Design Document

for

Pipelined Version Of ARM Simulator

Prepared by Rahul Nirania and Abhishek Yadav

April 10, 2018

Contents

1 Introduction

1.1 Abstract

We will be implementing pipelined version of arm-simulator in this assignment.Pipelining is fundamental concept which helps in increasing throughput.We are dividing our pipelining into 5 stages.Our simulator will be taking input from assembly file and latency.txt.Assembly file will consist of Instruction set while latency.txt will consist of latency(no. of clock cycle) particular instruction will take.Based on input from these two file a output file(*simulation.txt*) will be generated consisting of IPC,instruction count and clock cycle count.Current status of each and every instruction along with stage will be also displayed on terminal.

1.2 Technical Skills Used

(1)C++
(2)Pipelining

2 Overall Description

2.1 Software Perspective

This software is developed from educational purposes.This will help to simulate assembly code(ARM version).

2.2 Software Functions

This is ARM simulator which will simulating assembly code very fastly as compared to non-pipelined version.Pipelining helps to increase throughput while execution time of each and every instruction will remain same.We will be printing some important stastics after each instruction and at end of the program.

2.3 Unfolding Abstraction

We had divided our project into 5 stages of development.Since pipelined version of ARM processor consist of five cycle(as discussed in class) we are dividing our working into 5 stages.Each stage execution is written as c++ program.In main function we will be invoking these 5 function.

3 Parsing

3.1 Method

We are writing our own parser which will take input from *input.txt* and if file contains some error we will point error and further program will be not executed.

3.2 Instruction set

These type of instruction will be supported in our simulator.
1. Ldr rd,[r2,♯offset]
2. Ldr rd,[r2]
3. Str r1,[r2,♯offset]
4. Str r1,[r2]
5. Add rd, Rop1,Rop2]
6. Add rd, Rop1,♯imm
7. Sub rd, Rop1,Rop2
8. Sub rd, Rop1,♯imm
9. Mul rd, Rop1,Rop2
11. Mul rd, Rop1,♯imm
12. Mov rd, rop1
13. Mov rd, imm
14. Cmp rOp1,rOp2
15. Cmp rOp1,imm
16. Bne Label
17. Bge Label

4 Stages of Pipelining

4.1 Instruction fetch(IF)

For instruction fetch we use the data which is mentioned above(i.e. An array containing all the words in the file as its data). We will make an array of structure which will carry the information about the each instruction in an ordered way. This array will be used as INSTRUCION MEMORY for fetching each instruction. The reason why we are making this array of structure rather than using simply the array which we made during parsing is that it is very difficult to remember the index of the array in which instruction is stored. For example consider the following part of code:

Ldr r1, [r2]
Add r1, r1, r4
b loop

In this the array we get A = [Ldr , r1, [r2], Add , r1, r1, r4, b ,loop] and now the instruction starts at 0, 3, 7 and to know this we have to go through each array element and have to check whether its content matches any of Add, Sub, Ldr etc. To know the next instruction. One other way we thought to minimise the runtime is to do instruction fetch while parsing. But this idea do not work at all because there may be an error at 999th line of 1000 line code and this method will take a long time to detect it. It also do not give any data about the branch which can be can be called while Beq, Bne etc instruction.

4.2 Instruction Decode(ID)

In instruction decode we will be reading from register.Register is implemented as an 1-D array of size 12.Register no. is index of array.Corrosponding register data will be stored in ID/EX register file.This register file is implemented by an array which will be global in our c++ code.Destination register no. is also passed to register file,this is done so that destination register is not changed while executing other instruction and correct value is written in write back stage.We are storing our instruction in vector array of structure where each structure consist of a instruction.We will be storing Cmp and branch in one structure.In case of branch we will be moving to instruction following label.The instruction calculation will be done in Exe stage.Also for load and store instruction we storing offset in register file.

4.3 Execution(EXE)

Basically this section is used by almost all instruction. In this section we will do the arethematic on the data of registers provided by the previous section. The arithematic will include addition, subtraction and multiplication. This section will also include the calculation related to 'cmp' instruction and deduce the value of program counter(or in other words whether to take branch or not). This section will require the ALUSrc, ALUOp, RegDst values from control.

