Microprocessor Demonstration using Quartus and a DE2-115 FPGA

Reuben Beeler
Department of Physics, University of California Santa Barbara, Santa Barbara CA 93106
(Dated: March 19, 2025)

This paper presents an implementation of the UC127B microprocessor using Quartus and a DE2-115 FPGA to demonstrate fundamental aspects of microprocessor operation and assembly language programming. The UC127B is a simplified processor with a 4-bit word size and a minimal instruction set, making it an effective educational tool for understanding low-level computing concepts. The system is designed to execute custom programs and display key processor states on FPGA-mounted seven-segment displays. Three demonstration programs are implemented: (1) an add-by-3 program that takes a 4-bit input and outputs the result while handling overflow conditions, (2) a counter that increments by a user-defined step size and wraps around upon exceeding the maximum value, and (3) an extended 8-bit counter that utilizes memory-mapped outputs to display its state. Each program illustrates key principles such as instruction sequencing, conditional branching, and memory-mapped input/output. Through this hands-on implementation, the project provides valuable insights into the structure and operation of microprocessors encountered in real experiments.

INTRODUCTION and METHODS

Understanding the fundamental operation of a microprocessor is essential for comprehending the inner workings of modern digital computing systems. This lab focuses on programming the UC127B microprocessor, a simplified model designed to illustrate low-level computer architecture and instruction execution. With only 13 instructions and a limited memory capacity, the UC127B provides an accessible platform for studying assembly language programming and control flow.

In this experiment, the microprocessor is interfaced with an FPGA, where its internal state, including program counter, instruction register, data address, and data bus values, is visually represented on the FGPA's seven-segment LED displays. The provided demonstration program exemplifies fundamental microprocessor operations, such as reading input data, performing arithmetic operations, and controlling program flow using branching and jumping instructions. A key objective of the lab is to develop an appreciation for how a processor executes instructions sequentially and interacts with memory-mapped input and output components.

We implement and test three programs for the microprocessor, namely an add-by-3 program, a counter that increments based on an input-defined step size, and an extended 8-bit counter utilizing memory-mapped outputs. These exercises reinforce key concepts in low-level programming, including direct memory manipulation, condition-based execution, and instruction sequencing. Through hands-on implementation, this lab provides critical insights into the complexities underlying modern computational systems.

ANALYSIS

The microprocessor, or CPU, in this lab is the module labeled UC127B in the upper left of the schematic in Fig. 1. This CPU interprets our instructions, which are loaded into the program at compile time and read from the lpm_rom1 ROM submodule at runtime. The inputs on the left are push-buttons that control execution of the program by using one-push button to step through each instruction, while another push-button resets the program to the beginning. The program's read-address (or program counter) is labeled "opadr[5..0]" and starts at 0. Note that the CPU operates with 4-bit words. The data input to the RAM is also 4-bit, but the address is 3-bit. So, there are only 8 addressable RAM values but 16 values in the address space, so some addresses are not used (specifically addresses with MSB of 1). For debugging purposes, the address 0xF=15 (outside of range of the RAM) is mapped to a single-word memory unit whose output is displayed on the FPGA's green LEDs. This is achieved using the lpm_dec3 decoder for address 15 and a DFF that is enabled by the decoder and the write-enable "dwrite" CPU output. What makes the address 0xF special (besides the fact that it is outside the RAM and is visualized on the green LEDs) is that we cannot read from the 0xF address, because that input address is instead reserved for reading the value on the four lowest FPGA switches "SW[3..0]". I do not know why the assignment was setup this way, but I can work with it.

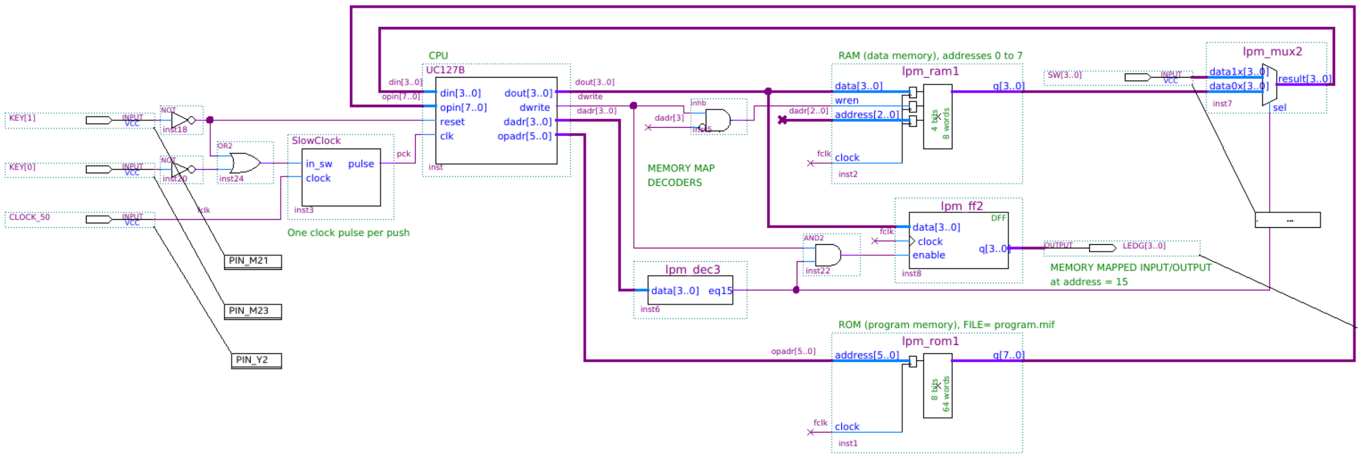

FIG. 1: The schematic of the microprocessor circuit.

The first task is to run the sample program below, `program.mif`, which is loaded into the ROM.

```
%Program to add 3 to input at dipswitches%
%  If output>F, then set output=F%

WIDTH=8;
DEPTH=64;

ADDRESS_RADIX=HEX;
DATA_RADIX=HEX;

CONTENT BEGIN  %modify this section for your program%

  0 :   A0 ;  %ANDC 0  clear accum (AND with 0)%
  1 :   9F ;  %ADD  F  load input (from memory map address F)%
  2 :   83 ;  %ADDC 3  add constant 3 to accum%
  3 :   70 ;  %SKC     skip next instruction if carried (>F)%
  4 :   06 ;  %JMP 06  jump so do nothing to accum%
  5 :   CF ;  % ORC F  set accum to F%
  6 :   5F ;  %STA  F  store value in memory map output (address F)%
  7 :   00 ;  %JMP 00  jump back to do again%

[8..3F]:  00 ;  %set rest of memory to 0%

END;
```

The program instructions are in the CONTENT clause, with the 6-bit ROM address on the left of the colon and the 8-bit output on the right. After each instruction is a semi-colon followed by a comment describing the instruction, which is very handy since this microprocessor definition is a toy instruction set that is not documented online. The main thing to know is that there is a single data register called `accum`, which is used for intermediate calculations. If an instruction says ANDC 0 to perform an AND operation with the Constant 0x00, the other operand that is AND'd with 0x00 is implied to be the data register `accum`. Let's now walk through the execution of this file, and then I will demonstrate it with a video. The first instruction, at address 0 has value 0xA0, corresponding to ANDC 0. This sets `accum` to 0 since anything bitwise-AND 0x00 is trivially 0x00. This is the go-to instruction for setting `accum` to zero. The next instruction is at the next address (1), which is ADD F. Rather than adding a constant value to `accum` like what would be done with ADDC, ADD F adds the value at *address* F to `accum`, which is zero. So, at the end of instruction 0x01, `accum` has the value stored at address F. But, remember that address F is a special address, where the input actually comes from the switches. So, whatever 4-bit word is set by the 4 right-most switches on the FPGA becomes the value in `accum`. Again, the program counter increments, so instruction 0x02 is next. Instruction 2 adds *constant* 3 (not an address) to accum using ADDC 3. Now, the value of accum is 3 more than the value that was on the slide switches at instruction 2 (call it X), so accum $= X + 3$. Realize that for $X > 12$, accum overflows since it is 4-bit and ends up (incorrectly) with a value less than X. To acknowledge this behavior, instruction 0x03 checks if the previous ADDC 3 instruction carried out, and then skips instruction 4 if so. This conditional check allows for multiple possible futures of the program, called branches. If ADDC 3 did not carry out, the program executes instruction 4, which is a jump instruction that skips to instruction 6. If the ADDC 3 instruction carried, then the program skips instruction 4, executes instruction 5 (ORC F) setting accum to F as a way to manage overflow, and then goes on to instruction 6. At this point, both branches are at instruction 6, so effectively what happened is that we have set accum $= \min(X + 3, 15)$. Finally, we execute instruction 6 which stores accum to address F (which is displayed on the green LEDs), and then the program returns to the beginning. A brief yet complete summary of this program is that takes 4-bit input X from the 4 switches and then outputs $\min(X + 3, 15)$ to 4 green LEDs in binary representation.

The video at this link demonstrates the execution of this program, with the program counter "opadr[5..0]" on the two left 7-segment LEDs, the instruction "opin[7..0]" on the next two 7-segments, and the output data address (where applicable) on the next 7-segment. Additionally, the last 7-segment display illustrates either the 4-bit CPU input "din[3..0]" (when switch SW[4] is HI) or CPU output "dout[3..0]" (when SW[4] is LO). Again, the right push-button is the clock, and the left-neighboring push-button resets the program.

The next program, program2.mif, has the following definition.

```
%Program to count up to F incrementing by input at dipswitches%
%  If output>F, then set output=0%

WIDTH=8;
DEPTH=64;

ADDRESS_RADIX=HEX;
DATA_RADIX=HEX;

CONTENT BEGIN  %modify this section for your program%

  0 :   A0 ;  %ANDC 0  clear accum (AND with 0)%
  1 :   9F ;  %ADD  F  load input (from memory map address F)%
  2 :   70 ;  %SKC     skip next instruction if carried (>F)%
  3 :   01 ;  %JMP 01  jump to next iteration%
  4 :   00 ;  %JMP 00  jump clear to 0 before next iteration%

[5..3F]:  00 ;  %set rest of memory to 0%

END;
```

This program counts up starting from zero, by steps given by the input number. When the count exceeds F, have the count resume from zero again. The first two steps are the same, which set accum to 0. Then, if it did not carry out, it counts up again by the value at the switches at instruction 1. If it did carry out, it first resets accum to 0 and then proceeds counting again. The counter value can be seen on the rightmost 7-segment display, so I decided not to write the value to address F to display on the green LEDs. This, however, could easily be fixed by inserting a "5F; %STA F%" instruction at each of the branches. A narrated video demonstration for program 2 is linked here.

Finally, we have program 3 illustrated in Fig. 2. This program implements a counter like before which increments by the 4-bit input on the switches, but the counter output now is 8-bit, so it is a lot more robust to overflow yet it requires special care (since the CPU is designed to operate on 4-bit words). This involves two 4-bit memory locations for the count value, and an additional memory-mapped output for the green LEDs "LEDG[7..4]", which I chose to be address 0xE. In the spirit of our other special address 0xF outside of RAM, I also set a memory-mapped input for address 0xE by modifying the multiplexer in the upper right. This time, however, the input just reads the value stored previously at 0xE, so it behaves like a normal RAM address (except displayed on the higher 4 green LEDs). I acknowledge that it is possible to use another RAM address (since we have extra) instead of adding address 0xE, but this works too.

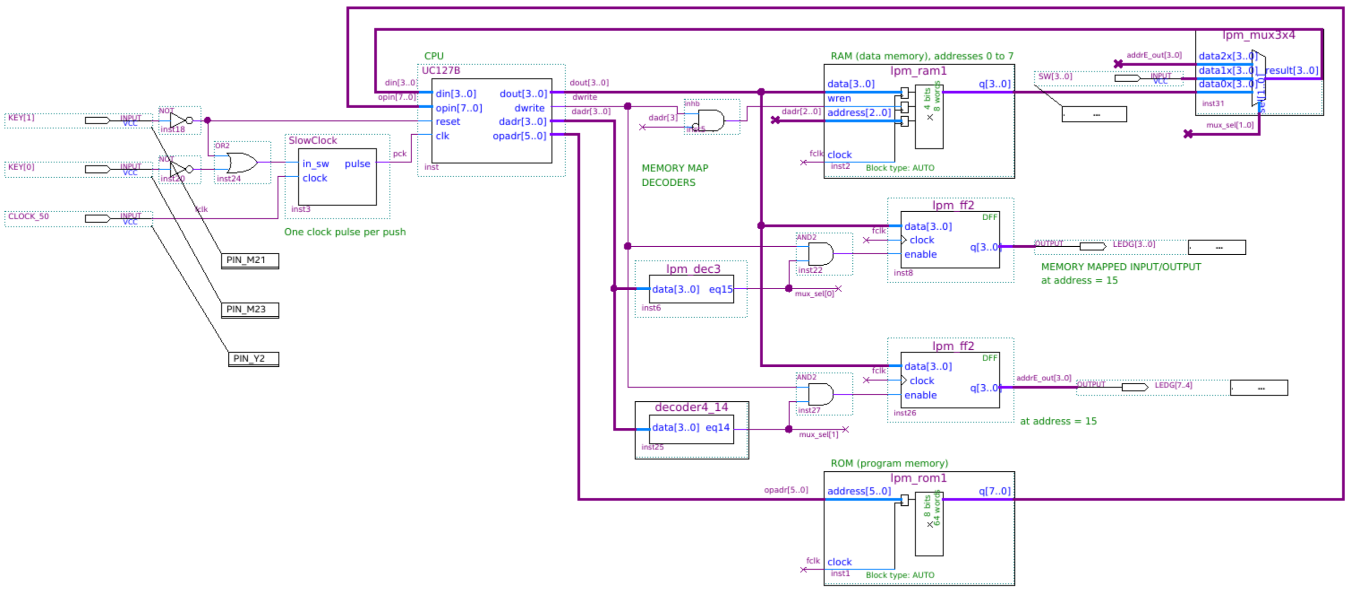

FIG. 2: The schematic of the microprocessor circuit with an extra memory address 0xE.

