CS 341 Computer Architecture Lab

8 Stage MIPS Pipeline Simulator
(Hurray! Finally after writing around 18,000 lines of code, We did It!)

Project Report

Astha Agarwal (110050018)

Anmol Garg (110050020)

Rahul Singhal (110050023)

Ved Ratn Dixit (110050044)

Introduction - About the Project

1. We have built an 8 stage MIPS pipeline simulator.
2. We take a MIPS code file as an input, parse it for data and text sections and display the cyclewise execution for each instruction. The MIPS code for the instruction is shown on the left side of the display window.
3. The Instruction set includes simple and multi-cycle arithmetic instructions, branch instructions, memory instructions(load/store) and "jump and link" instructions.
4. We have provided an option for forwarding and fast branching as well. These can be turned on or off on runtime depending on the requirement.
5. We have simulated all kinds of forwarding, showing arrows appropriately between stages where forwarding is happening.
6. We have simulated stalls due to register dependencies and unavailability of resources. Those dependencies also we have expressed in gui with the help of arrows and stage label.

Class Description and Implementation

1. **Program Parser :**

    a. We have made a parser which takes a MIPS program as input, identifies the instructions and creates corresponding instruction objects. It also stores corresponding registers/values to the instruction objects.
    b. It also recognises comments and ignores them.
    c. We have tried to implement as robust design of the parser as possible. Right now it can deal with almost any kind of syntax error(incorrect format instructions/registers, incorrect label name etc)

d. After detecting errors it prompts user to correct the syntax error after showing the line number where error was detected.

2. Classes Implemented:

a. System Class:

i. This class represents the computer system with all its resources and properties common to all the components of the system.

ii. This class maintains a set of static variables which include register with their values and dependency statuses, stages with their states and other properties of the system whether hardware forwarding /fast-branching is enabled or not.

iii. The class is then inherited in all those classes which required synchronised access to these system variables. Analogical to actual computer architecture this class represents the base hardware common to all the components of the system.

b. Program Class:

i. An object of this class will be instantiated for every program. It will hold together all the tools mentioned below (Instruction, Register, Stage, Parser) and use them to simulate the execution. The main function will just instantiate an object of this class with the code file and run execute on it.

ii. It will take care of branch instructions by detecting change in program counter, after which a flush of the pipeline along with undoing of some register dependencies is required.

iii. In one clock cycle multiple instructions are executed. The order of execution here is crucial. Order must be in the reverse order of their stages, because stage i needs stage i+1 to be free in order to successfully perform its execution even if none of its register dependencies are busy. All this is ensured in the program class.

c. Instruction Class:

i. We have made an Instruction class (stage_number, address) to store the state of an instruction at a particular cycle. We implement specific instructions like add, subtract, li by making inherited subclasses for each one of them.

ii. The main execute function is overridden in each of the inherited classes as per their requirement. Same goes for other instructions which have different implementation in different instructions.

iii. Since we are dealing with pointers of instructions, we had to understand and implement deep copy through the concept of copy constructor and clone methods.

d. Register Class:

i. We have implemented a register class which stores the corresponding register value, and its state (i.e. whether it can be read at a moment or not).

ii. The register file is maintained as an array(32) of Registers common to all the Instructions at all times(this is done through the System class mentioned earlier)
iii. The register dependencies keep a track of the instructions which are blocking the readability of the register because they are going to change the value of the register in their later stages
iv. These dependencies is maintained as a list in each of the register classes and is manipulated to identify the instruction id's ultimately responsible for the register stalls which may follow in the instructions who need these registers as input.
v. Also in case of forwarding, another list is maintained to keep record of which instructions have forwarded the value of some register along with the time at which they forwarded it.
vi. This list is then manipulated when the value of a register is read to figure out if the register's value is forwarded or not , and if yes then which instruction forwarded it. This is needed to correctly show the forwarding arrow in GUI.

e. **Stage Class:**
   i. We have implemented a stage class to store information for each of the stages. This kind of represents the state of the program. These stages hold address of the instruction they are currently trying to execute.
   ii. The execution happens in the reverse order of the stage number of instruction therefore the order of stages is crucial as it transforms to a stage dependency when executing in the program class.

f. Memory Class:
   i. To simulate a MIPS like memory, we have also created a memory class, which is basically a vector of bytes (char) and can load and store words and bytes.
   ii. It also maintains a map of the label of data members to the address in which they are actually stored. This is how we deal with the labels user often gives to data members.

