16 MAy, 2024

Documentation for the Computer Architecture Project

1 Introduction

This document is a report for the Computer Architecture (CA) project. The project aims to simulate a fictional processor design and architecture using C. The project at hand was one of four project packages offered. The project packet of intrest is the Double McHarvard with cheese circular shifts.

This package is using a Harvard architecture with one data bus and one instruction bus. The processor has a `8-bit` data bus and a `16-bit` instruction bus, while all the registers are `8-bits`. To elaborate, the processor has access to `64` General Purpose registers, a single status register, and a single program counter. Furthermore, the processor has a `8-bit` ALU.

2 Methodology

The project was implemented using `C` and was designed to be as modular as possible. To expand on this, the entire project consists of multiple header files, each containing a specific part of the project, and a single `main.c` file that contains the main function. The project was divided into three main parts, which will be explained in detail below:

- The Memory Architecture.
- The Instruction Set Architecture.
- The Data Path.

2.1 Memory Architecture

The memory architecture was further divided into three parts to ensure modularity. Each part was implemented in a separate header file. A brief description will be provided below:

- The Data Memory.
- The Instruction Memory.
- The Register File.

2.1.1 Data Memory

The data memory was implemented using a pointer to an array of `8-bit` integers. To be more specific, the data type of the pointer was `uint8_t`. The data memory was implemented in the `data_memory.h` header file. Below is a brief description of the functions implemented in the `data_memory.h` header file:

- Init Data Memory: This function 'callocs' memory for the data memory and returns a pointer to the allocated memory. The function uses the `calloc` function instead of the `malloc` function to ensure that the allocated memory is initialized to zero.

- Kill Data Memory: This function 'frees' the memory allocated for the data memory. The function takes a pointer to the data memory as an argument and 'frees' the memory allocated for the data memory.

- Set Data Memory: This function sets the data at a specific address in the data memory. The function takes a pointer to the data memory, the address of the data to be set, and the data to be set as arguments.

- Get Data Memory: This function returns the data at a specific address in the data memory. The function takes a pointer to the data memory and the address of the data to be retrieved as arguments.

- Copy Data Memory: This function copies the data memory. The function takes a pointer to the data memory as an argument and returns a pointer to the copied data memory using deep copy.

- Pretty Print Data Memory: This function prints the data memory. The function takes a pointer to the data memory as an argument and prints the data memory in a human-readable format. For all consecutive addresses with `0x00` data, the function prints the address range instead of printing each address individually.

- Pretty Print Diff Data Memory: This function prints the difference between two data memories. The function takes two pointers to the data memories as arguments and prints the difference between the two data memories in a human-readable format. If the data at a specific address is different between the two data memories, the function prints the address and the old and new data. If there is no difference between the two data memories, the function prints 'No Difference'.

2.1.2 Instruction Memory

The instruction memory was implemented using a pointer to an array of `16-bit` integers. To be more specific, the data type of the pointer was `uint16_t`. The instruction memory was implemented in the `instruction_memory.h` header file. For the sake of brevity, the functions implemented in the `instruction_memory.h` header file will not be described in detail as they are identical to the functions implemented in the `data_memory.h` header file with the exception of the data type of the pointer, and the lack thereof of the *Pretty Print Diff Data Memory* function.

2.1.3 Register File

The registers were implemented using a pointer to a structure that contained an array of `8-bit` integers for the general-purpose registers, an `8-bit` integer for the status register, and an *8-bit* integer for the program counter. The register file was implemented in the `register_file.h` header file.

To be as specific as possible, the structure used to represent the registers was defined as follows:

```
typedef struct {
  uint8_t general_purpose_registers[64];
  uint8_t status_register;
  uint8_t program_counter;
} RegisterFile;
```

Finally below is a brief description of the functions implemented in the `register_file.h` header file:

- Init Register File: This function initializes the register file, and returns a pointer to the allocated memory. This function is identical to the *Init Data Memory* function with the exception of the data type of the pointer.
- Kill Register File: This function 'frees' the memory allocated for the register file. This function is identical to the *Kill Data Memory* function with the exception of the data type of the pointer.
- Set Status Register flag: This function sets a specific flag in the status register. The function takes a pointer to the register file, a character representing the flag to be set, and a boolean representing the value to set the flag to as arguments.
- Get Status Register flag: This function returns the value of a specific flag in the status register. The function takes a pointer to the register file and a character representing the flag to be retrieved as arguments.
- Copy Register File: This function copies the register file. This function is identical to the *Copy Data Memory* function with the exception of the data type of the pointer.
- Pretty Print Register File: This function prints the register file. This function is identical to the *Pretty Print Data Memory* function with the exception of the data type of the pointer.

2.2 Instruction Set Architecture

The instruction set architecture was implemented using a set of header files, each containing a specific stage in the execution of an instruction. A brief description of each stage will be provided below:

- The Fetch Stage.
- The Decode Stage.
- The Execute Stage.
- The Assembly Code Parser*

2.2.1 Fetch Stage

The fetch stage was implemented in the `fetch.h` header file. The fetch stage is responsible for fetching the instruction from the instruction memory and incrementing the program counter.

