Processor Instructions Sets and Pipelining

1 Architecture Dependent Instruction Sets

Different processor architectures often each have specific instruction sets. An instruction set provides functions to the programmer whilst allowing efficient implementation on the chip. Programmers do not usually want to program in machine code and use higher level languages which have a higher level of abstraction. The semantic gap between high-level language and machine instructions is bridged by a compiler. The compiler translates the high-level language program into a sequence of machine instructions. The instruction set needs to be a good compiler target rather than human readable/useable.

There are two general categories for processor instruction sets:

A large instruction set that performs complex sequences of operations.

This leads to the CISC (complex instruction set computer):

- Only one instruction fetch per instruction → fewer instruction fetches in a program.
- Each instruction could last for many clock cycles.
- Can lead to complex chips which are limited by the need for large Read Only Memory on chip to hold the instruction set (a by-product of older minicomputers).

A small instruction set that performs a reduced number of instructions.

This leads to the RISC (reduced instruction set computer):

- Requires more instructions to construct a program.
- Instructions chosen to fit in a pipelined processor.
- Can lead to the use of faster processor speeds with smaller on chip memory
- When first introduced in the 1980's RISC's substantially outperformed current CISC's.

2 Pipelining

The aim of this section is to introduce the concept of pipelining. How pipelining is implemented, pipelining hazards, the effects of pipelining complexity and the effects of pipelining on CPU performance will be covered.

2.1 Introduction

A common misconception is that computers spend most of their time computing. Actually, they spend a significant amount of time locating data items and program routines and then

moving them around. An example of the time spent on different tasks running a print preview program on an ARM emulator[1] is shown below.

Instruction type	Dynamic usage
Data movement	43%
Control flow	23%
Arithmetic operations	15%
Comparisons	13%
Logical operations	5%
Other	1%

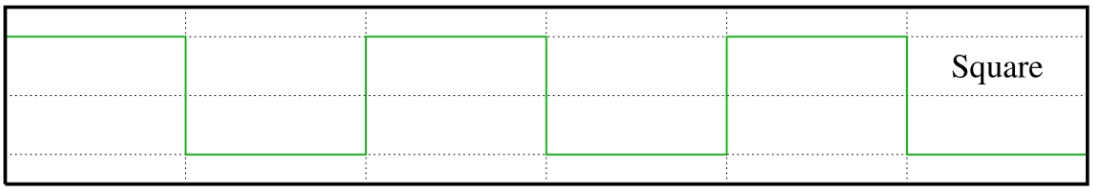
To improve the efficiency of a single processor a technique called pipelining was developed (1985). Pipelining allows several instructions to be executed in parallel by a single processor. The speed-up over an un-piped machine equals the number of pipe stages (this is the ideal case).

How is it implemented?

An individual instruction is executed as a series of steps by a processor. Each step is independent and uses a different part of the processor, i.e. a processor is built with several different circuits, each specifically designed to perform one of the steps in the pipeline. Once each stage is complete the next instruction can use that part of the processor. In other words, the instructions, once broken down, can be overlapped. The process of starting the next instruction before the last one has finished is called pipelining. The process of completing each stage and moving to the next stage is determined by the processor clock rate.

2.2 Clock cycle

The clock rate is the fundamental rate at which a processor performs its operations. Clock rate is the term use to define the "speed" of the processor – frequency of the clock cycle. A clock signal oscillates between a high and low state (a square wave).



[1] Furber, S., ARM: system-on-chip architecture, 2nd Edition, Addison-Wesley, London, 2000.

This signal is used to synchronise the actions of more than one circuit – they all pick up and sync to the up (or down) step in the clock signal. All pipeline stages advance at the same time (determined by the fundamental processor cycle) which is ideally 1 clock cycle.

The STM32F407 processor can be set to run with a fundamental system clock of up to 168MHz. It then provides several other clocks (AHB and APB) that can be configured to run at the different speeds required by the peripheral functions, such as the SPI bus and ADC.

2.3 Pipeline steps

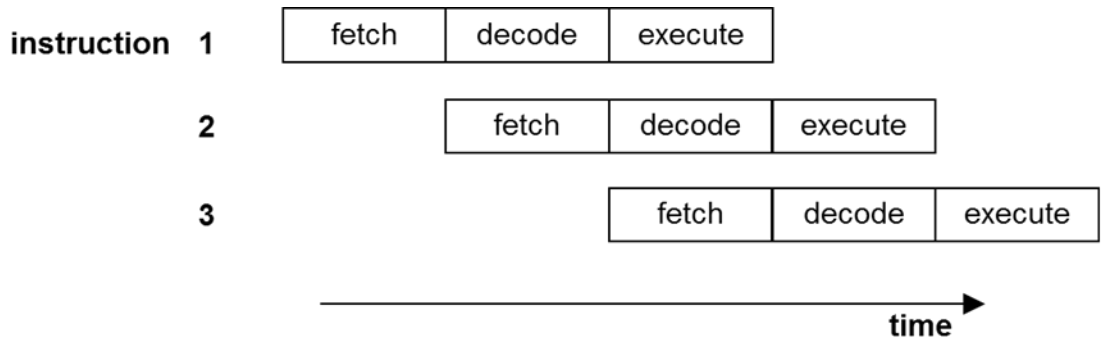
A typical instruction sequence might be:

