实验 3：指令流水线仿真程序

背景知识

1. Instruction pipeline

An **instruction pipeline** is a technique used in the design of computer and other digital electronic devices to increase their instruction throughput (the number of instructions that can be executed in a unit of time).

The fundamental idea is to split the processing of a computer instruction into a series of independent steps, with storage at the end of each step. This allows the computer's control circuitry to issue instructions at the processing rate of the slowest step, which is much faster than the time needed to perform all steps at once. The term pipeline refers to the fact that each step is carrying data at once (like water), and each step is connected to the next (like the links of a pipe).

Most modern CPUs are driven by a clock. The CPU consists internally of logic and register. When the clock signal arrives, the flip flops take their new value and the logic then requires a period of time to decode the new values. Then the next clock pulse arrives and the flip flops again take their new values, and so on. By breaking the logic into smaller pieces and inserting flip flops between the pieces of logic, the delay before the logic gives valid outputs is reduced. In this way the clock period can be reduced. For example, the classic RISC pipeline is broken into five stages with a set of flip flops between each stage.

1. Instruction fetch
2. Instruction decode and register fetch
3. Execute
4. Memory access
5. Register write back

When a programmer (or compiler) writes assembly code, they make the assumption that each instruction is executed before execution of the subsequent instruction is begun. This assumption is invalidated by pipelining. When this causes a program to behave incorrectly, the situation is known as a hazard. Various techniques for resolving hazards such as forwarding and stalling exist.

A non-pipeline architecture is inefficient because some CPU components (modules) are idle while another module is active during the instruction cycle. Pipelining does not completely cancel out idle time in a CPU but making those modules work in parallel improves program execution significantly.

Processors with pipelining are organized inside into stages which can semi-independently work on separate jobs. Each stage is organized and linked into a 'chain' so each stage's output is fed to

another stage until the job is done. This organization of the processor allows overall processing time to be significantly reduced.

A deeper pipeline means that there are more stages in the pipeline, and therefore, fewer logic gates in each stage. This generally means that the processor's frequency can be increased as the cycle time is lowered. This happens because there are fewer components in each stage of the pipeline, so the propagation delay is decreased for the overall stage.

Unfortunately, not all instructions are independent. In a sample pipeline, completing and instruction may require 5 stages. To operate at full performance, this pipeline will need to run 4 subsequent independent instructions while the first is completing. If 4 instructions that do not depend on the output of the first instruction are not available, the pipeline control logic must insert a stall or wasted clock cycle into the pipeline until the dependency is resolved. Fortunately, techniques such as forwarding can significantly reduce the cases where stalling is required. While pipelining can in theory increase performance over an unpipelined core by a factor of the number of stages (assuming the clock frequency also scales with the number of stages), in reality, most code does not allow for ideal execution.

Pipelining does not help in all cases. There are several possible disadvantages. An instruction pipeline is said to be fully pipeline if it can accept a new instruction every clock cycle. A pipeline that is not fully pipelined has wait cycles that delay the progress of the pipeline.

Advantages of Pipeline:

1. The cycle time of the processor is reduced, thus increasing instruction issue-rate in most cases.
2. Some combinational circuits such as adders or multipliers can be made faster by adding more circuitry. If pipelining is used instead, it can save circuitry vs. a more complex combinational circuit.

Disadvantages of Pipeline:

1. A non-pipelined processor executes only a single instruction at a time. This prevent branch delays (in effect, every branch is delayed) and problems with serial instructions being executed concurrently. Consequently the design is simpler and cheaper to manufacture.
2. The instruction latency in a non-pipelined processor is slightly lower than in s pipelined equivalent. This is because extra flip flops must be added to the data path of a pipelined processor.
3. A non-pipelined processor will have a stable instruction bandwidth. The performance of a pipelined processor is much harder to predict and may vary more widely between different programs.

2. Classic RISC pipeline

In the history of computer hardware, some early reduced instruction set computer central processing

units (RISC CPUs) used a very similar architectural solution, now called a **classic RISC pipeline**. Those CPUs were: MIPS, SPARC, Motorola 88000, and later DLX.

