Project Plan module 5
Design of JFEGS, a virtual CPU

Joost Buursink *s2790882*
Fabian Widlund *s3100235*
Emil Imhagen *s3108139*
Guus Branderhorst *s2795132*
Stach Redeker *s2758695*

Contents

I. Introduction

The aim of this project is to design and realize a rudimentary processor on the DE1-SoC board. We discuss the set criteria in this project plan and elaborate on our proposed concepts and ideas.

A. Criteria

Our processor should comply with a number of criteria. In no particular order, these are: [1]

1) The processor has to be a 1-address machine (accumulator) or a 2-address machine.
2) The processor shall have a separate entity-architecture for the data path.
3) The processor shall have a separate entity-architecture for the controller. This can be a finite state machine (FSM).
4) The processor shall have a test pin that can be used for debugging.
5) The main memory is byte-addressable and aligned addressing is used. The main memory may be defined in the FPGA itself, and not on (external) SRAM. A separate entity-architecture for main memory is required.
6) After resetting the first instruction is read from the main memory at address 0.
7) The processor shall be accompanied by a simple example program.
8) The instruction set shall consist of approximately 10 instructions. The instruction set shall at least contain instructions for *branch always*, *branch on a status bit*, *load from main memory*, and *store to main memory*.
9) For the processor, the timing information shall be given (INTEL/Quartus timing analyzer).

B. Name

The working name of our processor is *JFEGS*, which is an acronym consisting of the first letters of the creators' names.

II. Application

Although we do not aim to design an Application Specific Integrated Circuit (ASIC), it helps if we take a single application as a starting point. The application will be described by our example code (requirement 7) that is able to run on the processor. Of course, we intend to make our instruction set as broad as possible to facilitate executing other processes on our system.

We opt for taking the calculation of the Fibonacci sequence as the main application. The Fibonacci sequence is widely used in (the programming) industry to validate hardware and software applications. In addition to that, The Fibonacci sequence is well-known and can be printed using a limited instruction set, as will be discussed later. We created the following Fibonacci function in MIPS assembly:

```
.begin
.org 0
add %r0, %r0, %r1
add %r0, 1, %r2
fun: add %r1, %r2, %r3
add %r0, %r2, %r1
add %r0, %r3, %r2
ba fun
.end
```

III. Instruction set

We can use the aforementioned code to compose a set of required instructions. Observe that we need at least an *add* instruction, as well as *branch always* instruction. Furthermore, we need a register with a permanent 0 value for this script to work.

It could be useful to display the results of our code on one or more of the seven segment displays that are present on the DE1-SoC board. Hence, we shall also implement a display instruction. Taking into account the instructions required by the Fibonacci application, some nice-to-have instructions, and the instructions that are in the requirement list, we devised the following set:

- Branch always

- Branch on condition
- Load from memory
- Store to memory
- ADD
- AND(CC)
- OR(CC)
- SHIFTR
- Display

IV. Global datapath schematic

We designed a global overview of the data-path. We opted for a von Neumann architecture with one In-bus and one Out-bus. The overview can be seen in fig. 1.

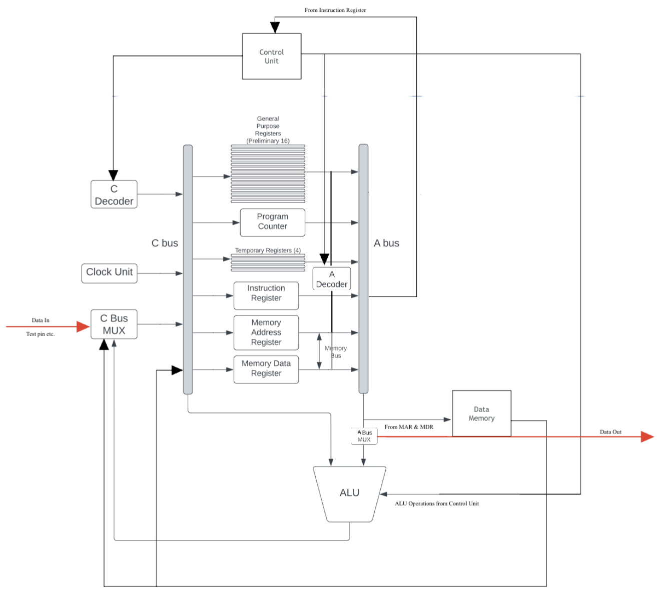

Fig. 1. Our global data path schematic.

V. Controller

We will implement a controller within this processor to inform the data path what operation the controller needs to execute and what data the operation is using. We keep updating the controller with status information. The status information tells the controller whether the result of a operation contains a carry bit (c), if the result is negative (n), if overflow occurs (v), or if the outcome is zero (z). The control unit will be based on the schematic that can be seen in see fig. 2.

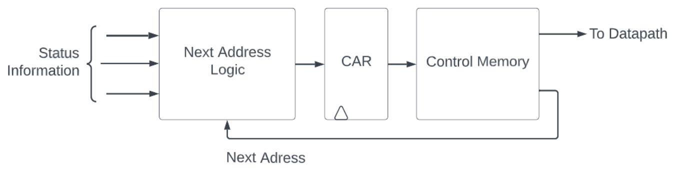

Fig. 2. Micro-programmed control unit schematic.

