|  |
| --- |
| **PDP-8 ISA Simulator Project** |
| ECE 486 – Computer Architecture |
|  |
| Evan Sprecher, Meng Lei, Sean Hendrickson, Ommaimah Hussein Mohammed |
|  |

Winter 2015

Table of Contents

[1. Project Overview 3](#_Toc413720986)

[2. Project Requirements 3](#_Toc413720987)

[3. Design Specification and Implementation 3](#_Toc413720988)

[4. Testing Procedures and Results 8](#_Toc413720989)

[5. PDP-8 ISA Simulation Source Code 33](#_Toc413720990)

# Project Overview

The purpose of this project is to simulate the PDP-8 Instruction Set Architecture. The PDP-8 ISA simulator should be capable of generating memory trace files.

**Inputs:**

This simulator will take an ASCII object code file. The various input file we will use to test the simulator will be produced by the PDP-8 Assembler provided.

**Outputs:**

After the completion of the execution, the simulator should generate a brief summary including:

* The total number of instructions executed.
* The total number of clock cycles consumed.
* The number of times each instruction type (by mnemonic) was executed.

In addition, the simulator must generate a trace file of all memory access (instruction and data) in order.

# Project Requirements

The following are requirements that need to be implemented into the project:

1. The simulator must correctly simulate the entire PDP-8 instruction set.
2. The simulator must be “clock accurate”.
3. The simulator must output correct value of all instructions executed.
4. The simulator must correctly implement memory read/write (store/load).
5. The simulator must correctly calculate the Effective Address.
6. The simulator must correctly implement each instruction type.

# Design Specification and Implementation

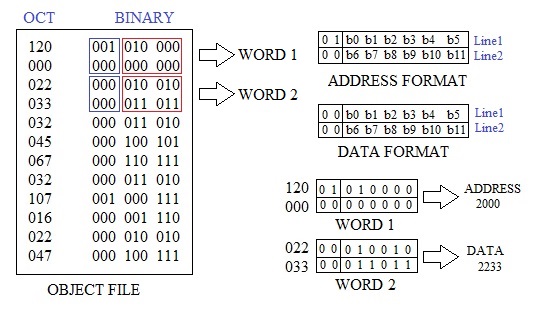
1. **Overview of PDP-8 Architecture:**

The PDP-8 instruction set has a word size of 12bits. It has 4K words of memory. Its main registers are:

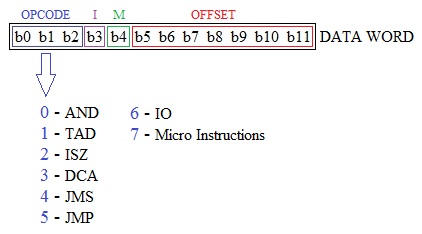
* PC (Program Counter) = a 12 bit register that holds the value of the next instruction.
* AC (Accumulator) = a 12 bit register.
* L (Link) = 1 bit, usually attached to accumulator for carry out of accumulator.

1. **Instruction Format and Decoding Module (Input Object File):**

The PDP-8 object file consists of a column of 3 character octal numbers. Each octal every 6 characters identify whether if the 12 bit address is an instruction or data word. The format of the object file is shown in the picture below. Upon program execution the object file is loaded into memory. If the word found is an address format all following data words will be stored starting at that address. Each subsequent data word will be stored in the next address until a new address is specified or the end of the object file is encountered.



After the file is loaded into memory, the data can be fetched and decoded from memory. All operations depend on the value of the Opcode which is given by the 3 most significant bits (note: for pdp8 bit 0 is the most significant bit). The corresponding instruction for each Opcode is shown in the figure below. While the first six instructions use the I, M, and OFFSET bits, the IO and MICROINSTRUCTIONS break down the bits differently then what is shown in the figure.



1. **Opcode Module:**

There are 8 different Opcodes. The first 6 Opcodes are memory references.

* AND (Opcode = 0) - Logical AND
* TAD (Opcode = 1) – Two’s complement Add
* ISZ (Opcode = 2) – Increment and skip on zero
* DCA (Opcode = 3) – Deposit and clear accumulator
* JMS (Opcode = 4) – Jump to subroutine
* JMP (Opcode = 5) – Jump
* <IO> (Opcode = 6) – Input/Output. This is not implemented in the simulation and will be treated as a NOP (No Operation).
* Micro-Instruction (Opcode = 7)
* Group 1 Microinstruction (Bit 3 = 0)
  + NOP - No operation
  + CLA – Clear accumulator
  + CLL – Clear link
  + CMA – Complement accumulator
  + CML – Compliment link
  + IAC – Increment accumulator
  + RAR – Rotate accumulator and link right
  + RTR - Rotate accumulator and link right twice
  + RAL - Rotate accumulator and link left
  + RTL - Rotate accumulator and link left twice

