IIT Bombay

CS-305/341 Computer Architecture

8-Stage Pipeline Simulator

Submitted To:
Prof. Bernard L. Menezes
Professor
Computer Science Department

Submitted By :
Mayankya Medhe:
160050027
Maitrey Gramopadhye:
160050049
Shreyash Meena:
160050058
Neelesh Verma:
160050062
Suseendran Bhaskaran:
160050105

Contents

Project Report

Abstract

The goal of our final project is to simulate a 8 stage pipelined processor.Instruction pipelining is a technique for implementing instruction-level parallelism within a single processor. Pipelining attempts to keep every part of the processor busy with some instruction by dividing incoming instructions into a series of sequential steps (the eponymous ”pipeline”) performed by different processor units with different parts of instructions processed in parallel. The 8 stages in our pipeline are- IF, IS, RF, EX, DF, DS, TC and WB.

Keywords: 8 stage pipeline, MIPS

1. Introduction

In the history of computer hardware, some early reduced instruction set computer central processing units (RISC CPUs) used a very similar architectural solution, now called a classic RISC pipeline. Those CPUs were: MIPS, SPARC, Motorola 88000, and later the notional CPU DLX invented for education. They used a 5 stage pipelined processor. In order to increase the speed of the processor even further, it can be broken down into more stages. We have implemented a 8 stage pipelined simulator. The 8 stages are as follows- **IF, IS, RF, EX, DF, DS, TC and WB**.

2. Related Work

MIPS-SIM[5]- The following project simulates a MIPS R4000 CPU in Python with 5 stage pipeline. It takes a MIPS code as input in the form of a file. It then applies the 5 stage pipeline and output the detailed pipeline stages of each instructions including the number of stalls.

3. Architecture

3.1. Parser

The parser will analyze the data segments and populate the main memory and memory file. It will then analyze text segments and populate code file, removing indents and comments. Then it will read back code file in the buffer. Finally, it will convert all instructions into 32-bit strings and populate the instruction memory.

3.2. Preprocessing

The input is a 32 bit MIPS Code saved in a file. The MIPS code will be parsed as 3 files: registers, memory and code. All the 32 R and 32 F registers in the file and the memory locations will be initialized to 0. Then it will assign program counter (PC) to the code file.

3.3. 8 Stage Pipeline

IF.

- Checks if the PC is valid
- Attempts to fetch the instruction pointed to by PC from the I-cache.
- Forward the relevant variables to IF/IS.
- Update PC by 4 Bytes.

IS.

- Checks for No Operation condition and pipeline stalls
- Checks for branching condition in the previous cycle
- Instruction lookup can take up to 2 cycles.
- Instruction cache hit detection and I-Memory access in case of Cache Miss.
- I-cache update.
- Forward the relevant variables to IS/RF

RF.

- Checks for No Operation condition and pipeline stalls
- Branching Check.
- Instruction Decode
  - Decode opcode and Sign extend immediate.
  - Set ALU-control signals using funct and ALUop.
- Forwarding/Stall Check - Checks if the data of input registers is outdated and forwwards the data or stalls based on user argument for forwarding
- Register Fetch - Fetches the data from relevant registers and forwards to RF/EX register.
- In case of R-type instruction, calculates the type of operation to be performed by ALU in the EX stage.

EX.

- Checks for No Operation condition
- Branching check
- Shift immediate and compute branch target.
- Compute ALU outputs. This operation can take more than 1 cycles.
- Forward the relevant variables to EX/DF

DF.

- Checks for No Operation condition
- Gets data location to be read or written to, along with the data to be written to it.
- In case of load instruction, attempts to read from the D-cache and gets the cache-hit or cache-miss.
- In case of store operation, writes into D-cache.
- Forward the relevant variables to DF/DS.

DS.

- Checks for No Operation condition
- Memory lookup can take up to 2 cycles.
- In case of load instruction and cache-miss, gets data from main memory
- In case of store operation and write through policy (based on user argument), write into the main memory
- In case write back policy was followed and there was a replacement while writing into D-cache, write the data back into the main memory.
- Forward the relevant variables to DS/TC

TC.

