Advanced Parallel Architecture

Lesson 5

Annalisa Massini - 2016/2017

Pipelining

Hennessy, Patterson

Computer architecture A quantitive approach

Appendix C – Sections C.1, C.2

Pipelining

- *Pipelining* is an implementation technique whereby multiple instructions are overlapped in execution
- *Pipelining* takes advantage of parallelism that exists among the actions needed to execute an instruction
- In a computer pipeline:
  - **Each step** in the pipeline completes a part of an instruction
  - **Different steps** are completing different parts of different instructions in parallel.
  - Each of these steps is called a ***pipe stage*** or a ***pipe segment***
  - The stages are connected one to the next to form a pipe →instructions ***enter*** at one end, ***progress*** through the stages, and ***exit*** at the other end, as cars in an assembly line

Pipelining

- The **throughput** of an instruction pipeline is determined by how often an instruction exits the pipeline
- Because the ***pipe stages are hooked together***, all the stages must be ready to proceed at the same time, just as we would require in an assembly line
- The time required between moving an instruction one step down the pipeline is a ***processor cycle***
- The length of a processor cycle is determined by the time required for the ***slowest pipe stage***
- In a computer, this processor cycle is usually 1 clock cycle

Pipelining

- The goal is to **balance** the **length of each pipeline stage**
- If the stages are perfectly balanced, *assuming ideal conditions*:
  - The time per instruction , on the pipelined processor is

$$\frac{\text{time per instruction on the unpipelined processor}}{\text{number of pipeline stages}}$$

  - the ***ideal* speedup** due to pipelining is equal to the **number of pipeline stages**

Pipelining

- Usually, however, the *stages will not be perfectly balanced*
- Thus, the time per instruction on the pipelined processor will not have its minimum possible value (it can be close)

- Pipelining yields a *reduction in the average execution time per instruction*
- The reduction can be viewed as:
  - decreasing the number of clock ***cycles per instruction*** (CPI)
  - decreasing the clock cycle time
  - a combination

Pipelining

- Pipelining:
  - is an implementation technique that exploits parallelism among the instructions in a sequential instruction stream
  - is not visible to the programmer

- In the follow, we use a RISC architecture characterized by a few key properties, which simplify its implementation:
  - All operations on data apply to data in registers
  - The only operations that affect memory are **load** (move data from memory to a register) and **store** (to memory from a register) operations
  - The instruction formats are few in number

Pipelining

- Most RISC architectures have three classes of instructions:
  - *ALU instructions*—These instructions take either two registers or a register and a sign-extended immediate, operate on them, and store the result into a third register
  - *Load and store instructions*—These instructions take a register source, called the *base register* and an *offset*, to compute *effective address,* as well as a second register operand
  - *Branches and jumps*—Branches are conditional transfers of control. Unconditional jumps are provided in many RISC architectures

Pipelining

- Every instruction in this RISC subset can be implemented in at most 5 clock cycles:
  - ***Instruction fetch*** *cycle* (IF)
  - ***Instruction decode/register fetch*** *cycle* (ID) - Decode the instruction and read the registers. Do the equality test on the registers as they are read, for a possible branch. Compute the possible branch target address by adding the sign-extended offset to the incremented PC

Pipelining

- Every instruction in this RISC subset can be implemented in at most 5 clock cycles:
  - ***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 (ALU opcode) on the values read from the register file
    - Register-Immediate ALU instruction—The ALU performs the operation (ALU opcode) on the first value read from the register file and the sign-extended immediate

Pipelining

- Every instruction in this RISC subset can be implemented in at most 5 clock cycles:
  - ***Memory access*** (MEM): If the instruction is a ***load***, the memory does a read using the effective address. If it is a ***store***, then the memory writes the data from the second register using the effective address
  - ***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)
- Branch instructions require 2 cycles, store instructions require 4 cycles, and all other instructions require 5 cycles

Pipelining

	Clock number								
	1	2	3	4	5	6	7	8	9
Instruction number									
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