\*\*All microinstructions in group 1 can be combined except for (RAR, RTR, RAL, and RTL)

* Group 2 Microinstruction (Bit 3 = 1, Bit 11 = 0)
  + SMA – Skip on minus accumulator
  + SZA – Skip on zero accumulator
  + SNL – Skip on non-zero link
  + SPA – Skip on positive accumulator
  + SNA – Skip on nonzero accumulator
  + SZL – Skip on zero link
  + SKP – Skip always
  + CLA – Clear accumulator
  + OSR – Or switch register with accumulator
  + HLT – Halt

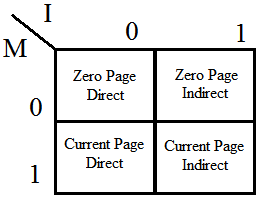
\*\*All microinstructions in group 2 can be combined except for OR and AND subgroups.

* Group 3 Microinstruction (Bit 3 = 1, Bit 11 = 1)
  + This group is not implemented in the simulation and will be treated as a NOP (No Operation).

1. **Addressing Modes Module (Calculating the Effective Address):**

The PDP-8 has 8 instructions, which requires the Opcode to be 3 bits. The PDP-8 word size is 12 bits. This leaves 12 - 3 = 9 bits left to specify the address. However 29 = 512 words left to address, therefore special addressing modes are used. There are 5 addressing modes used to calculate the *Effective Address*.

The 8th bit of the instruction is the ‘I’ bit, and it states whether it is direct (I = 0) or indirect (I = 1) addressing. The 7th bit of the instruction is the ‘M’ bit, and it is used to specify whether it is on the zero page (M = 0) or on the current page (M = 1).



\*Special Case: Zero-page auto-indexing

The zero-page auto-indexing is a special case that uses 8 special addresses in the zero page. The auto-indexing registers will auto-index only when it is indirectly accessed. It is located in register 8-15 decimal or 10-17 octal and will automatically increment the content of the Effective Address by 1 and return the contents of the Effective Address. This can be used to iterate through an array for example.

1. **Memory Module:**

The PDP-8 memory system has 4K of memory, 32 pages, and 128 lines.

* It reads (Loads) a 12-bit word from selected location.
* It writes (Stores) a 12-bit word from selected location.

1. **Summary Module:**

The simulator will keep track of each time an instruction is encountered and will be included in a brief summary which will display at the end of the simulation. The summary will include:

* Total number of instructions executed.
* Total number of clock cycles consumed.

|  |  |
| --- | --- |
| **Mnemonic** | **Clock Cycles** |
| AND | 2 |
| TAD | 2 |
| ISZ | 2 |
| DCA | 2 |
| JMS | 2 |
| JMP | 1 |
| <IO> | 0 |
| uInstruction | 1 |

* Total number of times each instruction type (by mnemonic) was executed.
* Indirect Addressing will have an additional clock cycle.
* Auto-increment will have two additional clock cycles.

1. **Output Trace File Module:**

The simulator will keep track of all memory accesses (instruction and data) in order. At the end of the simulation, it will generate a trace file in the following format:

<type> <address>

Type:

0 - Data Read

1 - Data Write

2 - Instruction Fetch

Address:

Address will be displayed in octal.

# Testing Procedures and Results

The essential part of the design was testing to insure all aspects of each module operated correctly separately, then together as many of the modules interfaced with each other. Below are all the testing procedures and strategies used to validate the simulation along with expected results and actual simulator results.

In the simulator we have 3 different verbose modes that can be turned on/off. This was useful for debugging, and depending on which verbose mode was chosen, the simulator will go more or less in depth with each step the simulator implements.

The different verbose modes are as follows:

* 1 = less detailed
* 2 = more detailed
* 3 = everything

1. **Testing Procedure and Results for “Instruction Decoding from Input File” Module:**

* **Parsing Function (Correct Formatting):**

Test Description:

This test passes values to functions used to parse data from an object file that follows the format specified in are project documentation. There is an individual test for each function. These functions include octalToBinary(), genOpcodeString(), createBinaryString(), findOpcode(), isOctal(), and getOffset().

Test Procedure:

TEST1: two character strings will be passed to octalToBinary(). The first string

Is all octal values and the second string is not all octal values.

TEST2: integers from 0 to 7 will be passed to genOpcodeString(). The

Integer will be converted into a binary string.

TEST3: findOpcode() will be tested by passing a series of 12 bit binary strings

And checking if the found opcode is correct.

TEST4: Numbers will be passed to isOctal() to see if the function gives correct

Results.

TEST5: Similar to findOpcode(), 12 bit binary strings will be passed to getOffset()

too check of the returned offset is correct.

