CSL216: Computer Architecture
Design Document
ARM Assembly Simulator

Vaibhav Vashisht 2016CSJ0002
Pratik Parmar 2016CSJ0049

Contents

1 Introduction

1.1 Purpose

The purpose of this document is to describe the implementation of ARM Assembly Program Simulator application developed by us.

1.2 Scope

The software acts as a Assembler for ARM Assembly Code. It runs the ARM assembly code and display the resister's value at any desired instruction in the Assembly code. It also give statistics about the instruction count, CPI, IPI etc at the end of the output. Due to incorparation of pipelining in the simulator,it produces the output in minimum number of cycles possible(At which giving correct ans is feasible) for the assembly code.

1.3 Definitions

1. **Pipelining:** Pipelining is an implementation technique where multiple instructions are overlapped in execution. The computer pipeline is divided in stages. Each stage completes a part of an instruction in parallel. 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.

2. **Hazard:** 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.

   (a) **Data Hazard:** They arise when an instruction depends on the result of a previous instruction in a way that is exposed by the overlapping of instructions in the pipeline.

   (b) **Control Hazard:** They arise from the pipelining of branches and other instructions that change the PC.

   (c) **Structural Hazard:** They arise from resource conflicts when the hardware cannot support all possible combinations of instructions in simultaneous overlapped execution.

3. **Forwarding:** Operand forwarding (or data forwarding) is an optimization in pipelined CPUs to limit performance deficits which occur due to pipeline stalls.

4. **Stall:** A bubble or pipeline stall is a delay in execution of an instruction in an instruction pipeline in order to resolve a hazard.

2 Overall Design

In this Simulator input file(”input.txt”) which contains the ARM assembly code will be given. The Simulator will interpretes the input and also generator any error occuring in the code (We have implemented Panic Mode error recovery).

2.1 Components of Simulator

1. **Instruction Class:** The class **Instruction** consists store information about each Instruction:
   (a) **Type**
   (b) **Destination Register:**
   (c) **Register Operand1:** Returns the index of the component i.e if resistor name is r1 then it returns 1.
   (d) **Register Operand**
   (e) **Offset:** Offset information for LDR command.

2. **Pipeline Class:** This class consists of Register values information.
   (a) It contains all the information related to the instruction in it.

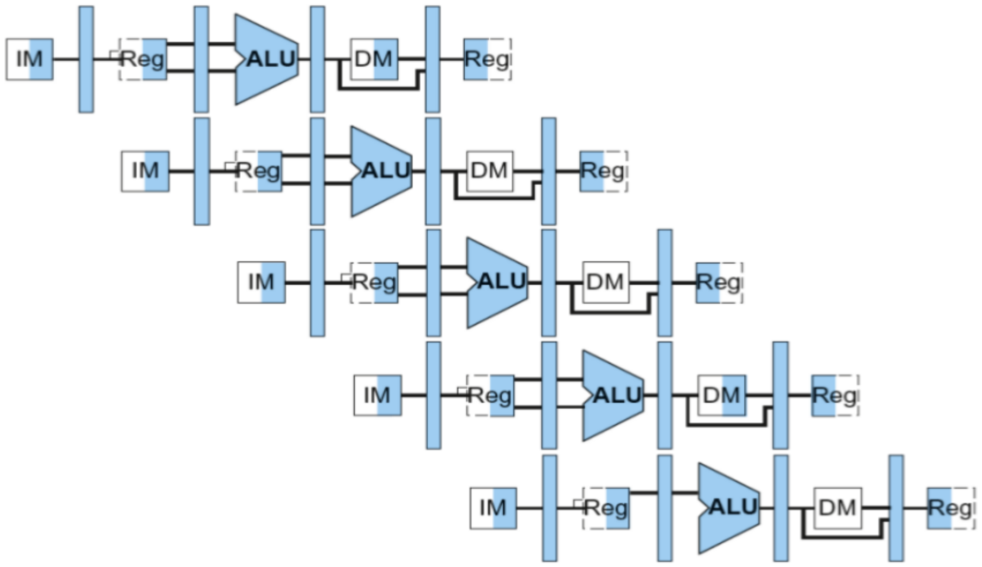
3 Algorithm

We will divide each instruction into five stages as done in pipeline.

