EEDG/CE 6302

Microprocessors and Embedded Systems

Electrical and Computer Engineering

Erik Jonsson School of Engineering and Computer Science

The University of Texas at Dallas

Dr. Tooraj Nikoubin

Project 1 (Part 1): Data Memory and Instruction Memory

Submitted By:

Muripa Uppaluri (MXU220008)

Chandanam Sai Nived (SXC210186)

Table of Contents

In this first lab project, we've designed two memories for MCU – the Data Memory and the Instruction Memory. We will be explaining about the implementation of each memory in the following sections.

Data Memory Implementation:

The Data Memory in a microcontroller refers to the memory used to store data and variables that change dynamically during execution. In this project, we've used Verilog HDL to design a data memory with **8-bit address bus input** that can access to every location of the memory array. This is a RAM with **8-bit data_in input** bus that can be used to write data into the memory and **8-bit data_out output** bus that is used to read the value out from the given memory location. We also have a **1-bit wr_rd_en input** signal which helps to understand whether we want a write operation (write data into the specified address location) or a read operation (read data out from the specified address location). Also, we've used a **clk input** signal that is used to synchronize or coordinate the transfer of data between memory and microprocessor.

Since the data_in bus is 8-bit wide, we've created a **memory array** where, **each location is 8 bits wide**, which means it can store 1 byte (8-bit) value and the array to have a **total of 256 locations**. So, this memory array can store up to 256 bytes of data.

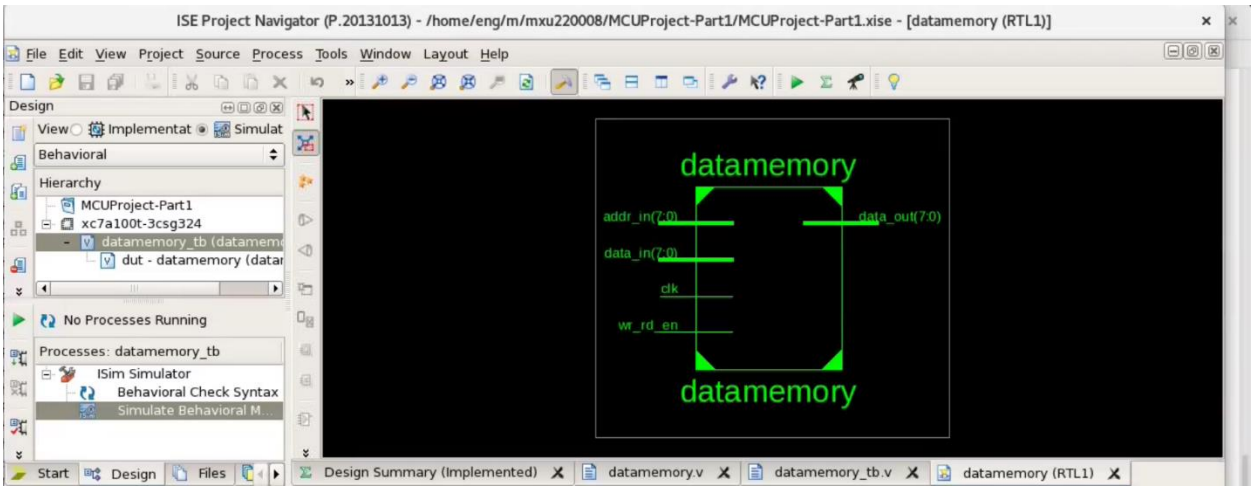
To avoid any erroneous data from the memory, we first initialized every location of the memory array with 0 using a simple Verilog for loop that would iterate over all the 256 locations and write 0 into these locations.

The memory is updated on the **rising edge of the clk signal only if the wr_rd_en signal is asserted**. So, a write operation occurs only when the wr_rd_en signal is asserted. During this **write operation**, the data from the **data_in input signal is transferred to the memory location specified by the addr_in** input bus. When the **wr_rd_en signal is deasserted, read operation occurs**, the value stored in the address location specified by addr_in bus is assigned to the data_out output bus. we've used **non-blocking assignments** for this sequential logic block.