2.2.2 Decode Stage

The decode stage was implemented in the `decode.h` header file. The decode stage is responsible for decoding the instruction and extracting the opcode and operands and creating a structure to represent the instruction which will be denoted as a **Process Control Block** (PCB). The PCB was defined as follows:

```
typedef struct {
  uint8_t opcode; // 4 bits
  uint8_t operand1; // 6 bits
  uint8_t operand2; // 6 bits
  uint8_t R1; // 8 bits
  uint8_t R2; // 8 bits
  uint8_t IMM; // 8 bits
  bool is_R_format; // 1 bit
} PCB;
```

The decode stage had two functions: the `decode` function and the `pretty_print_pcb` function. The `decode` function takes a pointer to the instruction memory and a pointer to the register file as arguments and returns a pointer to the PCB. The `pretty_print_pcb` function takes a pointer to the PCB as an argument and prints the PCB in a human-readable format.

2.2.3 Execute Stage

The execute stage was implemented in the `execute.h` header file. The execute stage is responsible for executing the instruction represented by the PCB. The execute stage had a single function called `execute` which takes a pointer to the PCB and a pointer to the register file as arguments and returns nothing.

The execute function was implemented using a switch statement that switches on the opcode of the PCB. The switch statement calls the appropriate function based on the opcode of the PCB, and handles the updating of the status register if necessary.

2.2.4 Assembly Code Parser

This is the only header file that is located in the `ISA` directory but is not part of the instruction set architecture. The assembly code parser was implemented in the `parser_asm.h` header file. This header file is responsible for parsing the assembly code line by line. For each line of assembly code, the parser tokenizes the line and converts the tokens to an instruction in the form of a `uint16_t` integer. The parser then writes the instruction to the instruction memory. Finally, as the parser reaches the `EOF` of the assembly code file, it appends to the instruction memory a `0xf000` instruction to signify the end of the program and a halt instruction. Once the parser has finished parsing the assembly code, it returns a pointer to the instruction memory.

2.3 Data Path

The data path was implemented in the `main.c` file. The data path is responsible for fetching the instruction from the instruction memory, decoding the instruction, executing the instruction, and updating the register file and data memory accordingly.

The main function first initializes the data memory, instruction memory, and register file. The main function then calls the assembly code parser to parse the assembly code and write the instructions to the instruction memory. Then the main function enters the main loop of the data path, which will be described in detail in the next paragraph. Finally, the main function frees the memory allocated for the data memory, instruction memory, and register file, And prints the final state of the data memory and register file.

The main loop of the data path consists of three stages: the fetch stage, the decode stage, and the execute stage. All these stages are occurring within the same clock cycle. The main loop keeps track which instruction should be fetched, decoded, or executed by using the program counter and deviations from it. Thus it would take exaclty $3 + (n - 1)$ clock cycles to execute the program where n is the number of instructions in the assembly code. This holds if there are no jumps or branches in the assembly code.

To elaborate more on the algorithm that made this level of concurrency possible, the main loop has within it a
i variable that is firstly initialized to zero. With every iteration of the main loop, the `i` variable is incremented by one. The `i` variable is then used to determine whether a decode or execute stage should be executed this iteration or not. The algorithm is as follows:

Algorithm 1: The concurrency algorithm

```
Input: Instructions ∈ ℤ^max
Output: NULL
i, HALT ← 0
input_for_decoder ← NULL
input_for_executor ← NULL
while HALT ≠ 1 do
    if i ≥ 2 then
        execute(input_for_executor)
        if input_for_executor → opcode = 0xF then
            HALT ← 1
        end
        if input_for_executor → opcode = BRANCH then
            i ← 0
            PC ← input_for_executor → IMM
        end
    end
    if i ≥ 1 then
        input_for_executor ← decode(input_for_decoder)
    end
    if i ≥ 0 then
        input_for_decoder ← fetch_instruction(PC)
    end
end
```

As can be seen from the algorithm above, whenever a branch instruction is encountered, the `i` variable is reset to zero and the program counter is updated to the immediate value of the branch instruction. Furthermore, the algorithm also checks for the `HALT` instruction and terminates the program if it is encountered.

3 Results

The project was tested using a set of assembly code files that were designed to test the functionality of the processor. The first test was a simple addition test that added two numbers and stored the result in a register. The second test was a factorial test that calculated the factorial of a number and stored the result in memory. The third test was a cumulative sum test that calculated the cumulative sum of a number and stored the result in memory.

The first test was designed to insure that the processor did manage to do the assembly code with in the estimated number of clock cycles. The second test and the third test were designed to test the logic of the processor, in particular the branching and jumping logic. The tests were successful and the processor was able to execute the assembly code correctly.