**Computer Organization and Architecture**

**(21ECSC201)**

|  |
| --- |
|  |

**Activity Report**

**Course details**

|  |  |
| --- | --- |
| **Course Name** | Computer Organization and Architecture |
| **Course Code** | 21ECSC201 |
| **Semester** | III |
| **Division** | D |
| **Year** | 2023-24 |

**Team Details**

**Group No: 01**

|  |  |  |  |
| --- | --- | --- | --- |
| **Si. No.** | **Roll No.** | **S. R. N.** | **Student Name** |
| 1. | 431 | **01FE22BCS204** | Avinash Nayak |
| 2. | 427 | 01FE22BCS164 | Veerraj Chitragar |
| 3. | 424 | 01FE22BCS148 | Yallamma Pujar |
| 4. | 411 | 01FE22BCS057 | Sujal Kalal |

**1. Problem Statement**

**Specifications:** “**Design an 8-bit processor with 2 address format and having Register indirect addressing mode.**”

Design and simulate a **2-stage pipelined processor**, which can perform Load/Store, Arithmetic & Logical operations on a set of data. Design for a Harward Architecture, separate code memory & separate data memory to store program instructions and operands respectively. Include all the control & status registers like PC (to hold address of instruction), MAR (to hold the memory address), MBR (to hold the data from memory), PSW (to hold status of condition codes) and IR (to decode the instruction). Include a **Register file of 16 Registers** (R0—R15).

**2. Brief description of the functional specifications of the Processor designed**

* Data Width:
  + The processor has an 8-bit data width, allowing it to process data in 8-bit chunks.
* Instruction Set:
  + Supports a specific set of instructions tailored to an 8-bit architecture.
  + Includes instructions for arithmetic operations, logic operations, data movement, and control flow.
* Registers:
  + The processor has a set of general-purpose registers to store intermediate results and operands.
  + The 2-address format allows an instruction to specify two operands and an operation that operates on both operands.
* Register-Indirect Addressing Mode:
  + Supports register-indirect addressing, allowing instructions to use the content of a register as an address to access data in memory.
* Memory Address Space:
  + Defines the memory address space that the processor can access. In an 8-bit system, this may be limited to 2^8 (256) memory locations.
* ALU (Arithmetic Logic Unit):
  + The ALU is responsible for performing arithmetic and logic operations on data.
* Control Unit:
  + Manages the execution of instructions and control signals for various components within the processor.
* Instruction Execution:
  + Follows a fetch-decode-execute cycle where instructions are fetched from memory, decoded, and executed.
* Data Movement:
  + Supports instructions to move data between registers and memory locations, facilitating data processing.

**3. Code snippet executed on the machine**

Define registers

R0 DB 0x00

R1 DB 0x00

R2 DB 0x00

R3 DB 0x00

R4 DB 0x00

R5 DB 0x00

R6 DB 0x00

R7 DB 0x00

R8 DB 0x00

R9 DB 0x00

R10 DB 0x00

R11 DB 0x00

R12 DB 0x00

R13 DB 0x00

R14 DB 0x00

R15 DB 0x00

Define control and status registers

PC DW 0x000 ; Program Counter

MAR DW 0x000 ; Memory Address Register

MBR DB 0x000 ; Memory Buffer Register

PSW DB 0x00 ; Program Status Word

IR DB 0x000 ; Instruction Register

START:

