# **Memory Interfacing**

***C8051 Microcontroller***

Jeffrey Pistacchio

Christopher Pybus

Microprocessor Systems

14 November 2016

# **Introduction**

Memory is an integral portion to any computer program. Specifically, in the C8051 microprocessor memory can be a precious commodity. Programmers need to be conscientious of how they use their on chip memory. One way to expand the amount of memory available is to integrate off-chip RAM into the hardware. In this lab, two Am9128 chips, and two Am91L14 chips to expand the total amount of memory available on the microcontroller. Part one of the lab required that the team interface two Am9128 chips to add 4k additional memory addresses (2k per chip). Part two of the lab required the team to interface two Am91L14 chips to store an additional 2k memory addresses.

# **Methods and Procedures**

Each part of the lab was divided into two goals, a hardware goal and a software goal. In order for each chip to properly interface with the microprocessor, a chip may only be enabled when the specific set of memory addresses that is assigned to that chip is called. Therefore, combinational logic taking the memory address as an input and outputting the specific chip enable sequence was needed.

## **Part 1**

Part 1 of the lab required that the team interface two Am9128 memory chips to add an additional 4k of external memory to the 8051 microprocessor. This task was accomplished in two discrete tasks, a hardware task and a software task. Both are described in more detail below.

**Part 1 - Hardware**

The external memory chip had to be wired up to the 8051 microprocessor using the crossbar and many inputs and outputs. The 8051 has a 16 bit memory addressing bus that functions on the 8 bits of Port 6 (low byte) and the 8 bits of Port 5 (hi byte). The least significant 11 bits of the 16 bit address bus were connected to the 11 address bus of the Am9128. The remaining 5 bits were connected to glue logic which is described in the next paragraph. The 8 data bits from the 8051 were connected the the Am9128 using Port 7 on the crossbar. Write enable and read enable lanes were connected to the Am9128 using pins P4.7 and P4.6 on the crossbar, respectively. A schematic of the Am9128 and crossbar can be seen in Appendix A.

Because the team was attempting to add extra memory address locations on top of the 0x2000 that was already there inside the 8051, the team had to develop a method to make the memory location of the two added memory chips appear to be in the 0x2000 to 0x3000 range from the perspective of the 8051 microprocessor. This was done using glue logic, or rather, hard wired logic that directed address signals to different chips depending on the input. In this case, the team designed glue logic that turn off or turn on the correct memory chips depending on what address was read in from the 5 most significant bits on the address bus. View Table 1 for specific ranges, and Appendix A for a schematic of the glue logic used.

Table 1: Ranges of Glue logic

|  |  |  |
| --- | --- | --- |
| HEX | Binary | Action |
| 0x1FFF | 0001 1111 1111 1111 | Turn off both memory chips |
| 0x2000 | 0010 0000 0000 0000 | Turn on chip 1, turn off chip 2 |
| 0x27FF | 0010 0111 1111 1111 | Turn on chip 1, turn off chip 2 |
| 0x2800 | 0010 1000 0000 0000 | Turn on chip 2, turn off chip 1 |
| 0x2FFF | 0010 1111 1111 1111 | Turn on chip 2, turn off chip 1 |
| 0x3000 | 0011 0000 0000 0000 | Turn off both memory chips |

As seen in the table, the boundaries that changed glue logic output were at 0x2000, 0x2800, and 0x3000. Bits 11 through 16 were used to determine the output.

**Part 1 - Software**

On the software side, the team only needed a very simple program that accessed external memory locations. First, the team configured the 8051 so it would properly function with any external memory. They slowed down the system clock so it would provide ample time for any external signals/chips to process and return output. The specific baud rate used was 28800, which is significantly slower than the normal 115200. Ports 4, 5, 6, and 7 were all configured for push pull as those ports are used for external memory signals, including address bus and data bus.

The main control function contained only a simple while loop with a nested for loop. The inner for loop ran from 0x1FF0 to 0x3200, reading and writing characters to each address location. It did this by using two functions, writeMem and readMem. The writeMem function took in an address location and a data byte, and wrote to that location using \_\_xdata variable. The readMem function took in an address location and then returned a data byte from that location.

