Lecture 7 - Pipeline

Gidon Rosalki

2025-05-11

1 Recap - single cycle, multi cycle

Single cycle has 5 stages to execute an instruction (fetch, decode, execute, memory, writeback), and each instruction is executed in a single cycle. This results in long cycle times, especially when some instructions need not one of the stages (consider memory for example.
Multi cycle takes the single cycle implementation, and adds registers between the stages. This allows us to separate the single cycle into multiple smaller cycles for the stages for executing an instruction, allowing us to have a much higher cycle time.

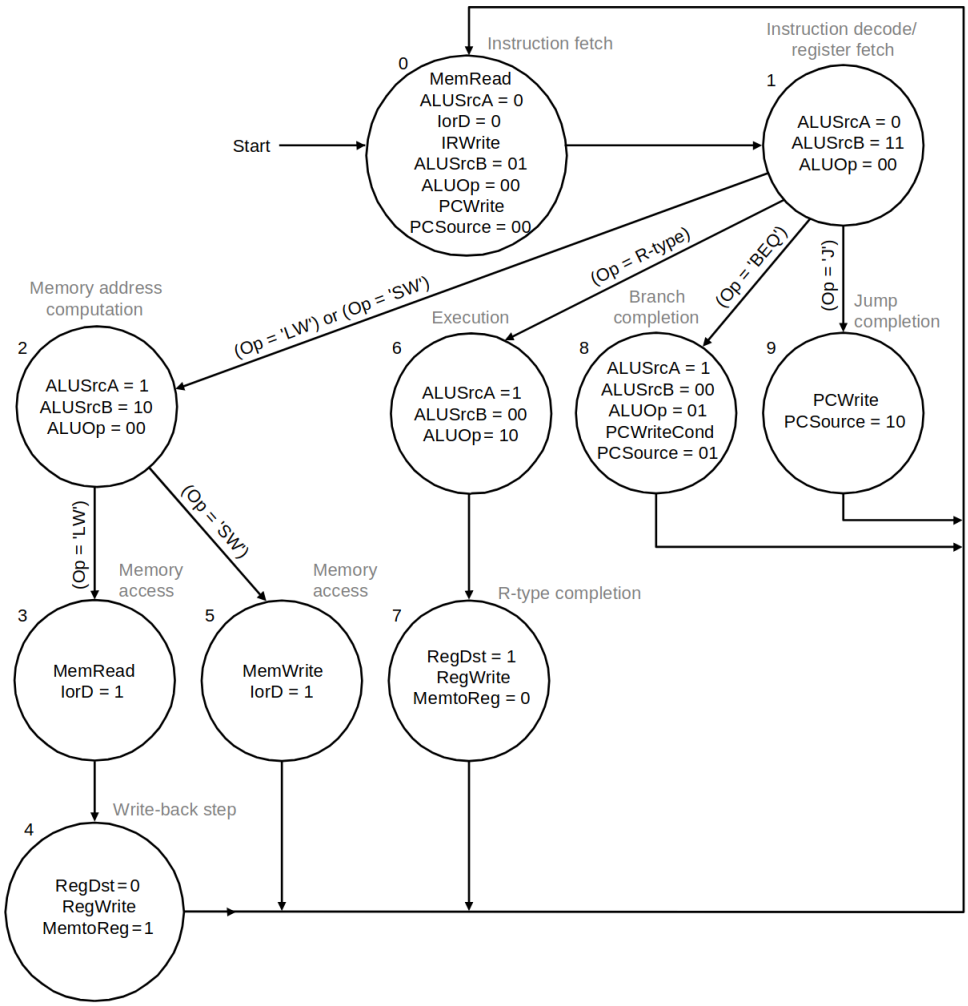

Figure 1: MIPS state machine

However, despite there being many benefits to multi-cycle, such as saving hardware elements through reuse in different stages, and having a shorter cycle time, it also has a higher CPI (cycles per instruction), which is bad, especially since currently the frequency increase is lower than the CPI increase, resulting in lower performance (for now). Today we will keep the short cycle time, and reduce the CPI.