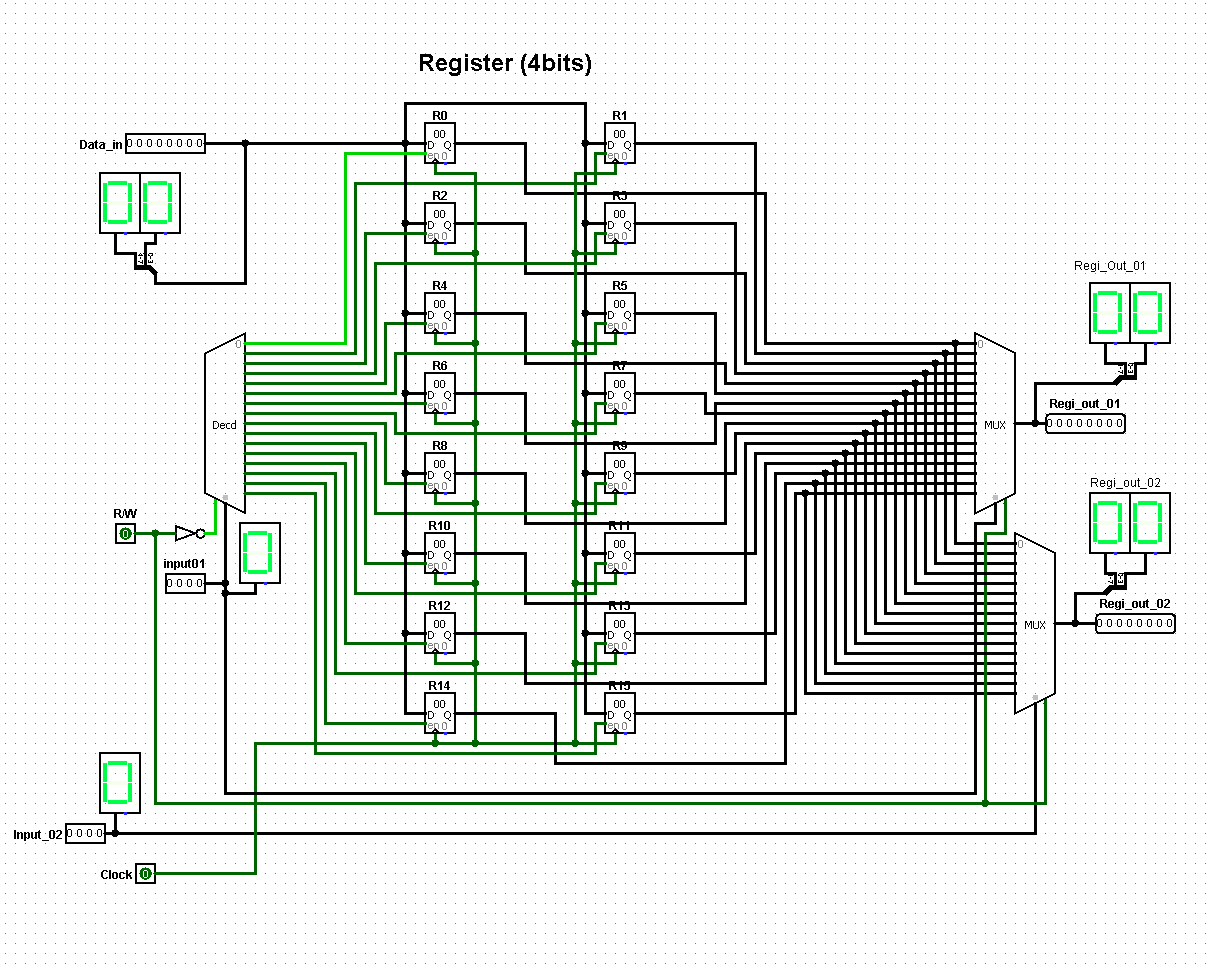
ADD R0, R1 ; Add values in R0 and R1, store in R0

