Note 15: Pipelining and Hazards

Kevin Moy

Abstract

This note dives into the specifics of pipelining and how it increases performance. However, doing so also introduces possibilities of hazards that we must account for.

1 Introduction

Now armed with a basic understanding of a computer's model, from both the highest level (HLLs) and lowest (CPU), we want to now focus on maximization of performance- essentially making instructions run as fast as possible.

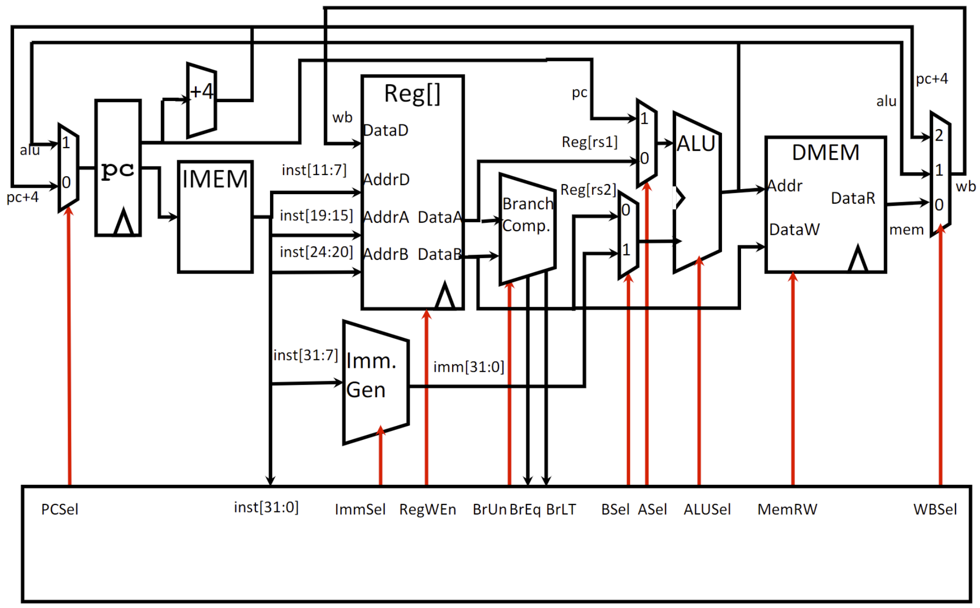
Parallelism is an important concept here, specifically **instruction-level parallelism** the idea of exploiting as much of our datapath components (”hardware”) as possible, which ideally would make things *run* as fast as possible.

Recall in the instruction decode (ID) phase of our data path, we take a 32-bit instruction and decode it based on the appropriate instruction format. This gives us important information such as return/operand registers as well as immediates. Then, based off the specific instruction, we generate our control signals which we feed into various datapath components: MUXes, ALU, DMEM, RegFile to actually execute the instruction physically and correctly.

Recall that pipelining was just breaking a datapath into different stages. Effectively, it allows you to execute all those stages at the same time, and thus increases throughput, as it's like running that many instructions at once!

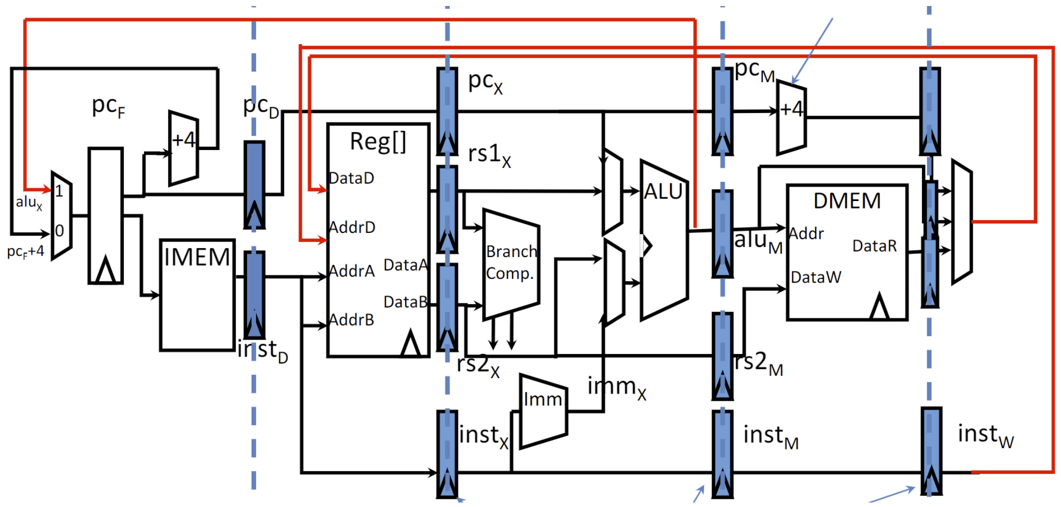
2 Pipelining: A Review

Let's take a look at our finished single-cycle RISC-V datapath:



This is single-cycle. For each clock tick, the instruction specified by PC is executed, and that's it.

