The purpose of the midterm (fifth homework assignment) is to provide an experience for integrating a new instruction into a Multi-Cycle CPU with various instruction sets. The Datapath and Controller designs are given – all components are modeled in Verilog, and then verified through the use of a full testbench.

**Code Submission and Memory Files**

All Verilog code and memory files utilized in this expanded CPU design, with detailed comments that go into detail regarding the definition of all lines of code, can be found in the root folder of the Midterm ZIP file submitted for this assignment:

Verilog files:

*BM\_Controller.v, BM\_Datapath.v, BoothMultiplier.v*

*Controller.v, Datapath.v, CompArch\_CPU.v, CompArch\_CPU\_Tester.v*

Data (memory) files:

InstructionFile.mem, HexadecimalFile.mem

**Datapath and Controller Implementations**

The Datapath of the *CompArch\_CPU* is seen below, taken from the HW description:



In an effort to implement the new instruction, a few extra things were added to the Datapath diagram shown above:

* *reg [15:0] mp* – used to contain the multiplier operand for multiplication
* *reg [15:0] MCout* – used to contain address locations of result storage
* *wire [31:0] prod* – used to contain result from multiplication operation
* *wire busy* – used to note whether the multiplier is busy or not

The Datapath was developed all in one file, *Datapath.v*, with all of the different registers being instantiated and updated in *always* procedural statements and by their individual controller signals: *enACC*, *enIR*, *enPC*. The ALU, who is ultimately responsible for updating the *ACC* and *PR*, is updated whenever the *ALU\_op* changes or whenever *enACC* is asserted high. It is also responsible for multiplexing of signals on busses and for setting the *op\_code* coming in from the *IR*.

The Controller of the *CompArch\_CPU* is seen below, taken from the HW description:



In an effort to implement the new instruction, a few extra items were added to the Controller diagram shown above:

* The Controller will transition from Execute 🡪 Fetch if and only if the multiplier alerts the Controller that it is not busy.
* The Controller will keep looping back to Execute until multiplication is done, and then will store both multiplication results into fixed memory locations.

Using these four stages, the Controller was realized using a Huffman style of coding, with one procedural block responsible for tracking the *reset/next\_state* transition, and the other for performing the operations within the states themselves. Below are quick descriptions of each of the states and which signals they interact with across the Datapath:

**Reset**:

* While *reset* *== 1*, stay in *`Reset* state. Begin fetching instructions once *reset* deasserts its current value (*reset ==* 0).

**Fetch**:

* Place the *PC* address on the *addr\_bus*. By setting *rd\_mem = 1*, place information from *mem[addr\_bus]* on *data\_bus*. Load the 16’b received into the *IR*, and *increment PC by 1*.

**Execute**:

* LDA – Place the *IR* information on the *addr\_bus*, then read *mem[addr\_bus]* into *data\_bus*. Place *data\_bus* on *MuxB*, and have it pass through the ALU, and then load it into the *ACC*.
* STA – Place current *ACC* data onto the *data\_bus*. Place *IR data* onto the *addr\_bus*. Write into *mem[addr\_bus]*.
* ADD (SUM) – Place the *IR* information on the *addr\_bus*, then read *mem[addr\_bus]* into the *data\_bus*. Place *data\_bus* on *MuxB*, and add it to the current contents of *ACC*. Then store results back into *ACC*.
* SUB – Place the *IR* information on the *addr\_bus*, then read *mem[addr\_bus]* into the *data\_bus*. Place *data\_bus* on *MuxB*, and subtract it from the current contents of *ACC*. Then store results back into *ACC*.
* JMP – Place the *IR* information on the *addr\_bus*. Place *addr\_bus* on *MuxB*, then pass it through and store it into PC.
* JEZ – If the ACC is 0 (based on zero signal), place the *IR* information on the *addr\_bus*. Place *addr\_bus* on *MuxB*, then pass it through and store it into *PC*.
* HLT – If this instruction is seen, the next state is the halt state.
* MUL – Perform multiplication with what is currently located in the *ACC* along with the multiplier located at *mem[addr\_bus]*. Once the multiplication is done, store the LSBs and MSBs of the product into fixed memory location, and also store the MSBs into the *ACC*.

**Halt**:

* Stay in this state until *reset = 1*, then go back to `*Reset* state.