Each of these classic scalar RISC designs fetched and attempted to execute one instruction per cycle. The main common concept of each design was a five stage execution instruction pipeline. During operation, each pipeline stage would work on one instruction at a time.

Each of these stages consisted of an initial set of flip-flops, and combinational logic which operated on the outputs of those flops.

The classic five stage RISC pipeline

Instruction fetch

The instruction Cache on these machines had a latency of one cycle. During the Instruction Fetch stage, a 32-bit instruction was fetched from the cache.

The PC predictor sends the Program Counter (PC) to the Instruction Cache to read the current instruction. At the same time, the PC predictor predicts the address of the next instruction by incrementing the PC by 4 (all instructions were 4 bytes long). This prediction was always wrong in the case of a taken branch, jump, or exception. Later machines would use more complicated and accurate algorithms to guess the next instruction address.

Decode

Unlike earlier microcoded machines, the first RISC machines had no microcode. Once fetched from the instruction cache, the instruction bits were shifted down the pipeline, so that simple combinational logic in each pipeline stage could produce the control signals for the datapath directly from the instruction bits. As a result, very little decoding is done in the stage traditionally called the decode stage.

At the same time the register file was read, instruction issue logic in this stage determined if the pipeline was ready to execute the instruction in this stage. If not, the issue logic would cause both the Instruction Fetch stage and the Decode stage to stall. On a stall cycle, the stages would prevent their initial flops from accepting new bits.

If the instruction decoded was a branch or jump, the target address of the branch or jump was computed in parallel with reading the register file. The branch condition is computed after the register file is read, and if the branch is taken or if the instruction is a jump, the PC predictor in the first stage is assigned the branch target, rather than the incremented PC that has been computed.

The decode stage ended up with quite a lot of hardware: the MIPS instruction set had the possibility of branching if two registers were equal, so 32-bit-wide AND tree ran in series after the register file read, making a very long critical path through this stage. Also, the branch target computation generally required a 16 it add and a 14 bit incrementer. Resolving the branch in the decode stage

made it possible to have just a single-cycle branch mispredict penalty. Since branches were very often taken (and thus mispredicted), it was very important to keep this penalty low.

Execute

Instructions on these simple RISC machines can be divided into three latency classes according to the type of the operation:

- Register-Register Operation (Single cycle latency): Add, subtract, compare, and logical operation. During the execute stage, the two arguments were fed to a simple ALU, which generated the result by the end of the execute stage.
- Memory Reference (Two cycle latency). All loads from memory. During the execute stage, the ALU added the two arguments (a register and a constant offset) to produce a virtual address by the end of the cycle.
- Multi Cycle Instructions (Many cycles latency). Integer multiply and divide and all floating-point operations. During the execute stage, the operands to these operations were fed to the multi-cycle multiply/divide unit. The rest of the pipeline was free to continue execution while the multiply/divide unit did its work. To avoid complicating the writeback stage and issue logic, multicycle instruction wrote their results to a separate set of registers.

Memory Access

During this stage, single cycle latency instructions simply have their results forwarded to the next stage. This forwarding ensures that both single and two cycle instructions always write their results in the same stage of the pipeline, so that just one write port to the register file can be used, and it is always available.

Writeback

During this stage, both single cycle and two cycle instructions write their results into the register file.

实验目的

- 了解指令流水线的基本原理和特点
- 掌握传统简约指令集指令流水线的基本结构和运行过程
- 掌握传统简约指令集指令流水线处理阻塞的基本方法

实验内容

1. 定义 NK-CPU 指令流水线仿真的基本数据结构
2. 使用 C 语言实现 NK-CPU 指令流水线仿真程序

3. 仿真运行实验 2 中实现的随机数排序程序

实验要求

1. 仿真程序具有基本功能包括：
   - 可以编辑和查看“数据存储器”的内容
   - 可以查看“汇编语言”程序
   - 可以查看 NK-CPU 寄存器的内容
   - 可以显示指令流水线的运行状态
   - 可以控制程序“执行”、“单步执行”。

2. 仿真程序界面布局建议：