OR R0, [R3] ; Bitwise OR of the result with the value in R0

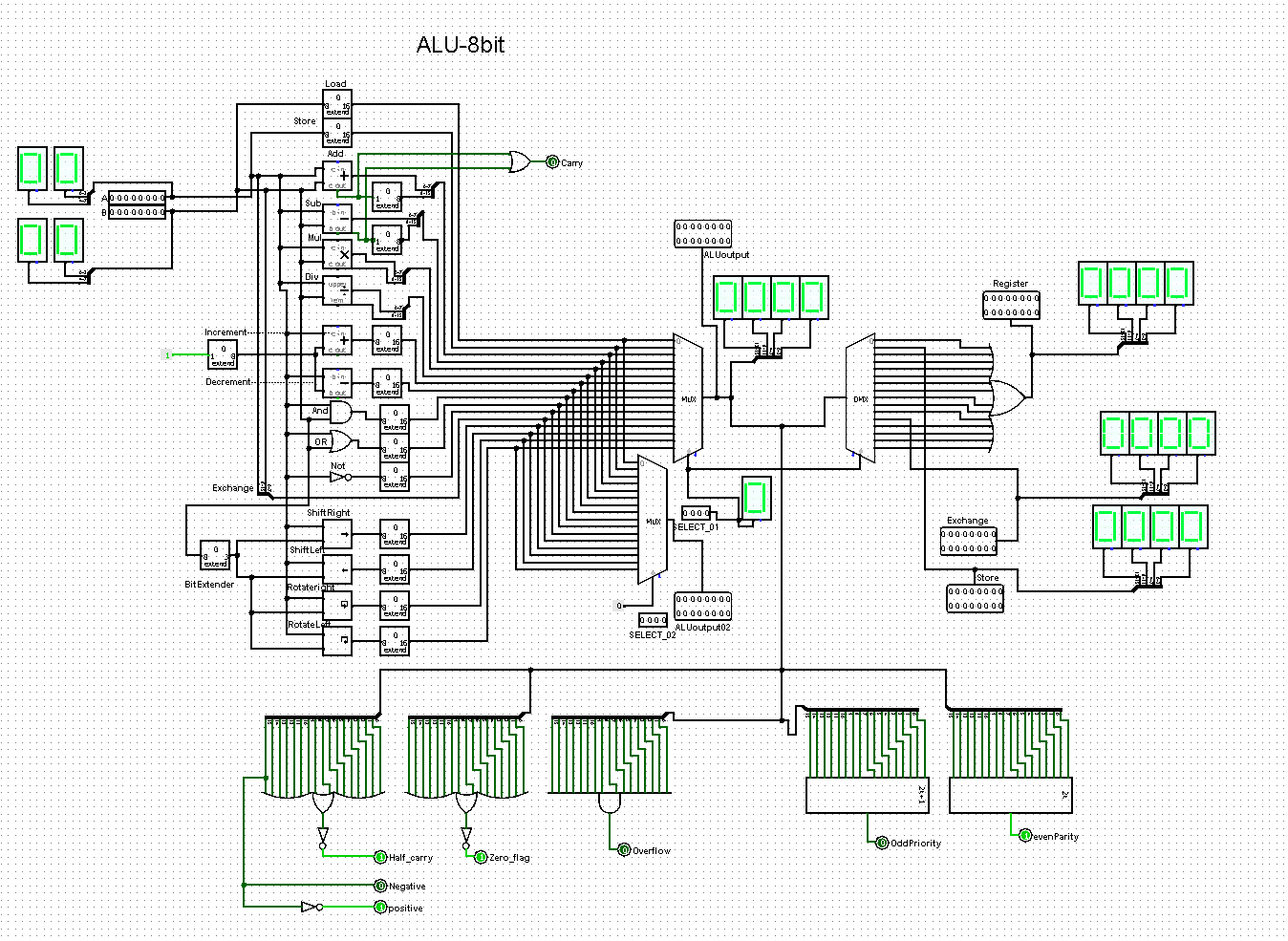
MUL R0, R10 ; Multiply the result by 10, store in R0

ST R0, [R10] ; Store the final result in memory at the indirect address in [R10]

