DESIGN DOCUMENT AND TEST PLAN

for

Assignment-5

CSL-215

Prepared by

Akshit Kansra(2016CSJ0001)
Abhishek Gupta(2016CSJ0012)

April 30, 2018

Contents

1 INTRODUCTION

1.1 What is this project about

In the project, we are trying to implement an ARM Assembly Simulator on our machine, which has Intel architecture. The simulator is being written in C++ language, and in this project, we plan to implement some basic instructions of the ARM Assembly. Our implemented instructions include *add*, *sub*, *ldr*, *str*, some branch instructions like beq, bne, bge, ble, b etc.In assignments 3 and 4, we did implement basic version of simulator, which implements each instruction one by one, and hence is a bit slower. Also, in assignment 4, we added the functionality of timing, where we got statistics such as Average CPI, total number of clock cycles etc.

Now in this assignment,we also plan to add pipelining. Pipelining is very useful to reduce the execution time, by keeping idle resources busy, and breaking complex instructions into smaller subparts, such that the clockcycle can be reduced, and execution takes place faster.

We plan to implement 5-stage pipelined version of ARMSim. The 5 stages are :

1. Instruction Fetch(IF)
2. Instruction Decode-Register Access(ID)
3. Execution(EXEC)
4. Memory Access(MEM)
5. Write Back(WB)

2 Work-Plan

We need to implement different stages of the pipeline in the same cycle, for which, the algorithm which we plan to implement is as follows

2.1 Basic Algorithm

We need to make sure that in each cycle, all the 5 stages do proceed, and also we can stall the pipeline if necessary. For this purpose, we have developed an algorithm to implement the pipelined version in C++, which is as follows:

We will keep on running a loop which continues until we reach the end of file, or SWI_EXIT command is found. In each loop, we plan to implement the stages of pipeline in reverse order, i.e, we will be having some functions for each stage of pipeline, which will be executed in the reverse order(i.e, first WB, then MEM, then EXEC ...), and by the end of the instruction, we forward it to the next stage. Why we are using this instead of forward execution is because in this manner, it is easy to detect an hazard, i.e. data hazard and control hazard, because we are able to check if previous instruction's data is affecting next instruction's data or not. Whenever we detect a hazard, we stall the pipeline. We wait for the instruction to finish, or atleast that part to finish which causes the hazard, and then again resume the pipeline.

In this implementation, there is no chance of structural hazard,as we will be able to stall the pipeline in case one of the resources is busy, and the previous stage has finished. Hence the implementation will continue.

3 COMPONENTS

THe processor consists of many smaller components that all gel together to carry out the instructions feeded into it. The high level component is named Processor which makes necessary calls to the actual components. The various components of the processor are listed below:

3.1 Instruction Memory

This contains the instructions that have to fed to the processor. The processor will pick up an instruction from the instruction memory based on the current value of PC.

3.2 Register File

This contiains a storage locations for items that need to be processed upon by the ALU. Currently there are 16 registers with some of them having special meaning. Eg. Link register and Stack register.

3.3 ALU

This is the brains of the processor and all the arithmetic and logical operations take place in the ALU. It takes input from the Register File and genrerates an output.

3.4 Data Memory

This holds the excess memory that the processor accesses time to time when it needs to process data not found in the registers. The output from the ALU can be stored in the Data Memory using a store operation. Similarly data from the fata memory can be loaded to a register using a load operation.

3.5 PC

This maintains the count of the current instruction that needs to be processed. The branch statement directly changes the PC value so that it can jump from one instruction directly to another. In general the PC updates its value by adding 1 every instruction cycle.

3.6 Controller

This maintains all the flags and does all the equality checks. It manages the flow of data between the different components and helps with Data Hazards.

3.7 Pipeline

The Pipeline consists of 5 stages

3.7.1 Instruction Fetch

3.7.2 Register Retrieval

3.7.3 ALU Processing

3.7.4 Data Memory Access

3.7.5 Write-back to Register

3.8 AIM

The pipeline helps to reduce the number of clock cycles taken to reduce an instruction. However at times ongoing instructions might clash due to the clash of data resources. We call this a Data Hazard. Data Hazards are often solved using efficient data forwarding techniques which have been implemented in the processor in focus.

4 TESTCASES

4.1 STRUCTURAL HAZARD

It means that the hardware cannot support the combination of instructions that we want to execute in the same clock cycle.

4.2 DATA HAZARD

Data hazard occur when the pipeline must be stalled because one step must wait for another to complete .

4.2.1 Example 1

add r1,r1,r2
sub r3,r1,r4

Forwarding paths are valid only if the destination stage is later in time than the source stage.

4.2.2 Example 2

ld r1,0(r2)
sub r5 , r1 ,r3

We need to stall even with forwarding when an R-Format instruction following a load tries to use the data.

4.3 CONTROL HAZARD

Arises from the need to make a decision based on the results of one instruction while others sre executing.

4.3.1 Example 1

ld r2, 0(r4) // r2 := memory at r4
ld r3, 4(r4) // r3 := memory at r4+4

```
sub r1, r2, r3 // r1 := r2 - r3
cmp r1, r7 // compare r1 and r7
beq L1 // if r1 is not equal to r7, goto L1
ldi r1, 1 // r1 := 1
L1: not r1, r1 // r1 := not r1
st r1, 0(r5) // store r1 to memory at r5
```

This code has the effect of comparing two memory locations and storing the result of that comparison to another location.
Since the cmp instruction is a control instruction, there are two problems here:

1. When the cmp instruction is in the decode stage, the sub instruction is in the execute stage. The branch can't read the output of the sub until it has been written to the register file; if it reads it early, it will read the wrong value.

2. More importantly, one cycle after the beqz instruction is fetched, what instruction should be fetched next? We don't know, because that depends on the outcome of the branch instruction. At this point, we don't even know that the branch instruction depends on the previous instruction, since it hasn't been decoded yet, so bypass can't help us. Even if we did know the condition, we still don't know where to fetch from if the branch is taken because the effective address computation for branches doesn't happen until the EX stage.

4.3.2 Example 2

```
add r4,r5,r6
cmp r1,r2
beq Label
or r7,r8,r9
```

5 TEST INPUT FILES

5.0.1 INPUT FILE 1

@Iteration over loop
mov r0, (0) @initialise loop index to 0
mov r1, (100) @number of iterations

Loop:
add r0, r0, (1) @increment loop index
cmp r0, r1
ble Loop

5.0.2 INPUT FILE 2

@Fibonacci numbers @Expects to be called with n in R0, and will return f (n) in the same register.
fibonacci:
mov r1, (0)
mov r2, (1)

fibloop:
mov r3, r2
add r2, r1, r2
mov r1, r3
sub r0, r0, (1)
cmp r0, (1)
bne fibloop

mov r0, r2
mov pc, lr

5.0.3 Latency Input File

ldr = 20;
ldr_pseudo = 1;
str = 20;
str_pseudo = 1;

add = 1;
sub = 1;
cmp = 1;
mul = 5;
bne = 2;
bge = 2;
b = 2;
bl = 2;
mov = 1;