Pipelined MIPS processor

Carl-Johannes Johnsen

Department of Computer Science
University of Copenhagen

April 18, 2017

In this lecture, we will be looking at pipelining our single cycle MIPS processor.

We will go through the motivation and background for pipelining, and the steps for implementing it in SME.

Finally, we will look at handling the problems introduced by pipelining, by adding two new units: the Forwarding Unit, and the Hazard Detection Unit.

The single cycle MIPS processor is not very efficient, as the clock rate is determined by the longest possible path in the processor.

In order to increase the clock rate, we must decrease the longest path in the processor, by introducing pipes.

Pipes are registers in the processor, which temporarely store all the values computed so far.

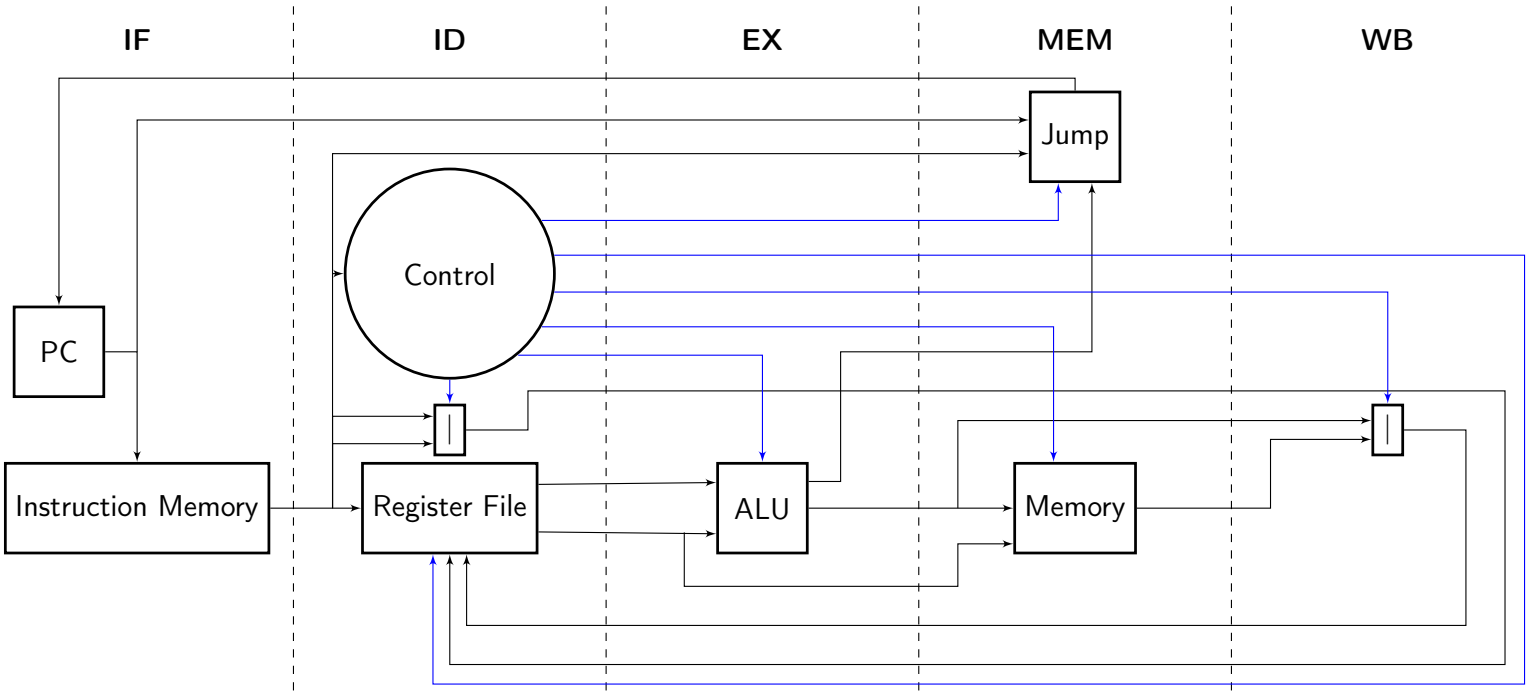
This ensures that the data does not have to travel as far, until it has reached a safe state.

