2 The Basics

This chapter reviews features that are found in all modern microprocessors: (i) instruction pipelining and (ii) a main memory hierarchy with caches, including the virtual-to-physical memory translation. It does not dwell on many details – that is what subsequent chapters will do. It provides solely a basis on which we can build later on.

2.1 Pipelining

Consider the steps required to execute an arithmetic instruction in the von Neumann machine model, namely:

1. Fetch the (next) instruction (the one at the address given by the program counter).
2. Decode it.
3. Execute it.
4. Store the result and increment the program counter.

In the case of a load or a store instruction, step 3 becomes two steps: calculate a memory address, and activate the memory for a read or for a write. In the latter case, no subsequent storing is needed. In the case of a branch, step 3 sets the program counter to point to the next instruction, and step 4 is voided.

Early on in the design of processors, it was recognized that complete sequentiality between the executions of instructions was often too restrictive and that parallel execution was possible. One of the first forms of parallelism that was investigated was the overlap of the mentioned steps between consecutive instructions. This led to what is now called *pipelining*.[1]

[1] In early computer architecture texts, the terms *overlap* and *look-ahead* were often used instead of *pipelining*, which was used for the pipelining of functional units (cf. Section 2.1.6).

2.1.1 The Pipelining Process

In concept, pipelining is similar to an assembly line process. Jobs A, B, and so on, are split into n sequential subjobs $A_1, A_2, \ldots, A_n$ ($B_1, B_2, \ldots, B_n$, etc.) with each A_i (B_i, etc.) taking approximately the same amount of processing time. Each subjob is processed by a different station, or equivalently the job passes through a series of *stages*, where each stage processes a different A_i. Subjobs of different jobs overlap in their execution: when subjob A_1 of job A is finished in stage 1, subjob A_2 will start executing in stage 2 while subjob B_1 of job B will start executing in stage 1. If t_i is the time to process A_i and $t_M = \max_i t_i$, then in steady state one job completes every t_M. Throughput (the number of instructions executed per unit time) is therefore enhanced. On the other hand, the latency (total execution time) of a given job, say L_A for A, becomes

$$L_A = nt_M, \quad \text{which may be greater than } \sum_{i=1}^{n} t_i.$$

Before applying the pipelining concept to the instruction execution cycle, let us look at a real-life situation. Although there won't be a complete correspondence between this example and pipelining as implemented in contemporary processors, it will allow us to see the advantages and some potential difficulties brought forth by pipelining.

Assume that you have had some friends over for dinner and now it's time to clean up. The first solution would be for you to do all the work: bringing the dishes to the sink, scraping them, washing them, drying them, and putting them back where they belong. Each of these five steps takes the same order of magnitude of time, say 30 seconds, but bringing the dishes is slightly faster (20 seconds) and storing them slightly slower (40 seconds). The time to clean one dish (the latency) is therefore 150 seconds. If there are 4 dishes per guest and 8 guests, the total cleanup time is $150 \times 4 \times 8 = 4800$ seconds, or 1 hour and 20 minutes. The throughput is 1 dish per 150 seconds. Now, if you enlist four of your guests to help you and, among the five of you, you distribute the tasks so that one person brings the dishes, one by one, to the second, who scrapes them and who in turn passes them, still one by one, to the washer, and so on to the dryer and finally to the person who stores them (you, because you know where they belong). Now the latency is that of the longest stage (storing) multiplied by the number of stages, or $40 \times 5 = 200$ seconds. However, the throughput is 1 dish per longest stage time, or 1 dish per 40 seconds. The total execution time is $40 \times 8 \times 4 + 4 \times 40 = 1360$ seconds, or a little less than 23 minutes. This is an appreciable savings of time.

Without stretching the analogy too far, this example highlights the following points about pipelining:

- In order to be effective, the pipeline must be *balanced*, that is, all stages must take approximately the same time. It makes no sense to optimize a stage whose processing time is not the longest. For example, if drying took 25 seconds instead of 30, this would not change the overall cleanup time.

- A job must pass through all stages, and the order is the same for all jobs. Even if a dish does not need scraping, it must pass through that stage (one of your friends will be idle during that time).
- *Buffering* (holding a dish in its current partially processed state) between stages is required, because not all stages take exactly the same time.
- Each stage must have all the resources it needs allocated to it. For example, if you have only one brush and it is needed some of the time, by both the scraper and the washer there will be some stalling in the pipeline. This situation is a form of (structural) *hazard.*
- The pipeline may be disrupted by some internal event (e.g., someone drops a dish and it breaks) or some external event (one of your coworkers is called on the phone and has to leave her station). In these cases of *exception* or *interrupt*, the pipeline must be flushed and the state of the process must be saved so that when the exception or interrupt is removed, the operation can start anew in a consistent state.

Other forms of hazards exist in processor pipelines that do not fit well in our example: they are *data hazards* due to dependencies between instructions and *control hazards* caused by transfers of control (branches, function calls). We shall return to these shortly.

