8086

Simulated 16-bit Intel Microprocessor

Hollinsky, Paul - Kozera, Stefanie
Electrical and Computer Engineering Department
School of Engineering and Computer Science
Oakland University, Rochester, MI
e-mails: phollinsky@oakland.edu, stefaniekozera@oakland.edu

***Abstract*—Create an homage to the Intel 8086 microprocessor from 1979 with the Nexys 4 DDR FPGA development board. We have emulated a portion of the instruction set from the original processor, these instructions consisting of add, add with carry, jump, exchange register, the carry flag instructions, and halt. The emulation of the 8086 offered insight into the methodology behind a CPU, and because of the 8086's historical relevance, insight into the architecture of modern day CPU's as well.**

I. Introduction

The Intel 8086, despite being a very early microprocessor in the age of home computers, was quite a complex chip, with special pipelining and optimization taking place to ensure performance [1]. The 8088, the 8086's 8-bit cousin, was used in IMB's model 5150 PC, which spiked these CPU's popularity. Because of this, all modern-day CPUs in personal computers are derived from this original architecture, appropriately named "The x86 Architecture". The 8086 was the first in this long line of CPU development [2].

The goal of this project is to create a high-level simulation of the 8086. To do this, we have emulated a portion of the 8086's instruction set. We then ran simulations of these processes in action, and eventually implemented a physical example on the Nexys 4 DDR FPGA. Our motivation for doing this was our interest in the inner workings of CPUs, and the 8086 offered excellent insight into this. We learned about many topics pertaining to CPU emulation in class. The programming language VHDL was essential in completing this emulation. We certainly had to do some digging on our own in order to complete this project. We did a significant amount of research into the instruction set of the 8086, and in general how it functions. The breakdown of our emulation is addressed below.

II. Methodology

Because we are creating a high-level simulation, and have the power to run at much higher clock frequencies than the original hardware if necessary, we are not focused on pipelining or optimization. Instead, we have two basic running states, fetch/decode, and execute.

A. Primary State Machine

i. Fetch/Decode State

The fetch/decode state aims to read the entire instruction into internal registers for later execution. This is not a simple task, as Intel's architecture has many flags and within instructions which change the length of the instruction. Our code allows for this, and contains a LUT which returns the length and type of an instruction. See figure 5.

ii. Execute State

The Execute state uses a large multiplexor to execute the proper instruction according to the decode phase. These modules would theoretically be all of the 8086's instruction set, but in our case, we have only implemented a limited amount of opcodes, mainly relating to addition. The opcodes we have implemented are the following:

1. ADD – Adds an immediate value to the register. Note that this instruction does consider the carry flag
2. ADC – Add with Carry, this adds and immediate value to the register and includes the carry flag in the operation. Thus the carry flag is essentially treated as $c_{in.}$
3. JMP – An unconditional Jump, will transfer control to another portion of the program, this allows us to move to a different instruction, instead of simply moving through the sequence
4. JZ – A conditional Jump is the zero flag (ZF) is set to 1
5. JNZ – A conditional Jump if the zero flag (ZF) is set to 0
6. STC – Sets the carry flag to 1
7. CLC – Sets the carry flag to 0, or in other words, clears the carry flag
8. CMC – Inverts the carry flag, therefore, if CF = 1, then CF = 0 and vice versa
9. XCHG – Exchange, this exchanges the values of two registers, their values will flip flop
10. HLT – This enters the halt state, all execution will stop until the CPU is reset

Detailed descriptions of all of the original 8086 opcodes can be found in the Intel 8086 Family User's Manual. The created opcodes above are based off of the originals from this manual [1]. For our purposes, these are the instructions that can be executed during the execute state. Depending on what the decode state specifies, one of the following will be executed. The instruction is skipped if it is invalid.

These two primary states contain most of the functionality. When actually functioning, the CPU toggles between these two states, loading an instruction from memory in the decode state, and then executing the instruction in the execute state. These two states make up the bulk of emulation, however there are two other states that are used.

iii. Halt State

The halt state is entered with the HLT opcode is executed. When the halt state is entered, all functionality ceases. In our case, the program then stays put until the reset button is pressed, at which point the state machine restarts and functionality resumes as normal.

iv. Reset State

When the reset button is pressed, the reset state is entered. This sets us back at the beginning of the primary state machine, and sets all registers and flags back to their default values. Note that this is not necessarily zero, as not all registers and flags have a default of zero. From reset the CPU can then continue into the decode state.

See figure 5 for an illustration of this state machine.

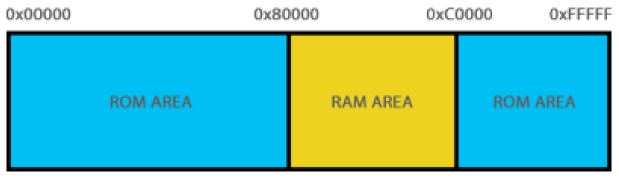
B. Address Segmentation Calculations

The processor works with segmented addresses, 16-bit addresses with 16-bit segment pointers. These are added to produce a 20-bit address, allowing for 1MB of addressable memory. The calculation shifts the segment by 4 bits to the left and then adds it to the selected address to create a 20-bit address.

C. Memory Controller

The memory controller connects directly to the 1MB addressable memory bus on the CPU. In this case, the memory controller devotes the area from 0x80000 to 0xC0000 to RAM and the remaining area to ROM, as shown in the following diagram. Simulating this much RAM was unnecessary and slowed down the simulation, so we have a mere 64 bytes of RAM being emulated. Any RAM outside these 64 bytes will not be written to and will return 0x0000 when read.

Figure 1: System Memory Map



D. ROM

ROM is generated using a NodeJS script which creates a VHDL case statement returning the values corresponding to the code. In this way, the ROM functions more like a very large LUT within the FPGA. The CPU could, however, execute instructions from the RAM as well, if user programmability was desired.

E. Instruction Info Module

The instruction info module connects directly into the main finite state machine. It takes the first two bytes of instruction from the processor, and combinationally returns a value for length and the operation defined for that opcode. This allows the processor to read the correct number of bytes for the instruction from memory. Again, this works as a large LUT.

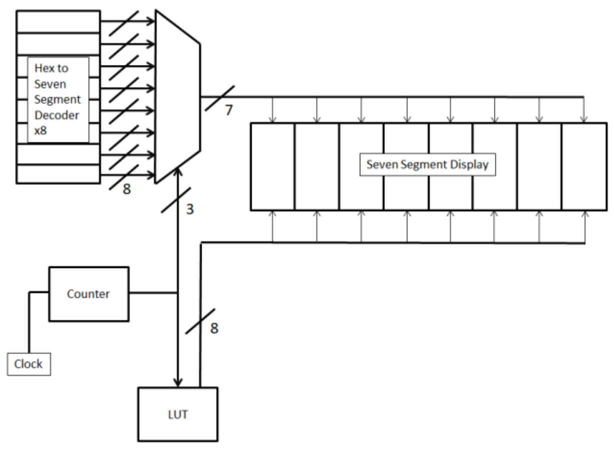
F. Clock Divider Module

The clock divider module is a variable speed clock generator used for our project. The output from this generator is either 10MHz or 4Hz and is controller with the "slow" signal. The generator expects a clock of 100MHz as an input. The clock speed can be changed on the fly and the circuit will guarantee no clock pulse is shorter than intended for maximum stability.

G. Display Module

The display module drives a bank of 8 seven segment displays at a refresh rate of 240Hz. It runs on the full 100MHz clock and has a self-contained clock divider for the 1.92 kHz necessary to drive all 8 displays at the required speed. It takes two 16-bit words as inputs and outputs all of the necessary display signals. The VHDL code generates 8 of the Hex to 7 Segment LUT converters, and then the 1.92 kHz clock is used to switch between outputs.

Figure 2: 7 Segment Display Driver Circuitry

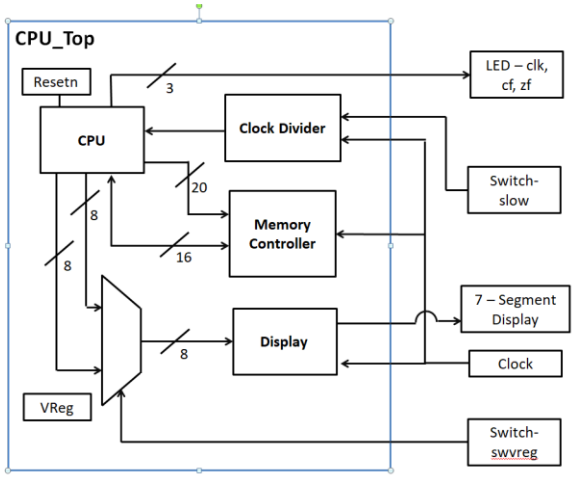


H. *Operation: Add Immediate*

The AddImmediate instruction is used to show how larger instructions would be structured. Currently this instruction is complete in one clock cycle; however it has a done flag which is checked by the state machine, so it could be configured to take multiple clock cycles if desired and would continue to execute properly, with the main FSM waiting on its completion. An instruction such as divide may take several clock cycles. Smaller instructions such as jump that can be expressed in just a few lines are not abstracted to their own file, and are executed directly in the execute state of the main FSM.

Below is a circuit diagram of the top file of this system. This links all portions of the simulation together, so they may work in unison.

Figure 3: CPU_Top, a circuit diagram showing the relationship between all portions of the emulation



III. Experimental Setup

Project testing has taken place using GHDL and GTKWave due to size concerns with the Vivado software package. GHDL is a VHDL analysis tool, and GTKWave is a waveform viewer. We used both in the creation of this program. This setup is cross platform and allows quick iterative testing.

NASM (The Netwide Assembler) was used to compile a testbench of assembly code into machine code. Special padding instructions are used to make sure that the first instructions fall on the reset vector. The generated file will be exactly 1MB, and the addresses of bytes in the file will be identical to the ones built into the ROM. NASM must be set, in this case, to only use the 8086 instruction set, rather than later editions used in more modern processors

Vivado was used for the timing simulation, as well as implementation and generating the bitstream. We then uploaded the program to the Nexys 4 FPGA development board using the SD card.

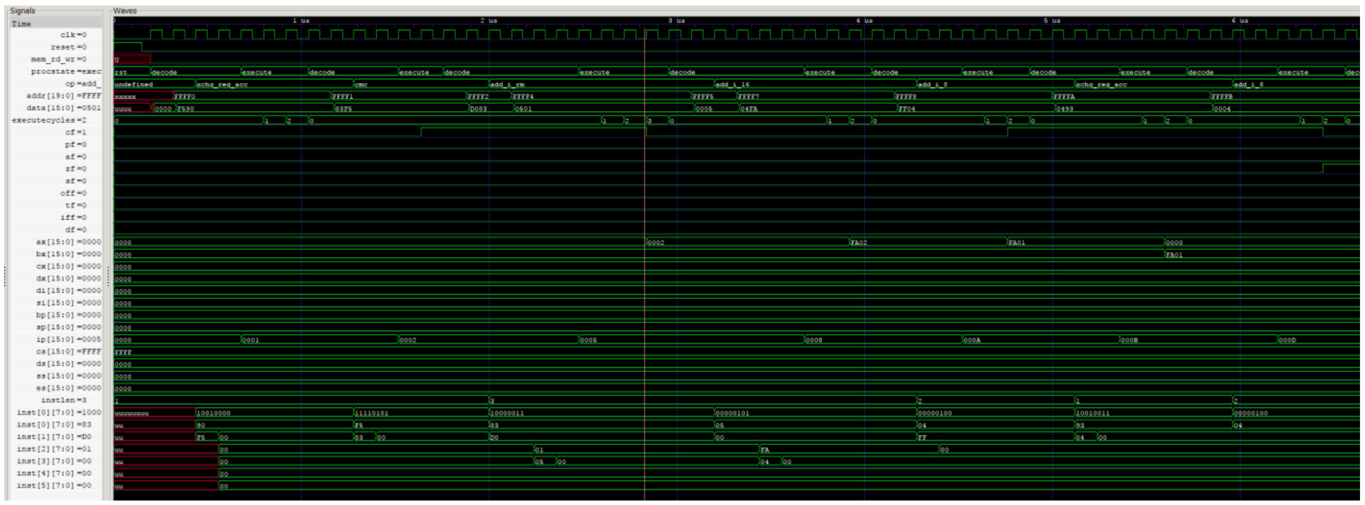
The testbench simply works by simulating CPU_Top in its entirety and briefly activating resetn. Then, GHDL allowed us to view all internal signals and watch program execution occur.

We expected that using these packages we would be able to simulate a portion of the CPU accurately, and that it would behave roughly like we expected. We found this to be true.

IV. Results

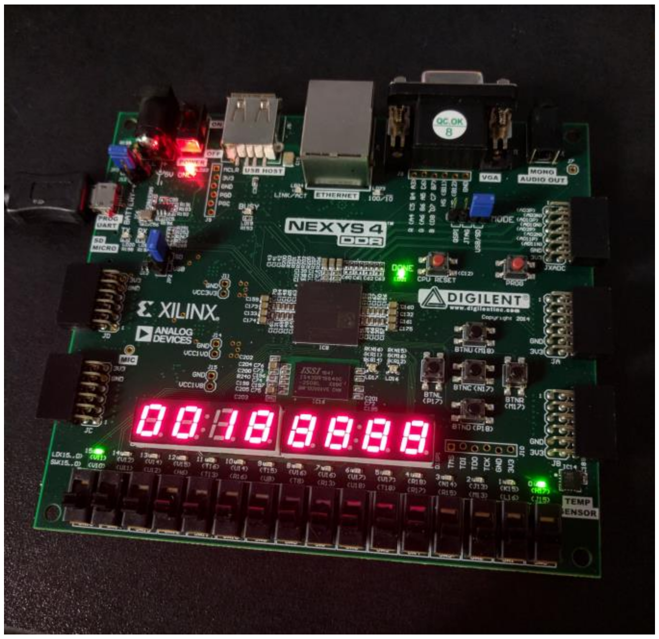
We discovered that with diligence, it is indeed possible to simulate a more advanced CPU such as the 8086 on an FPGA. The logic to do so is often quite complicated, and with the vast instruction set of the 8086, impossible to completely implement in our time limit. Despite this, we were still able to implement a portion of the instruction set, which demonstrates the general idea how the CPU works.

Figure 4: Picture of the finished simulation



The testbench is excellent for examining various processor states that you would not be able to see when watching the processor run on the FPGA.

Below is a picture of the Nexys 4 board running the emulation. The leftmost four seven segment displays always displayed the current instruction pointer. The rightmost four could be set to display either ax or bx. The displays are switching so quickly that it is impossible for the camera to capture the values on screen.



CONCLUSIONS

This emulation offered valuable insight into the inner workings of a rather complicated CPU. We learned through this process that although CPU's are incredibly complicated engineering wise, when broken down they can be understood, and eventually emulated. To further this project, we could implement additional opcodes from the 8086's original instruction set. Because of the scope of this project, we were only able to implement a limited amount and the uses for this CPU were highly limited. With all opcodes implemented, this emulation could act as a fully functioning CPU, and would be able to run programs created with the x86 architecture for the original 8086.

We have interest in adding VGA support to this project. VGA support would allow us to use an external monitor. With this, we could write the display code in such a way that we could see all of the registers at once. The displays were a limitation with the development board for us. As the project stands now, we are limited to only seeing a few registers at a time. This project itself would be very impressive if one could see all the registers undergo modifications simultaneously.

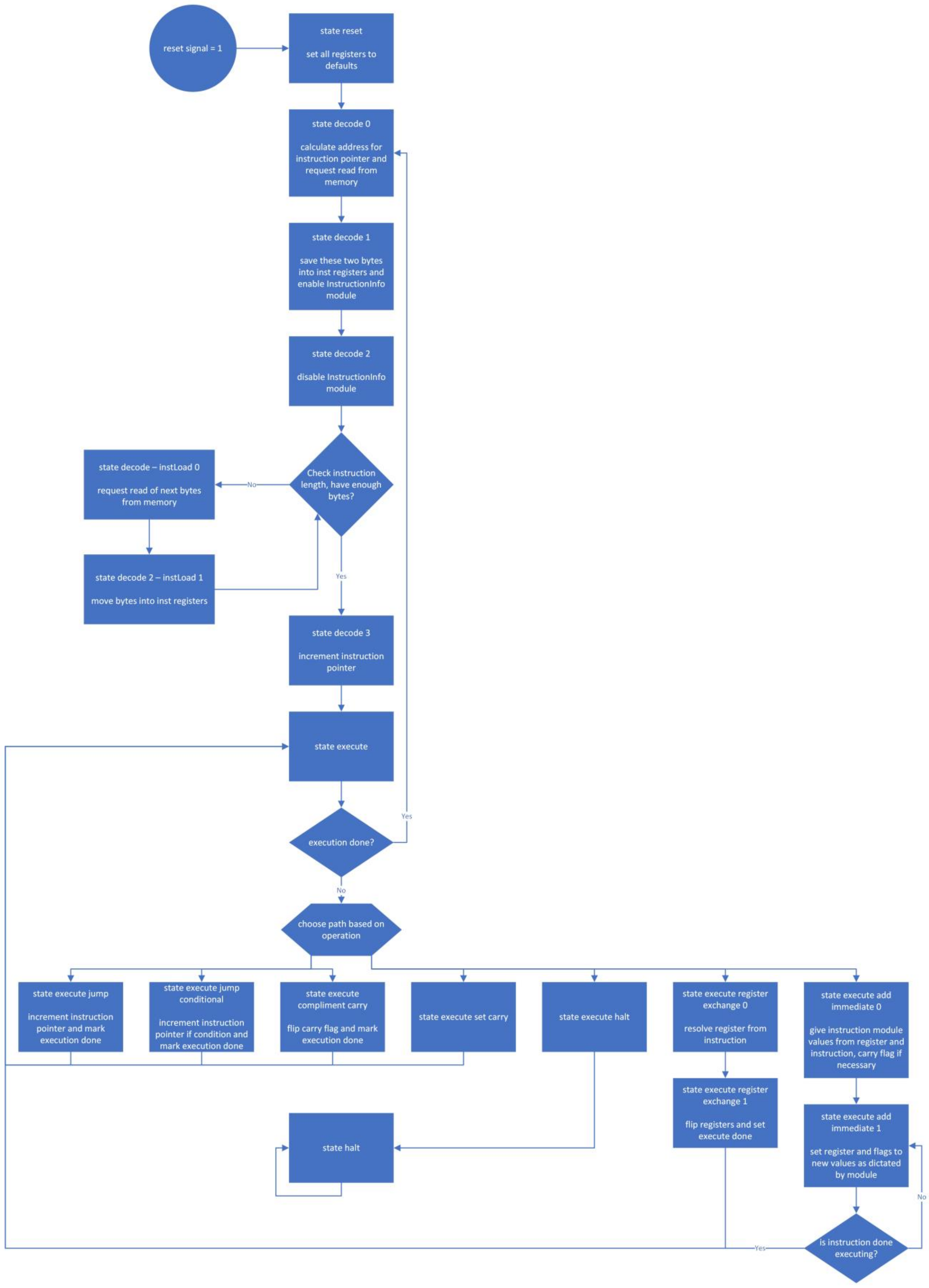
As well, if attempting to implement programs meant for the original 8086, there would likely be compatibility issues with certain interrupts. Due to the fact that our code is running on the "bare metal" CPU currently and the system is not running a BIOS, any code which made use of BIOS functions would not work until a suitable BIOS ROM was built.

REFERENCES

[1] Intel Corporation. (1979). Intel 8086: Family User's Manual. MA: Intel Corporation.

[2] Edwards, Benj. "Birth of a Standard: The Intel 8086 Microprocessor" PC World. 16 Jun. 2008, www.pcworld.com. Accessed 7 Dec. 2015.

Figure 5: Primary State Machine in the CPU Emulation


reset signal = 1
state reset
set all registers to defaults
state decode 0
calculate address for instruction pointer and request read from memory
state decode 1
save these two bytes into inst registers and enable InstructionInfo module
state decode 2
disable InstructionInfo module
Check instruction length, have enough bytes?
No
state decode – instLoad 0
request read of next bytes from memory
state decode 2 – instLoad 1
move bytes into inst registers
Yes
state decode 3
increment instruction pointer
state execute
execution done?
Yes
No
choose path based on operation
state execute jump
increment instruction pointer and mark execution done
state execute jump conditional
increment instruction pointer if condition and mark execution done
state execute compliment carry
flip carry flag and mark execution done
state execute set carry
state execute halt
state execute register exchange 0
resolve register from instruction
state execute add immediate 0
give instruction module values from register and instruction, carry flag if necessary
state execute register exchange 1
flip registers and set execute done
state execute add immediate 1
set register and flags to new values as dictated by module
state halt
No
is instruction done executing?
Yes