Non-Pipelined Processor

ECE 5367 Term Report

Faheem Quazi, Arien Stanley, Gabriel Torres
ECE 5367
University of Houston
Houston, Texas

***Abstract*— In order to test software for a new platform or processor, one might want to emulate said platform before making a purchase, since hardware is quite expensive. This provides the purpose for this project: to write an emulator for non-pipelined RISC processors, specifically those which use the MIPS instruction set. This emulator was produced with some caveats, namely the lack of byte-specific memory and limitations on overall program size. This project can eventually be expanded to fix those problems, and given more development resources, could lead to emulation of actual RISC-based systems, such as the PlayStation 1.**

Keywords—MIPS; risc; emulation; cpu; University of Houston; ECE 5367;

I. INTRODUCTION

In the time when computer devices were operating in the Megahertz range, it was very difficult to emulate other systems because the translation and processing of the instructions would be extremely slow if not impossible. Now with modern processors and speeds much faster, it has become possible to completely emulate older systems, not just the processors but also the audio systems, and graphics. This is primarily possible due to these older CPUs running at significantly lower frequencies, so even though the translation and processing of native instructions is happening, it's fast enough that it would be considered native speed on the older system. In the case of this project, the MIPS instruction set provides a more challenging, yet simple enough architecture to build a processor around compared to x86 or x64. Since access to processor building hardware is difficult to obtain, the goal of this project was to emulate a RISC-style non-pipelined processor which operates on the MIPS instruction set.

II. METHODOLOGY

A. Defining the Scope

After having defined the scope of the project we outlined the following objectives:

- To be able to identify and decode a subset of the MIPS instruction set
- To provide a system of executing said MIPS instructions, utilizing a method which will allow for simple expansion to the full instruction set in the future.
- To emulate the behavior of a non-pipelined CPU within the code, following the standard steps of Fetch, Decode, Execute, etc.

The professor provided us with a format for initializing CPU registers and memory, and this format includes an area for storing instructions. Namely, the section consists of binary MIPS instructions separated by newline characters. These instructions are stored in plaintext (i.e., not a binary format).

B. Code

The code can be divided into three main components, the main function, the file_load function, and the CPU function. While this report will attempt to provide a detailed analysis of the code, the actual code contains comments (as visible in some figures), and those would provide the best reference to understand how the program works.

a. the main function

The "main()" function is where execution of the program begins. It starts by asking the user to type in the input file name (which is the text file that contains all the data of the registers, the memory, and the code instructions) by printing the prompt "Input File:" to the screen and storing the user-entered input file name into a string (fni). It then asks the user to type in the output file name (which is the test file that will be used to store the clock cycles information, as well as the updated register values and values stored in the memory) by printing the prompt "Output File:" to the screen and storing the user-entered output file name into a string (fno). The program then runs the "file_load()" function, which we created, using the input file name (fni) as an input to extract all the information of the registers, values stored in memory, and the code instructions. More detail on this function later. Finally, the program will run the "cpu()" function, another function we created, to simulate the functionalities of the CPU of a MIPS processor such as decoding and executing instructions.

```c
int main(int argc, char const *argv[])
{
    /* set initial values */
    REGISTERS[0] = 0;

    /* Get settings from user */
    char* fni = (char *)malloc(sizeof(char) * 32);
    char* fno = (char *)malloc(sizeof(char) * 32);

    // Input
    printf("Input File: ");
    scanf("%s", fni);
    // Output
    printf("Output File: ");
    scanf("%s", fno);

    /* Load the file specified */
    printf("Load Program\n");
    file_load(fni);

    /* Run CPU */
    printf("CPU Run\n");
    cpu(fno); // This will run until code execution completes

    printf("Done\n");
    return 0;
}
```

Figure 1: The Main Function

```c
// convert binary string to integer
int fromBinary(const char *s) {
  return (int) strtol(s, NULL, 2);
}

/* Handle loading of input file */
void file_load(char* input_file) {
    char *line = (char *)malloc(sizeof(char) * 48); // Line read from input file
    char dsc1, dsc2, dsc3; // Misc. characters read from line to be discarded
    FILE *fin; // Input file pointer
    int i1, i2; // Used for register number and content, memory location and content
    fin = fopen(input_file, "r"); // Open input file

    fgets(line, 48, fin); // Read the first line from file

    int j = 0; // Used to count registers read from file
    while(strcmp(line, "MEMORY\r\n")!=0 && j++ < 32) {…

    while(strcmp(line, "CODE\r\n")!=0){…
    // Read lines to the end
    while(!feof(fin)) {
        // Process CODE section
        if (strcmp(line, "CODE\r\n")!=0) {
            // Get instruction from line
            CODE[CODE_LENGTH] = fromBinary(line);
            ++CODE_LENGTH;
        }

        fgets(line, 48, fin); // Read the line from file
    }
    fclose(fin); // Close input file
}
```