1. Fetch the instruction from memory (fetch)
2. Decode it to see what sort of instruction it is (decode)
3. Access any operands that may be required from the register bank (register)
4. Combine the operands to form the result or a memory address (execute/ALU)
5. Access memory for a data operand if necessary (memory)

This is a six step pipeline. Not all steps will be used for every instruction, but the majority of instructions will require most of these steps. As an instruction completes step 1 and moves on to step 2, the next instruction begins step 1. This is completely invisible to a programmer!

2.3.1 ARM pipeline

ARM (pre-1995, up to ARM7) uses a 3 stage pipeline consisting of fetch, decode and execute stages. A similar 3-stage pipeline is used in the STM32F407. The execution of the pipelined instructions can be viewed in the following manner.



Each stage of the pipeline requires one clock cycle for a typical instruction. A consequence of pipelined behaviour is the program counter must run ahead of the current instruction.

In this case a full instruction executes over three clock cycles, a time known as the latency of the instruction execution (time from start to finish of an instruction). A three stage pipeline should, in theory, deliver a three times speed up compared with non-overlapped instruction execution for a typical program. In practice however, there are cases where the pipeline does not flow so smoothly. These are called pipeline hazards. They reduce the achievable computational speed by causing data stalls.

2.4 Pipeline hazards

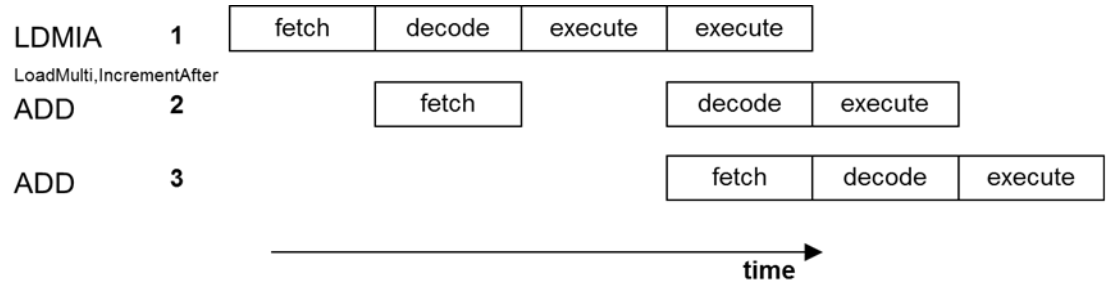
Pipelines work best when their contents flow smoothly. However particular instructions and stages can disrupt the flow:

- Multi-cycle instruction stages require more than one clock cycle to execute.
- Read-after-write instructions require the result of the previous instruction to execute a subsequent instruction. This requires waiting for the first instruction to complete.
- Branch behaviour is even worse as it delays not the execution step but the fetch step of a subsequent instruction.

2.4.1 Multi-cycle instructions

Examples of multi-cycle instructions are multiple load, data store and all floating point arithmetic instructions.

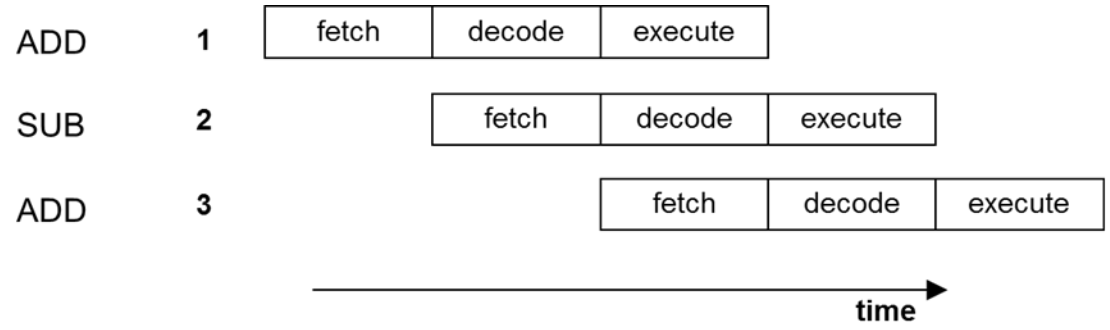
A series of additions interspersed with a multiple load is shown below:



Since there are two registers to load then the instruction stays in the execution cycle for two clock cycles.