2.1.2 A Basic Five-stage Instruction Execution Pipeline

Our presentation of the instruction execution pipeline will use a RISC processor as the underlying execution engine. The processor in question consists of a set of registers (the register file), a program counter PC, a (pipelined) CPU, an instruction cache (I-cache), and a data cache (D-cache). The caches are backed up by a memory hierarchy, but in this section we shall assume that the caches are perfect (i.e., there will be no cache misses). For our purposes in this section, the state of a running process is the contents of the registers and of the program counter PC. Each register and the PC are 32 bits long (4 bytes, or 1 word).

Recall that a RISC processor is a load–store architecture, that is, all instructions except those involving access to memory have register or immediate operands. The instructions are of same length (4 bytes in our case) and can be of one of three types:

- Arithmetic–logical instructions, of the form $R_i \leftarrow R_j op R_k$ (one of the source registers R_j or R_k can be replaced by an immediate constant encoded in the instruction itself).
- Load–store instructions, of the form $R_i \leftarrow Mem[R_j + disp]$ or $Mem[R_j + disp] \leftarrow R_i$.
- Control instructions such as (conditional) branches of the form $br\ (R_j op R_k)$ *displ*, where a taken branch (R_j *op* R_k is true) sets the PC to its current value plus the *displ* rather than having the PC point to the next sequential instruction.

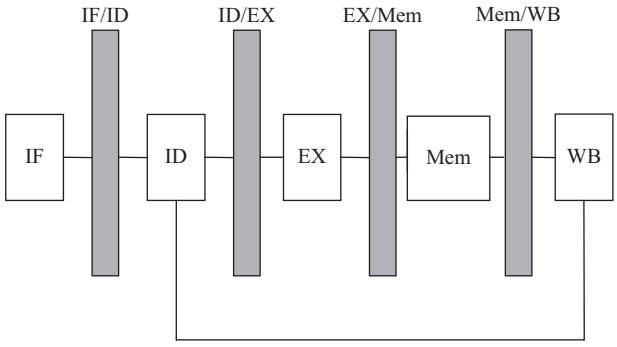

Figure 2.1. Highly abstracted pipeline.

As we saw at the beginning of this chapter, the instruction that requires the most steps is a load instruction, which requires five steps. Each step is executed on a different stage, namely:

1. Instruction fetch (IF). The instruction is fetched from the memory (I-cache) at the address indicated by the PC. At this point we assume that the instruction is not a branch, and we can increment the PC so that it will point to the next instruction in sequence.
2. Instruction decode (ID). The instruction is decoded, and its type is recognized. Some other tasks, such as the extension of immediate constants into 32 bits, are performed. More details are given in the following.
3. Execution (EX). In the case of an arithmetic instruction, an ALU performs the arithmetic or logical operation. In the case of a load or store instruction, the address $addr = R_j + disp$ is computed (*disp* will have been extended to 32 bits in the ID stage). In the case of a branch, the PC will be set to its correct value for the next instruction (and other actions might be taken, as will be seen in Section 2.1.4).
4. Memory access (Mem). In the case of a load, the contents of $Mem[addr]$ are fetched (from the D-cache). If the instruction is a store, the contents of that location are modified. If the instruction is neither a load nor a store, nothing happens during that stage, but the instruction, unless it is a branch, must pass through it.
5. Writeback (WB). If the instruction is neither a branch nor a store, the result of the operation (or of the load) is stored in the result register.

In a highly abstracted way, the pipeline looks like Figure 2.1. In between each stage and the next are the pipeline registers, which are named after the left and right stages that they separate. A pipeline register stores all the information needed for completion of the execution of the instruction after it has passed through the stage at its left.

If we assume that accessing the caches takes slightly longer than the operations in the three other stages, that is, that the cache access takes 1 cycle, then a snapshot

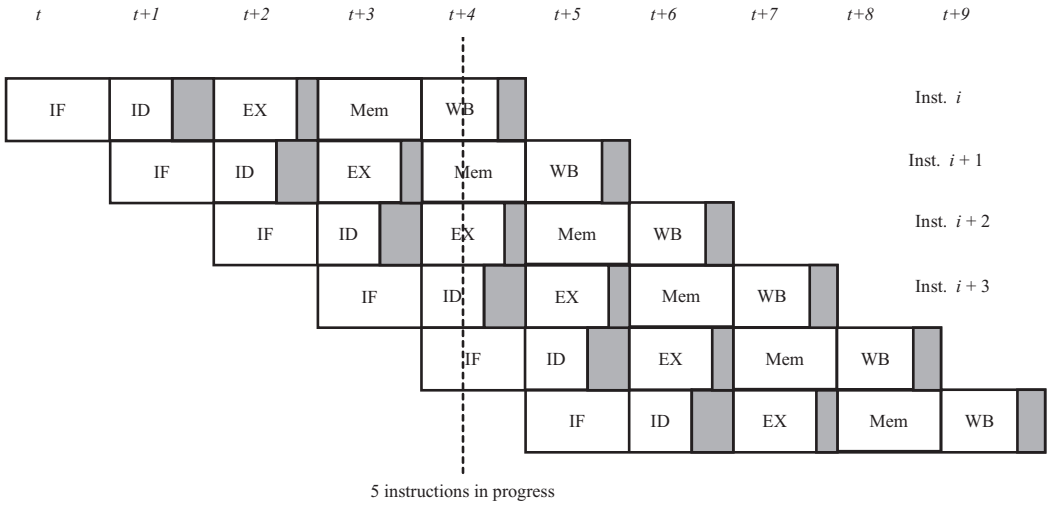

Figure 2.2. Snapshot of sequential program execution.

of the execution of a sequential program (no branches) is shown in Figure 2.2 (the shaded parts indicate that some stages are shorter than the IF and Mem stages).

