Pipelined MIPS processor

Carl-Johannes Johnsen

Department of Computer Science
University of Copenhagen

March 27, 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.

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 pipelind 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.