2 Pipeline overview

The idea is to keep everything busy with useful work. It is a general purpose technique to improve efficiency, not just specific to CPUs. We organise the pipeline by splitting the process into independent stages, this allows a stage to start executing the next workload while the following stages process previous ones. There are many examples in real life, such as assembly lines, or security at airports, and so on. Consider laundry, if laundry has 4 stages, washing, drying, folding, and storing, let us state that every stage takes half an hour. If I want to do 4 loads of laundry, then it will take $4 * 2 = 8$ hours, however, if I pipeline, and do the next stage for every load while the previous is executing, then we may cut it down to 3.5. In short, I wash the first load, then I wash the second, while the first is drying, I wash the third, while the second is drying, and I'm folding the first, and so on.

2.1 Definitions

Latency: The time between when something is initiated, and when its effects begin / become dtectable. This can also be considered the time to completely execute a specific task, from the start to the end. For example, the latency to travel by the motorway from Jerusalem to Tel Aviv is 50 minutes (no traffic).
Throughput: Units of work that can be done in a period of time, for example, the Jerusalem to Tel Aviv motorway has a throughput of 30 cars per minute.

CPI: Cycles per instruction, the *average* number of clock cycles to execute an instruction. $\text{IPC} = \frac{1}{\text{CPI}}$

Following is a diagram making use of pipelining on the multi cycle MIPS implementation:

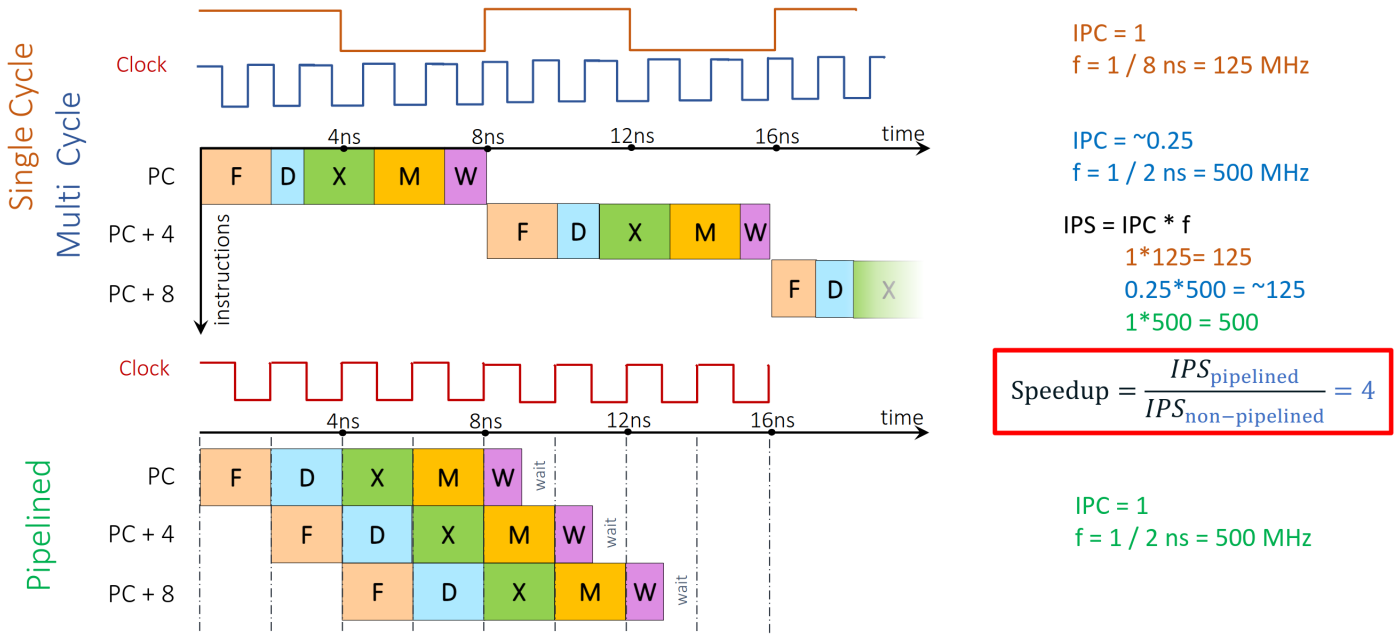

Figure 2: Pipelined vs non pipelined

Pipelining does not reduce the latency of a single workload (instruction in our case), but rather increases the total throughput (number of completed workloads per time unit). The pipeline throughput (= rate) is limited by the slowest stage. The potential speedup is

$$\text{speedup} = \frac{\text{Throughput}_{\text{non pipelined}}}{\text{Throughput}_{\text{pipelined}}} = \frac{\text{workload time}_{\text{non pipelined}}}{\text{slowest stage time}_{\text{pipelined}}}$$

In the ideal case, where all stages have the same length:

$$\text{speedup} = \frac{\text{workload time}_{\text{non pipelined}}}{\text{slowest stage time}_{\text{pipelined}}} = \text{Number of stages}$$

n order to increase the speedup, we partition the pipe to many pipe stages, and balance the work in the pip stages, as in we make the longest stage as short as possible.

3 Pipeline implementation

Let us begin by ignoring branching, and control signals.

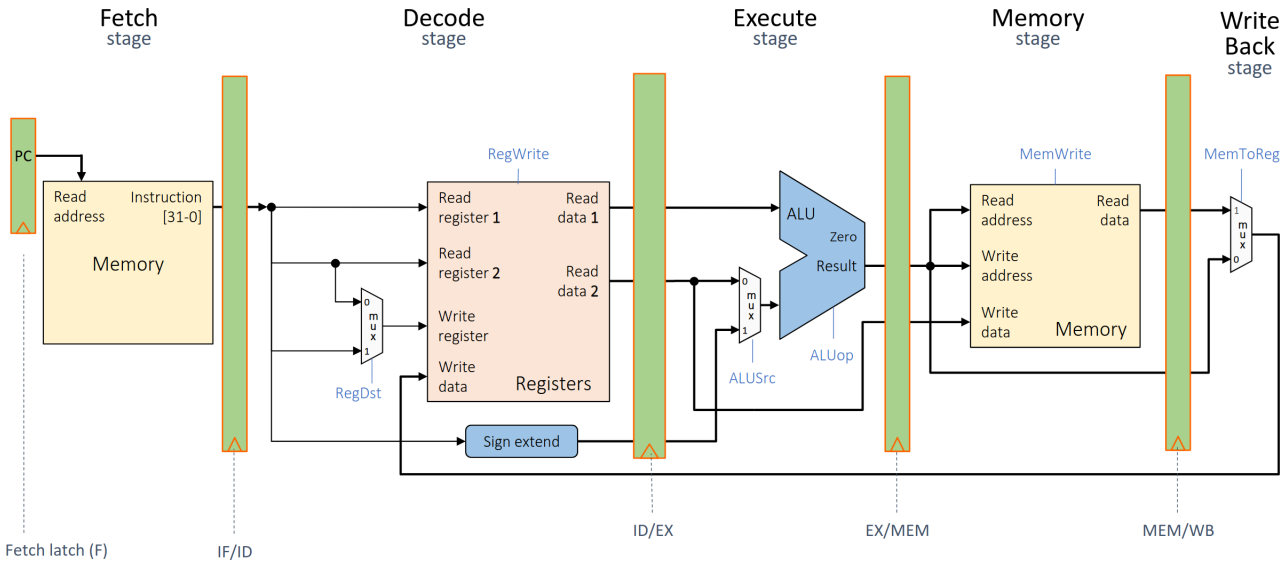

Figure 3: Adding registers to create stages

Every green pipe is a clock cycle and pipeline stage, and at each clock cycle we are locking the relevant registers, and storing the output from the previous stage in them.

