PDP-8 ISA Design

ECE586

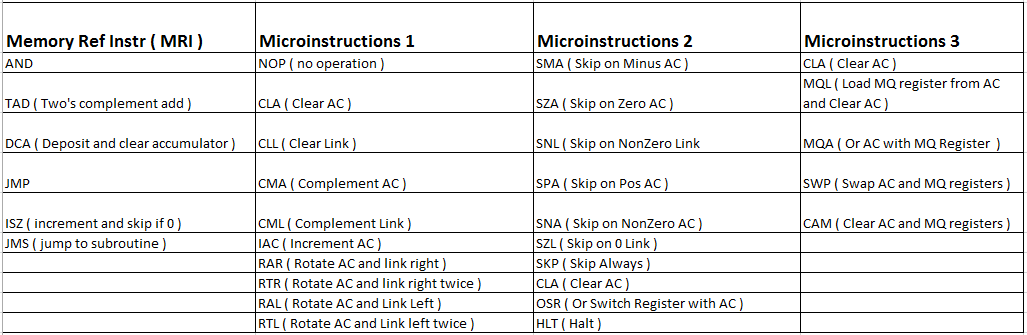
**Table of Contents…**

**Introduction**

The PDP-8 Architecture is a 12-bit minicomputer based on simple instruction set and powerful operations in one instruction cycle. It was designed by Digital Equipment Corporation as a “Straight-8” and was later designed with CMOS microprocessors and known as the “CMOS-8s.” PDP-8’s significance is with its simplicity, low cost, and expandability. The PDP-8 registers are Accumulator (AC), Link (L, carry from AC ), Program Counter (PC), Memory address register (MAR), and Memory buffer register (MBR). It has 4K main memory with 32 pages of 128 12-bit words.

The instruction set is made of 12-bit instruction words. The 3 high order bits encode the operation. Bit 3 determines whether the address is direct or indirect. Bit 4 indicates if the page is current of page 0. Bits 5 to 11 gives the 7-bit offset address. The assembler places constant values for arithmetic in the current page. For cross-page jumps and subroutine calls, indirect addressing in the current page is used.

The instructions are listed below:

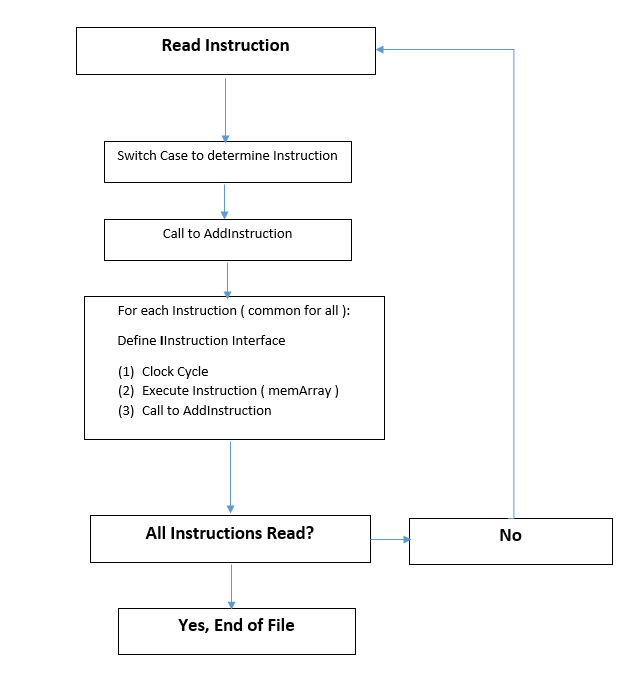


The objective of this project is to create a PDP-8 Instruction Set Architecture simulator to execute instructions as defined for PDP-8.

**Methods**

**Program Design:**

Our first step was to design the framework. A layout flowchart of the framework is shown below, Fig.



The program is designed to read in instructions, determine instructions, add them into an array of instructions, and execute. This is done until the end of file is reached. For each instruction, clock cycles will be added accordingly.

The designed framework is separated into different modules defined below.

There are defined libraries that C# allows us to access directly. These references ae added into the References directory as needed.

The Classes directory includes the modules for defining variables for instruction items and accessing memory, ie. PC, AC, Mem Address, and access to the memory array that was read in from the ObjectLoader class. The main modules are the InstructionItems to define variables to access the PC, BranchType, Memory Address, and whether a branch was taken. The InStructionResults, for keeping track of the number of executions for each Opcode. The MemArray class accesses the memory array read in from the ObjectLoader class. The Operation class defines the variables for instructions, finalMemAddress after execution, clock cycles, and addressing mode.

The Instructions directory includes separate modules for each instruction, with executions for each. All of the microcode instructions are grouped into the OprInstrution class. The executions are defined as separate functions of that class.

The Constants module define the opcode value for each instruction. The Logger class writes to a MemTrace file.

The ObjectLoader class is where the object file is read in. All lines are read into the MemArray, until the end of file is reached. The first address recognized is read into MemArray position 200, and each incremental address will be read in at indexed arrays of 50. Each data line read in will follow the starting address. Once all lines are read into the MemArray, the FinishedArray is created and can be accessed in MemArray and LoaderResult.

ProgramExecuter class:

The Utils class decodes the address segments, for opcode, I/D, 0/Current page, and address.

The Emulater Tests Project allows for separate tests to be created for each instruction and run for debugging.

**Instructions**

All Instruction Items are defined in the InstructionItems class:

public int accumulatorOctal;

public int pcCounter;

public int MemoryValueOctal;

public int MemoryAddress;

public int MicroCodes;

public int IOCodes;

public bool LinkBit

These items are passed into each instruction, manipulated upon execution of the respective instruction, and passed back with continued execution. The BranchTraceRow class manages whether a branch was taken and what type of branch is taken. The program will provide a branch trace to indicate whether an instruction was a conditional branch, unconditional branch, or subroutine. Each branch instruction is labeled as follows:

|  |  |
| --- | --- |
| **Conditional:** |  |
|  | ISZ |
|  | SMA |
|  | SZA |
|  | SNL |
|  | SPA |
|  | SNA |
|  | SZL |
|  |  |
| **Unconditional:** |  |
|  | JMP |
|  | SKP |
|  |  |
| **Subroutine** |  |
|  | JMS |

Each opcode is provided in a separate module. The Opcode and cycles are defined as follows:

|  |  |  |
| --- | --- | --- |
| **Mnemonic** | **Opcode** | **Cycles** |
| AND | 0 | 2 |
| TAD | 1 | 2 |
| ISZ | 2 | 2 |
| DCA | 3 | 2 |
| JMS | 4 | 2 |
| JMP | 5 | 1 |
| IO | 6 | 0 |
| u-instructions | 7 | 1 |

The MRI instructions each take 2 cycles to execute. These instructions access memory by accessing the contents of the Accumulator, Effective Address, and Program Counter. They are defined as the first 6 instrutions in the table above. Each microinstruction is included in the OprInstruction class and takes 1 clock cycle to execute. They operate on the accumulator link pair. The microinstructions group 1 function to clear, complement, rotate and/or increment the accumulator-link pair. The microinstruction group 2 perform conditional branching/skip. The miroinstruction 2 operate on the MQ register. The I/O instructions operate on devices such as a keyboard stroke.

Designed Architecture ( Source codes, outline/overview of code )….

**Algorithms for Instructions? ( may not be necessary )**

**Testing:**

Tests for each instruction is defined in the Emulator Test Project. Multiple tests are created to determine if the expected results are produced for each instruction. An example test program is shown below:

**Conclusion:**