Expected and Simulated Results:

octalString1 = 1234567

output for string1 = 001010011100101110111

octalString2 = 1239765

ERROR - non octal number in string

0 in binary with 3 bits = 000

1 in binary with 3 bits = 001

2 in binary with 3 bits = 010

3 in binary with 3 bits = 011

4 in binary with 3 bits = 100

5 in binary with 3 bits = 101

6 in binary with 3 bits = 110

7 in binary with 3 bits = 111

string 000000000000 OPCODE = 0

string 001000000000 OPCODE = 1

string 010000000000 OPCODE = 2

string 011000000000 OPCODE = 3

string 100000000000 OPCODE = 4

string 101000000000 OPCODE = 5

string 110000000000 OPCODE = 6

string 111000000000 OPCODE = 7

string 000100000000 OPCODE = 0

string 001100000000 OPCODE = 1

string 111111111111 OPCODE = 7

0 is OCTAL

1 is OCTAL

2 is OCTAL

3 is OCTAL

4 is OCTAL

5 is OCTAL

6 is OCTAL

7 is OCTAL

8 NOT OCTAL

9 NOT OCTAL

: NOT OCTAL

; NOT OCTAL

< NOT OCTAL

= NOT OCTAL

> NOT OCTAL

? NOT OCTAL

@ NOT OCTAL

A NOT OCTAL

B NOT OCTAL

C NOT OCTAL

address1 = 000000100000, offset = 32

address2 = 110001000000, offset = 64

address3 = 010110001100, offset = 12

address4 = 010001000010, offset = 66

Press ENTER to end program

* **Parsing Function (Error Checking and Verification):**

Test Description:

This test checks that the data word from memory is being parsed correctly before being passed the IO or microinstruction functions.

Test Procedure:

Hard coded values will be passed to the parser functions. The output will be checked against expected results. A message will be returned if a mismatch is found, else no errors are returned.

Expected and Simulated Results:

\*\*\*BEGINNING MICRO TEST\*\*\*

\*\*\*FINISHED MICRO TEST\*\*\*

\*\*\*BEGINNING IO TEST\*\*\*

\*\*\*FINISHED IO TEST\*\*\*

The output above shows no errors were found

1. **Testing Procedure and Results for “Opcode” Module:**

* **AND - Logical AND**

Test Description:

Testing correct implementation of AND, then verifying the correct value is in the Accumulator.

Test Procedure:

Choose random values, AND them together and see results compared to expected results.

Expected and Simulated Results:

//0 AND 0 = 0

C(EAddr)=0000o AC=0000o

AC Expected: 0000o

AC result: 0000o

//0 AND 1 = 0

C(EAddr)=0000o AC=7777o

AC Expected: 0000o

AC result: 0000o

//1 AND 0 = 0

C(EAddr)=7777o AC=0000o

AC Expected: 0000o

AC result: 0000o

//1 AND 1 = 1

C(EAddr)=7777o AC=7777o

AC Expected: 7777o

AC result: 7777o

//0101 AND 0011 = 0001

C(EAddr)=0707o AC=0077o

AC Expected: 0007o

AC result: 0007o

* **TAD – Two’s complement Add**

Test Description:

Testing correct implementations of Two’s complement ADD and verify correct results in the Accumulator.

Test Procedure:

Choose random values, including negative numbers, and verify the results matches the expected values.

Expected and Simulated Results:

//0 + 0 = 0

C(EAddr)=0000o AC=0000o link=0b

AC Expected: 0000o link: 0b

AC result: 0000o link: 0b

//1 + 0 = 1

C(EAddr)=0001o AC=0000o link=0b

AC Expected: 0001o link: 0b

AC result: 0001o link: 0b

//0 + 1 = 1

C(EAddr)=0000o AC=0001o link=0b

AC Expected: 0001o link: 0b

AC result: 0001o link: 0b

//7777 + 1 = 0 + link

C(EAddr)=7777o AC=0001o link=0b

AC Expected: 0000o link: 1b

AC result: 0000o link: 1b

//1 + 7777 = 0 + link

C(EAddr)=0001o AC=7777o link=0b

AC Expected: 0000o link: 1b

AC result: 0000o link: 1b

//1 + 7777 + link = 0

C(EAddr)=0001o AC=7777o link=1b

AC Expected: 0000o link: 0b

AC result: 0000o link: 0b

//24+ 47 = 71

C(EAddr)=0030o AC=0057o link=0b

AC Expected: 0107o link: 0b

AC result: 0107o link: 0b

//-24 + 47 = 23 + link

C(EAddr)=7750o AC=0057o link=0b

AC Expected: 0027o link: 1b

AC result: 0027o link: 1b

//24 + -47 = -23

C(EAddr)=0030o AC=7721o link=0b

AC Expected: 7751o link: 0b

AC result: 7751o link: 0b

//-24 + -47 = -71 + link

C(EAddr)=7750o AC=7721o link=0b

AC Expected: 7671o link: 1b

AC result: 7671o link: 1b

//1977 + 1985 = -134

C(EAddr)=3671o AC=3701o link=0b

AC Expected: 7572o link: 0b

AC result: 7572o link: 0b

//-1977 + -1985 = 134 + link

C(EAddr)=4107o AC=4077o link=0b

AC Expected: 206o link: 1b

AC result: 206o link: 1b

* **ISZ – Increment and skip on zero**

Test Description:

Testing correct implementation of ISZ, and then verifying that it skips if the result is zero.

Test Procedure:

Choose random values and see results compared to expected results.

Expected and Simulated Results:

//0 + 1 = 1

C(EAddr)=0000o

C(EAddr) Expected: 0001o

C(EAddr) result: 0001o

//1 + 1 = 2

C(EAddr)=0001o

C(EAddr) Expected: 0002o

C(EAddr) result: 0002o

//7776 + 1 = 7777

C(EAddr)=7776o

C(EAddr) Expected: 7777o

C(EAddr) result: 7777o

//7777 + 1 = 0

C(EAddr)=7777o

C(EAddr) Expected: 0000o

C(EAddr) result: 0000o

* **DCA – Deposit and clear accumulator**

Test Description:

Test correct implementation of DCA, and then verifying that AC is cleared.

Test Procedure:

Choose random values and see results compared to expected results.

Expected and Simulated Results:

pc=203, ac=5, link=0

[DCA |0|1|0052]

Expected Results: AC = 0

pc=204, ac=0, link=0

pc=207, ac=1, link=0

[DCA |0|1|0050]

Expected Results: AC = 0

pc=210, ac=0, link=0

* **JMS – Jump to subroutine**

Test Description:

Test correct implementation of JMP.

Test Procedure:

Choose random values and see results compared to expected results.

Expected and Simulated Results:

pc=231, ac=3, link=0

[JMS |0|1|0040]

pc=241, ac=3, link=0

pc=232, ac=0, link=0

[JMS |0|1|0034]

pc=235, ac=0, link=0

* **JMP – Jump**

Test Description:

Test correct implementation of JMP.

Test Procedure:

Choose random values and see results compared to expected results.

Expected and Simulated Results:

pc=233, ac=0, link=0

[JMP |0|1|0044]

pc=244, ac=0, link=0

pc=243, ac=0, link=0

[JMP |1|1|0040]

pc=232, ac=0, link=0

pc=237, ac=0, link=0

[JMP |1|1|0034]

pc=233, ac=0, link=0

pc=233, ac=0, link=0

[JMP |0|1|0044]

pc=244, ac=0, link=0

* **Micro-Instruction**
* Group 1 Microinstruction
  + **CLA**:

Test Description:

Test correct implementation of CLA, and verify that AC is cleared.

Test Procedure:

Choose random values and see results compared to expected results.

Expected and Simulated Results:

pc=255, ac=1, link=1

[MICRO| 200]

Expected Results: AC = 0

pc=256, ac=0, link=1

pc=271, ac=4, link=0

[MICRO| 200]

Expected Results: AC = 0

pc=272, ac=0, link=0

* + **CLL**:

Test Description:

Test correct implementation of CLL, and verify that link is cleared.

Test Procedure:

Choose random values and see results compared to expected results.

Expected and Simulated Results:

pc=252, ac=0, link=0

[MICRO| 020]

Expected Results: Link = 1

pc=253, ac=0, link=1

* + **CMA**:

Test Description:

Test correct implementation of CMA, and verify that accumulator is complemented.

Test Procedure:

Choose random values and see results compared to expected results.

Expected and Simulated Results:

pc=250, ac=7776, link=1

[MICRO| 040]

Expected Results: AC = 1

pc=251, ac=1, link=1

pc=246, ac=0, link=0

[MICRO| 040]

Expected Results: AC = 7777

pc=247, ac=7777, link=0

* + **CML**:

Test Description:

Test correct implementation of CML, and verify that link is complemented.

Test Procedure:

Choose random values and see results compared to expected results.

Expected and Simulated Results:

pc=252, ac=0, link=0

[MICRO| 020]

Expected Results: L = 1

pc=253, ac=0, link=1

pc=253, ac=0, link=1

[MICRO| 020]

Expected Results: L = 0

pc=254, ac=0, link=0

* + **IAC**:

Test Description:

Test correct implementation of IAC, and verify that AC = AC + 1.

Test Procedure:

Choose random values and see results compared to expected results.

