Notes-01

Informal: Hello students. This is your first “class” on Computer Architecture. You need some introductory material; I will provide it later.

Computer Architecture is fascinating. We will see how speed can be improved. There are obvious hardware limitations. The basic strategy is to do many things simultaneously, i.e. in parallel. Intuitively, if two processors share the workload, we should expect the work to be done twice as fast. This is too optimistic but yet it **is** the approach.

Pipelining

Consider a hypothetical processor that executes every instruction in 2 clock cycles. We are accustomed to the execution model in which an instruction begins execution when its preceding instruction has completed. Pipelining takes the approach of **overlapped execution** of consecutive instructions, i.e. an instruction is allowed to begin execution **before** its preceding instruction has completed. How is it possible ? Suppose an instruction has two stages s1 and s2, performed one after another. The two stages are **independent**. Then if we have two consecutive instructions i1 and i2, they can be executed as follows:

	Cycle-1	Cycle-2	Cycle-3
i1	s1	s2	
i2		s1	s2

Figure 1.1: Overlapped execution of i1 and i2

We see that the sequence of two instructions gets completed in 3 cycles. With the traditional (non-pipelined) processor, it would require 2+2 = 4 cycles. So the average execution time is 1.5 cycles.

Things improve as more instructions are considered. If we have 3 consecutive instructions i1, i2, and i3, the timing diagram becomes:

	Cycle-1	Cycle-2	Cycle-3	Cycle-4
i1	s1	s2		
i2		s1	s2	
i3			s1	s2

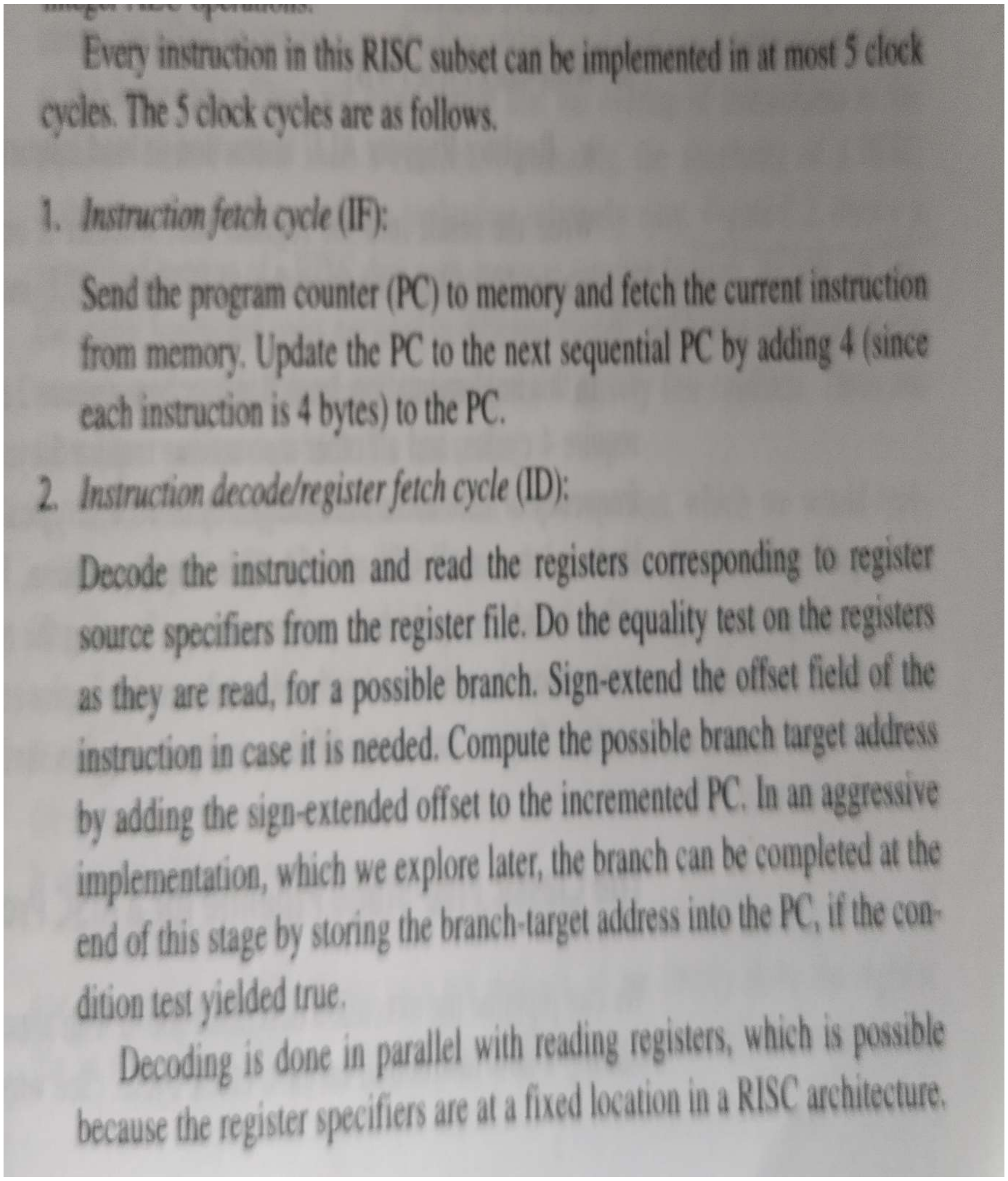
Figure 1.2: Overlapped execution of i1, i2 and i3

This time, 3 instructions are completed in 4 cycles which implies an average execution time of 1.33 cycles !! This is definitely an improvement over 2 cycles.