2.4.2 Read-after-Write

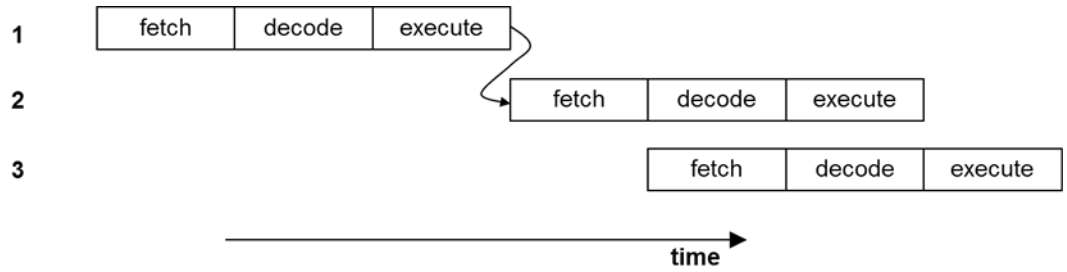
A read-after-write pipeline hazard occurs when the result of an instruction is required in the next instruction. It is a very common hazard and stalls the pipeline until the result is available – **note that a 3 stage pipeline has NO data stall due to a read-after-write (see below).**



Imagine, for example, that the result of the first ADD above is required to be subtracted (SUB) from another variable in instruction 2.

2.4.3 Branch behaviour

A worse hazard is branch behaviour, where the stall affects the fetch stage of the subsequent instruction. Consider the result of a compare instruction which decides the ensuing sequence of instructions (e.g. an 'if x is less than y' statement in C)

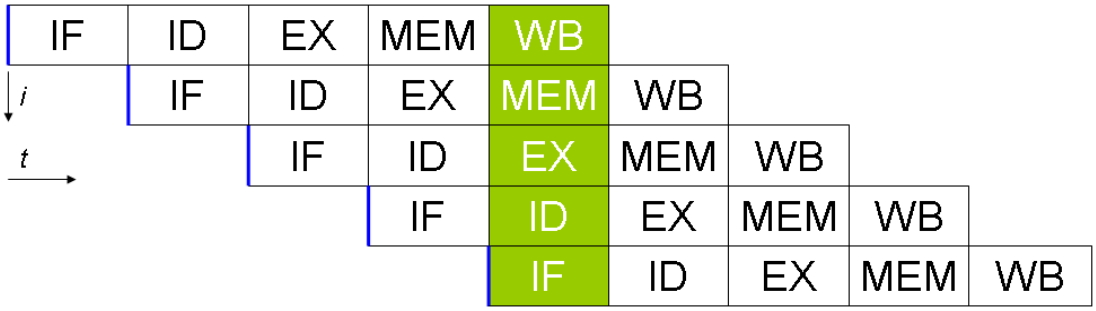


Branch instructions can cause a complete pipeline stall (commonly called the branch penalty). However, one possible solution is to introduce a delayed branch. This requires similar instructions which are to be completed on all possible branches, to be placed in the pipeline. These similar instructions are processed regardless of the outcome. If there are not enough instructions to fill the stall window then the delay is not resolved. Other solutions include *branch prediction*.

2.5 Pipeline complexity

We have used a 3 stage ARM pipeline to illustrate ideas so far. But pipelines may be designed with more stages so as to reduce the work performed during each stage. This increase in the number of stages and allows a higher clock frequency to be used because less work is done in each stage. Some high performance ARM processors use "the classic RISC 5 stage pipeline", breaking the execute stage into three separate stages (execute, memory, write-back). As a result more stages allow an increased clock frequency, but increase the chance of data stalls (as well as adding extra complexity into the processor design).

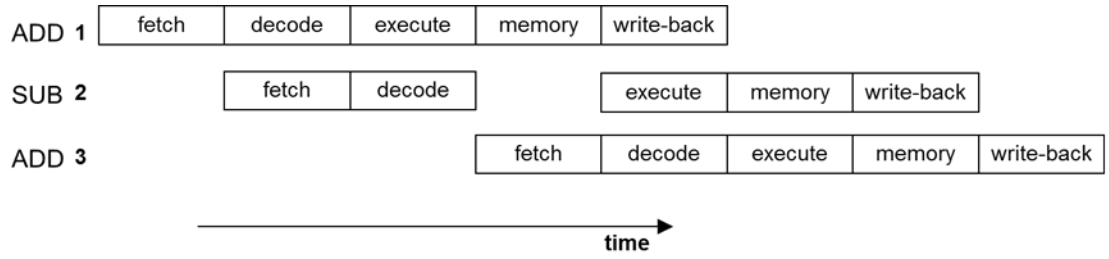
The classic RISC pipeline is shown below:



In the pipeline above:

- IF is instruction *fetch*
- ID is instruction *decode*
- EX is *execute*
- MEM is *memory access*
- WB is *write-back*

With the 5 stage pipeline the read-after-write data stall hazard becomes an issue. This is illustrated below, where again the result of the first ADD is required to be subtracted (SUB) from another variable.



2.5.1 Program execution time

The following is a list of factors which determine the execution time of a program:

- The clock cycle time (frequency/clock rate)
- The number of pipeline stages
- The complexity of the instructions (number of clock cycles per instruction execution)
- The number of instructions in the program
- The design of the program (data stalls)
- Memory access speeds – can variables be provided fast enough for the processor?