Expected and Simulated Results:

pc=254, ac=0, link=1

[MICRO| 001]

Expected Results: AC = 1, Link = 0

pc=255, ac=1, link=1

pc=216, ac=7777, link=0

[MICRO| 001]

Expected Results: AC = 1, Link = 1

pc=217, ac=0, link=1

* + **RAR:**

Test Description:

Test correct implementation of RAR, and verify accumulator shifted right once and link bit displays correct value.

Test Procedure:

Choose random values and verify correct results.

Expected and Simulated Results:

pc=265, ac=3, link=0

[MICRO| 010]

pc=266, ac=1, link=1

pc=277, ac=6, link=0

[MICRO| 010]

pc=300, ac=3, link=0

* + **RAL:**

Test Description:

Test correct implementation of RAL, and verify accumulator shifted left once and link bit displays correct value.

Test Procedure:

Choose random values and verify correct results.

Expected and Simulated Results:

pc=266, ac=1, link=1

[MICRO| 004]

pc=267, ac=3, link=0

pc=274, ac=3, link=0

[MICRO| 004]

pc=275, ac=6, link=0

* + **RTR:**

Test Description:

Test correct implementation of RTR, and verify accumulator shifted right twice and link bit displays correct value.

Test Procedure:

Choose random values and verify correct results.

Expected and Simulated Results:

pc=267, ac=3, link=0

[MICRO| 012]

pc=270, ac=4000, link=1

pc=275, ac=6, link=0

[MICRO| 012]

pc=276, ac=1, link=1

* + **RTL:**

Test Description:

Test correct implementation of RTL, and verify accumulator shifted left twice and link bit displays correct value.

Test Procedure:

Choose random values and verify correct results.

Expected and Simulated Results:

pc=270, ac=4000, link=1

[MICRO| 006]

pc=271, ac=3, link=0

pc=276, ac=1, link=1

[MICRO| 006]

pc=277, ac=6, link=0

* Group 2 Microinstruction
  + **SMA**:

Test Description:

Test correct implementation of Skip on Minus Accumulator.

Test Procedure:

Choose values and see if results compared to expected results.

Expected and Simulated Results:

//is 0 negative?

Instuction=7500o AC=0000o

Expected: [nothing]

Result:

//is -1 negative?

Instuction=7500o AC=7777o

Expected: SKIP!

Result: SKIP!

//is the lowest number negative?

Instuction=7500o AC=4000o

Expected: SKIP!

Result: SKIP!

* + **SZA**:

Test Description:

Test correct implementation of Skip on Zero Accumulator.

Test Procedure:

Choose values and see if results compared to expected results.

Expected and Simulated Results:

//is 0 = 0?

Instuction=7440o AC=0000o

Expected: SKIP!

Result: SKIP!

//is 7777 = 0?

Instuction=7440o AC=7777o

Expected: [nothing]

Result:

//is 1 = 0?

Instuction=7440o AC=0001o

Expected: [nothing]

Result:

* + **SNL**:

Test Description:

Test correct implementation of Skip if Link is not zero.

Test Procedure:

Choose values and see if results compared to expected results.

Expected and Simulated Results:

//is 1 = 1?

Instuction=7420o link=1b

Expected: SKIP!

Result: SKIP!

//is 0 = 1?

Instuction=7420o link=0b

Expected: [nothing]

Result:

* + **SPA**:

Test Description:

Test correct implementation of Skip on Positive Accumulator.

Test Procedure:

Choose values and see if results compared to expected results.

Expected and Simulated Results:

//is 1 positive?

Instuction=7510o AC=0001o

Expected: SKIP!

Result: SKIP!

//is lowest number positive?

Instuction=7510o AC=7777o

Expected: [nothing]

Result:

//is -1 positive?

Instuction=7510o AC=4000o

Expected: [nothing]

Result:

* + **SNA**:

Test Description:

Test correct implementation of Skip on Nonzero Accumulator.

Test Procedure:

Choose values and see if results compared to expected results.

Expected and Simulated Results:

//is 0 not 0 ?

Instuction=7450o AC=0000o

Expected: [nothing]

Result:

//is 1 not 0?

Instuction=7450o AC=0001o

Expected: SKIP!

Result: SKIP!

//is 7777 not 0?

Instuction=7450o AC=7777o

Expected: SKIP!

Result: SKIP!

* + **SZL – Skip on zero link**

Test Description:

Test correct implementation of Skip if link is zero.

Test Procedure:

Choose values and see if results compared to expected results.

Expected and Simulated Results:

//is 1 = 0?

Instuction=7430o link=1b

Expected: [nothing]

Result:

//is 0 = 0?

Instuction=7430o link=0b

Expected: SKIP!

Result: SKIP!

* + **SKP**:

Test Description:

Test correct implementation of Skip.

Test Procedure:

Choose a value that will implement the SKIP and verify that it works.

Expected and Simulated Results:

//skip skips.

Instuction=7410o

Expected: SKIP!

Result: SKIP!

* + **HLT – Halt**

Test Description:

Test correct implementation of Halt.

Test Procedure:

Run several different programs and verify that execution ends.

Expected and Simulated Results:

For each program run, the execution stopped correctly and outputs the summary and trace file, which verifies the Halt function.

* + **Testing OR subgroups:**

Test Description:

Testing the correct implementation of the OR subgroups.

Test Procedure:

Choose values and see if tested results compare to expected results.

Expected and Simulated Results:

//is 1 = 1 and 0 = 0?

Instuction=7460o link=1b AC=0000o

Expected: SKIP!

Result: SKIP!

//is 0 = 1 and 1 = 0?

Instuction=7460o link=0b AC=0001o

Expected: [nothing]

Result:

1. **Testing Procedure and Results for “Addressing Modes” Module:**

* TEST 1: Zero Page Direct Addressing:

Test Description:

Testing correct implementation of Zero Page Direct Addressing and verifying that when adding values of specific offsets, it gets calculated correctly in the Effective Address.

Test Procedure:

Test with offset = 1111111b

Test with offset = 0000000b

Expected and Simulated Results:

//testing 000001111111b

Bit3=0 Bit4=0 offset=0177o

EAddress expected: 0177o

EAddress result: 0177o

//testing 000000000000

Bit3=0 Bit4=0 offset=0000o

EAddress expected: 0000o

EAddress result: 0000o

* TEST 2: Current Page Direct Addressing:

Test Description:

Testing correct implementation of Current Page Direct Addressing and verifying that when adding values of specific offsets, it gets calculated correctly in the Effective Address.

Test Procedure:

Test with different offsets and add to PC.

Format of testing is //testing PC and Offset

Expected and Simulated Results:

//testing 111110000000 and 000001111111

Bit3=0 Bit4=1 offset=0177o oldpc=7600o

EAddress expected: 7777o

EAddress result: 7777o

//testing 111100000000 and 000001111111

Bit3=0 Bit4=1 offset=0177o oldpc=7400o

EAddress expected: 7577o

EAddress result: 7577o

//testing 111110000000 and 000000000000

Bit3=0 Bit4=1 offset=0000o oldpc=7600o

EAddress expected: 3968o

EAddress result: 3968o

//testing 111100000000 and 000000000000

Bit3=0 Bit4=1 offset=0000o oldpc=7400o

EAddress expected: 3840o

EAddress result: 3840o

//testing 000010000000 and 000000000000

Bit3=0 Bit4=1 offset=0000o oldpc=0200o

EAddress expected: 0200o

EAddress result: 0200o

//testing 000000000000 and 000000000000

Bit3=0 Bit4=1 offset=0000o oldpc=0000o

EAddress expected: 0000o

EAddress result: 0000o

//testing 000010000000 and 000001111111

Bit3=0 Bit4=1 offset=0177o oldpc=0200o

EAddress expected: 0377o

EAddress result: 0377o

//testing 000000000000 and 000001111111

Bit3=0 Bit4=1 offset=0177o oldpc=0000o

EAddress expected: 0177o

EAddress result: 0177o

* TEST 3: Testing Zero Page Indirect Addressing:

Test Description:

Testing correct implementation of Zero Page Indirect Addressing and verifying that when adding values of specific offsets, it gets calculated correctly in the Effective Address.

Test Procedure:

Test with offset = 1111111b

Test with offset = 0000000b

Expected and Simulated Results:

//testing 000001111111

Bit3=1 Bit4=0 offset=0177o

EAddress expected: C(0177o)

EAddress result: C(0177o)

//testing 000000000000

Bit3=1 Bit4=0 offset=0000o

EAddress expected: C(0000o)

EAddress result: C(0000o)

* TEST 4: Testing Current Page Indirect Addressing:

Test Description:

Testing correct implementation of Current Page Indirect Addressing and verifying that when adding values of specific offsets, it gets calculated correctly in the Effective Address.

Test Procedure:

Test with different offsets and add to PC.

Format of testing is //testing PC and Offset

Expected and Simulated Results:

/testing 111110000000 and 000000000000

Bit3=1 Bit4=1 offset=0000o oldpc=7600o

EAddress expected: C(3968o)

EAddress result: C(3968o)

//testing 111100000000 and 000000000000

Bit3=1 Bit4=1 offset=0000o oldpc=7400o

EAddress expected: C(3840o)

EAddress result: C(3840o)

//testing 000010000000 and 000000000000

Bit3=1 Bit4=1 offset=0000o oldpc=0200o

EAddress expected: C(0200o)

EAddress result: C(0200o)

//testing 000000000000 and 000000000000

Bit3=1 Bit4=1 offset=0000o oldpc=0000o

EAddress expected: C(0000o)

EAddress result: C(0000o)

//testing 000010000000 and 000001111111

Bit3=1 Bit4=1 offset=0177o oldpc=0200o

EAddress expected: C(0377o)

EAddress result: C(0377o)

//testing 000000000000 and 000001111111

Bit3=1 Bit4=1 offset=0177o oldpc=0000o

EAddress expected: C(0177o)

EAddress result: C(0177o)

* TEST 5: Testing Zero Page Auto-Indexing Addressing:

Test Description:

Testing correct implementation of Zero Page Auto-Indexing Addressing and verifying that when adding values of specific offsets, it gets calculated correctly in the Effective Address.

Test Procedure:

Test with different offsets and verify that +1 is added to the Effective Address.

Expected and Simulated Results:

//the register before autoindexing

Bit3=1 Bit4=0 offset=0007o

EAddress expected: 0007o

EAddress result: 0007o

//the first autoindexing register

Bit3=1 Bit4=0 offset=0010o

EAddress expected: C(0010o)+1

EAddress result: C(0010o)+1

//the last autoindexing register

Bit3=1 Bit4=0 offset=0017o

EAddress expected: C(0017o)+1

EAddress result: C(0017o)+1

//the register after autoindexing

Bit3=1 Bit4=0 offset=0020o

EAddress expected: 0020o

EAddress result: 0020o

1. **Testing Procedure and Results for “Memory” Module:**

* TEST 1: Store Values (Assume address does not exceed 12 bits)

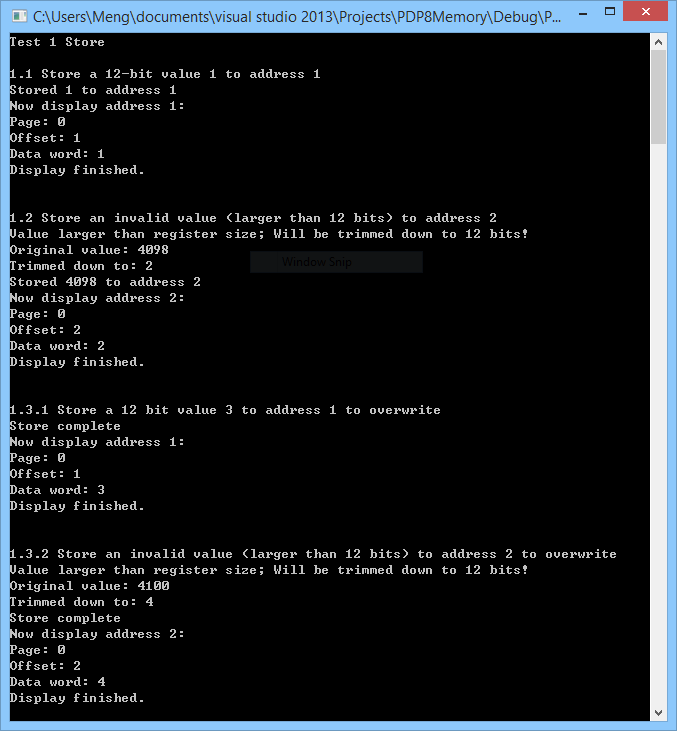
Test Description:

Use store and display function to test (1) if correct values are stored into the correct addresses (2) invalid values are trimmed down then stored to correct addresses (3) overwrite functionality without prompting user

Test Procedure:

1.1 Store a valid value (12 bits or less) to an empty memory entry  
1.2 Store an invalid value (larger than 12 bits) to an empty memory entry  
1.3 Store a valid and an invalid value to an occupied entry to overwrite

Expected and Simulated Results:

1.1 Page 0 Offset 1 Value 1  
1.2 Page 0 Offset 2 Value 2 (4098 trimmed down)  
1.3.1 Page 0 Offset 1 Value 3 (Overwrite)  
1.3.2 Page 0 Offset 2 Value 4 (Overwrite, 4100 trimmed down)  


* TEST 2: Load Values (Assume address does not exceed 12 bits)