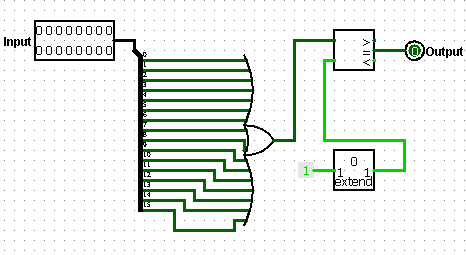
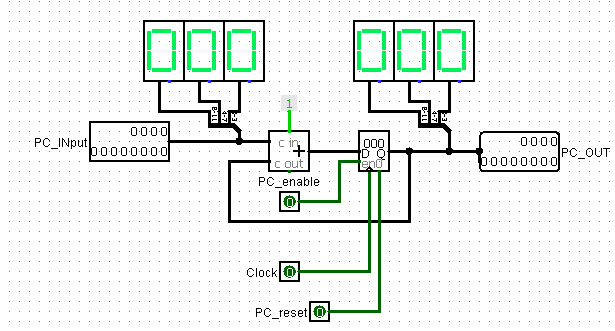
**4. Processor Design**

****

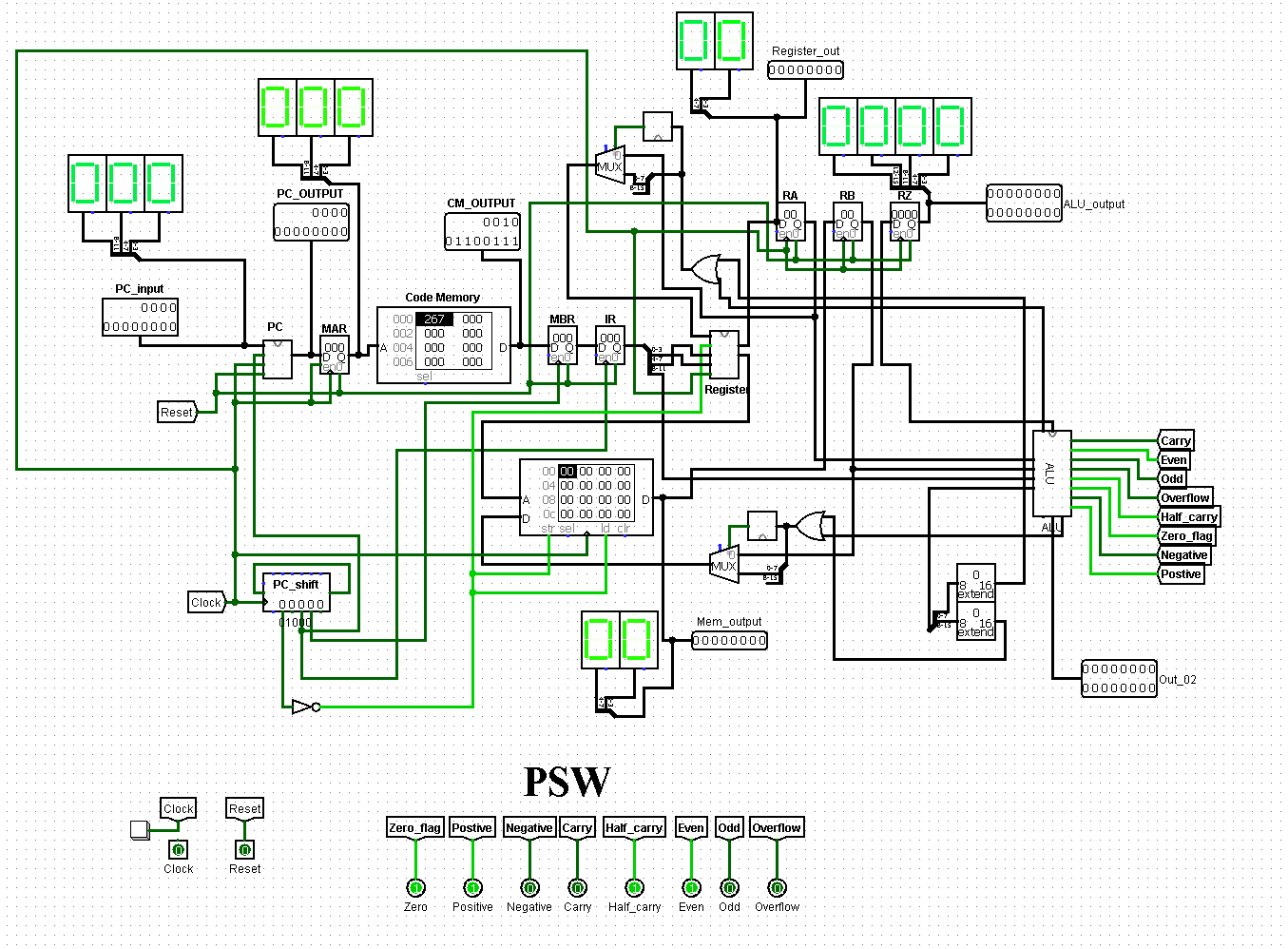
**Register (Select 4-Bits and Address is 8 bits)**