- Each of the clock cycles from the previous section becomes a *pipe stage*—a cycle in the pipeline
- This results in the execution pattern above, which is the typical way a pipeline structure is drawn

Pipelining

	Clock number								
	1	2	3	4	5	6	7	8	9
Instruction number									
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

- Each instruction takes 5 clock cycles to complete
- During each clock cycle the hardware:
  - will initiate a new instruction
  - will be executing some part of the five different instructions

Pipelining

	Clock number								
	1	2	3	4	5	6	7	8	9
Instruction number									
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

- Pipelining seems simple, but it's not
  - two different operations **cannot** be performed 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

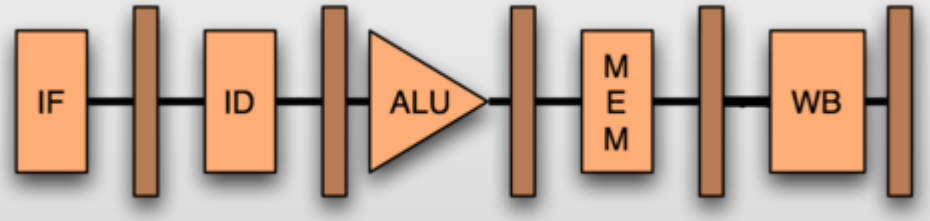
Pipelining

Observations

- The use of **separate caches** eliminates a conflict for a single memory that would arise between instruction fetch and data memory access
- The register file is used in the two stages: one for **reading** in ID and one for **writing** in WB. These uses are distinct
- To start a new instruction every clock, we must increment and store the PC every clock (IF stage). Furthermore, we must also have an adder to compute the potential branch target during ID. One further problem is that a **branch** does not change the PC until the ID stage. This causes a problem

Pipelining

- To ensure that instructions in different stages of the pipeline do not interfere with one another ***pipeline registers*** are introduced between successive stages of the pipeline:
  - at the end of a clock cycle all the results from a given stage are stored into a register that is used as the input to the next stage on the next clock cycle



Performance Issues in Pipelining

- Pipelining increases the CPU **instruction throughput** — *the number of instructions completed per unit of time* — but it does not reduce the execution time of a single instruction

- ***The increase in instruction throughput*** means that a ***program runs faster*** and has lower total execution time, even though ***no single instruction runs faster***!

Performance Issues in Pipelining

- In fact, the *execution time of each instruction is slightly increased* due to:
  - **imbalance** among the pipe stages
  - **overhead** in the control of the pipeline

- **Imbalance** among the pipe stages reduces performance since the clock can run no faster than the time needed for the slowest pipeline stage
- Pipeline **overhead** arises from the combination of pipeline register delay and clock skew

Example

- Assume that an unpipelined processor has a 1 ns clock cycle and that it uses 4 cycles for ALU operations and branches and 5 cycles for memory operations
- Assume that the relative frequencies of these operations are 40%, 20%, and 40%, respectively
- Suppose that due to clock skew and setup, pipelining the processor adds 0.2 ns of overhead to the clock
- How much speedup in the instruction execution rate will we gain from a pipeline?

Example

- The average instruction execution time on the **unpipelined** processor is:

$$\text{Average instruction execution time} =$$
$$= \text{Clock cycle} \times \text{Average CPI} =$$
$$= 1\ \text{ns} \times [(40\% + 20\%) \times 4 + 40\% \times 5] =$$
$$= 1\ \text{ns} \times 4.4 = 4.4\ \text{ns}$$

Example

- In the **pipelined** implementation, the clock must run at the speed of the slowest stage plus overhead
- Average instruction execution time is (1 + 0.2)ns = 1.2 ns
- Thus, the **speedup** from pipelining is

$$\text{Speedup from pipelining} =$$

$$= \frac{\text{Average instruction time unpipelined}}{\text{Average instruction time pipelined}} = \frac{4.4\text{ ns}}{1.2\text{ ns}} = 3.7 \text{ times}$$

- The 0.2 ns overhead establishes a limit on the effectiveness of pipelining