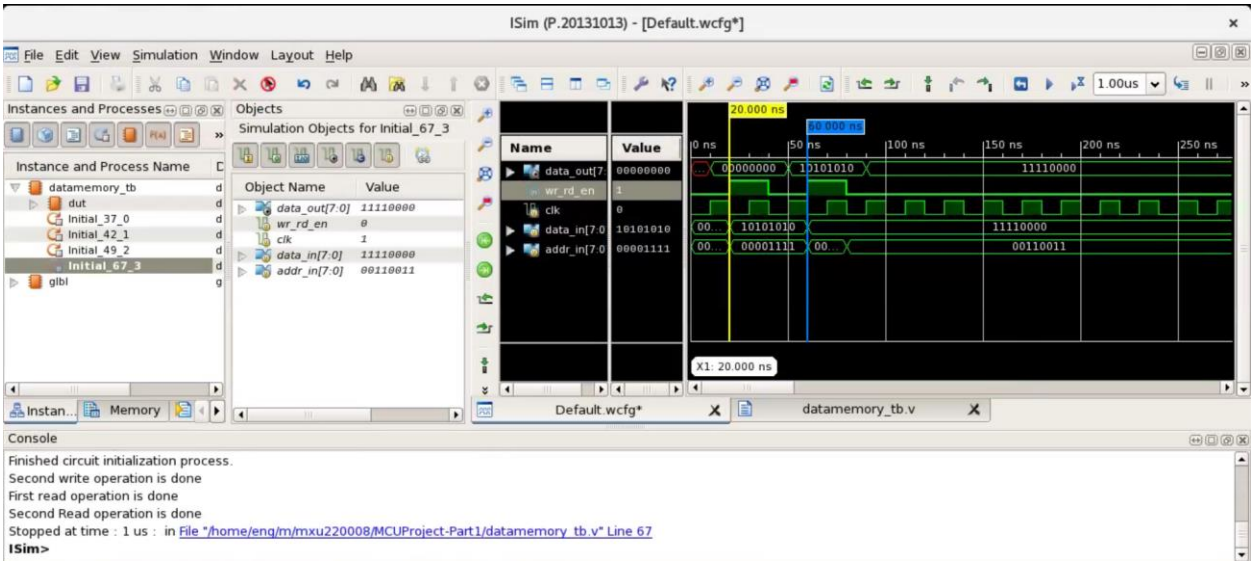
Data Memory Testbench –

We've created a simple testbench to verify this data memory. In this testbench module, we've declared all **the input ports as reg datatype** so that they can hold the changing values of stimulus and the output port as wire data type. The device under test is instantiated using **port-map instantiation**. Initially, all the inputs are initialized to 0 and then, random value is assigned to the addr_in and data_in inputs while the wr_rd_en signal is held high. This verifies the write operation. The wr_rd_en signal is then deasserted to verify the read operation. **$stop** system task is used to end the program. The testbench can be developed to be more random using **$urandom** system task to generate data. We've also used **$display** statements to check the read write operations and print the result statement using if condition. This helped me to check the result quickly instead of viewing the waveforms directly to analyze the result.

Schematic view of the Data memory –



Data memory waveform with Write Read transactions –



Instruction Memory Implementation:

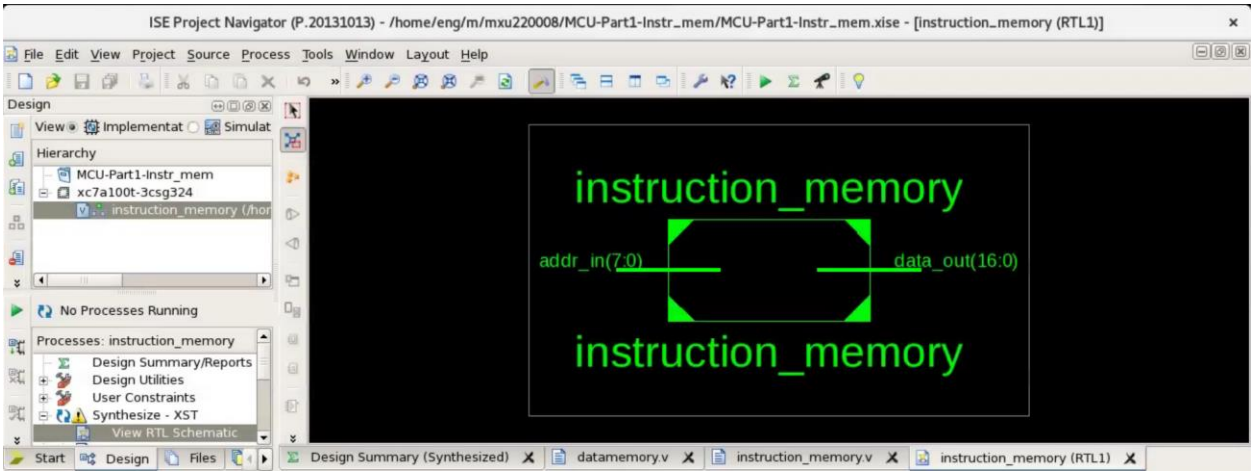
The instruction memory stores the instructions to be implemented by the processor. It is usually ROM or flash memory and ensures the instructions are not lost or overwritten during runtime. In this project, we've used Verilog HDL to design an instruction memory **with 8-bit addr_in bus input** and **17-bit data_out bus**.

We've created a **memory array** where, **each location is 17 bits wide**, which means it can store 17 bits instruction and the array to have a total of **256 locations**. So, this memory array can store up to 256 bytes of data. We've developed a simple instruction memory using **always block that trigger with a change in addr_in value**. Whenever addr_in value changes, the data/instruction in memory array in the given addr_in location is assigned to the data_out bus using **nonblocking** assignment. We've used a separate initial block to load in the instructions into the memory.