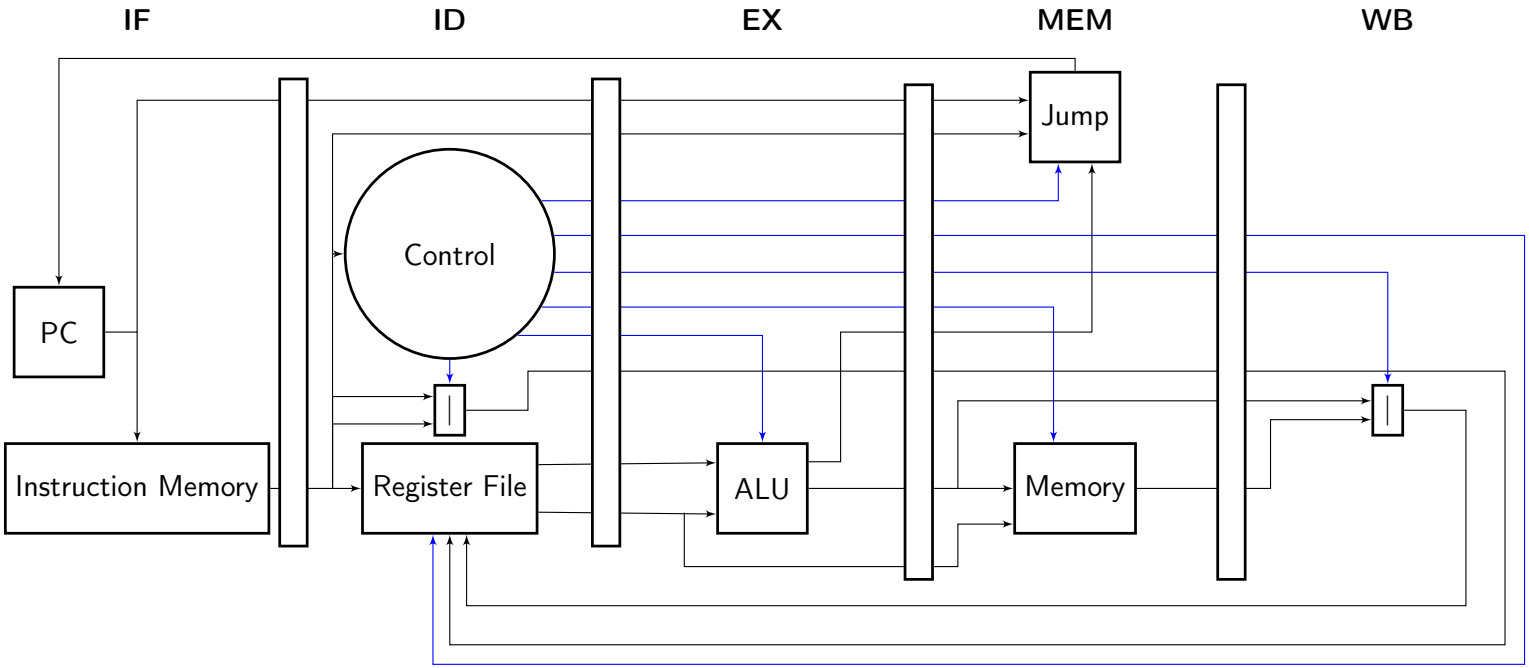
Determining where to place the pipes, is done by dividing the processor into stages.

We will follow the classic MIPS example, and divide the processor into 5 stages:

- Instruction Fetch (IF)
- Instruction Decode (ID)
- Execute (EX)
- Memory (MEM)
- Write Back (WB)

We are going to insert a pipe between each state, i.e. 4 pipes.


IF
ID
EX
MEM
WB
Jump
Control
PC
Instruction Memory
Register File
ALU
Memory


IF
ID
EX
MEM
WB
Jump
Control
PC
Instruction Memory
Register File
ALU
Memory

We have two ways of implementing pipes in SME:

- Clocked Busses - for busses which only traverses 2 stages
- Clocked Processes - for dividing busses traversing more than 2 stages

Just adding the `ClockedBus` attribute to the busses seems simpler. However, it can become more explicit, by adding additional busses, and to have a process, which explicitly touches all of the busses, which should go into the pipe.