3. Graphics:

a. For GUI we have used openGl as it is not only an easily installable, independent and highly potent graphics module, it is also cross platform. This will facilitate any extension we plan to do in the future on any platform.

b. The graphics drawing program is kind of independent of the simulation going on in the backend because it won't change the state or modify anything regarding the simulation.

c. This program when given a set of instruction objects along with their states as they were after execution in the previous clock cycle, will draw appropriate objects depending on the instruction states.

d. The drawing includes a sidebar to show the instruction, stage label to show which stage is going on and whether it is stalled or not. By means of arrows which instruction is responsible for register stalls if any. is shown.

e. Also the forwarding arrows shown represent the forwarding going on between stages. There could be cases when multiple forwarding is happening directed to one instruction stage, in this case the forwarding which ultimately led the execution process to resume is shown.

f. This program will be called once after simulation of every clock cycle. Thus allowing us to run the simulation and draw it in steps of 1 clock cycle.

Instructions and Their Usage

Name	**Mnemonic**	**Operation**
Add	add	R[rd]=R[rs]+R[rt]
Add Immediate	addi	R[rt]=R[rs]+SignExtImm
Subtract	sub	R[rd]=R[rs]-R[rt]
And	and	R[rd]=R[rs]&R[rt]
And Immediate	andi	R[rt]=R[rs]&ZeroExtImm
Nor	nor	R[rd]=R[rs]|R[rt]
Or	or	R[rd]=R[rs]|R[rt]
Or Immediate	ori	R[rt]=R[rs]|ZeroExtImm
Xor	xor	R[rd]=R[rs]^R[rt]
Xor Immediate	xori	R[rt]=R[rs]^ZeroExtImm
Shift Left Logical	sll	R[rd]=R[rs]shamt
Shift Right Logical	srl	R[rd]=R[rs]shamt
Shift Right Arithmetic	sra	R[rd]=R[rs]>shamt
Shift Left Logical Var.	sllv	R[rd]=R[rs]R[rt]
Shift Right Logical Var.	srlv	R[rd]=R[rs]R[rt]
Shift Right Arithmetic Var.	srav	R[rd]=R[rs]>R[rt]

Set Less Than	slt	R[rd]=(R[rs]<R[rt])?1:0
Set Less Than Imm.	slti	R[rt]=(R[rs]<SignExtImm)?1:0
Branch On Equal	beq	if(R[rs]==R[rt]) PC=BranchAddr
Branch On Not Equal	bne	if(R[rs]!=R[rt]) PC=BranchAddr
Branch Less Than	blt	if(R[rs]<R[rt]) PC=BranchAddr
Branch Greater Than	bgt	if(R[rs]>R[rt]) PC=BranchAddr
Branch Less Than Or Equal	ble	if(R[rs]<=R[rt]) PC=BranchAddr
Branch Greater Than Or Equal	bge	if(R[rs]>=R[rt]) PC=BranchAddr
Jump	j	PC=JumpAddr
Jump And Link	jal	R[31]=PC; PC=JumpAddr
Jump Register	jr	PC=R[rs]
Move	move	R[rd]=R[rs]
Load Byte	lb	R[rt]={24'b0, M[R[rs]+ZeroExtImm](7:0)}
Load Word	lw	R[rt]=M[R[rs]+SignExtImm]
Load Immediate	li	R[rd]=immediate
Load Address	la	R[rd]=Address of label
Store Byte	sb	M[R[rs]+SignExtImm] (7:0)=R[rt](7:0)
Store Word	sw	M[R[rs]+SignExtImm]=R[rt]
Multiply	mult	R[rd]=R[rs]*R[rt]
Divide	div	R[rd]=R[rs]/R[rt]
Exit	exit	programOver=true

Learning Experience

1. The main motivation behind taking up this project was to accomplish the challenge of simulating an 8 stage MIPS pipeline, exactly the way it is done in machines. The real challenge was to implement all the parts of a MIPS machine and getting them to work in sync such that a given MIPS program runs the way it should really run.
2. In the process, we learnt MIPS grammar, and the art of writing a parser which takes into account syntax errors, comments and the actual code.