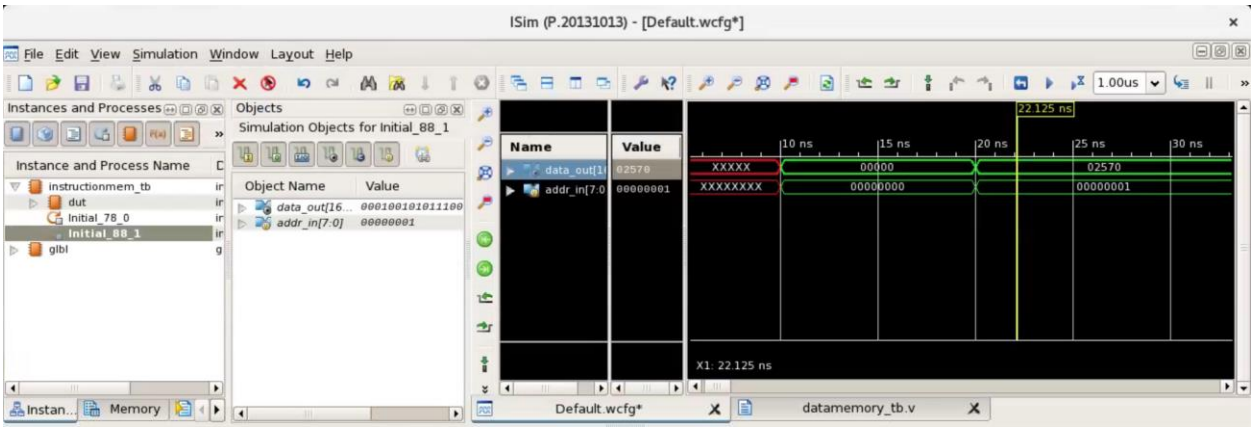
Instruction Memory Testbench –

We've created a simple testbench to verify this instruction memory. In this testbench module, we've declared all the input ports as reg datatype so that they can hold the changing values of stimulus and the output port as wire data type. The device under test is instantiated using port-map instantiation. The testbench logic is simple, just **input random values on addr_in bus and check the instruction on data_out bus**. This **data_out must be equal to the expected instruction** that we've loaded initially in the DUT code.

Schematic View of Instruction Memory –



Instruction Memory waveform –



Opcode Table for the Instructions –

Since we have 5 bits for opcode (Instruction [17:12]), we can have upto 32 opcodes for this MCU. A few of them can be:

Number	Operation	Symbolic notation	Opcode	Action	ALU req
1	No Operation	NOP	0	None	No
2	Input	IN	2	R[DR] <- Input Port	No
3	AND	AND	1	R[DR] <- R[SA] ^ R[SB]	Yes
4	Move A	MOV	3	R[DR] <- R[SA]	No
5	Addition	ADD	4	R[DR] <- R[SA] + R[SB]	Yes
6	Subtraction	SUB	5	R[DR] <- R[SA] - R[SB]	Yes
7	Increment	INC	6	R[DR] <- R[SA] + 1	Yes
8	Decrement	DEC	7	R[DR] <- R[SA] - 1	Yes
9	Shift Right	SHR	8	R[DR] <- sr R[SB]	Yes
10	Shift Left	SHL	9	R[DR] <- sl R[SB]	Yes
11	Load	LD	10	R[DR] <- M[SA]	No
12	Store	ST	11	M[SA] <- R[DR]	No
13	Complement	NOT	12	R[DR] <- Complement of R[SA]	Yes
14	OR	OR	13	R[DR] <- R[SA] v R[SB]	Yes
15	XOR	XOR	14	R[DR] <- R[SA] xor R[SB]	Yes
16	Add immediate	ADI	15	R[DR] <- R[SA] + se IM	Yes
17	Subtract with borrow immediate	SBI	16	R[DR] <- R[SA] + complement se [IM] + 1	Yes

Lab Handout Questions

1. ROM means Read Only Memory
   PROM means Programmable Read Only Memory
   EPROM means Erasable Programmable Read Only Memory
   EEPROM mean Electrically Erasable Programmable Read Only Memory

Memory Type	Pros	Cons
ROM	a) Permanent storage of data. b) No loss of data even when power is off. c) More reliable. d) Less prone to failures.	a) Data cannot be modified during runtime. b) If changes are needed in memory, chip must be replaced with new memory and reprogrammed.
PROM	a) Programming can be done using software. b) Can be programmed only once.	a) Limited to one-time programming.
EPROM	a) Non volatile. b) Can be erased and reprogrammed. c) Cost effective compared to PROM	a) Cannot erase parts of memory, full content needs to be erased. b) Static power consumption is high because of transistors.
EEPROM	a) Can be reprogrammed and erased and electronically. b) Can be reprogrammed multiple times. c) No need for UV light to erase data.	a) Slower than other memory types. b) More expensive than other memory types.

   Flash Memory can be erased and reprogrammed multiple times. It is generally slower than RAM but faster than ROM. Used in many types of storage devices such as USB drives, memory cards, SSDs, etc.

2. Data stored in program memory is larger than in data memory because program memory is used to store both the program instructions and constant data, while data memory is used to store only variable data. Program memory has a larger size compared to data memory in MCUs to accommodate the larger size of program instructions.

In this project, we've taken data memory array with 256 locations, each 8 bit wide while program memory has 256 locations, each with 17 bits of data. Hence data memory can store upto 2048 bits while program memory can store 4352 bits of data.