Introducing the pipes is fairly straightforward. For each pipe, we add a copy of the bus, which the 'next' stage needs. Then, for each pipe, we add an SME process, which takes all the busses from the 'previous' stage, and outputs their data on the matching newly added pipe bus. Finally, the references in the 'next' stage should be updated to look at the piped busses.

This process can be repeated for all of the required pipes. There is only one problem: the Jump Unit. The processor do not know when to jump, until the MEM stage, as the adresses needs to be computed in the EX stage.

To solve this, the Jump Unit should be divided out to the different stages. The IF stage should handle incrementing the Program Counter, and choosing between the addresses from the MEM stage, and the incremented Program Counter.

The EX stage should as mentioned, compute the addresses, and finally, the MEM stage should hold the logic for choosing between the branch address and the jump address.

Finally, in the single cycle MIPS processor, we added a Write Buffer in order to eliminate the cycle from the Register File to the Register File.

However, by introducing pipes, we have also introduced buffers, and as such, we can remove the Write Buffer.

To test the pipelined processor, we could use any of the programs we have previously written. However, since we have pipelined the processor, we need to insert bubbles in our program, to ensure the data is ready for all of the instructions.

An bubble is a No Operation (`nop`) instruction, which performs no operation, and touches neither the Register File nor the Memory.

We are going to implement a simple program, which is easy to verify: a small loop, which computes n fibonacci numbers, and places them in memory.

As we have done previously, we are going to give pseudo low level C code.

```
void init(int *arr) {
    *(arr)   = 1;
    *(arr+1) = 1;
}

void loop(int *arr, int n) {
    int i, tmp1, tmp2, tmp3;
    for (i = 0; i < n; i++) {
        tmp1 = *(arr+i);
        tmp2 = *(arr+i+1);
        tmp3 = tmp1 + tmp2;
        *(arr+i+2) = tmp3;
    }
}
```

Note: if the program is written in actual C, the allocated array should be $n + 2$, due to initialization values.

Furthermore, when we port it to MIPS assembly, after each instruction, we should insert a bubble of 4 `nop`'s.

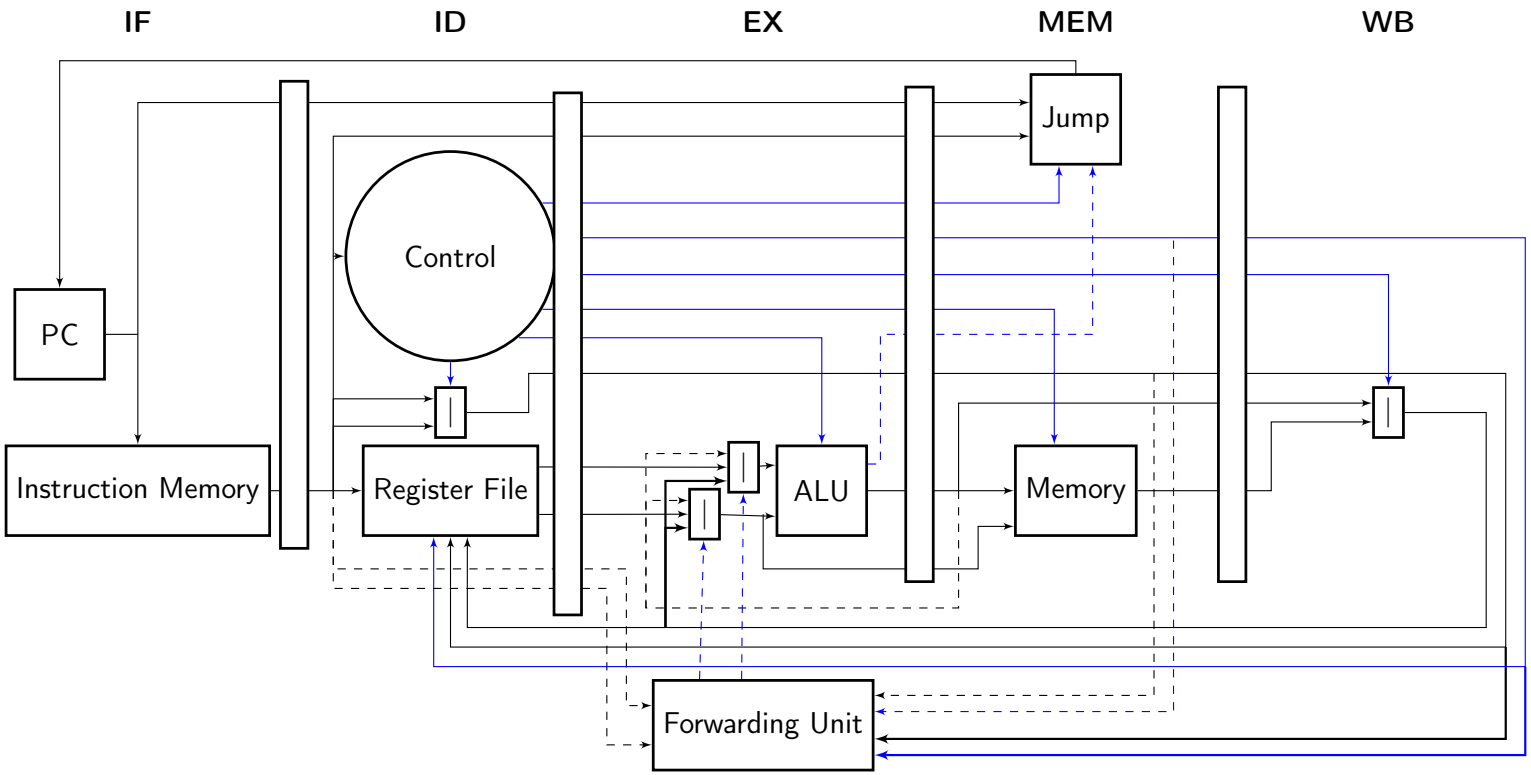
When the program has run, the $n + 2$ fibonacci numbers should be in memory, at the given address.

