Project - Structural Model

Part 3 - May 9, 2005

Jason Nemeth

I. Introduction

The purpose of this assignment was to re-implement a working model of the microprocessor using a structural VHDL implementation.

II. Requirements

The microprocessor must have the following capabilities:

1. Direct access to internal memory with at least 256 words.
2. An instruction set capable of executing at least the following two benchmarks:
   a. Output 10 signed integers in sorted (largest first) order where the numbers are input in random order.
   b. Output the mean (to the nearest integer) of 16 signed integers read into the processor.
3. One 8-bit input port, and one 8-bit output port.
4. Optional one single-bit Input enable line, and one single-bit Output ready line

The overall layout of the microprocessor is shown below in *Figure 1*. The behavioral model was created such that all these characteristics were met. The model is capable of modeling both provided benchmarks.

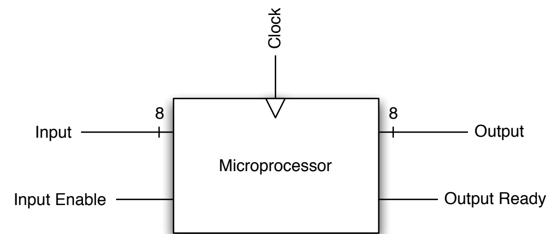

Figure 1. Layout of the Microprocessor

III. Design Decisions

Opcodes

Once the pseudo-code for the model's implementation was complete, the required opcodes were determined. Listed below in *Table 1* are the 8 opcodes utilized in the model.

Opcode	Operation
000	Push
001	Pop
010	JumpNot0
011	Jump
100	Add
101	Sub
110	ShiftR4
111	NoOp

Table 1. Opcodes and their associated operations

The above opcodes allow for the proper averaging and sorting of a number of 8-bit input values.

Structural Layout

The next step that needed to be completed was the actual layout of the microprocessor in structural terms. As could be noted by examining the opcodes listed above, this microprocessor utilizes

a stack machine-based architecture. The design allows for a Program Counter, Instruction Register, and Stack Pointer, each directly mapped to memory locations within RAM. Directly-mapping each of these components allowed for a design completely free of external registers above those contained in the main 256-word RAM.

A relatively shallow hierarchy was chosen due to the large number of components that needed to be directly mapped to RAM. A simplified overview of the system is shown in *Figure 2*. This figure shows the major interactions between the components, as well as the logic and control signals produced by the Controller that synchronize the system.

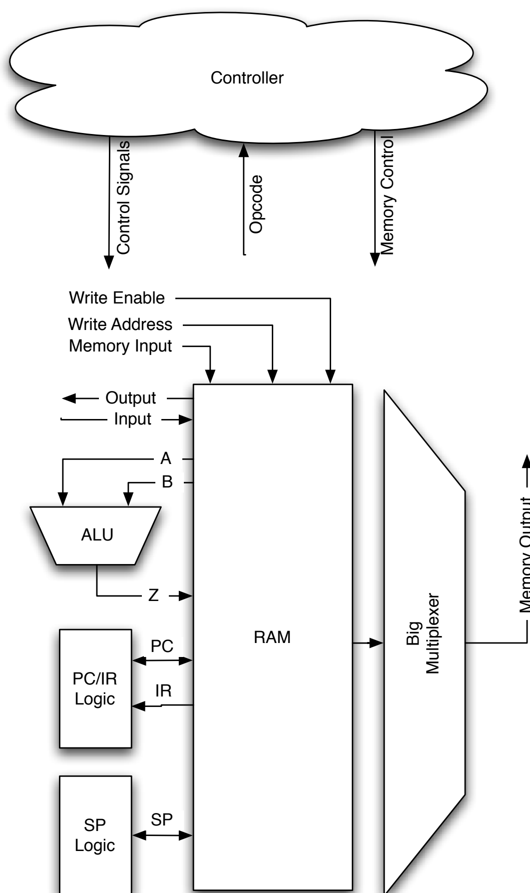

Figure 2. Overview of the microprocessor

Memory Implementation

Most of the time planning the model was spent working on the layout of the main memory. Due to the relative abundance of stack operations needed when using a stack architecture, a memory system was developed such that memory reads and writes can take place simultaneously. This allows for stack operations to be completed in a single clock cycle by simultaneously, for example, reading a memory address and pushing it onto the stack. A more detailed representation of the operation of the main memory core is shown in *Figure 3*. This operation is achieved by linking the output of the BigMux to the memory's input line during stack operations.

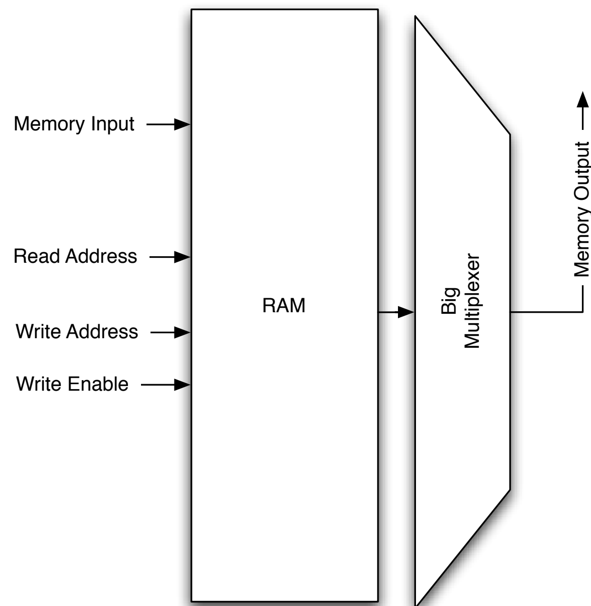

Figure 3. Memory access procedure layout

Memory Access

Since more than four, but less than eight opcodes were required for the system, each opcode must occupy three bits. Unfortunately, this leaves only five bits for addressing, covering a maximum of 32 memory locations. It was decided to directly map those 32 locations to