****

**ALU (Arithmetic logical unit)**

**Selector & Program Counter**



**Final Design**

**5. Challenges Faced and Solution**

Embarking on the processor design journey presented our team with a myriad of challenges that demanded ingenuity and resilience. From unforeseen hurdles that necessitated a strategic rethink to complexities in ensuring a robust and efficient architecture, our path was marked by continuous problem-solving. Whether optimizing performance, addressing power consumption concerns, or navigating issues related to memory access, parallel processing, and thermal management, each challenge required dedicated attention. Throughout the development phase, we embraced a flexible and adaptive strategy, fostering collaboration and continuous refinement of our design approaches. In overcoming these obstacles, we not only achieved our design objectives but also gained valuable insights and experiences that contribute to the ongoing evolution of our processor design expertise.

* Register file optimization:
  + Feature:
    - * Multi-Ported Register Files:
* Designing register files with multiple read and write ports allows for concurrent access by multiple instructions in the pipeline. This can enhance the processor's throughput and support higher degrees of parallelism.
  + - * Increasing Register File Size:
        + Expanding the size of the register file provides more storage for temporary data, reducing the likelihood of register spills to slower levels of memory. However, this needs to be balanced with considerations for chip area and power consumption.
      * Banked Register Files:
        + Dividing the register file into banks can reduce access conflicts and improve parallelism. Each bank can be accessed independently, allowing multiple instructions to access the register file simultaneously.
      * Load/Store Optimization:
        + Efficient handling of load and store instructions, including the use of load/store buffers and minimizing memory access latency, can alleviate pressure on the register file.
  + Solution:
    - We used two multiplexers to get indirect address and give two input port to register file for our required output. Using decoder to locate write back address to register file
* PC increment:
  + - Solution:
      * We used registered shifter for increment PC after 5 clock cycle in every instruction
* Pipeline:
  + - Feature:
      * We are designing two stage pipeline that help us to wait instruction in MBR and PC is pointing to next instruction
    - Solution:
      * We used registered shifter port for pipeline. The instruction will be stay in MBR until the instruction will be write back in of previous instruction.
* ALU Design:
  + - Feature:
      * We design ALU as perform for MISD (Multiple instruction for single data).
      * Multi-Ported ALU (Output): Designing register files with multiple read and write ports allows for concurrent access by multiple instructions in the pipeline.
      * Using gates design to perform all function like load, store, exchange and other operator also.
    - Solution:
      * Exchange: We used splitter to in reverse order.
        + For Load, store and exchange answer we used demultiplexer for dividing required output.
        + Data width: Bit extender for required output for data width.
        + Using demultiplexer and OR gate we get required output.
* Selector:
  + - This is used select one of the inputs for register file and memory.
    - We faced issue as when two input is high then select 2nd input as input for this we used it as selector.
    - Solution: Using or gate and comparator
* PSW (Program status word):
  + - Feature:
      * All of status of required design by us like all function (PSW).
    - Solution:
      * Using multiple OR gate and Odd and even parity

Using Multiple display and input and output we check the what is actually done in processor.