By introducing pipes, we also introduced data hazards and control hazards. We start by looking at handling data hazards.

Data hazards are when one instruction writes to a register that a following instruction reads from. This was not a problem in the single cycle processor, as all the data had been written in the same clock cycle.

We can eliminate some of the data hazards by implementing an additional unit: the Forwarding Unit.

This unit looks at the registers used by the instruction in the EX stage, and if the instruction in either the MEM or the WB stage writes to either of the registers, then the Forwarding will forward that data to the EX stage.


IF
ID
EX
MEM
WB
Jump
Control
PC
Instruction Memory
Register File
ALU
Memory
Forwarding Unit

Implementing the Forwarding unit should be done with an SME process, which should be put in the EX stage.

The unit should be controlling two multiplexors, which decides whether the EX stage should use the values read from registers, the value from the MEM stage or the value from the WB stage.

Testing the forwarding unit is straightforward: we just remove the `nop`'s from the fibonacci program, except for those following a load, branch or jump, as these hazards are not handled yet.

As we just mentioned, we cannot handle all of the data hazards with just fowarding. This is due to the forwarding destination being the EX stage, and that the data might not be ready at that point.

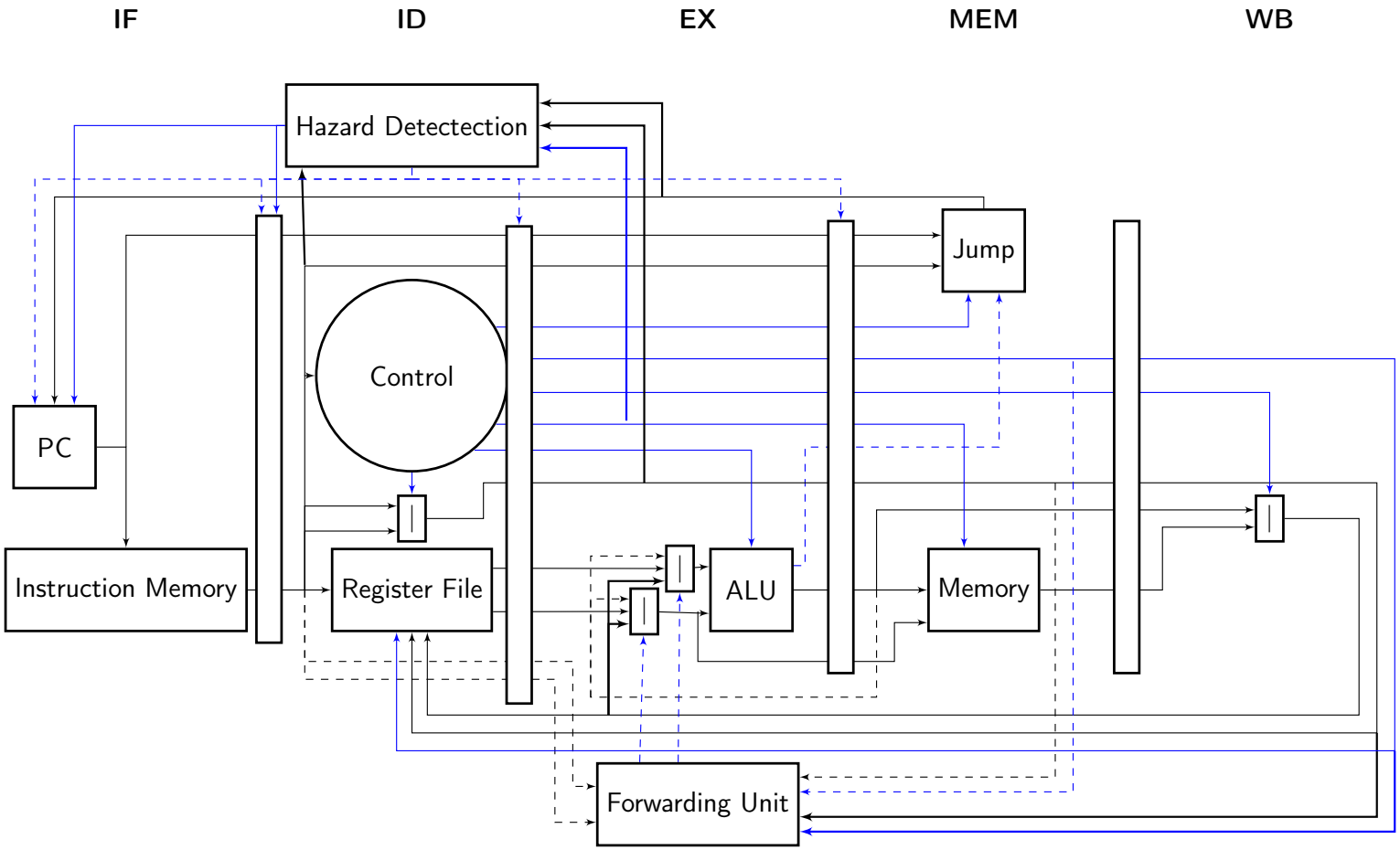
E.g. if we have a load instruction, which is followed by an instruction that uses the loaded data in its computation. In this case, the loaded data will not be ready until the load instruction has reached the WB stage.

To handle this, the processor needs to detect the hazard, and insert a bubble, and stall some of the pipes and registers.

We also need to handle an additional type of hazard: control hazards. Control hazards are when either a jump or a branch instruction has been executed.