3. Another highlight was to use C++ Object Oriented Programming. We gained useful insights about class inheritance, static variables and virtual functions.
4. We have designed the instructions in such a way that we can add any other instructions easily. We can also add details to be passed on to GUI as and when required.
5. We also learnt how to manage a code that is approximately 18000 (a "wc -l *" in src folder gives a count of 17767) lines big, and get every part of it to run as desired.
6. We encountered major issues in implementation, which are listed below. Being able to solve each of them successfully was a learning experience of its own kind.

Problems Encountered

1. **Problem of Implementing Multi-cycle Instructions**

    a. We know that MIPS simulator runs code on its different stages simultaneously. Now, writing a code in c++, we did not have an option to implement it that way. Converting the functionality of simultaneous execution into that of a sequential c++ code was a real problem.
    b. What complicated the matters worse was the multi-cycle instructions like mult which had a multi-cycle stage of their own (MULT AND DIV ALU) which was independent of the EX stage of normal pipeline.
    c. This forced us to consider the dependencies between stages and made it impossible to just run the instructions in the order in which they were inserted.
    d. Finally we solved the problem by sorting the instruction queue on the basis of the stage they want to execute and executing them in that order. This also involved deciding the order between stages as that would transform directly into a stage dependency in our implementation.

2. **Problem of Register Stalls and Forwarding**

    a. Implementing forwarding and stalls was another such issue. How to keep note of which register was being written and how to make it available for read by other instructions in appropriate stage depending on whether forwarding is on or off, was another issue.
    b. Note that readability of a register could be blocked by multiple instructions, and it must not be read until all the dependencies are resolved. We solved this issue by maintaining a list of instruction Ids in the register class which are responsible for blocking the register. This gave us direct data in the register class itself to see whether a register is blocked or not and if yes, then by which instructions.
    c. Same goes for forwarding, it is possible that an instruction has to read to registers and both of them are being forwarded, then to show which instruction actually held the instruction back, we must keep a track of the time at which forwarding value was available.

d. We solved this issue by not only noting down what instructions forwarded a register but also at what clock cycle was their forwarded value available. This gave us means to manipulate the collected data and obtain whatever information required.

3. **Problem of Branching Instructions**

a. Implementing branching was also a major issue. We have also added the functionality of choosing fast branching, which computes the branch in ID stage itself.
b. Now once we know that the branch is to be taken or not, we must flush the pipeline and undo any side effects the few clock cycles of execution which shouldn't have taken place may have done.
c. Case 1: fast branching is enable. In this case the predicate will be calculated in ID so the subsequent instruction could at most be at IF2. Therefore there is no possibility of them blocking any readability. So all we need to do is flush the pipeline and remove the subsequent instructions from the pipeline.(This is precisely what we did)

d. Case 2: fast branching is disable. In this case the predicate will be calculated in EX so the subsequent instructions may have reached ID and blocked the readability of some register. There now when we'll flush the pipeline we also need to unstall that register. (We wrote a specialized function in the register class precisely for this and made it to work)

Programming Tools Used

1. OpenGL for GUI
2. C++ for simulation and algorithm implementation

Important Note

1. **We have designed this project in such a way that we can easily merge the functionalities of MARS (the mips editor that we used in lab) and WinDLX (the 5 stage pipeline simulator for windows), and create a new application that provide both the functionalities at the same time.**
2. Such an application would be a really nice tool to be used for academic and teaching purposes. No need to fire up multiple applications on different terminals and handle them simultaneously.
3. We are very **enthusiastic to extend this project to next level** and present this application as an open source application to public out there. Almost everything is implemented and the backend-frontend binding is so strong that converting this project to a product won't require much effort.