Here's the program file for program 3.

```
%Program to count 8-bit values (at RAM[E..F]) incrementing by 4-bit input at dipswitches%

WIDTH=8;
DEPTH=64;

ADDRESS_RADIX=HEX;
DATA_RADIX=HEX;

CONTENT BEGIN  %modify this section for your program%

  0 :   A0 ;  %ANDC 0  set data register to 0%
  1 :   5E ;  %STA  E  write data (0x00) to upper 4 LED bits%
  2 :   5F ;  %STA  F  write data (0x00) to lower 4 LED bits%

  3 :   9F ;  %ADD  F  load input (from memory map address F)%
  4 :   70 ;  %SKC     skip next instruction if carried (>F)%
  5 :   0F ;  %JMP 0F  goto NOT CARRIED block%
  6 :   5F ;  %STA  F  CARRIED:     write data to lower 4 LED bits%
  7 :   50 ;  %STA  0               write data backup at addr 0%
  8 :   A0 ;  %ANDC 0               set data register to 0%
  9 :   9E ;  %ADD  E               set data register to value at addr 0xE%
  A :   81 ;  %ADDC 1               add overflow to upper bits (increment by 1)%
  B :   5E ;  %STA  E               store incremented value back to addr 0xE%
  C :   A0 ;  %ANDC 0               set data register to 0%
  D :   90 ;  %ADD  0               restore lower 4-bit counter value from backup addr 0%
  E :   03 ;  %JMP 03               goto next iteration%
  F :   5F ;  %STA  F  NOT CARRIED: write data to lower 4 LED bits%
 10 :   03 ;  %JMP 03               goto next iteration%

[11..3F]:  00 ;  %set rest of memory to 0%

END;
```

Note that the first three instructions just set the count value and data register to 0. The main counter loop begins at instruction 3. We add the switch value to the current count and check if it overflowed the lower 4-bits. If so, we go to the CARRIED block and increment the upper 4 bits by one. (Note that we do not check if we overflow the upper 4-bits.) This requires reading the upper 4 bits from address E (instructions 8-9), then adding 1, then storing the result back to address E. But, beware that reading from address E overwrites the data register (previously called "accum"), so we first need to store it to the 4-bit output at address F because we need it later. We also need to store it somewhere else (I choose address 0) because we cannot read the values written to address F, since input from address F reads from the switches instead. These two "store" instructions that are required before incrementing the top 4 bits are done at instructions 6-7. Then, once storing the new upper 4 bits, we must restore the lower 4 bits to the data register for the next loop, which is done in instructions C-D. Finally, for the CARRIED branch, we finish with instruction 03 to return to the beginning of the loop where we increment again. If our increment did not carry over, we instead go the NOT CARRIED branch, store the new lower 4-bits (updating the green LEDs), and then similarly return to instruction 3, which is the beginning of the loop. A narrated video demonstration is linked here.

DISCUSSION

The implementation of the UC127B microprocessor on the DE2-115 FPGA provides a tangible demonstration of how a processor executes instructions and manipulates memory. The three implemented programs highlight essential aspects of low-level computing, including arithmetic operations, conditional branching, and the use of memory-mapped I/O for interfacing with hardware components.

The add-by-3 program exemplifies fundamental computation with overflow management, demonstrating how conditional branching allows the program to correctly cap values at the 4-bit limit. This exercise reinforces the significance of handling numerical overflow, a concept applicable to real-world processor design. Similarly, the second program showcases dynamic user-defined control over computation by allowing input-dependent step sizes. This introduces the concept of real-time user interaction with a processor, as commonly seen in embedded systems. Finally, the third program expands on previous concepts by extending the counter to 8 bits, requiring careful management of multiple memory locations and demonstrating how a limited-word-size processor can be adapted for more complex tasks.

One of the notable challenges in this project was the unconventional use of address 0xF as a memory-mapped I/O location for input and output, which necessitated careful handling of read and write operations. Additionally, the expansion of the counter to 8 bits required a workaround due to the processor's 4-bit word size, demonstrating the practical necessity of memory management techniques in microprocessor design. The ability to visualize program execution on the FPGA's seven-segment displays significantly enhances comprehension of instruction sequencing and control flow.

Overall, this project underscores the educational value of implementing a microprocessor on an FPGA platform. The hands-on experience gained from designing, coding, and debugging assembly-level programs solidifies foundational concepts in computer architecture. Future improvements could include expanding the instruction set, incorporating additional memory-mapped peripherals, or designing a more complex multi-cycle processor to extend computational capabilities.