- Checks for No Operation condition
- In case of cache-miss and load operation, update D-cache with data fetched from main memory.
- Forward the relevant variables to TC/WB.

WB.

- Checks for No Operation condition
- WB data =Read data or ALUoutput1 depending on MemtoReg.
- Write WB data to WB register.

3.4. Pipeline Representation

Presently, we are printing the output on terminal. We plan to build a Graphical User Interface that depicts Registers, Memory, Stall Related Information and Stage Diagram

4. Experiment

The input file will have a data and a text segment, they can be in whatever order the user wants. The parser will parse the file and generate the following files :

- **Main Memory file** : It is the picture of main memory and will contain the data residing in the main memory. The addresses will start from 0 and will go to a MAX value (size of main memory). We will just put the first data found in data segment in the address starting from 0, then the rest of data in the serial order follows as well. We are not leaving spaces between two consecutive data in the data segment, e.g. if user has used 2 variables in the data segment, say a and b, each of size 5 bytes, then in main memory, a will reside from 0 to 4 and b from 5 to 9; no spaces between them.
- **Memory Mapping file** : It is the mapping of the variables declared in the data segment, to their corresponding starting addresses, e.g. if the value of variable a in main memory file starts from address 10, then memory mapping file will contain *a : 10.* We are not storing the last address of a, as this is not required.
- **Parsed input file** : This will contain the user code, after removing comments, data segment, labels and indents.
- **Instructions file** : This file will contain the instructions encoded in 32 bits form. The encoding is done the same way as MIPS does it. The main python code will read this instructions file and store it in an array of strings.

5. Work Completed

We have designed the Parser and are able to convert the user code into MIPS 32 bit instruction format. We are able to create Memory and Register Files from the given MIPS-32 code. We have written code for all the 8-stages with forwarding supported as well. We have made separate classes for memory and caches. So the associativity and other factors can be changed as required. There will be proper stalls in case of hazards. We haven't tested our code yet, it is still in debugging stage. Right now, our parser is fully completed. It will generate some files as mentioned above.

6. Work Remaining

We have implemented very basic 8 stage-pipeline simulator. We are planning to add some more features to it as described below :

- **GUI interface** We haven't implemented the GUI interface yet, that will show all the stages to the user, in somewhat the same way as WINMIPS does it.
- **User input** Right now, there is no functionality for taking user input as WINMIPS also doesn't provide it (as far as we know from the assignments). But we are planning to take user input using 4 system calls : 2 for input and output of integers and 2 for input and output of strings. To implement this, we need to have a different encoding algorithm for these 4 syscalls as it is possible that it's encoding might collides with the encoding of our instructions.
- **Main memory** As mentioned earlier, there are no spaces in main memory between 2 consecutive variables declared in the data segment. It can lead to some problems, suppose the user changes some of it's variables and now they have different lengths. So we are planning to leave some spaces in main memory between 2 consecutive variables. But it's not a permanent solution, as it is possible that this space may also not be sufficient. In that case, if time permits, we will allocate memory to that variable anywhere in the main memory and note the starting address of this memory corresponding to this variable in memory mapping file. So, now in our memory mapping file, instead of a single starting address for each variable, we may have multiple starting addresses in a sorted order.

7. Conclusion

From this project we came to know how an 8 stage pipeline is different from a 5 stage pipeline. We also understood what happens in each and every stage of the pipeline. More importantly we understood the importance of teamwork and how it helped us achieve our common goal. We haven't tested yet, but we are hoping that 8-Stage pipeline will take less time than a 5-Stage pipeline. We are inferring this from what we read in the resources we cited. 8-Stage pipeline gives more power to the user by adjusting the number of branch-delayed slots but also increases the probability of structural hazards due to increase in number of stages.

8. Challenges Faced

The first task was to parse the code given by the user. There were several challenges that we faced when we actually did the parsing. They are as listed below :

- Since we are converting the user code into 32 bit instructions, there are no standard conversions (in 32 bits) for *Pseudo Instructions*. In real processors they are further decomposed into basic instructions (atomic instructions). To handle this, we have actually encoded pseudo instructions into special 32 bit instructions of our own that don't matches with encoding of other atomic instructions.