The problem is that the branch or jump is not performed until the MEM stage, which means that all of the previous states of the pipeline may have been filled with instructions, which should not be executed.

To solve this, we should detect the hazard, and in such case flush the pipeline. Flushing is the action of resetting the registers in the pipes, so they output `nop` instructions.


IF
ID
EX
MEM
WB
Hazard Detectection
Jump
Control
PC
Instruction Memory
Register File
ALU
Memory
Forwarding Unit

The Hazard Detection Unit should be implemented in its own SME process.

To handle the data hazards, the process should read the `memread` flag from the EX stage, the destination register from the EX stage, and the two source register address from the ID stage. If the flag has been set, and the destination address match either of the two source adddresses, the ID/EX pipe should be flushed, the IF/ID and the PC register should be stalled.

Stalling is setting the registers not to update, so they will output the same data in the next clock cycle.

To solve the control hazards, the unit should look at the signal from the MEM stage, which indicates whether or not jumping should be performed.

In such a case, the Hazard Detection Unit should send a signal to flush the IF/ID, ID/EX and EX/MEM pipes. Note: the PC register should perform normally, even though it might have been instructed to stall, as the stall is now invalid.

Testing this is straightforward, as the processor should be able to handle all of our previous programs, albeit with more clock cycles.

Note: if these programs are tested against the single cycle processor, the performance will be worse, when simulating. This is due to more processes and busses, which gives the simulation more work. There should not be any performance hit, when the processor is run on hardware.