As can be seen, as soon as the pipeline is full (time $t+4$), five instructions are executing concurrently. A consequence of this parallelism is that resources cannot be shared between stages. For example, we could not have a single cache unified for instruction and data, because instruction fetches, that is, accesses to the I-cache during the IF stage, occur every cycle and would therefore interfere with a load or a store (i.e., with access to the D-cache during the Mem stage). This interference would be present about 25% of the time (the average frequency of load–store operations). As mentioned earlier, some stage might be idle. For example, if instruction $i+1$ were an add, then at time $t+4$ the only action in the Mem stage would be to pass the result computed at time $t+3$ from the pipeline register EX/Mem where it was stored to the pipeline register Mem/WB.

It is not the intention to give a detailed description of the implementation of the pipeline. In order to do so, one would need to define more precisely the ISA of the target architecture. However, we shall briefly consider the resources needed for each stage and indicate what needs to be stored in the respective pipeline registers. In this section, we only look at arithmetic–logical and load–store instructions. We will look at control instructions in Section 2.1.4.

The first two stages, fetch (IF) and decode (ID), are common to all instructions. In the IF stage, the next instruction, whose address is in the PC, is fetched, and the PC is incremented to point to the next instruction. Both the instruction and the incremented PC are stored in the IF/ID register. The required resources are the I-cache and an adder (or counter) to increment the PC. In the ID stage, the opcode of the instruction found in the IF/ID register is sent to the unit that controls the settings of the various control lines that will activate selected circuits or registers and cache read or write in the subsequent three stages. With the presence of this control unit (implemented, for example, as a programmable logic array (PLA)),

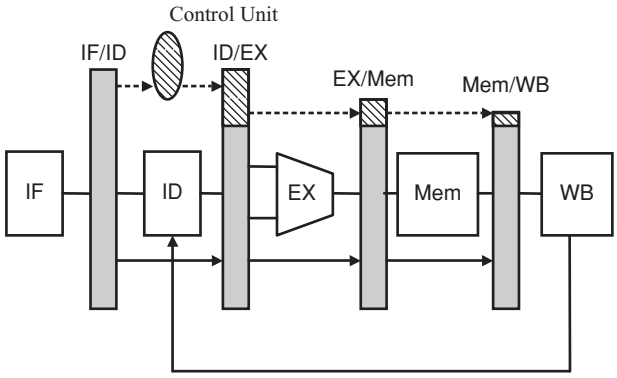

Figure 2.3. Abstracted view of the pipeline with the control unit.

the abstracted view of the pipeline becomes that of Figure 2.3. In addition to the opcode decoding performed by the control unit, all possible data required in the forthcoming stages, whether the instruction is an arithmetic–logical one or a load–store, are stored in the ID/EX register. This includes the contents of source registers, the extension to 32 bits of immediate constants and displacements (a sign extender is needed), the name of the potential result register, and the setting of control lines. The PC is also passed along.

After the IF and ID stages, the actions in the three remaining stages depend on the instruction type. In the EX stage, either an arithmetic result is computed with sources selected via settings of adequate control lines, or an address is computed. The main resource is therefore an ALU, and an ALU symbol for that stage is introduced in Figure 2.3. For both types of instruction the results are stored in the EX/Mem register. The EX/Mem register will also receive from the ID/EX register the name of the result register, the contents of a result register in the case of a store instruction, and the settings of control lines for the two remaining stages. Although passing the PC seems to be unnecessary, it is nonetheless stored in the pipeline registers. The reason for this will become clear when we deal with exceptions (Section 2.1.4). In the Mem stage, either a read (to the D-cache) is performed if the instruction is a load, or a write is performed if the instruction is a store, or else nothing is done. The last pipeline register, Mem/WB, will receive the result of an arithmetic operation, passed directly from the EX/Mem register, or the contents of the D-cache access, or nothing (in the case of a store), and, again, the value of the PC. In the WB stage the result of the instruction, if any, is stored in the result register. It is important to note that the contents of the register file are modified only in the last stage of the pipeline.

2.1.3 Data Hazards and Forwarding

In Figure 2.2, the pipeline is ideal. A result is generated every cycle. However, such smooth operation cannot be sustained forever. Even in the absence of exceptional conditions, three forms of hazards can disrupt the functioning of the pipeline. They are: