Chapter VI
Processor design

VI.1 Single cycle processors

There are no notes for this section as the material is fully covered by the P&H textbook in sections 5.1 – 5.4 of the 3rd edition or sections 5.1 – 5.3 of the 2nd.

VI.2 Multi-cycle processors

There are no notes for this section as the material is fully covered by the P&H textbook in section 5.5 of the 3rd edition or section 5.4 of the 2nd.

VI.3 Introduction to pipelining

Let's assume for now that all processor instructions require exactly the same number of cycles to execute and go through exactly the same processing steps. We could then speed up the execution by overlapping the execution of several instructions, so that, at any point, only one instruction is at a particular execution stage. The implementation technique which achieves this instruction overlapping is called *pipelining*.

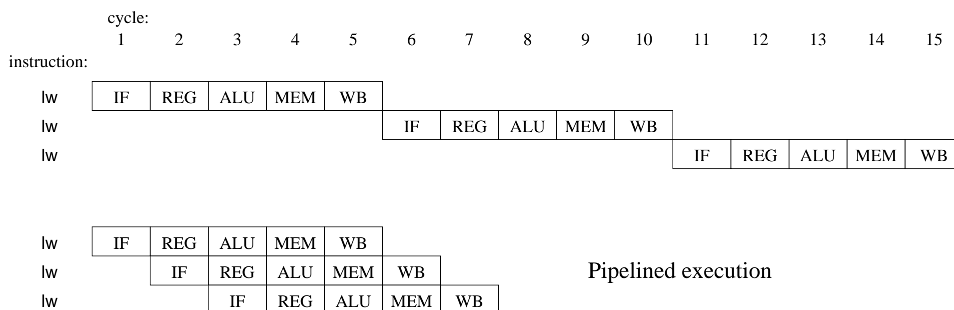

Figure VI.1: Comparison of simple multi-cycle and pipelined execution.

Figure VI.1 shows the way we commonly depict pipelined execution and how the execution time of a program is affected. It is interesting to observe that the execution of a single instruction is taking the same amount of clock cycles as before. Pipelining only speeds up sequences of instructions and, as an instruction is completed at every cycle, the Cycles Per Instruction (CPI) metric of the processor becomes equal to one. Luckily real programs execute billions

of instructions, so we can take full advantage of the pipeline speedup. Assuming that all stages take about the same amount of time, *the speed-up achieved is roughly equal to the number of stages*.

As we have already seen, different instructions require somewhat different processing steps and complete in a different number of cycles. So the important question is: can they be pipelined? The answer is positive, because we can always stretch the execution of some instruction types so that they all take the same number of cycles. These instructions do not do useful work at all stages, but this is not a problem. For example, we can force R-type arithmetic instructions to pass through a memory access stage — doing nothing — before they write their result back to the destination register.

A pipelined MIPS processor can therefore be pipelined into five, classic, stages:

IF Instruction fetch and calculation of the next PC

REG Instruction decode and register fetch

ALU ALU operation

MEM Data memory access

WB Write back the result into the register file

Pipeline hazards

The above have depicted pipelining as a panacea. In practise, the average CPI achieved by pipelining is considerably more than 1, because of complications known as *pipeline hazards*, of which there are three varieties: *structural hazards*, *data hazards* and *control hazards*.

VI.3.1 Structural hazards

Structural hazards arise when two stages of the pipeline share a hardware resource. Problems can then arise if the two different instructions in those pipeline stages both want to use that resource at the same time. For example, the register bank is used by both the REG stage and the WB stage. This is not a problem however, as the register bank has two read ports and one write port, which can all operate simultaneously. The REG stage can read two register values in the same clock cycle as the WB stage writes one register value.

Accessing the memory is more of a potential problem. The IF stage fetches instructions from the memory, while the MEM stage reads or writes data words. If the memory can only support one read or write per clock cycle, which is usually the case, the pipeline cannot operate at full efficiency. When a load or store instruction is in the MEM stage, reading or writing memory, the IF stage would

not be able to fetch an instruction, resulting in a pipeline *stall* in the IF stage. A *pipeline bubble* would pass through the pipeline as a result (see figure VI.2). Pipeline stages are doing no useful work as the bubble passes through them.

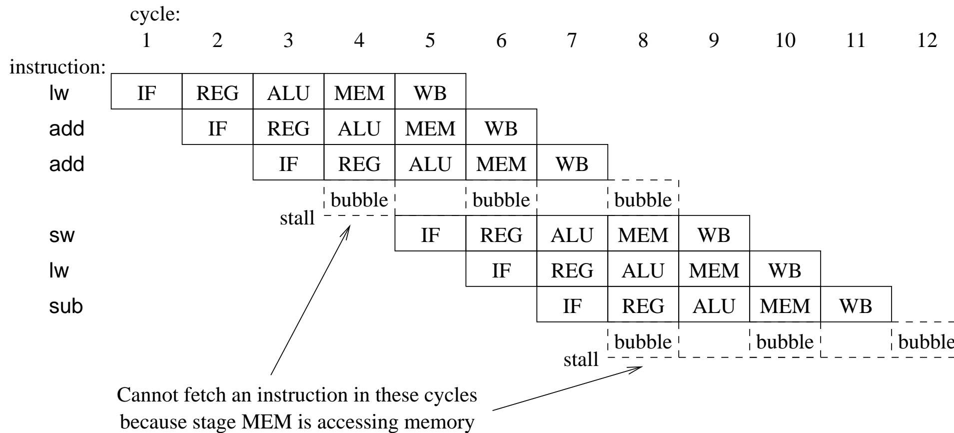

Figure VI.2: Pipeline bubbles caused by structural hazard.

There is a straightforward solution to this however: use two memories[1]. The IF pipeline stage reads from the instruction memory, and the MEM stage reads or writes from/to the data memory, and both can be accessed in the same clock cycle.

In general, structural hazards can be solved by using more hardware. The task of the architect is to determine how frequently a hazard can occur in order to justify or not the cost of adding the extra hardware.

VI.3.2 Data hazards

Data hazards arise because instructions in a program are not independent. The data written into a register by one instruction is often read from the register by a closely following instruction. There is a *data dependency* between the instructions. For example, in the following sequence of instructions:

```
lw $t0, 0($t4)
lw $t1, 4($t4)
add $t3, $t2, $t1
```

the `add` instruction reads from `$t1` the value written into it by the immediately preceding `lw` instruction. A data hazard arises in the pipeline because when the `add` instruction reaches pipeline stage REG and wants to read register `$t1`, the immediately preceding `lw` instruction has just reached pipeline stage ALU,

[1]In reality two memory *caches* are used. Caches are described in later lectures.

therefore it has not yet read the word from the memory. In order for the program to execute correctly, the `add` instruction must stall in stage REG of the pipeline for three clock cycles, while the `lw` instruction works through stages ALU, MEM and WB. For those three clock cycles, no more instructions can be fetched and the result is a three stage bubble in the pipeline (fig VI.3).

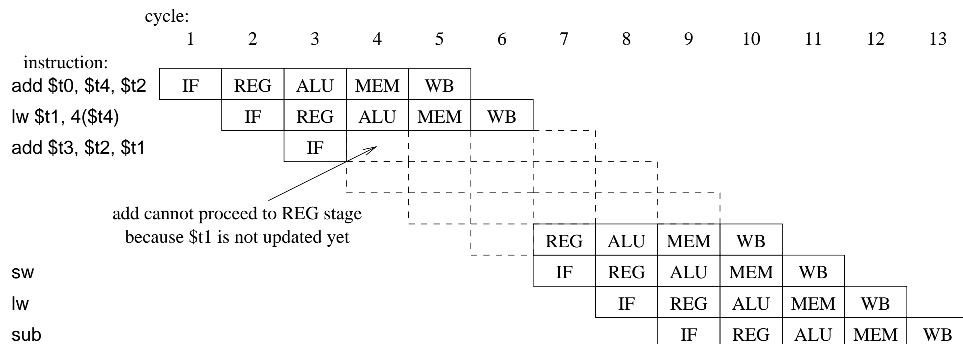

Figure VI.3: Data hazard.

The pipelined processor needs to incorporate special control logic to detect data dependencies between instructions and stall the pipeline for the necessary number of cycles.

Data hazards are very difficult to avoid. The compiler can attempt to order instructions so that instructions that read registers are placed as far as possible after the ones that load up those registers (as far as is possible without changing the effect of the program, of course). In the example above, if the two first instructions were swapped round, the pipeline would only need to stall for two clock cycles, because the `lw` instruction that loads register `$t1` would already be arriving in stage MEM when the `add` instruction arrives in stage REG. Because programs do so many operations on data, there are a lot of data dependencies between close instructions, and this reordering by the compiler is not particularly effective at reducing pipeline stalls.

There are ways of reducing the length of pipeline stalls caused by data dependencies between instructions, by adding more complex hardware, but they are beyond the scope of the Inf2C course.

VI.3.3 Control hazards

Control hazards are caused by branch instructions. A conditional branch instruction, `beq $t1, $t2, offset` (which compares registers `$t1` and `$t2`, and if they are equal adds `offset` to the PC to do a branch), would do the following in each pipeline stage:

pipeline stage	**operation**
IF	Fetch the instruction and increment PC by 4
REG	Get the values of `$t1` and `$t2` from the registers
ALU	Calculate `$t1-$t2` and also `PC+offset`
MEM	If the result of `$t1-$t2` was zero, update the PC with the result of `PC+offset`
WB	Do nothing (no result to write back)

(Note that in stage ALU, two calculations are done at once, so that stage of the datapath pipeline will need to contain two ALUs. Also, the operation in stage MEM here does not involve an access to memory.) The control hazard arises because the `beq` instruction does not update the program counter, if the branch is taken, until the instruction is in stage MEM. Normally, by the time an instruction reaches stage MEM, three subsequent instructions have been fetched, and are in stages ALU, REG and IF. But the processor must stop fetching instructions as soon as it fetches a conditional branch instruction into IF, because it does not know where to fetch the next instruction from — it does not yet know whether the branch will be taken or not. Instruction fetching must stall until the branch instruction completes stage MEM, at which point the PC is updated if the branch is taken. The result is a three stage bubble in the pipeline after each conditional branch instruction (fig VI.4).

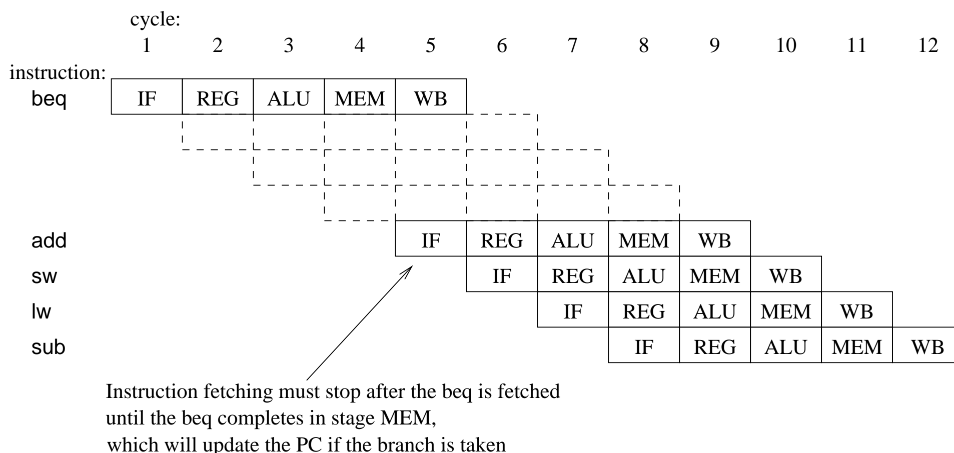

Figure VI.4: Control hazard.

Again, there are ways of reducing the performance penalty caused by branches, by adding more complex hardware. The basic idea here is for the IF hardware to *predict* whether the branch will be taken or not, and then to carry on fetching instructions from the predicted place. If the prediction turns out to be right, no bubble occurs in the pipeline. If the prediction turns out to be wrong, the wrongly fetched instructions already in the pipeline must be

discarded and new instructions fetched, causing a pipeline bubble, and a corresponding performance penalty.

VI.3.4 Summary

The effect of data and control hazards is that the average CPI for a pipelined processor is considerably greater than 1, even with complex extra hardware to minimise the number and length of stalls. There are ways of reducing the average CPI yet further, increasing processor performance, by adding more hardware. One is to have the IF stage fetch more than one instruction per clock cycle, and then feed these into several execution pipelines operating in parallel (this arrangement is known as a *superscalar* processor design). Another is to allow the hardware to *reorder* instructions, i.e. to execute the instructions in a different order to the order in which they were fetched. These methods are covered in more detail in the CS3 Computer Architecture course.