There is an extensive example of calling commands with pipelines in the slideshow, but replicating it here would be tedious.

4 Pipeline hazards

Hazards prevent the next instruction from executing during its designated cycle. There are 3 types:

1. **Structural** hazards: hardware can't support a combination of instructions (for our laundry examples, a single person to fold and put clothes away)
2. **Data** hazards: Instruction depends on the result of prior instruction which is still in the pipeline
3. **Control** hazards: Branch resolution depends on the result of a previous operation

4.1 Structural hazards

Consider for examples Instruction/Data memory, or read and write GPRs.

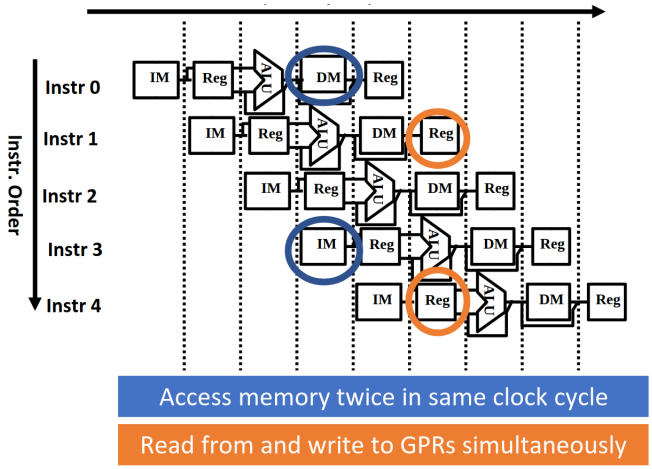

Figure 4: Structure hazard abstract

To resolve the problem where we try and read and write to the same register simultaneously (writeback from a prior cycle, and a read from a subsequent cycle), we will add a comparison to the internal structure of the register file. We will check if we are reading and writing to the same register, and if so, we will return the value of the write to the read, instead of the original value of the register.

4.2 Data hazards

An instruction depends on the result of a prior instruction that is unfinished, and is still in the pipeline. There are 3 types:

1. Read after Write (RAW): An operand is modified, and read soon after.

   ```
   add $2, $1, $3
   add $4, $2, $3
   ```

2. Write after Read (WAR): Read an operand, and write soon after.

   ```
   add $4, $1, $3
   add $3, $1, $2
   ```

3. Write after Write (WAW): Two instructions write in the same operand.

   ```
   add $1, $2, $3
   add $1, $4, $5
   ```

The second 2 are not real hazards, since the second instruction is not dependent on the first, but in the first where the read after read (on the last register of the instruction) is not a hazard, the RAW on register $2 is indeed a hazard.

4.2.1 RAW data hazards

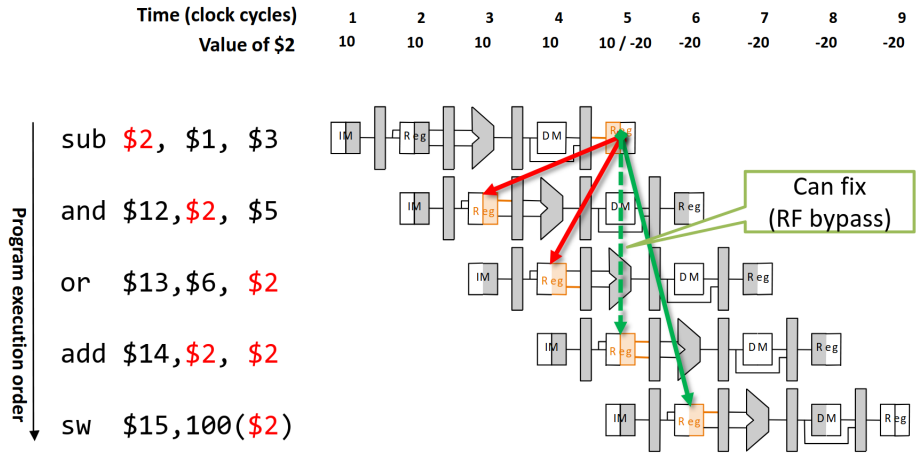

Figure 5: RAW data hazard

We see that we have subsequent instructions attempting to read the value from a register, before that value has been written to it. We have a few solutions available:

1. NOP generated bubble. The compiler adds a NOP instruction after instructions on which subsequent instructions are dependent, to ensure that the data is called in writeback before a subsequent instruction calls read. In the above example, we would add 3 NOPs after the first `sub` to allow this. However, this results in a big performance hit, since we must add many instructions that do nothing. Perhaps we could rearrange the instructions to resolve this, but this results in changing the program

2. The hardware can detect hazards, and add stalls to add in this delay, so in the above example, the hardware would add 3 stalls after the first sub, however, this also results in a big performance hit.

3. Forwarding: Instead of waiting for results to be written into the register file, use them as soon as they are calculated. We achieve this by adding a multiplexer connecting the output register of the ALU to the input, allowing us to call directly from the output of the previous command:

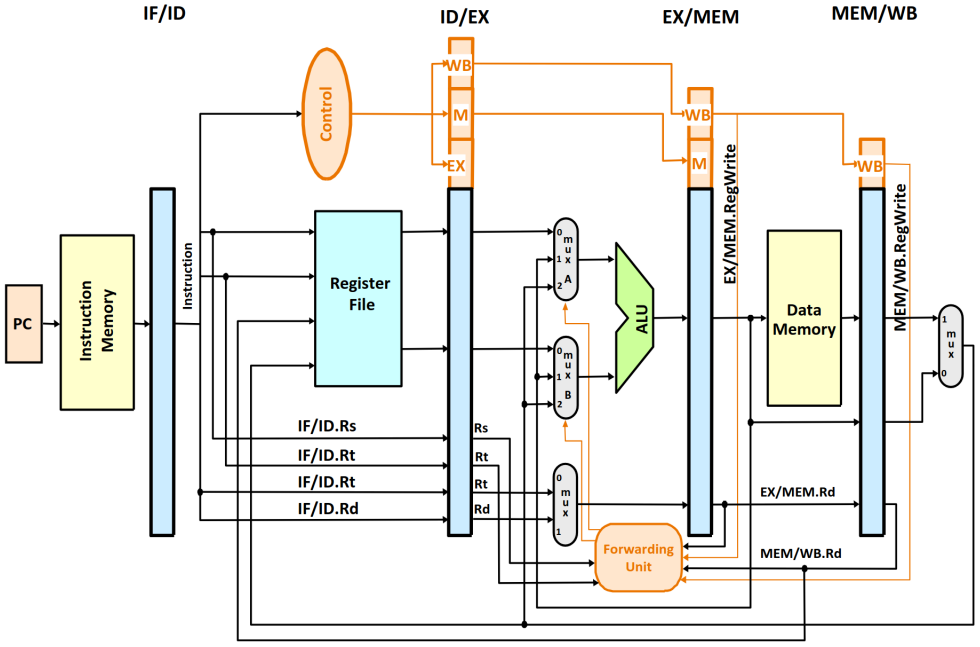

Figure 6: Forwarding the ALU output to the execute stage

- From MEM to EXE
- if (EX/MEM.RegWrite and (EX/MEM.WriteReg == ID/EX.ReadReg1)) then ALUSelA = 1
- if (EX/MEM.RegWrite and (EX/MEM.WriteReg == ID/EX.ReadReg2)) then ALUSelB = 1

