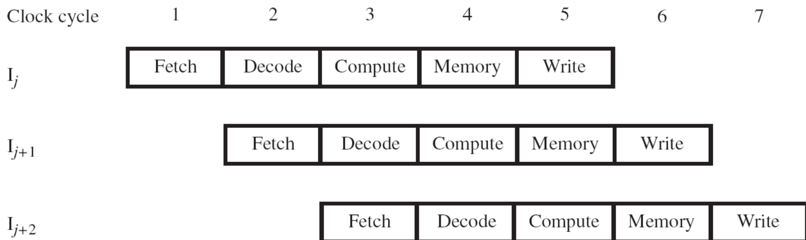
What is pipelining?

Pipelining is applying the “assembly line” concept to the execution of instructions:

- Instruction execution is divided into distinct steps (like we’ve already done)
- Multiple instructions are executed simultaneously by overlapping the steps of different instructions:
  - Only one instruction is started at a time
  - Each hardware stage is working on a different instruction
  - This keeps all stages busy, dramatically improving performance

Ideal Pipelining

In the ideal case, a new instruction is started each clock cycle, and each instruction only takes a single cycle in each step.



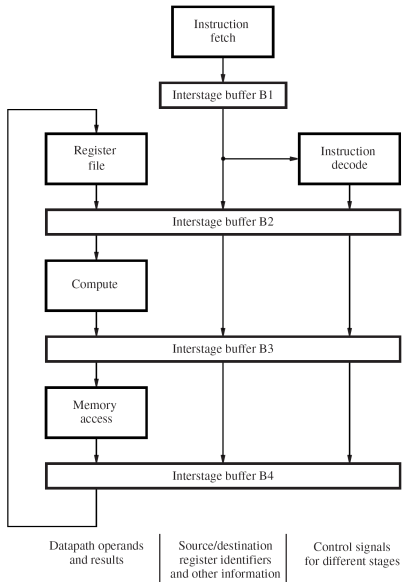
What are some reasons why this ideal may not be always achievable?

Pipeline Organization

- Use `PC` to fetch a new instruction every* cycle
- Instruction-specific information moves with instructions through the different stages
- Interstage buffers (pipeline registers) hold this information, incorporating `RA`, `RB`, `RM`, `RY`, `RZ`, `IR`, and `PC-Temp` registers
- The buffers also hold control signals: *e.g.*, mux inputs are determined during decode, but applied when appropriate

* Except when something prevents an instruction from advancing!

Pipeline Organization



Pipeline Stall

Textbook§6.4-6.7

What can stall the pipeline?

Instructions advance, one stage per cycle, unless something occurs to stall an instruction. Circumstances in which one instruction causes a delay in another instruction are called hazards, and they come in three flavors.

- Structural hazards: caused by contention for a shared resource (*e.g.*, memory)
- Data hazards: occur when one instruction must wait for the result of another
- Control hazards: caused by branch instructions delaying instruction fetch

Instructions may also be delayed when our assumption that each stage takes a single cycle is violated (*e.g.*, when a memory access results in a cache miss).

Data Dependencies

Consider the following assembly.

```
ADD   R2, R3, R7  // R2 <-- R3 + R7
SUB   R9, R2, R8  // R9 <-- R2 - R8
```

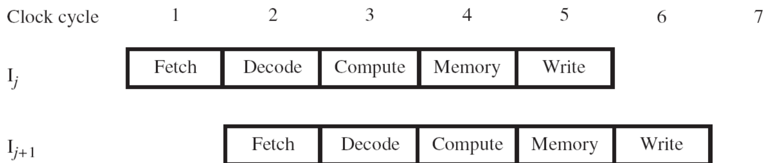
R2 is the (a) destination of the add instruction, and (b) source for the subtract instruction.

- There is a data dependency between ADD and SUB: SUB cannot be executed until we have the result of the ADD.
- With no pipelining, there's no problem: the result is in R2 because ADD completes before SUB begins.
- With pipelining, SUB starts before ADD finishes.

