Simple Example — Immediate Addressing

Consider the computation $(1 + 2) * 3$.

1. Write an assembly program for the above computation and store the result in register r1. You can assume that there are an unlimited number of regsiters available.
2. Describe what happens in the pipeline stages when various types of instructions (data-movement, data-processing) are processed on a five-stage pipeline: fetch IF, decode ID, register read RR, execute EX and write back WB.
3. Draw a diagram showing the in-order execution of the code and identify the delay slots.

Simple Example — Memory Addressing

Recall that modern processors use (several levels of) caches so that instructions and data can be loaded into the CPU faster than from main memory. In this exercise we will see the benefit of caches when a simple code is executed on a pipelined processor. In particular, we will assume that there are separate, onboard caches for data and instructions, and see that this helps getting rid of some delay slots (vacant pipeline stages during the execution of the code).

Consider the computation

$$(M1 + M2) * (M3 + M4)$$

where $M1, \ldots$ denote direct (memory) addressing.

1. Write an assembly program typical of RISC machines for this computation. First load the data and then process the arithmetic operations. You can assume that there are an unlimited number of registers available and store the result in register $r1$.

2. Show the execution of your program from item 1 on a pipelined processor. There are five pipeline stages: IF, ID, RR, EX, WB. Assume that instructions and data have to be loaded from main memory. Explain what happens during the pipeline stages for the various instructions and whether certain instructions can skip some of the pipeline stages.

3. Repeat the previous item, but this time assume that instructions and data are fetched from onboard instruction and data caches, respectively, thus there is no resource conflict on the bus.

Example — Immediate Addressing

Consider the computation $(((10 * 8) + 4) - 7)^2$.

1. Write an assembly program for the above computation by using a minimal number of registers.

2. Draw a diagram showing the in-order execution of the code on a five-stage pipeline and identify the delay slots. Assume that when an instruction cannot be further processed (e.g., because of some dependency or resource conflict), then it stalls the pipeline.

3. Now assume that there is an *instruction window* IW, where fetched and decoded instructions can be stored before further processing (so that the pipeline does not have to stall). In addition, allow an instructions to "jump the queue" (out-of-order execution), provided that this does not alter the outcome of the computation. Draw a diagram showing the execution of the code and identify the delay slots.

Example — Memory Addressing

Consider the computation $((M1 * M2) + M3) - M4$ where each Mi indicates the content of a memory location.

1. Write an assembly program for the above computation using minimum number of registers.
2. Show the delay slots when the code is executed on a five-stage pipeline. Assume that instructions and data have to be loaded from main memory. Assume that there is an instruction window IW, where fetched and decoded instructions can be stored until they can be further processed (out-of-order if possible).
3. Repeat the previous item, but now assume that instructions and data are loaded from onboard caches.

Comprehensive Example

Consider the computation $(M1 * M2) + (M3 * M4)$.

1. Write a program in assembly language that performs the above computation. Try to use a minimal number of registers.
2. Draw a diagram showing the execution of your program on a five-stage pipeline using in-order execution.
3. Draw a diagram showing the execution of your program on a five-stage pipeline using out-of-order execution.
4. Identify the dependencies in your code.
5. Remove as many dependencies as possible from your code (by using register renaming) and reorder the code to minimize the number of delay slots when executed on a five-stage pipeline. Show the pipeline activity in a diagram.

You can assume that there is an onboard instruction cache from which the instructions are fetched (so there is no resource conflict between fetching an instruction and executing a data-transfer instruction) and that there is an instruction window, where fetched and decoded instructions are stored.