PDP-8 ISA Design

ECE 486/586

**Table of Contents…**

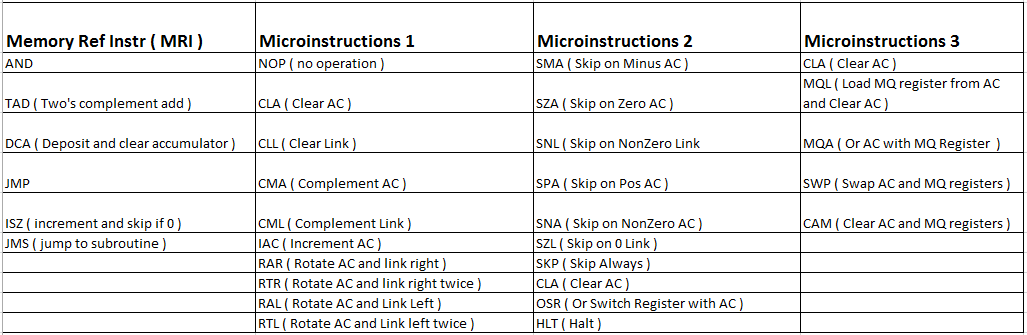
**Introduction**

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 include the Accumulator (AC) with Link (L, carry from AC ), Program Counter (PC) and Instruction Field (IF), the Data Field and Indirect Addressing, and the Multiplier Quotient (MQ). It has 4K main memory with 32 pages of 128 12-bit words. The 12-bit addressing defines the page ad offset.

Memory access uses the 12 bit Central Processor Memory Address register and the Memory Buffer Register, which holds the data. These two registers, along with the 2 bit Instruction Register to hold the opcode and the 12 bit Console Switch Register, is part of the CPU Register set.

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 list of instructions for PDP-8 are listed below:



Memory Reference instructions access memory by accessing the contents of AC,PC, and EA, along with manipulating the Effective Address. Group 1 and 2 instructions operate on the AC and Link. Group 3 instructions operate on the AC and MQ registers, which will not be implemented in our design. PDP-8 also includes IO Test Instructions that operate on device operations. These will also be excluded from this design.

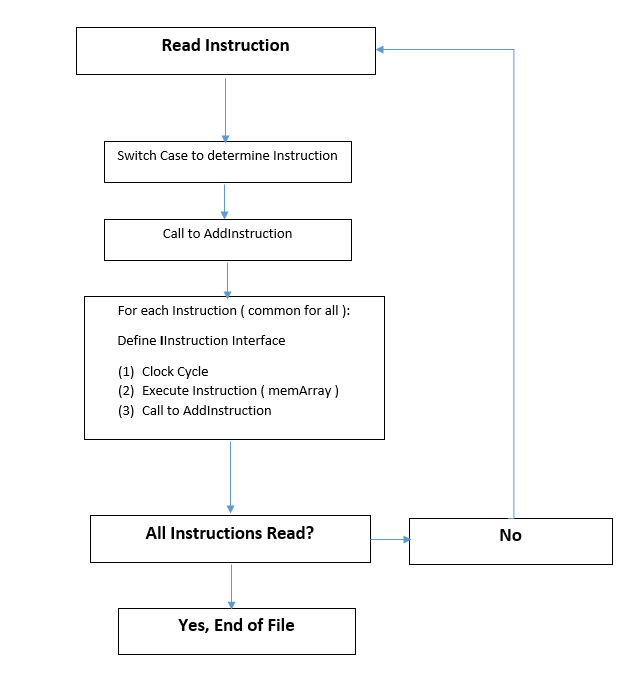
The objective of this project is to create a PDP-8 Instruction Set Architecture simulator to execute instructions as defined for PDP-8.

**Methods**

**Step 1:**

**Program Design and Layout:**

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 an object file that consists of octal addresses and data. The data defines the instructions. We have created a memory array that will hold the data at specified addresses, starting at 200. Addresses will be encoded with a starting 1, while data will follow with a leading 0. The instructions to execute are encoded in the data held in specified memory locations. Data is added at incrementing array locations, until another address is specified for the next set of instructions. This is done until the end of file is reached.

The designed framework is separated into different modules for modularity.

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

ProgramExecuter class executes the main program.

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 the specified address. Each data line read in will be placed into incremental addresses from the specified starting address. Once all lines are read into the MemArray, the FinishedArray is created and can be accessed in the MemArray and LoaderResult classes.

The Utils class operates to decode the instruction addresses. Each instruction will be decoded to determine the opcode, addressing ( Direct or Indirect and Page 0 or Current Page ), and address offset. An address is decoded from memory by first determining the page by analyzing bit 4, whether is it not set for page 0 or whether it is set and is the current page. If the page is 0, the effective address is equal to 0 concatenated with the offset. If the page is the current page, the effective address is the page concatenated with the offset. Bit 3 of the address is analyzed for whether the addressing is indirect or direct. If bit 3 is not set, the address is direct. If bit 3 is set, the address is indirect and the effective address is the contents of the page concatenated with the offset, either page 0 or instruction page. Audoindexing is used for indexing through arrays, which uses indirect addressing from 00108 ( 810) to 00178 (1510).

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. The main modules are the InstructionItems to define variables to access the PC, BranchType, Memory Address, and whether a branch was taken. The InstructionResults class, keeps 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 Constants module define the opcode value for each instruction. The Logger class writes to a MemTrace file, which produces the results for memory access.

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

The Instructions directory includes separate modules for each instruction, with executions for each. The MRI instructions are defined in their separate modules since they access memory. These include the AND, TAD, ISZ, DCA, JMS, JMP instruction. All of the microcode instructions are grouped into the OprInstrution class. The executions are defined as separate functions of that class if more than one line of instruction is needed, else the operations will be executed as soon as the instruction opcodes are determined. The OprInstruction class will determine the opcode for each instruction to execute accordingly, if the instruction is not found in this list, they are considered IOs or microinstruction 3s. These instructions are not implemented in the design and will just be incrementing our PC to the next instruction and increase the number of instructions executed.

**Instructions**

All Instruction Items may operate on defined variables 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, operated on upon execution of the respective instruction, and passed back with continued execution. Not all items will be operated on, and thus will just be passed back as is for the next instruction. 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 group 3 operate on the MQ register. The I/O instructions operate on devices such as a keyboard stroke.

Instructions within each microinstruction group 1 and 2 will operate based on priority when there are combinations of instructions. Microinstruction 1s will give priority in the following order:

|  |
| --- |
| **Micro 1** |
| **1110| CLA | CLL | CMA | CML | RAR | RAL | 0/1 | IAC** |
| **| 1 | 1 | 2 | 2 | 4 | 4 | 4 | 3** |

For example, CLA has priority over CMA. Our design will mask and determine order of execution based on which instruction will need to execute first. This is done by checking for bit 8 ( mask 0x80 ) then bit 7 ( mask 0x40 ) first before checking for bit 6 ( mask 0x20 ), and so on respectively. Since the instructions will always execute in order,

Please refer to the source code, page.

**Step 2:**

**Testing:**

Tests for each instruction is defined in the Emulator Test Project. Individual test units are created to determine if the expected results are produced for each instruction. Each instruction will be tested using the template below. This is an example of a segment of the AND instruction test case.

[TestMethod]

public void TestAndArgumentPassthrough()

{

InstructionItems TestItems = new InstructionItems()

{

accumulatorOctal = 0000,

LinkBit = true,

MemoryAddress = 0,

MemoryValueOctal = Convert.ToInt32(7777.ToString(), 8),

pcCounter = 1,

InstructionRegister = Convert.ToInt32(7402.ToString(), 8)

};

InstructionResult ExpectedItems = new InstructionResult()

{

accumulatorOctal = 0000,

LinkBit = true,

MemoryAddress = 0,

MemoryValueOctal = Convert.ToInt32(7777.ToString(), 8),

pcCounter = 2,

InstructionRegister = Convert.ToInt32(7402.ToString(), 8),

SetMemValue = false

};

IInstruction TestAndInstruction = new AndInstruction();

InstructionResult ActualResult = TestAndInstruction.ExecuteInstruction(TestItems);

Assert.AreEqual(ExpectedItems.accumulatorOctal, ActualResult.accumulatorOctal);

Assert.AreEqual(ExpectedItems.LinkBit, ActualResult.LinkBit);

Assert.AreEqual(ExpectedItems.MemoryAddress, ActualResult.MemoryAddress);

Assert.AreEqual(ExpectedItems.MemoryValueOctal, ActualResult.MemoryValueOctal);

Assert.AreEqual(ExpectedItems.pcCounter, ActualResult.pcCounter);

Assert.AreEqual(ExpectedItems.InstructionRegister,

ActualResult.InstructionRegister);

Assert.AreEqual(ExpectedItems.SetMemValue, ActualResult.SetMemValue);

}

[TestMethod]

public void TestAndOperation()

{

//First is test 0 anded with 0 - need to initialize the instruction items for the first time

InstructionItems TestItems = new InstructionItems()

{

accumulatorOctal =0000,

LinkBit = true,

MemoryAddress = 0,

MemoryValueOctal = 0000,

pcCounter = 5649,

InstructionRegister = Convert.ToInt32(7402.ToString(),8)

};

InstructionResult ExpectedItems = new InstructionResult()

{

accumulatorOctal = 0000,

LinkBit = true,

MemoryAddress = 0,

MemoryValueOctal = 0000,

pcCounter = 5650,

InstructionRegister = Convert.ToInt32(7402.ToString(), 8),

SetMemValue = false

};

IInstruction TestAndInstruction = new AndInstruction();

InstructionResult ActualResult = TestAndInstruction.ExecuteInstruction(TestItems);

Assert.AreEqual( 0, ActualResult.accumulatorOctal);

/\* Test Cases for AND instruction, all cases produce PC + 1 \*/

//Test all 1s ANDed with all 0s, results in all 0s

TestItems.accumulatorOctal = 0000;

TestItems.MemoryValueOctal = Convert.ToInt32(7777.ToString(), 8);

TestItems.pcCounter = Convert.ToInt32(0.ToString(), 8);

ActualResult = TestAndInstruction.ExecuteInstruction(TestItems);

Assert.AreEqual( 0, ActualResult.accumulatorOctal);

Assert.AreEqual(1, ActualResult.pcCounter);

}

The TestAndArgumentPassthrough and TestAndOperation functions are used for an initial test and initialization of the instruction. Each test item will be defined as TestItems and a check for expected results will be defined in the Assert.AreEqual function. The example above shows one test case for the AND instruction. This instruction executes to AND the AC with the memory Value and places the result into the AC. For all instructions, there will be PC checks for each test case to ensure that the instruction execution is incrementing to the next instruction accordingly, with the exception of Conditional and Unconditional Branch instructions. These instructions will skip the next instruction if conditions are met, or unconditionally in the case of a SKP instruction. This would require the PC to increment by 2, to correctly skip the next instruction.

The test units were created for all instructions, tested, and debugged to correct instructions accordingly.

The test design used to test each instruction is shown below:

….

This chart lists all instructions and test cases required for each. The values to check and results are included for reference while testing the solution.

**Conclusion:**

**References**