Data Hazards

```
ADD   R2, R3, R7  // R2 <-- R3 + R7
SUB   R9, R2, R8  // R9 <-- R2 - R8
```

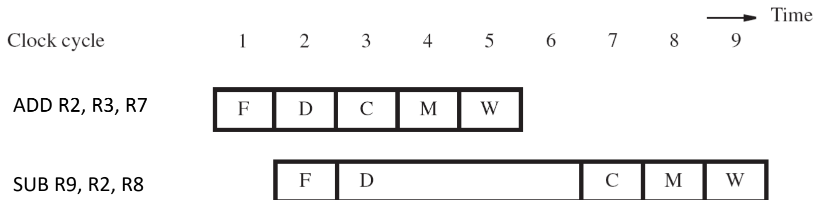
Suppose ADD is instruction I_j and SUB is instruction I_{j+1}:



- I_{j+1} reads its operands in cycle 3
- But the result of I_j is written in cycle 5 (to be read in cycle 6)
- I_j and I_{j+1} cannot execute simultaneously because of the data dependency
- This is a data hazard

To resolve this, we delay SUB until its operands are available.

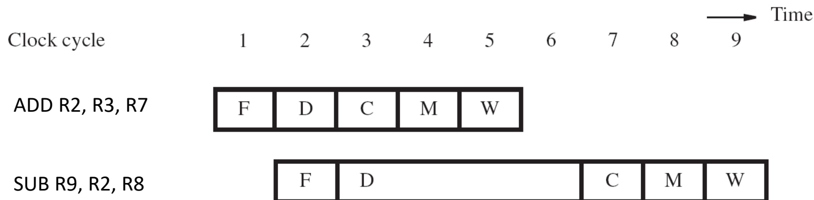
Stalling the Pipeline



We must delay the SUB instruction until it can read the result of the ADD from R2.

- R2 is written in cycle 5
- R2 can be read in cycle 6
- The CPU discovers the dependency during decode in cycle 3
- SUB stalls in decode for three cycles (3, 4, 5) before reading R2 in cycle 6

Stalling the Pipeline



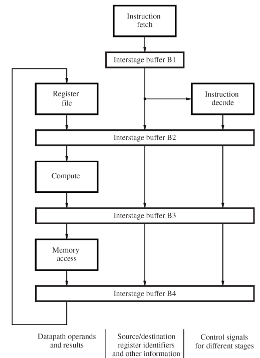
Control circuitry detects the dependencies during decode.

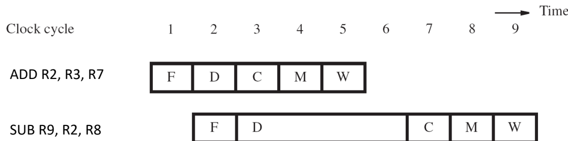
- Interstage buffers carry register identifiers for source(s) and destination of instructions
- In cycle 3, control compares the destination register in Compute (R2) against source(s) in Decode (R2 and R8)
- In this case, R2 matches; SUB is kept in Decode while ADD is allowed to continue

Stalling the Pipeline

What happens when ADD leaves Compute and enters Memory?

- B1 is not clocked, holding SUB in decode
- Meanwhile, control signals in Compute are set to create an *implicit* NOP (no-operation)
- These NOPs (also called *bubbles*) propagate through the pipeline
- Then, Control compares sources in Decode and destinations in later stages
- The dependency remains (ADD in Memory); SUB is stalled again (B1 not clocked)
- This repeats until the dependency clears