If we look at fig. 2, we notice how the control unit is responsible for the selection of instructions within the control memory. These micro-programs are shared with the data path. Status information is retrieved by the control unit from the Arithmetic Logic Unit (ALU). The status information is the input of the next address logic. The next address logic selects which address is the next address that is pointed to by the Control Address Register (CAR). The CAR points to an address within the control memory. At this address, a micro-program is stored. Once a micro-program is appointed, it is transferred to the data-path. In addition, the data at the appointed address aids to determine the next address. We designed the following micro-instruction format, see figure 3.

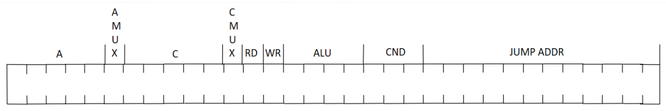

Fig. 3. Configuration of our micro-instructions.

VI. Memory

When designing memory, there are five things that we shall consider:

- How to realize the memory?
- What should be the memory size?
- What should be the word size?
- How much space will instructions take?
- How many I/O ports will the memory block need?

We will create the memory based on the von Neumann Architecture. The use of this architecture should reduce the complexity of the design. A programmer will have to hard-code programs into memory. Therefore, the programmer shall always know what is in a memory slot.

We will make the memory using a 2D array. This 2D array will consist of 4096 blocks. This array size will be large enough to cover the potential needs we may have when using our processor. The word size is 32 bits. This word size will allow for entire instructions to reside in one memory block.

The FPGA contains 64 MB of SDRAM and 1 GB of DDR3 RAM [2]. However, we cannot use this memory as it would be too complicated to communicate with the memory [1]. Therefore, the memory shall be synthesized using the VHDL subset, which means it will be built out of flip-flops.

We will not know whether this 2D array may be too big until the processor is largely synthesized in Quartus. Once the processor is synthesized, we can see the number of flip-flops the memory will take up, which tells us how much space is left for memory. The 32-bit words will be byte-addressable. The bytes shall be addressed using a 2-bit index. This will result in $\frac{log(4096)}{log(2)} + 2 = 14$ bits needed for addressing. We will use 16 registers, reducing the space taken in the store and loading instructions by rs1 and rd. This space allows for the increased size of the memory.

The memory will have 4 inputs and 1 output: read, write, data in, memory address, and data out respectively.

In summary, the memory will have a von Neumann architecture using a 2D array consisting of 4096 blocks of 4-byte words. Addressing the memory will take 14 bits of an instruction. We will use 5 I/O ports for the memory.

VII. Testing/debugging

As mentioned in the criteria list, number 4, the processor shall have a test pin that can be used to enter a debugging mode. In debug mode, the program execution shall halt. Although the system is allowed to finish the current instruction, it shall not move to the next. We are planning on implementing various features in the debug mode:

- the possibility to execute the code line by line
- the possibility to read data (show on the displays) from main memory using the onboard switches

This is going to be realized by having an asynchronous test pin. The instructions of our processor are going to be executed in one clock cycle. This removes the possibility to interrupt an instruction mid execution. We plan on using the program counter as a mean of navigating between the debug mode and the normal execution mode.

We plan on using the following structure (in pseudo code) when designing the logic for the debugging environment:

```
If test pin is high

    If step mode pin is high
        load contents of program counter
        execute loaded contents,
        increment program counter

    If display loaded address pin is high
         load contents of address from the
         switches, display the loaded
         contents on the 7-segment displays
```

When exiting the debug mode the program can continue executing as normal by fetching the next instruction using the program counter.

We intend on utilizing the LED lights on the DE1-SoC board to make it easier to understand what the processor is doing. These LEDs could indicate when the processor is writing to memory, reading from memory and can also display the status bits (c/n/v/z).

VIII. Planning and task division

We devised the following basic task division:

- Joost Buursink: primarily focused on memory management
- Fabian Widlund: primarily focused on the datapath
- Emil Imhagen: primarily focused on the testing and debugging
- Guus Branderhorst: primarily focused on the controller
- Stach Redeker: focused on testing, debugging and the controller, but also on the overlapping area, i.e. the instruction set (op code) and program

Then the planning. The project starts on Monday the 31st of October and ends with presentations on the 9th of November. The system should be handed in no later than the end of Tuesday the 8th. We aim to have the system ready Monday afternoon, in order to build in some buffer time if things go wrong. This also leaves Tuesday for creating the presentation. We plan to have fully working subsystems no later than Thursday afternoon, such that we can start combining on Friday.

We will use a Kanban board to track process. We will be following the priciples of Agile development, and hence we aim to have three daily stand-up meetings. The first meeting at the start of the day, the second before the lunch break, and the last at the end of the day.

IX. Conclusion

In this project plan, we summarized the criteria set for our JFEGS processor. In addition to that, we also elaborated on our design choices. We opted for a von Neumann architecture with one IN bus and one OUT bus. For the controller, we opt for using a 33 bits micro-instruction format. We will design a memory using a 2D array consisting of 4096 1-word blocks. We will realize debugging using a test pin that when pulled high, activates a debugging mode. In debugging mode, we are able to find potential problems with our program. In the end, we presented a planning and task division.

X. References

[1] G. Gillani, B. Molenkamp “Manual project Digital Hardware” 2022.
[2] Terasic Technologies Inc. “Terasic DE1-SoC User Manual” 2014.