## **Part 2**

For part 2, the team had to add additional memory on top of the two Am9128 memory chips they had just added. The memory to be added was Am91L14, which is different from the Am9128 in two ways. First, it only has 1024 memory locations, instead of the 2048 of the Am9128. Second, each memory location is only 4 bits wide, instead of the full byte wide memory location of the Am9128. This posed several challenges, as the team had already set up external memory logic to function with byte-wide memory at 0x0800 cutoff points. Again, the part was separated into hardware and software tasks detailed below.

**Part 2 - Hardware**

In order to add more memory to the already established byte wide memory setup, the team decided to use two Am91L14 chips working in parallel so that each chip would store half of the byte addressed so any specific location. The two chips would essentially function as one chip, except one would store the hi 4 bits and one would store the low 4 bits of any arbitrary data byte.

The next problem was adding the glue logic that would enable the two chips whenever an address between 0x3000 and 0x33FF was received, and then disable all chips for 0x3400 and above. The updated chip enable ranges table can be seen below in Table 2.

Table 2: Ranges of Glue logic

|  |  |  |
| --- | --- | --- |
| HEX | Binary | Action |
| 0x1FFF | 0001 1111 1111 1111 | Turn off both memory chips |
| 0x2000 | 0010 0000 0000 0000 | Turn on chip 1, turn off others |
| 0x27FF | 0010 0111 1111 1111 | Turn on chip 1, turn off others |
| 0x2800 | 0010 1000 0000 0000 | Turn on chip 2, turn off others |
| 0x2FFF | 0010 1111 1111 1111 | Turn on chip 2, turn off others |
| 0x3000 | 0011 0000 0000 0000 | Turn on chips 3 and 4, turn off others |
| 0x33FF | 0011 0011 1111 1111 | Turn on chips 3 and 4, turn off others |
| 0x3400 | 0011 0100 0000 0000 | Turn off all chips |

As you can see from the table above, the bits of interest now extend one bit lower than before. Instead of bits 11 through 15 driving the logic, we have to look at bits 10 through 15. Bit 10 specifically determines if an address is above or below the 0x3400 cutoff point, which wasn't necessary before because the team was working with memory that spanned 0x0800 addresses, and had no need for a 0x0400 cutoff point. The new glue logic can be seen in Appendix B.

**Part 2 - Software**

There were no changes to the software from the previous part, as all the changes were made to external memory, and no changes needed to be made to the code to access more external memory.

# **Results and Analysis**

Results were obtained by looping through all memory address printing the data byte on the memory address and observing whether the correct value is displayed. Furthermore, the chip operation was verified by turning the chip off (removing the Vcc connection) and observing the data value read on the addresses allocated to that chip.

The observed results can be seen in Table 3 below. All memory chips operated as intended and interfaced correctly with the C8051 microprocessor.

Table 3: Results Expected Value vs. Actual Value

|  |  |  |
| --- | --- | --- |
| Chip | Expected String | Read Value |
| Chip 1 (on) | “MPS is cool…..” | “MPS is cool…..” |
| Chip 1 (off) | “!!!!!!!!!!!!!!!!” | “!!!!!!!!!!!!!!!!” |
| Chip 2 | “MPS is cool…..” | “MPS is cool…..” |
| Chip 2 | “!!!!!!!!!!!!!!!!” | “!!!!!!!!!!!!!!!!” |
| Chips 3/4 (on) | “MPS is cool…..” | “MPS is cool…..” |
| Chips ¾ (off) | “!!!!!!!!!!!!!!!!” | “!!!!!!!!!!!!!!!!” |

# **Conclusion**

The team successfully accomplished all parts of Lab 5 by making several smaller programs that accomplished all the subgoals. Dividing the project into separate goals made the exercises significantly more approachable due to implementing the divide and conquer mindset. Each goal was simple enough that each group member could complete a goal by the time they came to class, and spend lab time debugging code and building hardware. Given more time the team would have liked to add enhancements, such as using parity bits and a parity check to be able to correct for misread data signals.