1. IF(Instruction Fetch)
2. ID(Instruction Decode)

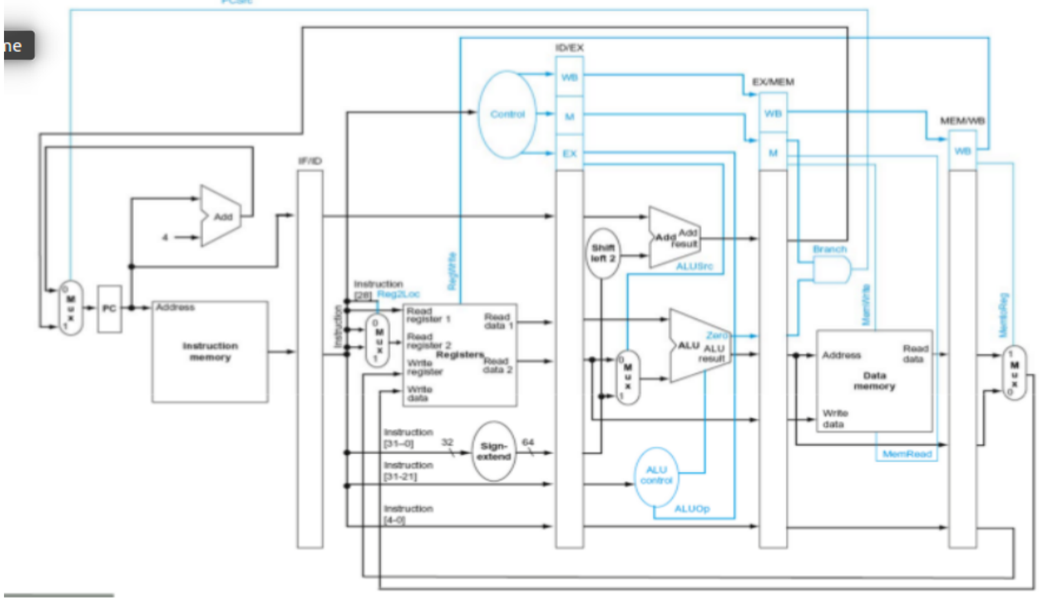
3. EX(Execute)
4. MEM(Memory)
5. WB(Write Back)

We will make generalized portion in a functions which will handle every type of instruction lets call them stage1,stage2,stage3,stage4,stage5.
stage1 will handle IF for every instruction,similarly stage2 will handle ID for each type of instruction and so on.
We have improved our parsing and have made the data of each instruction available as data portion of a class.
For eg if we want to extract instruction type we can easily extract it from Instruction Object.Type() function.These functions are used for making stage1,stage2 etc. Making f1,f2 etc portion is trivial as we just have to replicate the datapath of the final circuit which we made in processor chapter as given below.



Let's explain how will these functions work for instruction add. stage1 will extract the complete instruction from the class and pass on to IF/ID pipeline.In the next clock cycle stage2 will start decoding the instruction and will extract register number,instruction type(these will be extracted from object of class) and also read register if required and pass onto the next pipeline register. stage3 will add numbers if command being considered is add,stage4 will store the correspoding values in memory(array here) and stage5 will write

back to register.The above mentioned text is very trivial and seems to be very similar to what we studied in processor chapter but differnece will be illustarated through code.
Now comes the part of how to organise these functions so that we can get a pipeline.For this we will take help of recursion.Let's explain this - :



This image above illustrates how pipeline works. The above function completely illustrates our algorithm for implementing pipeline. They illustrate that ID function for ith instruction after its compeletion will call IF for the next instruction,this will ensure that we get the work done within a clock cycle.
For pipeline also we will have a class which will have pipeline registers as their data component.Also control values will also be contained in the pipeline.
Now lets move to handling hazards.

