PDP-8 ISA Simulator

Deborah Denhart  
Jeremiah Franke

# Purpose

The purpose of this project was to familiarize ourselves with the instruction set architecture (ISA) for the PDP-8 processor by creating a PDP-8 simulator. This simulator is required to implement all of the PDP-8 instructions besides Microcode 3 and IO system calls. This project was to give us hands on demonstration of the ISA of a fairly simple processor so that we could better grasp the complexities that go along with implementing a RISC type ISA.

# Requirements

* Write an ISA level simulator for the PDP-8 architecture
* Generate a memory trace file with the format:
  + <type> <address>
  + <type> can be:
    - 0 for data read
    - 1 for data write
    - 2 for instruction fetch
  + <address> will be in octal format
  + The data that is read or written will not be displayed
* The different types of input can be:
  + Binary
  + ASCII hexadecimal
  + ASCII octal
* All instructions will be implemented
  + Except for I/O and group 3 microinstructions
    - Handled as NOPs
    - Has a clock cycle of 0
    - Print a warning
  + Must be clock accurate
  + Indirect addresses add 1cycle
  + Auto increment adds 2 cycles
  + Start at address 200 or assume the first address is the start
* Generate a summary at the end of execution
  + Total number of instructions executed
  + Total number of clock cycles consumed
  + Number of times each instruction type (by mnemonic) was executed

# Design

This simulator was designed to conform to the specifications provided by the many documentation resources for the ISA of the PDP-8. We very carefully tried to stay true to the original design of the PDP-8 architecture by emulating the framework that was used internally in the PDP-8 to fetch, execute, and decode instructions passed to it in a variety of programming formats. These assembly code formats were designed with the notion that the simulator needed to be able to accept binary, octal, and hexadecimal formatting for the proper instructions. This simulator is not designed to stop a user from running code that is correct but results in an incorrect final product, i.e. a forever loop. This simulator also features clock accuracy by way of simulating the time required to finish the different operations codes. This simulator does not currently have a graphic user interface, GUI, nor does it simulate the IO operations.

The simulator was designed in C++ with some libraries custom built for handling the conversion of the register values from octal to binary, strings, numerals, and two’s complement. The major libraries that handle this we implemented to make it easier to construct a GUI that would wrap around the command line code. The design also requires that conditional flags be used for choosing the correct input information type between octal, hex, and binary. There are additional flags for debugging and trace file creation for statistics and memory dumps.

# Implementation

We started with the paged memory file as our base and expanded on that by building structured, class, subsystems that would handle access to and from the memory subsystem. We then built upon the basic register/memory class system to incorporate reading in a file that had the address format specified by the command line interface. We then built a variation of the bitset library to handle the implementation of the register values that would have to be easily converted from octal to decimal to binary, as well as finding the complements of the above. The class also provides the simulator to be able to produce the data in string formats for legibility, and to potentially be hooked into a GUI in the future. Finally we built the opcode structure directly within our execution control class.

The program is written in C++ with custom classes and a custom library to control the data conversions. This is currently a command line program that requires flags for handling file input flow as well as debugging. The file input can be specified as a local path preceded by the –f flag. The input data type is the –o, -v flags. O is for octal, v is for hexadecimal and if no flag is given then binary is the default input type. –d specifies the debug flag for the program.

# Testing

**Op codes:**

The op codes are plentiful in the PDP-8 , and the first six op codes are :

1. AND
   1. Bit by bit logical and of the accumulator and the contents of the effective address.
   2. DEBUG Execute: Memory Reference 0250 PC: 0204
   3. DEBUG opcode: 0, AND
   4. DEBUG offset: 412 Memory Reference
   5. DEBUG load instruction: Memory Reference
   6. DEBUG Execute: 0002
   7. DEBUG ALU: and is 0000
2. TAD
   1. Add the value pointed to by the effective address to the value in the accumulator
   2. Address 250 contains the value 2, Address 251 contains the value 3
   3. DEBUG Execute: Memory Reference 0250 PC: 0202
   4. DEBUG opcode: 1, TAD
   5. DEBUG offset: 412 Memory Reference
   6. DEBUG load instruction: Memory Reference
   7. DEBUG ALU: sum is 0002
   8. DEBUG Execute: Memory Reference 0251 PC: 0203
   9. DEBUG opcode: 1, TAD
   10. DEBUG offset: 412 Memory Reference
   11. DEBUG load instruction: Memory Reference
   12. DEBUG ALU: sum is 0005
3. ISZ
   1. Increment the contents of the effective address and skip the next instruction if the contents are zero.
   2. The contents of 253 are 7777 in octal which is -1. So the first run through it will skip the jump instruction beneath it.
   3. DEBUG Execute: Memory Reference 0253 PC: 0205
   4. DEBUG opcode: 2, ISZ
   5. DEBUG offset: 253 Memory Reference
   6. DEBUG load instruction: Memory Reference
4. DCA
   1. Deposit the value of the accumulator into memory and clear the accumulator
   2. The ALU was the value of 0005 in this test as the DCA was called after the two TAD ops.
   3. DEBUG opcode: 3, DCA
   4. DEBUG offset: 412 Memory Reference
   5. DEBUG load instruction: Memory Reference
   6. DEBUG ALU: clear is 0000
   7. DEBUG Execute: Memory Reference 0250 PC: 0204
5. JMS
   1. Jump to subroutine. Saves the return address at the jump destination address, then increments the pc to start on the instruction following the jump.
   2. DEBUG opcode: 4, JMS
   3. DEBUG offset: 300 Memory Reference
   4. DEBUG load instruction: Memory Reference
   5. DEBUG Execute: Memory Reference 0200 PC: 0200
6. JMP
   1. Direct jump with no return address. The effective address is the jump to address unless indirection is used.
   2. DEBUG opcode: 5, JMP
   3. DEBUG offset: 300 Memory Reference
   4. DEBUG load instruction: Memory Reference
   5. DEBUG Execute: Memory Reference 0200 PC: 0200
7. MicroInstructions

## Parsing different types of file input:

When the right file is run with the right flag ([default]/.bin, -o/.obj, -v/.mem), the three outputs should all be equal. Proper error handling should occur when a flag is incongruent with the data format within the file.

|  |  |  |  |
| --- | --- | --- | --- |
| Data type | Binary | Octal | Hexadecimal |
| Flag | [default] | -o | -v |
| File Input | -f add01.bin | -f add01.obj | -f add01.mem |
| -f add01.bin | DEBUG address: 0200  DEBUG address: 7300  DEBUG address: 1250  DEBUG address: 1251  DEBUG address: 3252  DEBUG address: 7402  DEBUG address: 5200  DEBUG address: 0250  DEBUG address: 0002  DEBUG address: 0003  DEBUG address: 0000 | Error: Exceeded memory space... | Error: Exceeded memory space... |
| -f add01.obj | Error: Exceeded memory space... | DEBUG address: 0200  DEBUG address: 7300  DEBUG address: 1250  DEBUG address: 1251  DEBUG address: 3252  DEBUG address: 7402  DEBUG address: 5200  DEBUG address: 0250  DEBUG address: 0002  DEBUG address: 0003  DEBUG address: 0000 | Error: Exceeded memory space... |
| -f add01.mem | Error: Invalid binary file format... | Error: Exceeded memory space... | DEBUG address: 0200  DEBUG address: 7300  DEBUG address: 1250  DEBUG address: 1251  DEBUG address: 3252  DEBUG address: 7402  DEBUG address: 5200  DEBUG address: 0250  DEBUG address: 0002  DEBUG address: 0003  DEBUG address: 0000 |

# Source Code