- Whenever there is a jump instruction to a label, we need to jump to the instruction just after the label. Whenever we find a branch instruction (and branch is taken), we then look at 0 to 25 bits of that instruction and start executing the instruction that has the program counter same as the decimal conversion of these bits.

- We are still figuring out how to implement syscalls in our code as there are a total of 12 syscalls. There has to be some special signal (or decoding of syscall instruction), so that in our python code we can figure out if the current instruction is a syscall. For that we also need to have v0 and a0 registers. But in WINMIPS, we don't have such registers nor these syscalls. So right now we have left it.

Now the next task was to implement all the 8 stages. The difficulties that we faced here are as follows :

- The first thing was to divide the whole process of pipelining into 8 stages, functionality in each stage. We looked at many papers for this. Finally we found a paper[1] that had a good content of the functionalities of the 8 stages. We have modified it at appropriate places to meet our requirements.

- There is a set of registers between consecutive pipeline stage to hold the values coming from previous stage and pass it on to the next stage. The problem here was to how to implement these registers functionalities. We have made 7 registers (7 strings because of 8 stages) each having 128 bits, initialized all to 0. Then we used these strings to hold the values between the pipeline stages.

- There can be hazards as well in the user code. We added some separate functionality to detect the hazards and stalling the instructions. For this we have made a separate class for stalls.
- We want the user to have option of forwarding as well. The problem was how to pass the values from one stage to the other as done in forwarding. To implement this, we took help of registers in between pipelines and passed values from them.
- Whenever there is a branch instruction, and the condition to take the branch is true, then we need to jump to that instruction and stopping other instructions just after the branch instructions. For this we designed a separate class for branch.
- The only policy that is being used right now for cache replacement is approximate LRU. We haven't implemented any other policy. But we want the user to pick the replacement policy. So we expect to implement this feature in the future.
- We also have a some sort of cache simulation in our code. We have made a separate class for each cache elements, i.e., cache entry, cache line, cache set, cache; each one using the objects of it's previous class. This was indeed a challenging task for us to think of designing a cache. We still were able to implement only l1 cache.
- Right now, we are just using just one read-back policy, i.e., whenever something is written to main memory, we also write that to D-cache. We haven't implemented any other policy here yet as we were not able to think how to implement them.

9. Acknowledgment

We would like to thank Prof. Bernard L. Menezes for giving us this opportunity to work on project. We would also like to thank all our Teaching Assistants (TAs) for their support and motivation.

10. Contribution

- **Mayanka Medhe:** Implemented the ALU functionality to handle all the ALU operations, maintaining control signals to multiplexers, and designed the WB stage.
- **Maitrey Gramopadhye:** Build all the classes for memory and cache and designed DF, DS and TC stages
- **Shreyash Meena:** Designing the IF and IS Stage, helped in making Parser.
- **Neelesh Verma:** Designed EX and RF stages, helped in making parser.
- **Suseendran Bhaskaran:** Designing the parser, helped in designing abstract for other stages.

11. References

[1] Otto Chiu, Charles Choi, Teddy Lee, Man-Kit Leung and Bruce Wang.
8-Stage Deep-Pipelined MIPS Processor
`https://people.eecs.berkeley.edu/~kubitron/courses/cs152-S04/projects/submit/project1_report6.pdf`

[2] Technical Manual Reference
Arm11 MPCore Processor
`http://infocenter.arm.com/help/topic/com.arm.doc.ddi0360e/DDI0360E_arm11_mpcore_r1p0_trm.pdf`

[3] TIan Finlayson, Gang-Ryung Uh, David Whalley and Gary Tyson
An Overview of Static Pipelining
`https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=arnumber=6086519`

[4] P. Bernardi, R. Cantoro, L. Ciganda, B. Du, E. Sanchez, M. Sonza Reorda, M. Grosso, O. Ballan
On the Functional Test of the Register Forwarding and Pipeline Interlocking Unit in Pipelined Processors
`https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=arnumber=6926101`

[5] Joel Tanzi
MIPS-Sim
`https://github.com/jtanzi/MIPS-Sim`