A few notes about the Controller:

* A forced time delay is built in whenever the *rd\_mem* and *wr\_mem* signals are asserted. This is done so that there is sufficient time for the *data\_bus* to be populated based on force timing delays seen in thetestbench *Memory\_Read\_Write* procedural block.
* Another forced time delay is built in for when the ALU must perform an operation. This is done to allow sufficient amount of time for the calculation to be performed in the *Datapath*, and allows for accurate transmission of data. This will cause a delay in the *ACC* receiving and storing data.

**Testbench Methodology and Planning**

Before going ahead and immediately programming the testbench, it was decided to implement an assembler to make it easier to load in programs to the CPU. The assembler would take in a specified instruction set, as provided by the HW description, and turn it into machine language that the CPU can understand. With a multi-cycle CPU and shared memory for both data and instructions, properly laying out a plan to validate the functionality of the CPU operation in the testbench is extremely important. The following layout is what was determined:

* Memory will contain 213 spaces, all initialized to 000016.
* The testbench will only utilize the first 30 memory spaces.
* Addresses in the memory 0 – 14 = Instructions
* Addresses in the memory 15 – 29 = Data (referenced by Instructions)
* The first 15 addresses will fully validate all functionality of the CPU.

**The change that will test the multiplier capability is highlighted in green:**



Utilizing the *PC* (Program Counter), the testbench will navigate through the Instructions contained within Lines 00 – 14, accessing (reading and writing) data to Lines 15 – 29 as necessary. Based on the linear set of instructions, the following section details what should be expected based on each step – instructions which test the new multiplication operation are highlighted in **green**:

Line 00: Load ACC with Data @ Line 25 > ACC = 10010

Line 01: Add value @ Line26 to ACC > ACC = 30010

Line 02: Subtract value @ Line 20 from ACC > ACC = 25010

Line 03: Add value @ Line 16 to ACC > ACC = 26010

Line 04: Store ACC in mem[001D] (Line 29) > mem[001D] = 010416

Line 05: Jump to instruction @ Line 08 > Skip HALT and SUB

**Line 08: Multiply ACC with data @ Line 28 > ACC = MSBs of product**

Line 09: Load ACC with Data @ Line 15 > ACC = 010

Line 10: Since ACC == 0, jump to Line 12 > Skip HLT

Line 12: Load ACC with Data @ Line 18 > ACC = 3010

Line 13: HALT the CompArch\_CPU > HALT until *reset = 1*

Line 14: Store ACC in mem[001C] (Line 28) > mem[001C] = 001E16

**Testbench Assembler (Machine Language)**

Utilizing the given instruction format, the following *InstructionFile.mem* file is created in order to populate the *HexadecimalFile.mem* file.

**The change that will test the multiplier capability is highlighted in green:**

****

**Expected Results of Multiplication Operation**

The following information, given all addresses supplied in the *HexadecimalFile.mem* input file, specify the expected results of the multiplication operation being tested:

ACC Value @ Start of Multiplication = 20010 = 00000000110010002

Multiplier from memory in Line 28 = 1337510 = 00110100001111112

**Product** = 0000000000101000\_11010001001110002 = 28\_D13816

**LSB** = D13816

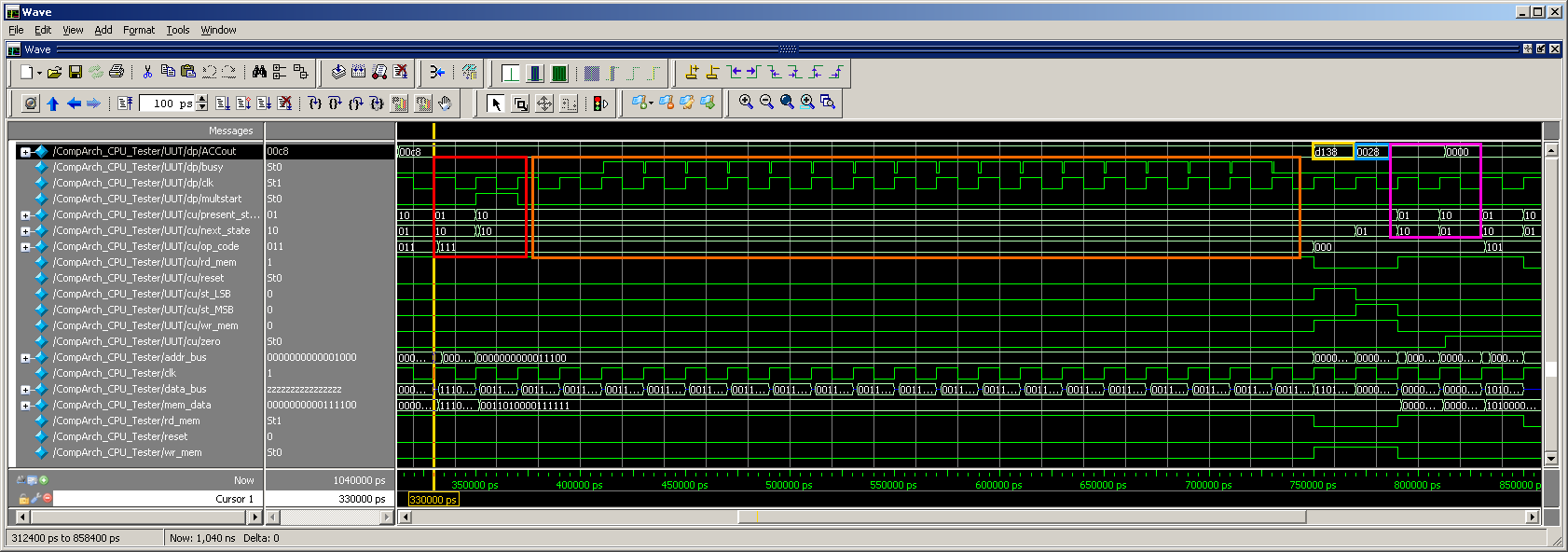
**MSB** = 2816

**Testbench Confirmation for Multiplication Operation**

Given that the other functionalities of this *CompArch\_CPU* were already validated in HW 3, this screenshot will specifically highlight the added multiplication functionality required for this midterm submission.

The screenshot of the waveform below showcases the output of the testbench created for the *CompArch\_ALU*. The boxes here highlight significant events in the timeline of the multiplication operation that is being verified. Zooming in will showcase all results correctly.

**Waveform Analysis of Multiplication Operation**



**RED** block: **CORRECT**

* The CPU is currently at Line 08 in the Hexadecimal instruction file.
* In order to begin preparation for multiplication, an op\_code of 111 is assigned by the Controller. The *multstart* signal is asserted for one *clk* cycle. The contents of the *ACC* and the data stored in *memdata[addr\_bus]* are sent to the *BoothMultiplier* as inputs, and the multiplication process commences.

**ORANGE** block: **CORRECT**

* After two pulses of *clk*, which correspond to the `Idle and `Init transitions located within the Controller of the *BoothMultiplier*, the multiplication process begins for a total of 18 *clk* cycles. This can be seen by monitoring the falling and rising edges of the *busy* signal, which is used in the *Controller* to determine the state of the *CompArch\_CPU*.
* It is important to note that the *present\_state* of the *CompArch\_CPU* always stays in the Execute state while the multiplication process is occurring.

**YELLOW** block: **NEED TO CONFIRM**

* As soon as the multiplication process is done, the LSBs of the 32 bit result are placed on the *ACC* and are sent out to be written into memory at locations specified by the *MC*, or Multiplication Counter.

**BLUE** block: **NEED TO CONFIRM**

* One the LSBs of the result are stored, the MSBs of the 32 bit result are placed on the *ACC* and are sent out to be written into memory at locations specified by the *MC*, or Multiplication Counter. This memory address is exactly one after the LSBs.
* Data for the MSBs also remains on the *ACC*.

**PURPLE** block: **CORRECT**

* Once both *clk* cycles of LSB and MSB storage are performed, navigate back to the `Fetch state and begin retrieving information from memory as before.
* The next operation in the Instruction memory is to LDA a value of 0, which is exactly what the *ACC* displays is occurring.

According to the waveform output, the multiplication operation occurred as intended. The last thing that needs to be verified is that data was properly written when the LSB/MSB instructions were issued. This can be verified by making a comparison between the *HexadecimalFile.mem* before the CompArch\_CPU operation was started, and after once all computations were performed in the simulation.

The comparison from both files can be seen here:



As can be seen, the Multiplier Counter (MC) starts storing multiplication results at memory address 30. Thus, the LSBs of the multiplication result are stored in address 30, and the MSBs of the multiplication result are stored in address 31.

Thus, all functionality and capabilities for the CompArch\_CPU, including the new multiplication operation, have been verified.

This concludes the analysis for the Midterm.