Lab Report 2

Computer Architecture Lab

201651015 Dipansh Khandelwal

201651008 Aman Singh

28th January, 2019

Aim

To understand the basic principles of how pipelining works, including the problems of data and branch hazards. Finally, we should have an understanding of how the instructions are used to control different parts of the data path through a control unit. (Pipelined Processors - Single Instruction)

Questions and Answers

1. What is a CPU ?

A central processing unit, also called a central processor or main processor, is the electronic circuitry within a computer that carries out the instructions of a computer program by performing the basic arithmetic, logic, controlling, and input/output operations specified by the instructions.

2. What is an assembly language program ?

The assembly language program is written in the low-level language which is developed by using mnemonics.

3. How does a compiler execute simple machine language instructions?

It translates the given instructions into Machine code that can be executed directly by a computer’s central processing unit (CPU).

4. What is the relation between assembly language and machine language?

Assembly language is a more human readable view of machine language. Instead of 2 representing the machine language as numbers, the instructions and registers are given names. Eg. for load use ld..

5. What does the instruction ‘add t0, t1, t2’ do?

The above instruction means: [t0] = [t1] + [t2]. The values are added as signed (2's complement) integers.

6. What does the instruction ‘beq t0, t1, Dest’ do?

The above instruction can be understood as:

If t0 == t1 then:

    go to Dest;

7. How does the pipelined CPU different from a non-pipelined?

Pipelining is an implementation technique where multiple instructions are overlapped in execution whereas in non-pipelining all the actions (fetching, decoding, execution of instructions and writing the results into the memory) are grouped into a single step.

8. Describe Hazard.

In the CPU design, hazards are problems with the instruction pipeline in CPU microarchitectures when the next instruction cannot execute in the following clock cycle, and can potentially lead to incorrect computation results.

Example

```
#include <iregdef.h>

.set noreorder

.text

.global start

.ent start

start: add t0, t1, t2

nop

nop

nop

nop

.end start
```

Solutions

1. In this stage, the CPU reads instructions from the address in the memory whose value is present in the program counter.

2. In this stage, the instruction is decoded and the register file is accessed to get the values from the registers used in the instruction.

3. In this stage, ALU operations are performed.

4. In this stage, memory operands are read and written from/to the memory that is present in the instruction.

5. In this stage, the computed/fetched value is written back to the register present in the instruction.

6. IF - Instruction Fetch, ID - Instruction Decode, EX - Execute, MEM - Memory, WB - Write Back. These are the five stages of RISC pipelining as explained in the above five answers.

7. 5 clock cycles required.

8. Execution Stage.

9. Memory(4th Stage) not used by arithmetic instruction as after computing it writes back to the desired location.

PROBLEM

1)#include <iregdef.h>

.set noreorder

.text

.global start

.ent start

start: lw t0, 0(t1)

nop

nop

nop

nop

.end start

---

a) Stage 1 (Instruction Fetch)

In this stage, the CPU reads instructions from the address in the memory whose value is present in the program counter.

Stage 2 (Instruction Decode)

In this stage, the instruction is decoded and the register file (t0, t1) is accessed to get the values from the registers used in the instruction.

Stage 3 (Instruction Execute)

In this stage, ALU operations are performed.

R[t0] <- MEM[R[t1] + s_extend(0)];

Stage 4 (Memory Access)

In this stage, memory operands are read from the memory that is present in the instruction.

Stage 5 (Write Back)

In this stage, the computed/fetched value is written back to the register present in the instruction.

---

b) ALU operations are performed.

R[t0] <- MEM[R[t1] + s_extend(0)];

MEM[R[t1] + s_extend(0)] The computation of this instruction takes place in ALU in order to find the exact location where the loaded data is to be written

c) 5 clock cycles

d) All stages used

2)

```
#include <iregdef.h>

.set noreorder

.text

.global start

.ent start

start: st t0, 4(t1)

nop

nop

nop

nop

.end start
```

---

a) Stage 1 (Instruction Fetch)6

In this stage, the CPU reads instructions from the address in the memory whose value is present in the program counter.

Stage 2 (Instruction Decode)

In this stage, the instruction is decoded and the register file (t0, t1) is accessed to get the values from the registers used in the instruction.

Stage 3 (Instruction Execute)

In this stage, ALU operations are performed.

MEM[R[t1] + sign_extend(4)] <- R[t0]

Stage 4 (Memory Access)

In this stage, memory operands are read from the memory that is present in the instruction.0

b) ALU operations are performed.

MEM[R[t1] + sign_extend(4)] <- R[t0]

MEM[R[t1] + s_extend(4)] The computation of this instruction takes place in ALU in order to find the exact location where the data is to be read.

c) 4 clock cycles

d) Write Back stage not used

---

3)

```
#include <iregdef.h>

.set noreorder

.text

.global start

.ent start

Dest;

start: beq t0, t1, Dest

nop

nop

nop

nop

.end start
```

a) Stage 1 (Instruction Fetch)

In this stage, the CPU reads instructions from the address in the memory whose value is present in the program counter.

Stage 2 (Instruction Decode)

In this stage, the instruction is decoded and the register file (t0, t1) is accessed to get the values from the registers used in the instruction.

Stage 3 (Instruction Execute)

In this stage, ALU operations are performed.

Checks if (R[t0] == R[t1])

b) ALU checks if the value of R[t0] is equal to the value of R[t1] and if true the address of ‘Dest’ is passed to the Program Counter (PC).

c) 3 clock cycles

d) 3 stages used - IF, ID,EX