- From WB to EXE
- if (MEM/WB.RegWrite and (ALUSelA != 1) and (MEM/WB.WriteReg == ID/EX.ReadReg1)) then ALUSelA = 2
- if (MEM/WB.RegWrite and (ALUSelB != 1) and (MEM/WB.WriteReg == ID/EX.ReadReg2)) then ALUSelB = 2

Figure 7: Forwarding the ALU pseudocode

4.2.2 Memory to memory copy

Here we load from a register immediately following a store from the same register:

```
lw $1, 4($2)
sw $1, 4($3)
```

So we have not finished loading the value into $1 the value we want to write. We cannot resolve this by forwarding the data, we have to complete the store first, so we will carry out **stalls** instead. We will use the following logic for stalls:

```
if (ID/EX.RegWrite && (ID/EX.opcode == lw) && ((ID/EX.WriteReg == IF/ID.ReadReg1) || (ID/EX.Wri
    {
        stall();
    }
```

but only if the instruction at ID really reads these registers. To add hazard detection to our forwarding, we will use the following implementation:

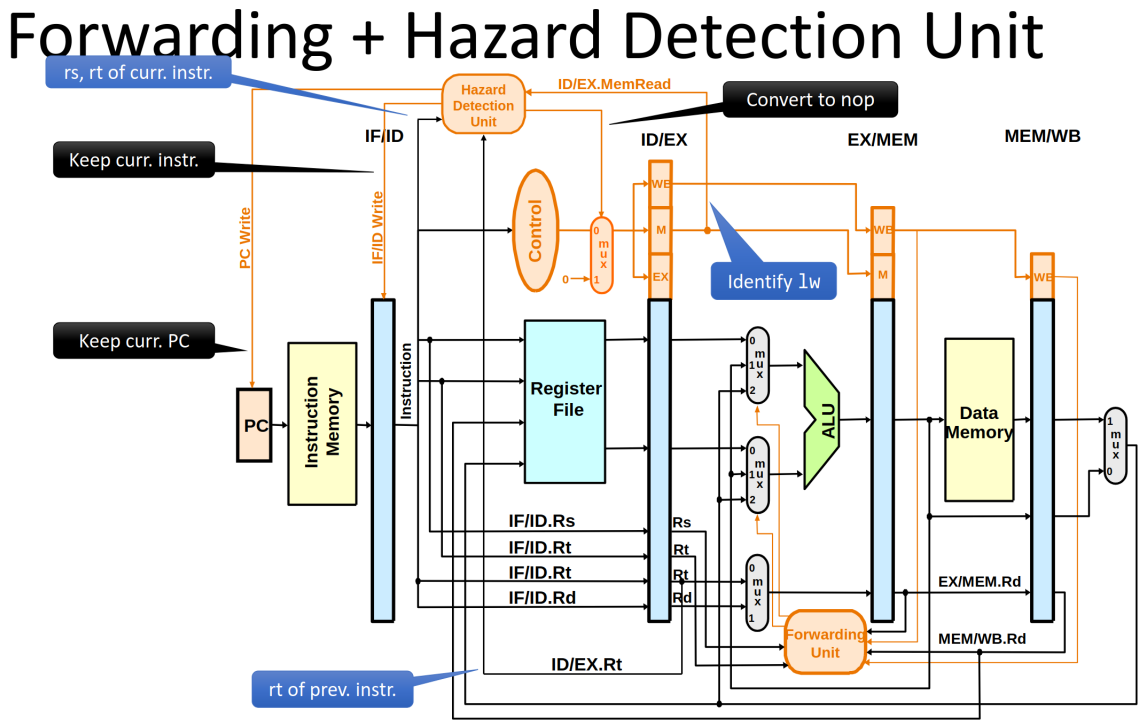

Figure 8: Forwarding with hazard detection unit

The instruction slot after a load is called the ”load delay slot”. If that instruction uses the result of the load, then the hardware interlock will stall it for one cycle. The compiler can avoid the stall by putting an unrelated instruction in that slot. For example:

```
        LW  Rb,b
        LW  Rc,c
Stall   ADD Ra,Rb,Rc
        SW  a,Ra
        LW  Re,e
        LW  Rf,f
Stall   SUB Rd,Re,Rf
        SW  d,Rd
```

We can convert this by moving two instructions as follows:

```
LW  Rb,b
LW  Rc,c
LW  Re,e
ADD Ra,Rb,Rc
LW  Rf,f
SW  a,Ra
SUB Rd,Re,Rf
SW  d,Rd
```

and so we no longer need the stalls.