Data Hazard We will compare registers for EX/MEM stage and ID/EX stage if we find destination register of EX/MEM stage to be used in ID/EX stage then we know its a data hazard.This can done in f3 function which simulates execute stage using an if condition for comparing two pipeline registers.Now this hazard needs to be removed.
This is done by extracting the needed value from the EX/MEM pipeline

register.This is incorporated by putting an if statement in the execute state function which checks if the given conditon holds then value is extracted from the pipeline register of the next stage .

```
MOV r3,#1
MOV r5,#111
MOV r6,#2
MOV r1,#12
SUB r2, r1, r3
ADD r12, r2, r5
SUB r13, r6, r2
ADD r11, r2, r2
```

Load-Use Hazard Similar to data hazard these are detected by comparing ID/EX and IF/ID pipeline registers.This is resolved by inserting a stall into the pipeline.How do we insert a stall ?
We maintain a bool variable which be passed onto all the stages of the pipeline and that varaible if true will not allow any data to be written into the regsiters and memory.
Alternatively we can also implement as done in arm pipeline.We will turn all control values to zero and hence nothing will be allowed to written to registers or memory.This can be done in the f2 function which then propagates to the next stages of the pipeline.
We also have to stop increment of pc until we have stalls, this can be done by decrementing pc in f2 which will again get incremented by default and hence will fetch the same instruction.
Different number of clock cycles for some instructions -:
This means that some stages of these instrcution will take more than one clock cycle hence we need to stall till that stage gets completed.Stalling is done as described in the load use hazard by setting control values to zero.

```
MOV r1,#1200
MOV r0,#4
STR r0, [r1,#20]
LDR r2, [r1,#20]
ADD r4, r2, r0
SUB r8, r2, r1
ADD r9, r4, r2
```

Branch Hazard One simple way is to stall the pipeline till we are not sure where will branch move pc to.As we have implemented branch instructions based on cmp statement placed above the branch statement hence we have to stall after cmp for it to be able to compare the values till execute stage.After that we can directly direct that value(true or false) to the first stage of branch and there on branch or not.But to make pipeline more aggressive we need to use branch prediction for which we may have to flush all the stages of pipeline extracted due to taking wrong prediction.This can be done by maintaining a global variable which can be turned true when we detect that we have taken the wrong branch(This can be detected upto EX or MEM stage),hence we need to flush atmost 4 pipeline stages.After the global variable is turned true we can simply zero all the wrongly taken stages by making all signals and values in them to be zero as a result no wrong values propagate in next stages.Also we need to change pc,this can also be done by an if condition on global variable which will hence change pc.

```
MOV r0,#1
MOV r1,#5
Loop:
ADD r2, r2, r0
ADD r0, r0,#1
CMP r0, r1
ADD r3, r2, r2
SUB r4, r3, r2
SUB r5, r4, r1
LDR r6, [r5]
MOV r7, r6
BNE Loop
ADD r7, r7, r1
SUB r8, r7, r5
```

Structural Hazard This are automatically removed by maintaining different arrays for storage.

Assumptions For each round in the pipeline, each stage in the pipeline will take maximum of cycles required by any pipeline.

Sequence of Execution in function Operate():

Write Back(Stage 5)
Memory (Stage 4)
Execute (Stage 3)
Instruction Decode (Stage 2)
Instruction Fetch (Stage 1)

For passing a NOP instruction, we pass 9999 into the function.(This works as a code for NOP instruction).
In the starting each pipeline register will be in nop state therefore no operation will be performed in stage 5 to stage 2, then only in stage 1 Instruction fetch will happen.
Then continuously instruction will be passed in the function and earlier instruction in the operate function will go up in stage and finally leave when write back stage would happen for them.
Similarly at the end 4 extra NOP instructions are passed in the pipeline to get last instruction to complete it's write back.

3.1 Instruction Handling

1. **ADD,SUB(etc arithmetic operations)**
   Decoding and fetching stages will be common to all type of instructions but EX stage will be dfiferent for arithmetic operations in this stage these instructions will use ALU to calculate the result and send to the next stage pipeline registers.

2. **STORE,LOAD(memory type operations)**
   In these type of operations memory stage will be active and other stage will be similar to the rest of the instructions but in memory stage these instructions will either read or write from the memory.These instructions may also require execute stage when load,str with offset are used.

3. **B,BL,BNE(etc)**
   In this in the first stage and second stage as in all instructions just fetch decode happens,nothing happens in IF and Execute stage of this instruction when BNE reaches MEM stage CMP would have reached the final stage and we would have already got the result of comparison

in the execute stage hence we would accordingly flush the instructions/continue with inserted instructions in the pipeline.