Now let's look at a **pipelined** RISC-V datapath:



Notice that we added 4 barriers, making this in a 5-stage pipeline- IF, ID, EX, MEM, WB. Each stage is effectively isolated from each other and controlled by a central CPU clock. For pipelining, we're effectively placing in registers between all the stages, as ”stopping points”, only releasing the values they hold on the next clock tick.

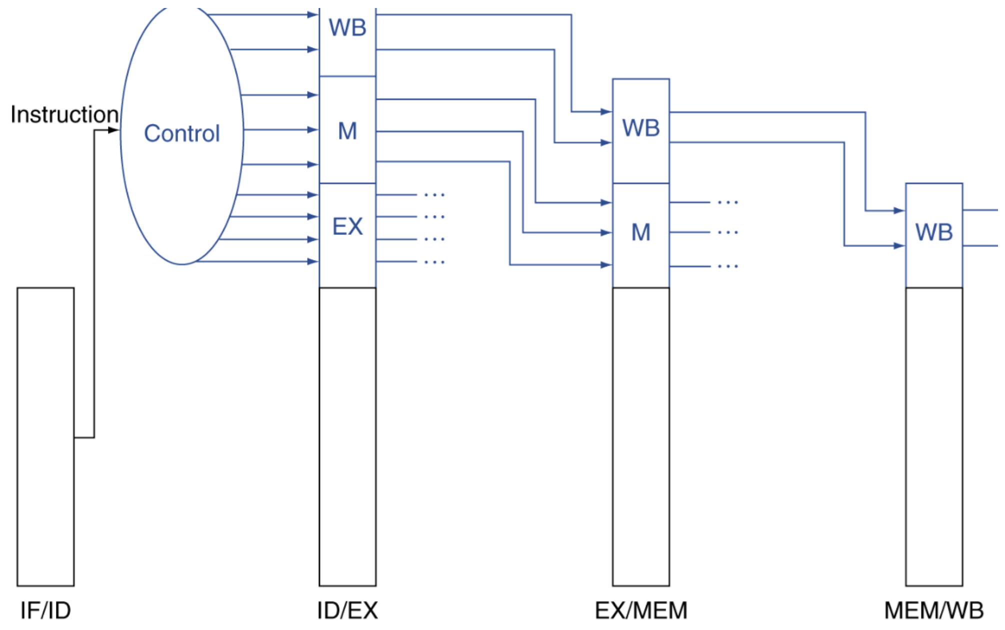
There are some important differences to note. In the MEM stage, we add 4 to `PC` in the MEM stage before letting it go to the write back MUX. `PC+4` is a possible write-back value to `rd`, in `jal` instructions. However, we ALSO need `PC`. So instead of creating four extra registers in our pipeline for both values, we just add the +4 adder in the MEM stage. Minimal hardware means a cheaper and usually faster CPU!

Also notice the **feedback** information into previous stages. For example, the output of the Write-back MUX feeds back information into the ID stage.

Interestingly, the instruction itself also pipelines and propagates through the control logic at each stage!

3 Pipelined Control

We can actually pipeline our control logic as well. All control signals and important information are derived, of course, from the instruction- same as in single-cycle. Consider this pipeline model below:



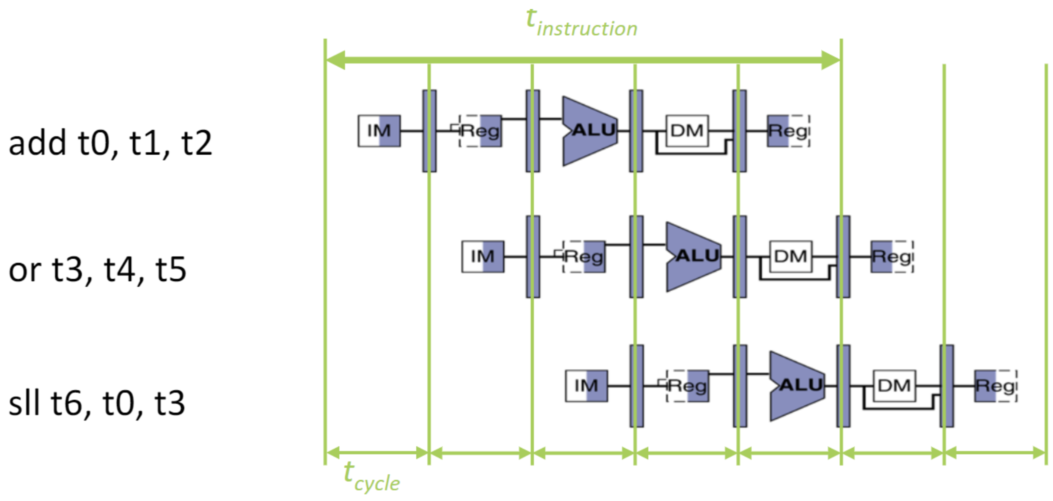
In this model, all instruction control signals are generated in the ID stage, then pipelined from one stage to another in registers based on where these control signals belong. Notice we don't care about EX signals in the MEM stage, and MEM signals in the WB stage, etc, so we don't need to keep those pipeline as we go on.

Another model is simply passing the entire instruction through the entire datapath, so you have access to the full contents at each stage. We see this in our pipelined datapath above.

Let's say we executed sequence of RISC-V instructions now:

```
add t0 t1 t2
or t3 t4 t5
sll t6 t0 t3
```

This diagram below shows the pipelined execution timing of these 3 instructions:



Here, x is time, and the y axis corresponds to our 3 instructions' timing as they pass through the pipelined stages.

Recall we are bottlednecked by our slowest stage in pipelining. That means our clock period has to be big enough just to account for that slowest stage, meaning a *single* instruction will take longer to fully execute in a pipelined datapath than a single-cycle, since only one stage is executed per clock cycle (per clock period seconds). However, our **clock rate** (frequency) will be higher- since we're dividing 1 over our clock period instead of the entire instruction execution time. Overall, our minimum clock period for pipelined CPUs will be FAR shorter than a single-cycle CPU- since we only account for the bottleneck stage.

Remember that we still have to account for hard-coded times like clk-to-q, setup time, and hold time for registers. So our speed improvements won't always be exact, but they will be there for sure.

However, **hazards** can arise from pipelining. Generally, a *hazard* is an unwanted byproduct of pipelining that prevents our next instruction from executing fully (and correctly) in our next clock cycle.

There are three (main) types of hazards:

- **Structural Hazard**: Required resource busy
- **Data Hazard**: Data dependency between instructions
- **Control Hazard**: Flow of execution depends on previous instruction.

4 Structural Hazards

Structural hazards arise when two instructions in (different stages of) the pipeline need to access the same resource at the same time, say, the `RegFile`. At any one time in our pipelined CPU, five instructions are pretty much executing at once, and there's a good chance at least two of them will try to access the same datapath component at the same time.

A possible solution: just have an instruction wait until the other(s) are done. However, if you think about it we're losing the pipelined aspect of our CPU, since now instructions aren't being run in parallel anymore.

So instead of being lazy, we add more hardware. Let's use our `RegFile` example. Both the `ID` stage and the `WB` stage use `RegFile`: we read from it in the decode stage and write to in the write-back stage. Specifically, each instruction can read up to 2 registers in ID, and write one value to a register in WB.

The most obvious solution is just *different ports* for `RegFile`! We specifically have two ports for reading and one port for writing in the `RegFile` component. With this, we can now do two reads and one write at the same time- and allow three accesses per cycle. And yes, this is of course what we have in our CPU currently!

Now let's say (hypothetically) we only have one port for all reading and writing though. What might be some other solutions? There is a mechanism called **double pumping**. On a rising clock edge, we write, and on the falling edge, we read. So we can handle both in a single clock cycle- registers allow this kind of speed!

Another structural hazard comes from memory- we can infer this means reading and writing to memory at the same time as well! But again we already solved this by splitting memory into IMEM (instruction memory) and DMEM (data memory). Remember the IF stage reads from IMEM and the MEM stage reads/writes to DMEM.