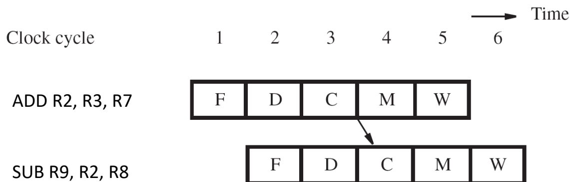




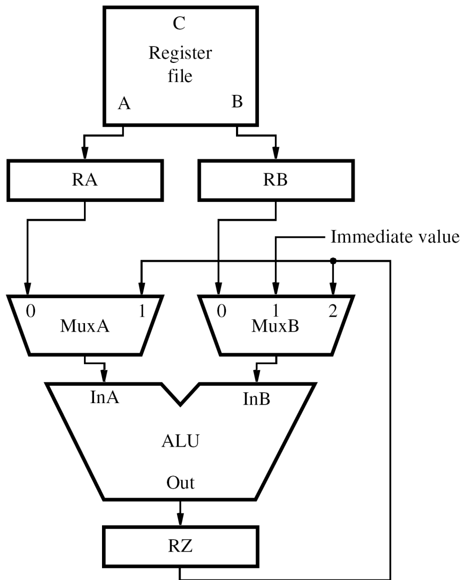
Can we avoid stalling?

We can avoid some hazards by adding extra hardware to the pipeline, and more complex logic to the control circuitry.

- Operand forwarding handles some data dependencies without stalling the pipeline
- In our example, ADD's result is in RZ (within B3) in cycle 4
- We can add inputs to our ALU operand muxes and forward the result from stage 4 to stage 3



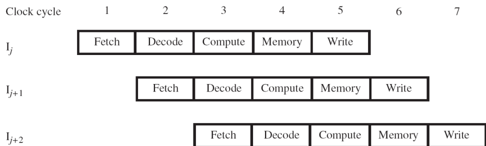
Forwarding: Memory to Compute

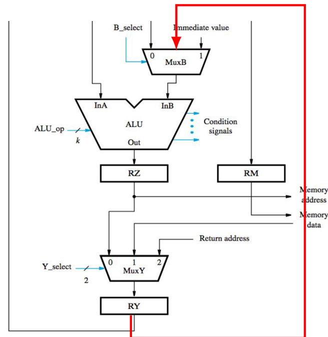


Forwarding: Write-back to Compute

If an instruction separates two with a dependency, we still must stall if we cannot forward. Solution: add more forwarding paths!

```
ADD   R2, R3, R7   // R2 <-- R3 + R7
ORR   R4, R5, R6   // R4 <-- R5 || R6
SUB   R9, R2, R8   // R9 <-- R2 - R8
```



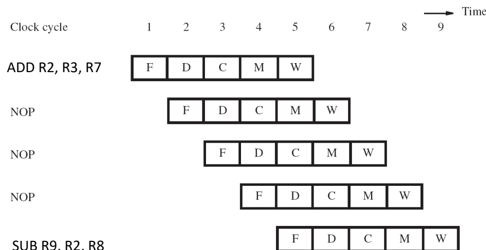


Mux inputs must also be added to accept forwarding from Write-back.

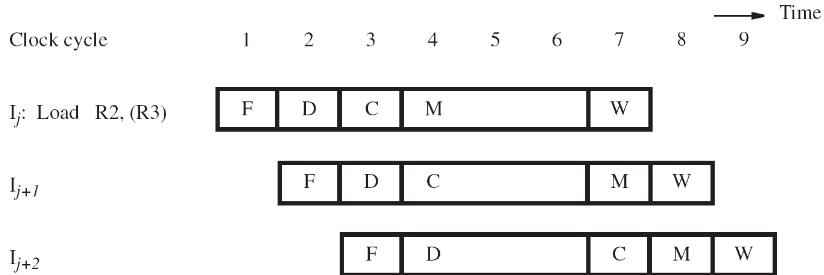
Handling Dependencies in Software

Data dependencies are evident during assembly, and can therefore be handled in software (if, *e.g.*, we do not intend to detect or mitigate them in hardware).

- The assembler inserts three *explicit* NOP instructions
- SUB does not enter decode until the result of ADD is availble
- The assembler can optimize, replacing NOP with independent instructions



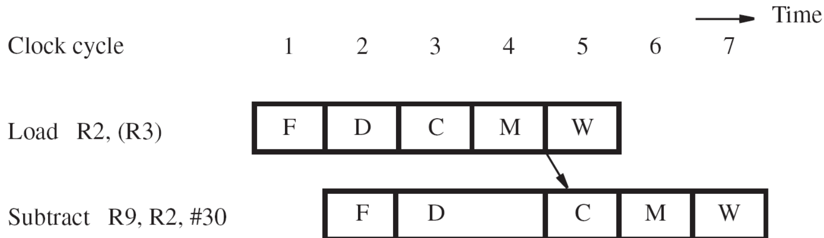
Cache misses can delay instructions in either the Fetch or Memory stages, *e.g.*,



Memory Delays

Even when a load hits in cache, there may be delay due to a data dependency.

- A one-cycle stall is required before the result can be forwarded from the Write-back stage
- Optimize by inserting a useful instruction between the two

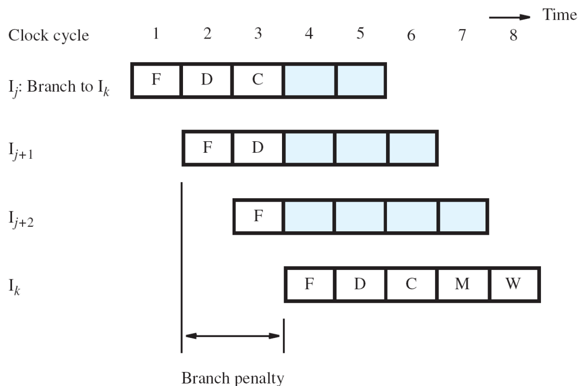


Control Hazards

Remember that ideal pipelining expects that we can fetch a new instruction each cycle, while the previous instruction is decoded.

- Branch instructions must (a) compute the target address, and (b) potentially compare registers
- This comparison determines whether to go to the target address, or execute the fall-through instruction
- A hazard occurs because these operations occur in later stages (*e.g.*, Compute)

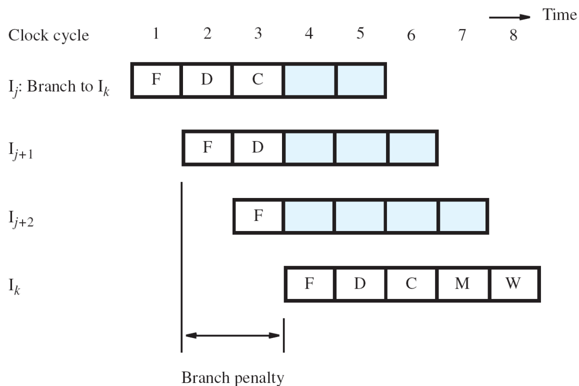
Unconditional Branches



- Target address ($offset + (PC + 4)$) is computed in cycle 3
- Meanwhile, fetch in cycles 2 ($PC + 4$) and 3 ($(PC + 4) + 4$)

- These instructions are discarded, resulting in a 2 cycle penalty

Reducing the Branch Penalty: hardware-based approach



We can reduce the branch penalty by computing the target earlier.

- Add an adder to the decode stage
- This shortens the branch penalty by one cycle

We are adding HW (*i.e.*, cost and energy) to improve performance.

Conditional Branches

```
BEQ R5, R6, label // If R5 == R6, PC <-- PC + displacement
```

- Conditional branches must compute the target address and compare registers
- We can compute the target in Decode with an extra adder
- We can also make a comparison in Decode with an extra comparator

We are adding hardware *again* to improve performance.

Reducing the Branch Penalty: software-based approach

An alternative to adding hardware consists, instead, of *always* leting the two instructions that follow a branch finish execution.

This is called a delay slot (there may be more than one) and this is a feature visible to the programmer.

Intended sequence of instructions:

```
...
ORR  R4, R5, R6
SUB  R3, R3, R8
AND  R1, R9, R8
BEQ  label1
MOV  R6, #7
...
```

Actual sequence of instructions written by the programmer:

```
...
ORR  R4, R5, R6
BEQ  label1
SUB  R3, R3, R8 // delay slot
AND  R1, R9, R8 // delay slot
MOV  R6, #7
...
```

The instructions in the delay slot always execute, because the processor finish executing all the instructions that have entered the pipeline (not wasting the fetch/decoding stage).

What's Next?

This lecture has introduced the basics of processor pipelining. We've looked at:

- Pipelining;
- Data hazards;
- Path forwarding.

Next we'll look at computer hardware for arithmetic.