4. **For 'B'**
   Firstly instruction is fetched and when 'B' instruction is decoded directly the instrcution after the label is fetched.

5. **For 'BL'**
   It is just similar to B instruction just that link register value is updated with pc+1.

6. **For CMP**
   In this first and second stage happen normally and in the third stage comparator is used to compare the two values when we reach the fourth stage in the comparator new instructions are fetched according to the result.

4 Testing

4.1 TestCases

1.
```
MOV r0 ,#4
MOV r1 ,#1100
ADD r2 , r1 , r0
LDR r3 , [ r2 ]
ADD r4 , r0 , r3
```
   Testcase with load-use and data hazard.

2.
```
MOV r3 ,#1
MOV r5 ,#111
MOV r6 ,#2
MOV r1 ,#12
SUB r2 , r1 , r3
ADD r12 , r2 , r5
SUB r13 , r6 , r2
ADD r11 , r2 , r2
```
   Testcase with Data hazard.

3.
   ```
   MOV r1,#1200
   MOV r0,#4
   STR r0,[r1,#20]
   LDR r2,[r1,#20]
   ADD r4,r2,r0
   SUB r8,r2,r1
   ADD r9,r4,r2
   ```

   Testcase with load-use and data hazard.

4.
   ```
   MOV r0,#1
   MOV r1,#100
    Loop:
   ADD r2,r2,r0
   ADD r0,r0,#1
   CMP r0,r1
   BNE Loop
   ADD r3,r2,r1
   SUB r4,r3,r2
   MUL r6,r3,r4
   ```

   Testcase with data and control hazard.

5.
   ```
   MOV r0,#4
   MOV r1,#1000
   STR r0,[r1]
   LDR r2,[r1]
   ADD r3,r2,r2
   SUB r4,r3,r2
   ADD r5,r4,r1
   STR r1,[r5]
   LDR r6,[r5]
   MOV r7,r6
   ```

   Testcase with load-use hazard.

6.
   ```
   MOV r0,#1
   MOV r1,#5
    Loop:
   ```

```
ADD r2 , r2 , r0
ADD r0 , r0 ,#1
CMP r0 , r1
ADD r3 , r2 , r2
SUB r4 , r3 , r2
SUB r5 , r4 , r1
LDR r6 , [ r5 ]
MOV r7 , r6
BNE Loop
ADD r7 , r7 , r1
SUB r8 , r7 , r5
```

Testcase with data and control hazard.

7.
```
MOV r0 ,#1
MOV r1 ,#4
Loop :
ADD r2 , r2 ,#4
ADD r0 , r0 ,#1
ADD r3 , r2 ,#1200
SUB r4 , r3 , r2
SUB r5 , r4 , r1
LDR r6 , [ r5 ]
MOV r7 , r6
CMP r0 , r1
BNE Loop
ADD r7 , r7 , r1
SUB r8 , r7 , r5
BL Loopp
ADD r5 , r7 , r8
ADD r9 , r5 , r5
Loopp :
ADD r10 , r10 ,#1
SUB r10 , r10 ,#1
SUB r11 , r10 , r13
CMP r10 , r11
BNE Loopp
ADD r12 , r10 , r11
```

Testcase with control and data hazard along with the use of link register.

8.
```
mov r0,#100
mov r1,#0
mov r2,#1
StoreNum:
Str r2,[r1,#1000]
add r2,r2,#1
add r1,r1,#4
sub r0,r0,#1
cmp r0,#1
bne StoreNum
mov r0,#100
mov r1,#0
mov r2,#0
LoadNum:
Ldr r2,[r1,#1000]
add r2,r2,#1
add r1,r1,#4
sub r0,r0,#1
cmp r0,#1
bne LoadNum
```

Testcase for iteration in an array using branch.

9.
```
ADD r0,r0,#0
ADD r1,r1,#1
loop:
add r2,r1,r0
mov r0,r1
MOV r1,r2
ADD r3,r3,#1
CMP r3,#5
BNE loop
ADD r10,r10,r10
MUL r11,r1,r2
```

Testcase for finding fibonacci at a given position.

5 Software Requirement

Following Software/Packages are Required for running this application.

1. GNU g++ Compiler for C++ 11
   For Compilation of C++ program