positions 0 through 31 in main memory. Doing so allows for the direct access of each of those locations by the Push and Pop commands. Indeed, Push and Pop can only be utilized on items contained in the first 32 words of memory. It was therefore necessary to locate every important direct-access item within that area, including data storage locations. *Table 2* below shows the orientation of those items in memory.

Memory Location		Function
31	00011111	A
30	00011110	B
29	00011101	Z
28	00011100	PC
27	00011011	IR
26	00011010	SP
25	00011001	Constant9
24	00011000	Constant10
23	00010111	Constant0
22	00010110	Constant-1
21	00010101	Input
20	00010100	Output
19	00010011	j
18	00010010	i
17	00010001	DataTemp
16	00010000	Data16
15	00001111	Data15
14	00001110	Data14
13	00001101	Data13
12	00001100	Data12
11	00001011	Data11
10	00001010	Data10
9	00001001	Data9
8	00001000	Data8
7	00000111	Data7
6	00000110	Data6
5	00000101	Data5
4	00000100	Data4
3	00000011	Data3
2	00000010	Data2
1	00000001	Data1
0	00000000	Data0

Table 2. Special Memory Locations

The 3-bit opcode also had an effect on the Jump and JumpNot0 operations. Since only five bits are available, it is not possible to send the Program Counter to a fully-qualified memory address. Instead, relative addressing was utilized. This signed system allows for jumping ahead 15 or back 16 memory locations from the current address in the Program Counter. This is a restrictive design, and causes extra operations if a jump needs to exceed 16 locations in either direction. This is an unfortunate side-effect, but was ultimately the best solution as merely assigning the upper three bits of the address to a specific value would not allow complete utilization of memory for program code.

ALU Implementation

The ALU was implemented completely memory-mapped. Its inputs A and B are found in memory locations 31 and 30, and its output Z can be found in location 29, as shown previously in *Table 2*. This allows for the utilization of direct-mapped memory addresses to perform the mathematical functions Add, Subtract, and Shift Right 4 Bits. The ALU gets passed opcode information, performs the associated operation, and stores the result in Z. This result is then directly accessible by the Push and Pop operations.

Design of Operation

The microprocessor goes through three distinct phases during the course of executing a single instruction. Each phase takes 30ns, which is the speed of a single clock cycle, meaning an entire operation takes 90ns. The Fetch phase allows for the retrieval of an instruction from memory. The Decode phase allows for the translation of the instruction into opcodes, and also the incrementing of the Stack Pointer during a Push operation. The Execute phase executes the instruction being performed, including Add, Subtract, and ShiftRight4. When an instruction has completed execution, the machine returns to the Fetch phase once again, allowing the process to be completed all over again. A 30ns cycle was chosen because the Adder produced in Homework 1 produced valid output

after about 25ns, and a few extra nanoseconds were provided for the added complexity. Since the adder and the RAM multiplexer are the components with the largest delay, it might be possible to decrease the clock cycle once the design is implemented at the RTL level by either optimizing those designs, or having them run asynchronously.

IV. Results

The structural model appears to be up and functioning properly. All eight of the opcodes produce correct outputs under all tested conditions, and most importantly the stack performs properly. The model produced correct output for Benchmark B, the average of 16 numbers. However, the system could unfortunately not produce a proper set of sorted outputs. I'm fairly confident the problem lies in the software, but hours of debugging could not uncover it. Further estimations of time and performance metrics for Benchmark A are based upon the incorrect system output, and should be viewed as rough estimates. The metrics for Benchmark B, on the other hand, are reproducible and reliable.

Benchmark	Time (μs)	Metric (Gate*μs)
Average	7.1575	107362
Sort	12.0125	180000

Table 3. Benchmark results and metrics

These estimated results were calculated using a 30ns, three phase clock cycle (Fetch, Decode, Execute).

The output and results of the microprocessor model can be viewed using the TextIO output file "results.txt", which also logs the inputs into the system.

V. Conclusions

The results of the structural model will be used to further implement the RTL model in the coming week. The model's individual behaviorally-designed structures will be replaced by structures implemented at the RTL level. This structural model proves that the system is capable of operating properly, producing proper output for the Average operation. Unfortunately, a likely bug in the Sort operation's software kept it from performing properly. After many hours of debugging, it eventually had to be abandoned in order to get the rest of the system, and the other benchmark, performing properly. I hope to have the operation fixed before beginning work on the RTL model so that the RTL model is capable of completing both benchmarks.

This model is a much better indicator of overall system performance than the previous behavioral model. In this model gates can be estimated, and execution times observed. In addition, this model naturally gives way to the RTL model that needs to be developed in the next step of the assignment.