Figure 2: The file load function, code section decoding

b. The file_load function

The "file_load()" function is the function that will be used to extract all the information from the input file and store its contents into three separate arrays: REGISTERS, MEMORY, and CODE. The function will first begin by opening the input file that the user typed in, and then reading in the first line of the input file into a string (line), which should read "REGISTERS\r\n", indicating the first portion of the input file that contains the data of the registers. The function then uses a while loop that reads in every line of the input file into the same string (line) and extracts the information of the registers and stores its contents in the REGISTERS array until the line read into the string reads "MEMORY\r\n", which indicates that the program has finished extracting the register contents and has reached the memory portion of the input file. The function then uses another while loop that reads in every line of the input file into the string and extracts the information of the memory and stores its contents in the MEMORY array until the line read into the string reads "CODE\r\n", which indicates that the program has finished extracting the memory content and has reached the code instructions portion of the input file. The function then uses one final while loop that reads in every line of the input file into the string and extracts all the code instructions and stores them in the CODE array until the end of the file is reached, which indicates that the program has finished extracting everything from the input file and storing them into their correct arrays.

c. The cpu function

The cpu function is what emulates the CPU, and takes the instructions from the CODE array, and executes them. The program has been divided up roughly into the steps of CPU execution as described in class, namely Instruction Fetch and Decode, ALU and Execution. The code takes liberties with this execution to allow for expandability in the form of adding more instructions.

The first part of our process is to fetch the instructions from the code that we are given and decode it into its specific instruction type. To do this, we used a while loop that runs until the program counter is greater than the code length. With a simple switch statement using the opcode, we can set cases for opcode values, and separate the specific lines of instructions into R, I and J. Using the case of zero for R, two or three for J and the default case being I type, since there are over a dozen different opcodes. Once we have decoded the opcode, and it enters the switch statement, the instruction type is then set into a variable included in the struct, so we are able to reference it in the next section for the ALU where we will execute our instructions.

Once we have determined the type of opcode, the decoding process begins. We store the instruction into a special union variable. This union consists of a raw integer (where the instruction to be decoded is inserted), as well as three unique structs. Each struct represents the different type of instruction present in MIPS: R, I, and J. Figure 3 below provides a screenshot of these structs. By storing the instruction in this union and utilizing the decoded opcode/instruction type from the previous step, we can access the unique fields of each instruction type by name, to greatly simplify the execution of instructions in the next step.

```
/* Instruction Type Definitions */
typedef struct  {
    unsigned int funct:6;
    unsigned int shamt:5;
    unsigned int regd:5;
    unsigned int regt:5;
    unsigned int regs:5;
    unsigned int opcode:6;
} instr_type_R;

typedef struct {
    int imm:16;
    unsigned int regt:5;
    unsigned int regs:5;
    unsigned int opcode:6;
} instr_type_I;

typedef struct {
    int imm:26;
    unsigned int opcode:6;
} instr_type_J;

/* store the current instruction here */
union {
    unsigned int raw;
    instr_type_R R;
    instr_type_I I;
    instr_type_J J;
} instr;
```

Figure 3: Using Structs and Union to process instructions

Now that we have decoded the instruction type in the previous instruction fetch and decode step, we are now able to plug this instruction type into a nested switch statement that will then using the function or opcode to carry out the required instruction. We have a switch statement for each instruction type, so we can control where the code is taken, and what instruction is executed. For R-type instructions, we are required to use the function value, as the opcodes for R-Types are all the same. For J-type and I-type instructions we can use the opcodes, and this will then carry out the correct instruction to match the opcode value given in the previous step. We then have print statements to print whether read write memory or write back register were set to high, and print that next to the instruction during the final output.

III. CONCLUSION AND FUTURE GOALS

The goal of this project was to successfully create a simulation of the un-pipelined processor that decodes given instructions into its instruction type and correctly carries out the arithmetic operation that was given. The project was successful, and the code was able to sort each instruction into their instruction type and carry out the instructions required by the project proposal and provide an output which matches the expected output given the instructions provided.

Looking to the future our team worked to complete the complete MIPS instruction set, but we did not get the chance to add every single MIPS instruction. Missing from our project were all the multiply instructions, the branch instructions including: BREAK, MFC0, MTC0, SYSCALL. From the memory instructions we were only able to currently add LW and SW.

This leads into our next goal of manipulating our code to correctly handle memory. Currently our memory is accessed in four-bit blocks, as expected our code has issues when you try to access memory bits that are not multiples of four. We were unable to complete some of the MIPS instructions as they require memory to be accessed by the bit, currently our code is not able to handle this, so we cannot correctly access the data in the way the instruction requires of us.

Along with our change to memory that we would like to make in the future, we also would like to add the ability to read actual RISC/MIPS code to our un-pipelined processor. Currently we are only able to read those sample text files; our code would not be able to handle actual MIPS code in its raw form. We would also like to add threading if possible, to our project, in an effect to reduce lag time and increase performance, setting up these subroutines to help us handle each of the instructions in a more efficient manner.

Lastly, in relation to our current project we would like to explore PSI emulation using our code, as PS1's inner code is based on MIPS architecture. We believe upon completion of the other goals we have, that we should be able to execute a working PS1 emulator. Unrelated to our unpipelined processor, we would like the ability to also explore the pipelined processor, as it's stated that the pipelined processor is more efficient, being able to run multiple instructions at the same time. This makes it faster than the un-pipelined processor which handles instruction one at a time, in a step-by-step process.

ACKNOWLEDGMENT

[1] A special thanks to Dr. Abu Baker for teaching us about CPU design and the fundamentals around how a CPU works, and providing inspiration and guidance for this project