Lab-5 (Pipeline Stall Detector/Simulator)

CS22BTECH11061

November 7, 2023

1 Coding Approach

The program is implemented in C for Pipeline Stall Detection and consists of several functions for this purpose. Here is an overview of the coding approach:

1.1 Struct Instruction data-type

The program defines a data structure called `Instruction`, which stores the instruction string and the number of NOP instructions required with and without forwarding.

Listing 1: struct Instruction

```
1    typedef struct Instruction{
2        char string[33];
3
4        int nop_without_forwarding;//nops before this instruction
5        int nop_with_forwarding;
6
7    } Instruction;
```

1.2 Parsing the Instructions

The program uses functions like `translate_register_name()`, `stop_at_character()`, `get_rs1_rs2()`, and `get_rd()` to parse the instructions, extract the registers involved, and determine the type of instruction (load, store, or other), as well as string.h library functions like `strtok()`.

1.2.1 `get_rd()`

The function `get_rd()` uses `strtok()`, `stop_at_character()` to parse the string for rd register name and then uses `translate_register_name()` to translate the register name from its alias.

Listing 2: get_rd function

```
1    int get_rd(char rd[10], char original_instruction[33]){
2        char instruction[33];
3        strcpy(instruction, original_instruction);
4
5        char *token = strtok(instruction, " ");//token contains
             instruction name
6        char name[10];
7        strcpy(name, token);
8        token = strtok(NULL, " ");//now it contains rd
9
10       strcpy(rd, token);
11       stop_at_character(rd, ',');
12
13       translate_register_name(rd);
14
```

```
        if(name[0] == 'l'){
            return 1;//for load
        }
        else if(name[0] = 's' && strlen(name) == 2){
            return 2;//for store
        }
        else{
            return 0;
        }
    }
```

1.2.2 get_rs1_rs2()

In this function similar approach as `get_rd()` is used.

Listing 3: get_rs1_rs2 function

```
    void get_rs1_rs2(char rs1[10], char rs2[10], char
       original_instruction[33]){
        char instruction[33];
        strcpy(instruction, original_instruction);//original inst
           passed by ref, so cannot make changes

        char *token = strtok(instruction, " ");//token contains
           instruction name
        char name[10];//instruction name
        strcpy(name, token);

        token = strtok(NULL, " ");//now it contains rd

        char rd[10];//dest name
        strcpy(rd, token);
        stop_at_character(rd, ',');

        token = strtok(NULL, " ");//now it contains rs1

        strcpy(rs1, token);
        stop_at_character(rs1, ',');

        for(int i=0; i<strlen(rs1); i++){//for load or store
            if(rs1[i] == '('){
                stop_at_character(rs1 + (i+1), ')');//end string at ')'

                char buffer[10];
                strcpy(buffer, rs1);//for undefined behaviour if any
                strcpy(rs1, buffer+(i+1));
                break;
            }
        }

        token = strtok(NULL, " ");//contains rs2 if it exists else null

        if(token != NULL){
            strcpy(rs2, token);
        }
        else{
            strcpy(rs2, "l/s");//for load or store
        }

```

```
        stop_at_character(rs2, '\n');//get rid of newline character

        if(name[0] == 's' && strlen(name) ==  2){// for s both of this
            are sources and not dest
            strcpy(rs2, rs1);
            strcpy(rs1, rd);
        }

        translate_register_name(rs1);
        //printf("rs1 = %s\n", rs1);
        translate_register_name(rs2);
    }
```

1.2.3 translate_register_name()

`translate_register_name()` is used to translate from callee convention to names like x0, x1, ..,etc. It is implemented by comparing strings and reassigning them as needed. You can see the Implementation in the c code.

1.3 get_instructions_with_stalls()

To detect data hazards, function examines the dependencies between instructions. It checks for hazards between the current instruction and the previous two instructions. If a hazard is detected, function inserts NOP instructions accordingly.

1.3.1 Without Forwarding

For all instructions other than store, there can be data hazards for instructions which come after them. Which need two NOPS or instructions between those instructions to resolve. So we add NOPS accordingly.

1.3.2 With Forwarding

There can be a data hazard after load instructions, which need 1 NOP or instruction gap to resolve. We add nops accordingly.

Listing 4: get_instructions_with_stalls function

```
    void get_instructions_with_stalls(int IC, Instruction instructions[
        IC]){

        for(int current=0; current<IC; current++){
            if(current == 0) continue;//no stalls before first
                instruction

            //getting rs1 and rs2 for current instruction
            char rs1[10], rs2[10];

            get_rs1_rs2(rs1, rs2, instructions[current].string);

            char prev_rd[10];
            int type = get_rd(prev_rd, instructions[current - 1].string
                );

            //when prev_rd is not x0
            //prev instruction is not store
            //rs1 or rs2 uses prev_rd
            if(strcmp(prev_rd, "x0") != 0 && type != 2 &&(strcmp(rs1,
                prev_rd) == 0 || strcmp(rs2, prev_rd) == 0) ){
```

```
                instructions[current].nop_without_forwarding = 2;//nop
                    for prev instruction withoud forwarding

                //for load
                if(type == 1){
                    instructions[current].nop_with_forwarding = 1;
                }
            }
            else{//only when no hazard in prev instruction we check for
                prev to prev
                if(current == 1) continue;//no prev to prev inst
                    present in this case
                if(instructions[current-1].nop_without_forwarding > 0)
                    continue;//more than equal to 2 inst gap already

                char prev_prev_rd[10];
                get_rd(prev_prev_rd, instructions[current - 2].string);

                //when match and not store inst
                if(type != 2 && (strcmp(rs1, prev_prev_rd) == 0 ||
                    strcmp(rs2, prev_prev_rd) == 0) ){
                    instructions[current].nop_without_forwarding = 1;
                }
            }
    }
}
```

1.4 Printing the Solution

The `print()` function prints the nops according to the values stored in `Instruction` data type. While printing the instructions and nops, it also calculates the rquired cycles and prints it.

Listing 5: get_rs1_rs2 function

```
    void print(int IC, Instruction instructions[IC]){

        int cycles = IC+4;
        printf("\nNo Forwarding : \n\n");
        for(int i=0; i<IC; i++){
            int nops = instructions[i].nop_without_forwarding;
            cycles += nops;

            for(int j=0; j<nops; j++){
                printf("nop\n");
            }

            printf("%s", instructions[i].string);
        }
        printf("\nCycles = %d\n", cycles);
        printf("\n");


        cycles = IC+4;
        printf("Forwarding : \n\n");
        for(int i=0; i<IC; i++){
            int nops = instructions[i].nop_with_forwarding;
            cycles += nops;

            for(int j=0; j<nops; j++){
```

```
                printf("nop\n");
            }

            printf("%s", instructions[i].string);
        }
        printf("\nCycles = %d\n", cycles);
        printf("\n");
    }
```

2 Conclusion

The program successfully identifies and resolves data hazards in the assembly instructions by inserting appropriate NOP instructions. It offers solutions both with and without data forwarding, providing insights into the impact of forwarding techniques on the pipeline execution. The program is tested for correctness with the provided test input files.