4.3 Control hazards

In the pipeline, the current instruction is decoded while the next one is fetched. For sequential reasons, the PC advanced by 4 during fetch. A control hazard happens when program flow is **not** sequential. For example, conditional branches such as `beq` and `bne`, or unconditional branches such as `j`, `jal`, and `jr`, or even exceptions. This leaves us with a problem, how can we identify a branch before we decode it?

We could add nops, but this involves a three cycle penalty per branch, a very large waste given the prevalence of the branch instruction in code.

We can instead insert a special branch comparator in stage 2. As soon as an instruction is decoded, and the opcode is identified as branch, we immediately make a decision, and set the new value of the PC. This has the benefit that branch is completed in stage 2, and thus only one unnecessary instruction is fetched, and so only one `nop` is needed.

Overall, we have a few solutions. We may **stall** the pipeline, and not execute instructions until a branch condition is resolved. This has a 3 cycle penalty for each branch, and since branches are very common (on average, every 8th executed instruction is a branch, and 80% of program execution time is spent in loops), thus our IPC would be severely impacted:

$$\begin{aligned} CPI_{\text{ideal}} &= 1 \\ CPI_{\text{real}} &= CPI_{\text{ideal}} + (\% \text{ of branches} = 12.5\%) \cdot (\text{penalty} = 3) \\ &= 1.375 \end{aligned}$$

We could **fetch not taken**, where we execute instruction from the not taken path, as if there is no branch, and that way if the branch is not taken, then there is no penalty. If the branch was taken, then we flush the wrong instructions before they change the CPU state (registers / memory), and fetch the instructions from the correct (taken) path. Assuming 50% of branches were not taken on average, then the IPC will be 0.84 of the ideal, on average. This is a 16% slowdown, and we need to find something better.

Next we have **delayed branches**. What if the software (compiler) could help? Then a change to the ISA is required. We redefine a branch to take place after n following instructions. The hardware therefore executes n instructions following the branch, regardless of whether or not a branch was taken. The compiler fills the n slots following the branch with instructions that should be executed regardless of branch resolution. These are either from before the branch instruction, or instructions that come after the branch paths have converged. If the compiler cannot find such instructions, then it can always put in nops. Unfortunately, filling in these slots is difficult, especially as the number of slots increases. Assuming that we can fill d% of the delayed slots, then

$$CPI_{\text{real}} = 1 + (0.2 \cdot (1 - d)) \cdot 3$$

So for example, for $d = 0.6$, we get $CPI_{\text{real}} = 1.2$. Additionally, by mixing our architecture, with our uArch, since new generations require different numbers of delays slots, this leads to countability issues between generations.

Finally, we have **Dynamic branch prediction**. As soon as an instruction is fetched (at the IF stage), we change the PC to the predicted path, and switch to the right path after the branch execution if the prediction was wrong. Complex hardware at the IF stage predicts whether or not the instruction is a branch, would a branch be taken or not, and if it is taken, what is the target. This is called the **Branch Target Buffer** (BTB). It is organised exactly like a cache.
This way, if the prediction is correct, then we change nothing, and a wrong prediction involves flushing the pipeline, and restarting from the correct PC. Assuming a correct prediction rate of p%:

$$CPI_{\text{real}} = 1 + (0.2 \cdot (1 - p)) \cdot 3$$

So for example, if $p = 0.85$, then $IPC_{\text{real}} = 0.92$. Modern predictors can even achieve 0.95+.