**7.** **Brief Comparative study of contemporary processors**

* Microarchitecture:
  + Compare the microarchitectures used in different processors. Consider aspects such as pipeline depth, branch prediction, and execution units.
* Manufacturing Process:
  + Explore the nanometer manufacturing process technology employed by each processor. Smaller nanometer processes generally imply better power efficiency and performance.
* Clock Speed:
  + Compare the clock speeds of processors. Higher clock speeds often lead to better single-threaded performance.
* Number of Cores and Threads:
  + Analyse the number of physical cores and threads each processor offers. More cores and threads contribute to better multitasking and parallel processing capabilities.
* Cache Hierarchy:
  + Examine the cache hierarchy, including L1, L2, and L3 caches. A well-optimized cache hierarchy enhances memory access efficiency.
* Instruction Set Architecture (ISA):
  + Understand the instruction set architecture used by each processor. Some processors may support advanced instruction sets, impacting their ability to execute specific types of tasks more efficiently.
* Power Efficiency:
  + Evaluate power consumption and efficiency. Processors that deliver high performance with lower power consumption are often preferred, especially in mobile and battery-powered devices.
* Integrated Graphics:
  + Consider whether the processor includes integrated graphics. Integrated GPU capabilities can be crucial for applications such as gaming and multimedia.
* Specialized Instructions or Features:
  + Look for any specialized instructions or features, such as SIMD (Single Instruction, Multiple Data) extensions, virtualization support, or security features like hardware-level encryption.
* Memory Support:
  + Examine the supported memory types, speeds, and maximum capacities. Memory support can impact overall system performance.
* Benchmark Performance:
  + Refer to benchmark results for real-world performance comparisons. Benchmarks like SPEC CPU, Geek bench, and Cinebench can provide insights into different aspects of performance.
* Price-to-Performance Ratio:
  + Consider the price-to-performance ratio to determine the value offered by each processor. This ratio is crucial for both budget-conscious and performance-oriented users.

**8. Conclusion**

The activity of designing an Assembly Language Program (ALP) for an 8-bit processor with a 2-address format and register-indirect addressing mode has been insightful and educational. Several key takeaways can be gleaned from this exercise.

Firstly, the exercise emphasized the importance of considering hardware constraints in programming. The limitations of an 8-bit processor, such as its constrained word size and limited number of registers, necessitate a thoughtful approach to resource management. This includes optimizing code to make the most efficient use of available resources.

Secondly, the exploration of a 2-address format underscored the significance of concise and expressive instructions. The ability to perform operations with two operands within a single instruction can lead to more compact and readable code. However, it requires careful consideration of the trade-offs between code size and execution efficiency.

The inclusion of register-indirect addressing mode added another layer of complexity and versatility to the learning experience. It demonstrated the flexibility of accessing memory indirectly through a register, providing more dynamic memory management options. This addressing mode, however, also introduces considerations of additional instruction cycles for memory access.

In terms of programming challenges, the activity highlighted the need for adaptability and strategic decision-making. The choice of addressing modes and the orchestration of instructions demand a balance between code readability and execution efficiency.

Lastly, the overarching takeaway from this activity is the critical role of optimization in achieving performance goals. Whether it is optimizing for speed or minimizing code size, understanding the nuances of the processor architecture and making informed choices in programming are integral to successful application development.

In conclusion, the learning and takeaways from this activity revolve around the nuanced interplay between hardware constraints, programming decisions, and the pursuit of optimized code. These insights contribute to a deeper understanding of processor architecture and the skills needed to design efficient and effective programs for constrained computing environments.