4.4 Memory(MEM)

This section is mainly used when we want some values from memory otherwise we just pass the data we computed in Execute stage to WriteBack stage. This section is used by instuction like "Load". In Load instruction we will fetch the data from the memory address which we have computed in the Execute part. This section will require MemRead, MemtoReg, MemWrite values from control.

4.5 Write back(WB)

Except store instruction all instruction will be using this stage to write back to register.The correct register number will be preserved as we are pipelining through the stages.This stage consist of one mux which decide whether we have to write or not.

5 Dealing with Hazard

5.1 Hazard

The pipelining is very effective in reducing the time taken to compile and execute the instruction. But the biggest problem it faces is Hazard. They may be of three type:
1.Data Hazard
2.Control Hazard
3.Structure Hazard

5.2 Data Hazard

Reason of occurence:When a we require a data which is not computed yet. Because of this we have to wait till that instruction is completed which is calculating that data.
Checking and Solution:We can divide our solution into two parts, one for R-R data hazard and other for R-Load data hazard. For both the cases we wil make a ckeck function which we will use to check that whether the data of any register which we are using is not been under process for writing at that time. If the function check is telling it to be true then we do this solution for above two cases. In first case we provide the value from the MEM stage of the instruction in which this value is been written and to the EX stage of that instruction in which it is required. This process is done by our biapassing function. And in the other case we first wait for one cycle and then biapass the value from WB stage to EX of the two instruction mentioned earlier respectively. We are palning to make some function for stall also.

5.3 Control Hazard

Reason of occurence:This hazard occur when there is branch related instruction like Beq, Bne etc except B instruction. As it takes time to choose whether to take branch or to go to the next instruction.
Checking and Solution: Whenever there is any branch related instruction except B there will be this hazard. As it is very difficult to solve so we hadn't planned for it yet.But as of now we will try to do it by two bit predictor method.

5.4 Structural Hazard

Reason of occerance:This is occured when two sections/parts of pipeline try to use the same memory at a single time.
Solution: We are making Instruction Fetch memory and Main memory seperately. By this if Instructiom fetch part require next instruction and Memory part want some data from main memory then we can do both in same cycle. One of the other case is when we try to write and read data from the same memory at the same time. This type is handled by choosing that minimum clock cycle length that writes the data in first half of the cycle and reads the data in the other half of the cycle and also takes care of other condition as mentioned above.

6 Output

6.1 simulation.txt

After end of simulation we will be displaying IPC(instruction per count),clock cycle count,register value and memory.A option will be provided to user whether he/she wants to run whole program at once or want to simulate step by step.When user simulate the program step by step,instruction residing in various pipelined stages will be shown on terminal.

6.2 Test Plans

We will be designing three test-cases to test our program.
1. Test-case with no hazard
LDR r1,[r0,♯0]
LDR r2,[r0,♯8]
LDR r4,[r0,♯16]
ADD r3,r1, r2
ADD r5,r1, r4

2. Test-case with data hazard

Case-1
Ldr r1, [r2,♯0]
Sub r4,r1,r5
These two instruction in continuation will cause data hazard.Value of r1 register will be available in r1 after write back stage.But we will be executing next instruction immediately after IF stage and till that time value of r1 will be not available in register.
To remove this hazard we will be *stalling* for one cycle and then *forwarding* value of r1 from **MEM** stage to **ID-EXE** stage of second instruction.

Case-2
Add r5,r0,r1
Sub r3,r5,r4
In these type of dependency on r5 value of r5 will be available in WB stage only.To remove this type of hazard we will forwarding result of EXE stage to ID stage.
To remove stall we can also do code scheduling but if time permits we can use this

approach to remove data hazard.

3.Test-case with control hazard

```
StoreIntegers: str r2 , [r3]
add r3, r3,♯4
add r2, r2,♯1
sub r1, r1,♯1
cmp r1,♯0
bne StoreIntegers
mov r1,♯5
mov r4,♯0
ldr r3, =AA
```

We are taking cmp and bne to take as one instruction,after executing these instruction we cannot conclude whether to take branch or not.To remove this hazard we can either uses static branch prediction or dynamic branch prediction.But we will be using two bit prediction to remove this hazard.