Once the pipeline is filled up, i.e. every stage is occupied in every cycle, **an instruction is completed in every cycle**. Thus, i1 completes in cycle-2, i2 in cycle-3, and i3 in cycle-4. Note that an individual instruction still requires 2 cycles to complete; however, since instructions are overlapped in time, the total time required to execute a sequence of instructions is significantly reduced ! In other words, pipelining increases throughput.

A 5-stage instruction pipeline

Now that we have understood the basic principle, let us consider a practical example. We will be considering a processor which executes an instruction in 5 cycles. The stages in the execution are as follows:



Every instruction in this RISC subset can be implemented in at most 5 clock cycles. The 5 clock cycles are as follows.

1. *Instruction fetch cycle* (IF):

   Send the program counter (PC) to memory and fetch the current instruction from memory. Update the PC to the next sequential PC by adding 4 (since each instruction is 4 bytes) to the PC.

2. *Instruction decode/register fetch cycle* (ID):

   Decode the instruction and read the registers corresponding to register source specifiers from the register file. Do the equality test on the registers as they are read, for a possible branch. Sign-extend the offset field of the instruction in case it is needed. Compute the possible branch target address by adding the sign-extended offset to the incremented PC. In an aggressive implementation, which we explore later, the branch can be completed at the end of this stage by storing the branch-target address into the PC, if the condition test yielded true.

   Decoding is done in parallel with reading registers, which is possible because the register specifiers are at a fixed location in a RISC architecture.

Continuing,

This technique is known as *fixed-field decoding*. Note that we may read a register we don't use, which doesn't help but also doesn't hurt performance. (It does waste energy to read an unneeded register, and power-sensitive designs might avoid this.) Because the immediate portion of an instruction is also located in an identical place, the sign-extended immediate is also calculated during this cycle in case it is needed.

3. *Execution/effective address cycle* (EX):

   The ALU operates on the operands prepared in the prior cycle, performing one of three functions depending on the instruction type.

   - Memory reference—The ALU adds the base register and the offset to form the effective address.
   - Register-Register ALU instruction—The ALU performs the operation specified by the ALU opcode on the values read from the register file.
   - Register-Immediate ALU instruction—The ALU performs the operation specified by the ALU opcode on the first value read from the register file and the sign-extended immediate.

   In a load-store architecture the effective address and execution cycles can be combined into a single clock cycle, since no instruction needs to simultaneously calculate a data address and perform an operation on the data.

4. *Memory access* (MEM):

   If the instruction is a load, the memory does a read using the effective address computed in the previous cycle. If it is a store, then the memory writes the data from the second register read from the register file using the effective address.

5. *Write-back cycle* (WB):

   - Register-Register ALU instruction or load instruction:

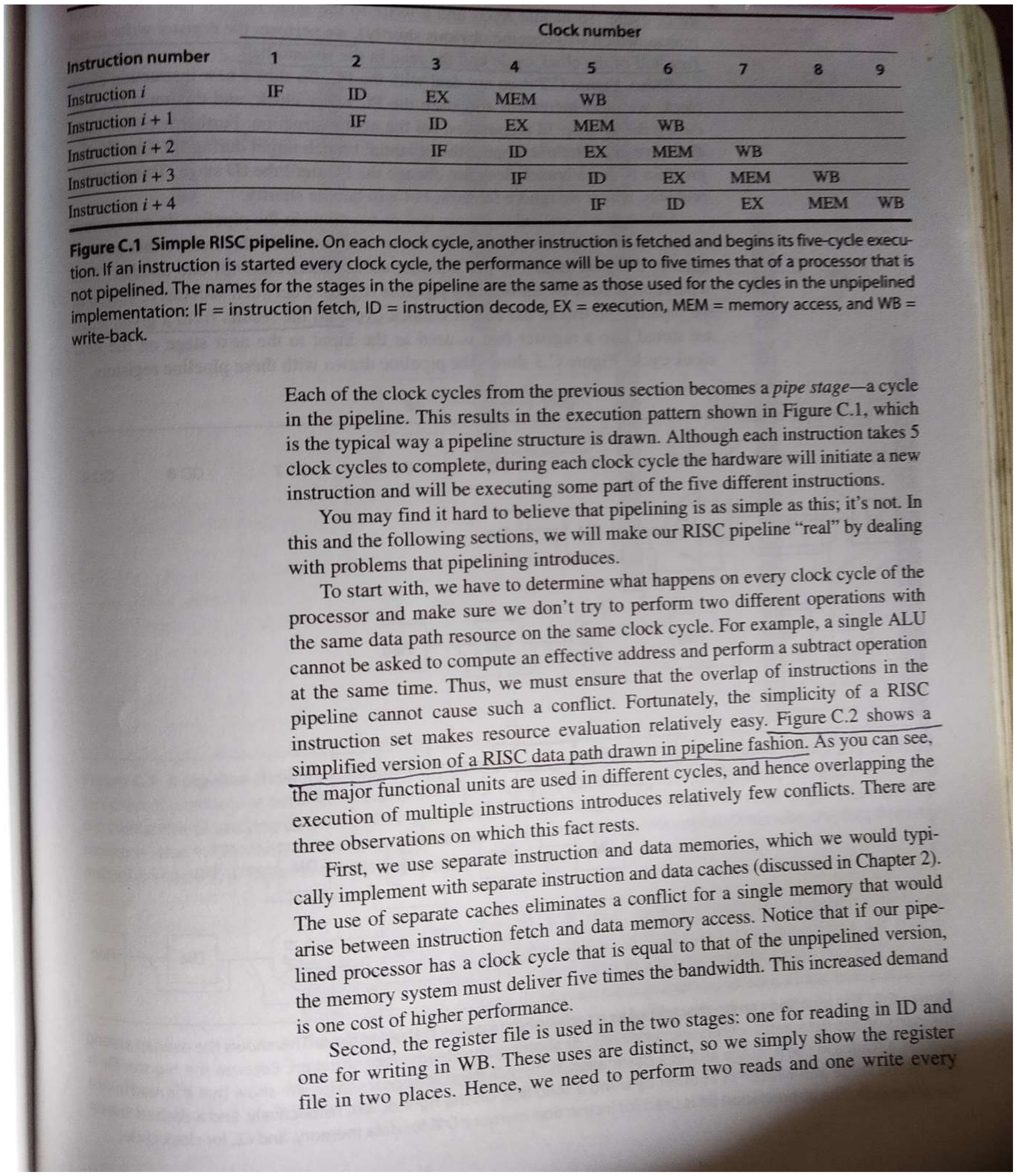
   Write the result into the register file, whether it comes from the memory system (for a load) or from the ALU (for an ALU instruction).

In this implementation, branch instructions require 2 cycles, store instructions require 4 cycles, and all other instructions require 5 cycles. Assuming a branch frequency of 12% and a store frequency of 10%, a typical instruction distribution leads to an overall CPI of 4.54. This implementation, however, is not optimal either in achieving the best performance or in using the minimal amount of hardware given the performance level; we leave the improvement of this design as an exercise for you and instead focus on pipelining this version.

The Classic Five-Stage Pipeline for a RISC Processor

We can pipeline the execution described above with almost no changes by simply starting a new instruction on each clock cycle. (See why we chose this design?)

Continuing,



Instruction number	Clock number 1	2	3	4	5	6	7	8	9
Instruction *i*	IF	ID	EX	MEM	WB				
Instruction *i* + 1		IF	ID	EX	MEM	WB			
Instruction *i* + 2			IF	ID	EX	MEM	WB		
Instruction *i* + 3				IF	ID	EX	MEM	WB	
Instruction *i* + 4					IF	ID	EX	MEM	WB

Figure C.1 Simple RISC pipeline. On each clock cycle, another instruction is fetched and begins its five-cycle execution. If an instruction is started every clock cycle, the performance will be up to five times that of a processor that is not pipelined. The names for the stages in the pipeline are the same as those used for the cycles in the unpipelined implementation: IF = instruction fetch, ID = instruction decode, EX = execution, MEM = memory access, and WB = write-back.

Each of the clock cycles from the previous section becomes a *pipe stage*—a cycle in the pipeline. This results in the execution pattern shown in Figure C.1, which is the typical way a pipeline structure is drawn. Although each instruction takes 5 clock cycles to complete, during each clock cycle the hardware will initiate a new instruction and will be executing some part of the five different instructions.

You may find it hard to believe that pipelining is as simple as this; it's not. In this and the following sections, we will make our RISC pipeline "real" by dealing with problems that pipelining introduces.

To start with, we have to determine what happens on every clock cycle of the processor and make sure we don't try to perform two different operations with the same data path resource on the same clock cycle. For example, a single ALU cannot be asked to compute an effective address and perform a subtract operation at the same time. Thus, we must ensure that the overlap of instructions in the pipeline cannot cause such a conflict. Fortunately, the simplicity of a RISC instruction set makes resource evaluation relatively easy. Figure C.2 shows a simplified version of a RISC data path drawn in pipeline fashion. As you can see, the major functional units are used in different cycles, and hence overlapping the execution of multiple instructions introduces relatively few conflicts. There are three observations on which this fact rests.

First, we use separate instruction and data memories, which we would typically implement with separate instruction and data caches (discussed in Chapter 2). The use of separate caches eliminates a conflict for a single memory that would arise between instruction fetch and data memory access. Notice that if our pipelined processor has a clock cycle that is equal to that of the unpipelined version, the memory system must deliver five times the bandwidth. This increased demand is one cost of higher performance.

Second, the register file is used in the two stages: one for reading in ID and one for writing in WB. These uses are distinct, so we simply show the register file in two places. Hence, we need to perform two reads and one write every