So in summary, structural hazards come from instructions competing for the same thing. The solution: more hardware- specifically, making separate components for different types of resource accessed to minimize conflict. As far as our pipelined RISC-V CPU so far, we've actually already solved pretty much all structural hazards that can arise.

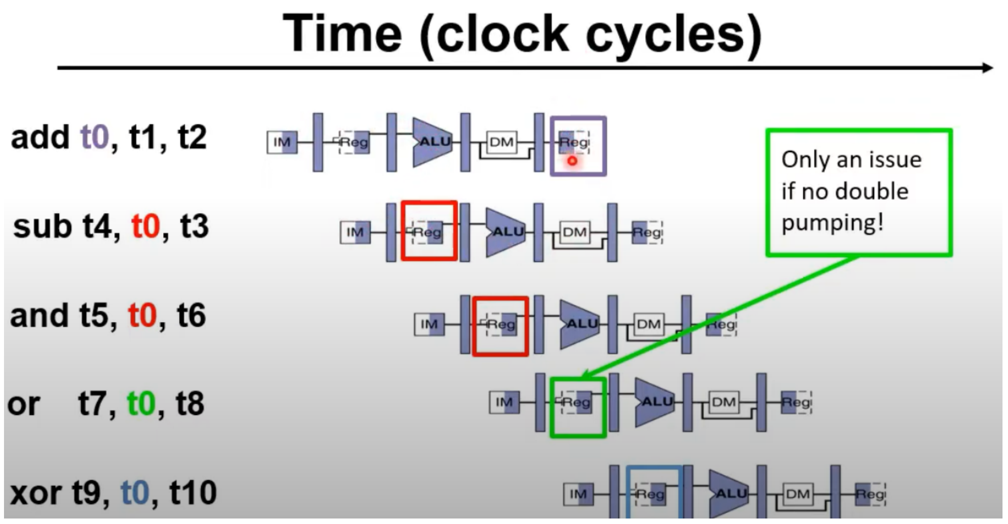
5 Data Hazards

Consider the following 5-piece meal of RISC-V instructions:

```
add t0 t1 t2
sub t4 t0 t3
and t5 t0 t6
or  t7 t0 t8
xor t9 t0 t10
```

The first instruction adds and stores in `t0` and all other instructions read from `t0`. How might this cause issues in a pipeline? Well, we have the very likely probability that `t0`'s value **isn't ready in time** for reading by the other instructions! Remember the `rs1,rs2` registers are read during the ID stage, while `rd` (`t0` in this case) is written to three stages later, in the WB stage.

Below is the pipeline diagram for these 5 instructions:



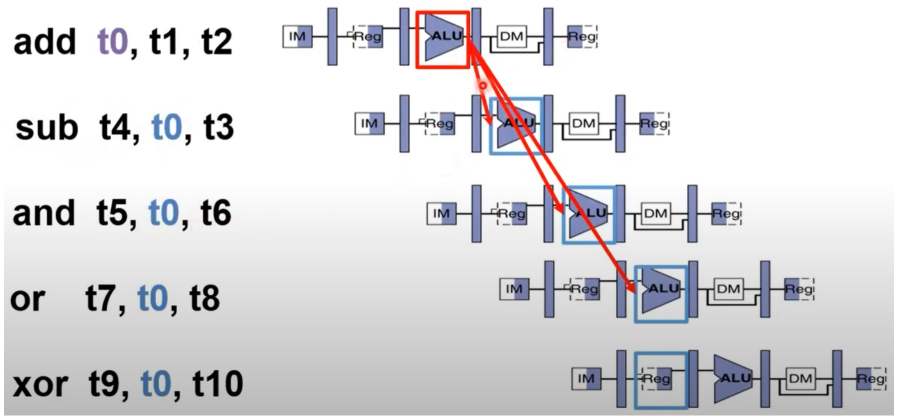
Notice that only at the fifth clock cycle, where the `add t0 t1 t2` instruction is at the WB stage, will `t0` be ready. However, our pipeline requires the second instruction `sub t4 t0 t3` to have `t0` ready at the THIRD clock cycle, in the instructions' ID phase- in other words, it must read from `t0` before its correct value is actually written!

This is an example of a **data hazard**: a problem that arises when an instruction tries to access some component that isn't ready yet. As a side note, double pumping is actually very important: for reads and writes to a register on the SAME clock cycle, it ensures we write before reading from a register, so our read will be of the most recent value. We'll assume double pumping is enabled unless specified otherwise.

One solution, and probably the easiest solution, for data hazards is to **stall**: wait until our data is ready before moving on. In our above example, we know that `t0` won't be ready until the 5th clock cycle, or its WB stage. However, the second instruction, `sub t4 t0 t3`, needs it at the THIRD clock cycle, or its ID stage. So we just stall our pipeline for two cycles! We pass in some bullshit `nop` instruction (a pseudoinstruction that literally does nothing) for two clock cycles, and only on the FOURTH clock cycle do we execute the `sub` instruction, for its ID stage and the WB stage of the first instruction will now be aligned on clock cycle 5. Effectively, each "bubble" passed in shifts the next instruction right by one clock cycle. The compiler actually takes the job of detecting data hazards and inserting no ops where needed.

Stalls aren't the only solution, and certainly not ideal- we waste a significant number of clock cycles for instructions that do nothing, just so we can get the actual instructions aligned correctly. Additionally, stalling increases the **latency** of the instruction: the `sub` instruction now takes two extra cycles to make it through the datapath.

A better solution is **forwarding**. This essentially is the process of forwarding results AS SOON as they are calculated or made available to where other instructions need them. In our above example, as soon as `R[t1] + R[t2]` is calculated (in the EX stage), we can directly wire it to the EX stage of the next (`sub`) instruction as the first operand in the ALU.



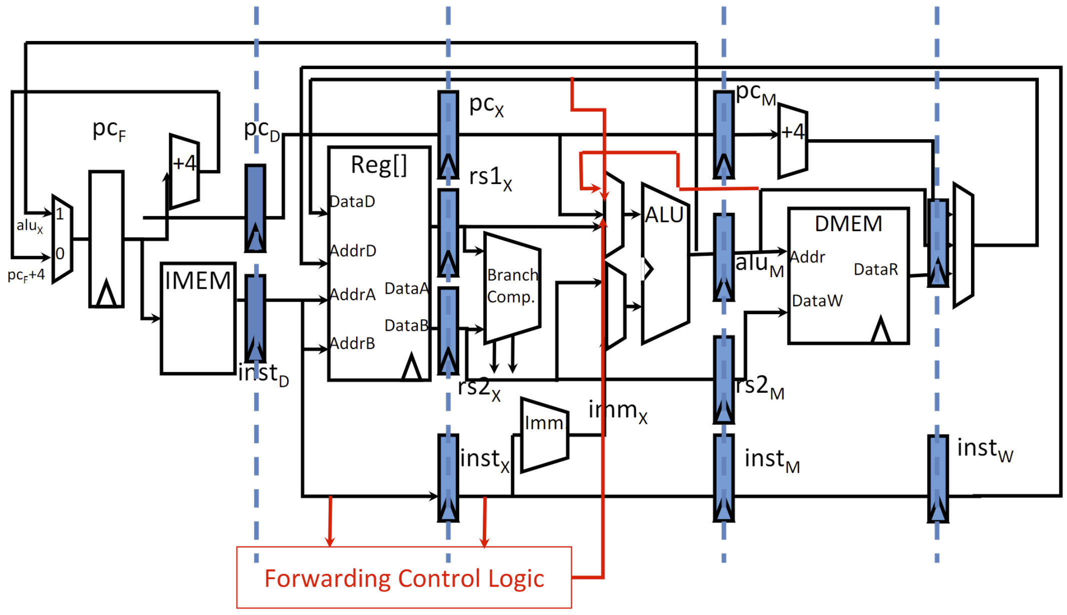
This way, the `sub` and `and` instructions has access to the correct value at the EX stage. The `t0` in their ID stages but this actually doesn't matter: in the EX stage, it will be overwritten with the direct wiring anyway.

So in summary, forwarding is just directly wiring the calculation of what we want to its appropriate inputs for subsequent instructions. The downside? If you think about it, at a lower level this is actually quite complex to implement: lots of wiring across different stages and many different cases to account for.

Let's try to implement forwarding, just for fun, though. How many subsequent instructions after, say, an `add` instruction could cause a data hazard (with double pumping)? Two instructions. That's the role of the compiler to analyze those two and ensure no data hazards can occur.

We want to **compare** the destination register (number) of the old instruction to the source register number(s) of the next two instructions. That way, we'll know if a data hazard will occur.

Below is the pipelined RISC-V CPU with forwarding implemented:



6 Forwarding Failures

Forwarding doesn't actually solve all of our data hazard problems.

We only looked at simple R-type instructions before. Now consider the following 2-piece:

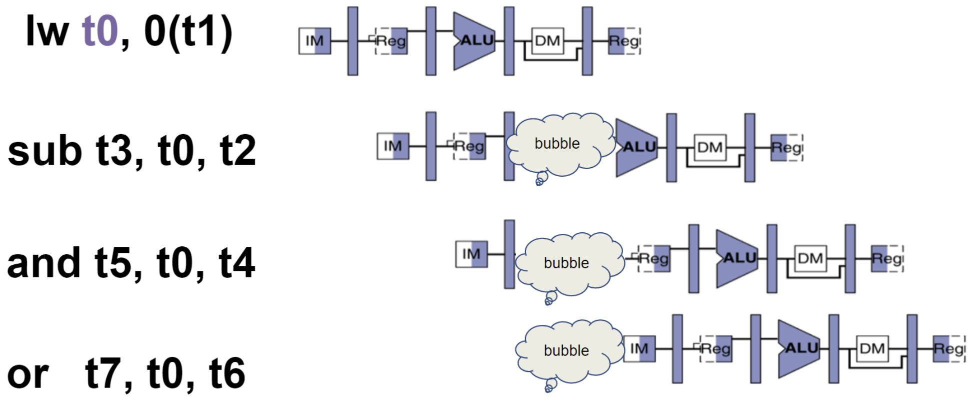
```
lw t0 0(t1)
sub t3 t0 t2
```

Now we have a LOAD instruction.

Notice now that `t0` will actually only be ready at the end of the MEM stage- on the fourth clock cycle. Which means that we can't actually "forward" it to the EX stage of the `sub` instruction- since its EX stage is also at the (beginning of) fourth clock cycle!

Remember forwarding was just a time-saving workaround for stalling. But in the case where we actually want to go *back* in time, what can we do? Forwarding on its own completely fails here. In this case, we *must* stall THEN forward.

If the compiler fails to detect or stall for us, the hardware takes over- this is called **hardware interlock**, where the **hardware stalls the pipeline**.



At the third clock cycle in this diagram, we can detect an issue: `lw` will be at its `EX` stage (when we officially know it's a `lw` instruction and we write to `t0`), and the subsequent instruction (`sub`) is in the ID stage, where it knows it will be reading from `t0`. So the hardware executes the first two stages of `sub` then forces it to wait a clock cycle before resuming. Notice how this has to carry for ALL subsequent instruction in order to maintain the pipeline.

The instruction immediately after a load instruction is called the **load delay slot**: it must stall for a clock cycle IFF it uses the result (`rd`) of the load. It's like just executing a single `nop` instruction in the slot. Any type of execution of no ops will inherently lead to performance loss, since we're just doing a lot of nothing for a clock cycle.

But if we can **re-arrange** our instructions such that the load delay slot(s) are taken up by non-data-hazard inducing instructions, we avoid the need for no ops altogether!

Let's look at two different implementations of `D=A+B; E=A+C;`. Here's the original (more intuitive) execution, which takes 13 clock cycles to execute in pipelined fashion:

```
lw t1 0(t0)
lw t2 4(t0)
add t3 t1 t2
sw t3 12(t0)
lw t4 8(t0)
add t5 t1 t4
sw t5 16(t0)
```

We have some data hazards here. The first load is fine, but the second has problems because we're reading from `t2` on the next instruction! So we need to force a stall for a clockc cycle for `add t3 t1 t2`. In the same way, the `lw t4 8(t0)` instruction has its next instruction read from `t4`, so ANOTHER stall is needed. Notice no problems exist with the last two instructions- we only have data hazard issues if we READ AND WRITE from the same register(s). Overall, 13 cycles if you draw out the diagram.

But what if we could switch some instructions around such that we avoid the stalls AND still have our final outputs (in `t3` and `t5`) containing the correct sums? This is called **code scheduling**. Let's rearrange the loads so they are all clumped together:

```
lw t1 0(t0)
lw t2 4(t0)
lw t4 8(t0)
add t3 t1 t2
sw t3 12(t0)
add t5 t1 t4
sw t5 16(t0)
```

We put `lw t4 8(t0)` in the load delay slot of the second load so it's BUFFERING it such that we take care of something else INSTEAD of a no op! We effectively "bubble" but bubble while doing something useful. Notice now NO load instructions incur hazards! So now with our two stalls avoided, this set will only take 11 clock cycles.

In general, moving the load instruction(s) earlier will generally eliminate the hazards they cause. This is no longer the job of the hardware- now it's the compiler's job.

Sometimes, though, this re-arrangement isn't possible- in that case, we must bite the bullet and accept some stalls inserted by the hardware. Modern compilers have actually developed where they've become *very* good at this instruction rearrangement.

So in summary, there's three possibilities for solving data hazards. The first is pipeline stalling- but this induces performance setbacks. The second is forwarding- but in the case of load instructions, it'll fail for instructions in the subsequent load delay slot. Finally, there is code scheduling or instruction rearrangment- rearranging instructions to avoid the load delay slots but also maintaining the overall logic.

7 Control Hazards

The final hazard that can arise are control hazards, which, accordingly arise from control instructions (branches).

Remember we compare our registers and get our comparison boolean at the end of the branch instruction's EX stage- the 3rd clock cycle. But all the instructions between the branch instruction and the label (instruction)? Obviously we don't want these to execute (since the point of a branch is to skip these middle instructions until we return).

So therein lies the issue with control hazards: we don't always know if our next instruction will be a branch or not, and what the next instruction after the branch will be. Everything depends on the outcome of the branch, which doesn't come until the third stage (EX)!

For this unique kind of hazard, this is a pure time constraint. We want to stop a certain number of subsequent instructions from starting entirely. Thus for the next 2 instructions, it's possible that control hazards can occur.

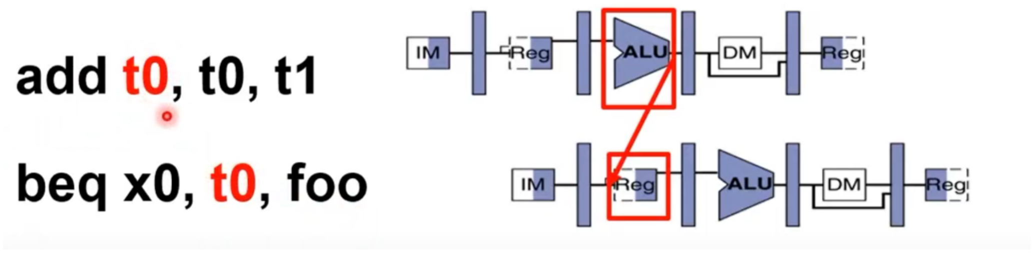
So for now, our (naive) solution will just be to stall on EVERY branch, i.e. TWO bubbles, for the next two instructions, or waiting 2 clock cycles, since we want to shift right by 2.

Is there a way we can fix this? A potential solution is moving the branching logic, i.e. branch comparator, to the ID stage instead. Immediately after reading the values of `R[rs1]` and `R[rs2]` in the ID stage, immediately compare them. This is kinda like forwarding! Of course, we also need to calculate `PC+imm` immediately in the ID stage (instead of EX) as well (if branch comparison, now in the ID stage, is true).

But this introduces a lot of new problems as well. Consider the two-piece below:

```
add t0 t0 t1
beq x0 t0 foo
(...)
foo:
```

We have an `add` instruction writing to `t0` and a subsequent branch instruction READING from `t0`. Read+Write from the same register in subsequent instructions should immediately trigger an alert in your head- and it is indeed an issue! Let's look at their pipeline diagram:



The ID stage of `beq` is in the same clock cycle as the EX stage of `add`: now that our comparison is being moved to the ID stage, forwarding can no longer occur and `beq` would be reading an incorrect value of `t0`. So we'd have to do another stall for this branch instruction, THEN forward. This means we really didn't eliminate any stalls and just added more hardware. No bueno.

So overall, this ID stage comparator thing won't work, unfortunately. So is it just back to stalling every 2 instructions after?

If the branch comparison is false, there are no problems: subsequent (2) instructions are executed as normal. However, if true, we basically need to flush that pipeline: we need to actually convert our subsequent two instructions to `nops` OR just insert them in.

It's the same logic for jumps, except we ALWAYS have to account for the control hazards in that case (no conditionals).

8 Branch Prediction

Are we doomed to `nops`?

Let's do a recap of what our dilemma is. We have branch instructions, and we don't know what our next instruction will be until the end of the EX stage, or at least the third clock cycle. Thus the immediate subsequent two instructions will run through some of their stages even though they shouldn't. Worst case: 2 cycle stall for these 2 instructions.

But there's an even better solution called **branch prediction**. This is basically just *guessing* the branch outcome. If we predict true (take the branch), we take it and see what happens. If we don't think we gonna take it, just don't. Seems kinda weird, but let's analyze further.

This branch predictor is a piece of hardware that is executed in the branch instruction's IF stage, and we immediately guess the next PC and then execute the corresponding instruction on the next clock cycle. If this predictor is right, there'll be absolutely no stalls needed! The next instruction in the pipeline will be the correct one.

In practice, this is excellent and is sensibly much faster than the standard stall procedures.

Branch control hazards are really more emphasized with pipelines with far more than five stages. Here, the delay that comes with stalling will be *far* greater than two clock cycles, so a good branch predictor becomes very important. So for CPUs with deeper pipelines, we want **dynamic branch prediction**: we keep a history table of past branch predictions. Whenever we predict (and execute) a branch, if our table is correct, life is good.

If our prediction wrong, however, we must update: we flush the pipeline and flip the prediction. Whenever we predict to take the branch, we "speculatively execute" it- sort of like "testing" its execution. After two more stalls or clock cycles, though, the branch instruction reaches the EX stage, which tells us if our prediction was actually right or not. If it turns out we shouldn't have taken the branch, we must clear our pipeline and restart at the branch instruction. We've just executed 2 more pipelined instructions for 2 clock cycles- the same error incurred with stalling.

"Eager execution" is another option where we have two parallel pipelines, one executing branch and other executing not. The correct one will be merged back into the "official" pipeline and the incorrect will be discarded. However, this leads to security vulnerabilities.

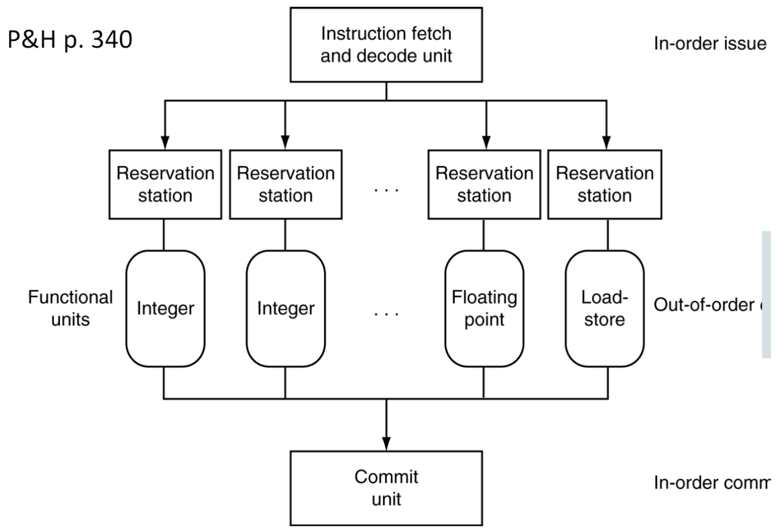
So in summary, to deal with most control hazards, branch predictors are definitely the way to go. Incredibly useful with very little risk.

9 Superscalar Processors

So the purpose of all of this information was to increase the performance of our processor. Through pipelining and utilization of ALL stages at each clock cycle, we were able to heavily increase clock rate and performance. Now let's talk about something cool.

Superscalar processors utilize adding even more hardware to our processor- for example, more ALUs, more execution units, more branch predictors, etc.

Below is a flowchart of superscalar processor functionality:



Same as normal processors, we have an instruction fetch and decode unit. But notice now we have many functional units: many for integers, some for loads/stores, and others. So we now can parallelize and run even MORE instructions at once. These processors also implement **out-of-order execution**, where they dynamically take an input of some series of instructions, then rearrange execution order so hazards are avoided and as many functional units are used as possible. With this massive amount of parallelization, the CPI (cycles per instruction) can actually be reduced from 5 to ONE. Each instruction still needs at 5 cycles to complete individually, but if we are doing all these instruction completions at the same time per clock cycle, we decrease the *average* significantly.

10 Summary

Today we analyzed three types of hazards: problems caused by pipelining, as well as solutions for all of them. The first type is structural hazards, which occur when two stages compete for a single datapath component. The second is data hazards, when some component (mainly registers) are in progress and aren't ready for (reading by) a subsequent instruction. Finally, control hazards are just unwanted instructions getting executed on branch or jump statements.

Then we covered solutions for all of them: stalling, forwarding, instruction scheduling, branch predictors. All contribute to the speed of modern processors, and the speed of the computer you have right now.