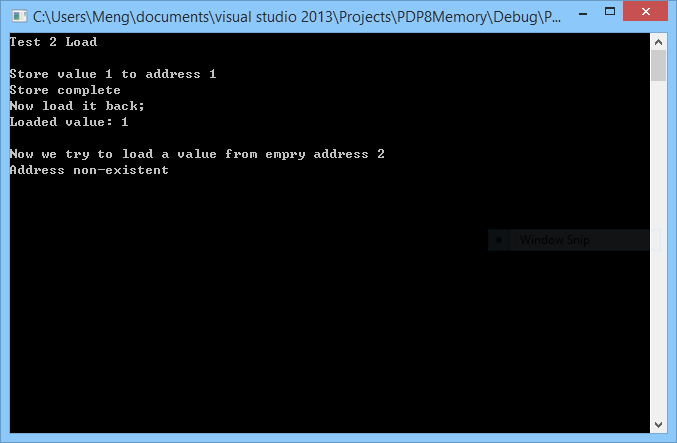
Test Description:

Given “store” function working, we now test if “load” works. We want to see if (1) if it loads correct value from correct address (2) if it complains when the address is empty

Test Procedure:

2.1 Store value 1 to address 1 and then load it back  
2.2 Load value from empty address 2

Expected and Simulated Results:

2.1 Loaded value: 1  
2.2 Address non-existent  


* TEST 3: Page table integrity

Test Description:

Given “store” and “load” working, we tested the full page table integrity; all 4,096 entries are tested so all boundary tests are done in one part.

Test Procedure:

We store some values to the page table at first (in decimal):

Address 0 (Page 0 Offset 0) Value 0  
Address 1 (Page 0 Offset 1) Value 1  
Address 2 (Page 0 Offset 2) Value 2  
Address 3 (Page 0 Offset 3) Value 3  
Address 5 (Page 0 Offset 5) Value 4  
Address 6 (Page 0 Offset 6) Value 5  
Address 7 (Page 0 Offset 7) Value 6  
Address 129 (Page 1 Offset 1) Value 7  
Address 130 (Page 1 Offset 2) Value 8  
Address 258 (Page 2 Offset 2) Value 9  
Address 259 (Page 2 Offset 3) Value 10  
Address 261 (Page 2 Offset 5) Value 11

Then use “display\_all” function, which will later be integrated with trace file to show the entire page table content for testing and debugging purposes.

Expected and Simulated Results:

“display\_all” function will display all values (page number, offset, value) in octal. So the expected result should be

Page 0000

Offset 0000

Data 0000  
Data 0001  
Data 0002  
Data 0003

Offset 0005

Data 0004  
Data 0005  
Data 0006

Page 0001

Offset 0001

Data 0007  
Data 0010

Page 0002

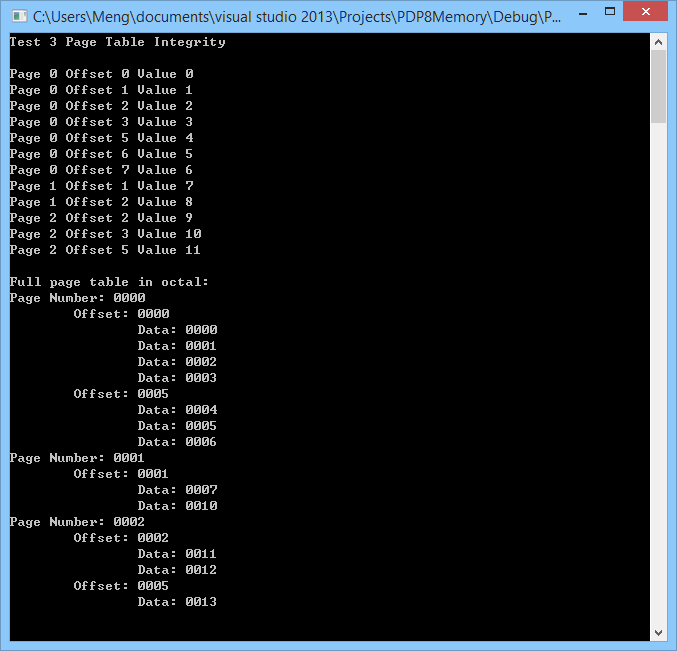
Offset 0002

Data 0011  
Data 0012

Offset 0005

Data 0013

Simulated result is shown as:



1. **Testing Procedure and Results for “Summary” Module:**

Test Description:

Test correct implementation of Summary Output module.

Test Procedure:

Insert an input object file, and verify the correct output summary.

Expected and Simulated Results:

Input Object File:

102

000

012

000

012

001

Output Summary Module:

-----------PDP-8 ISA Simulation Summary---------------

Total number of Instructions executed: 2

Total number of clock cycles consumed: 4

\*\*Number of times each instruction type was executed\*\*

|-----------------------------------------------------

| Mnemonic | Number of times executed

|-----------------------------------------------------

| AND | 0

| TAD | 2

| ISZ | 0

| DCA | 0

| JMS | 0

| JMP | 0

| <IO> | 0

| uInstructions | 0

------------------------------------------------------

1. **Testing Procedure and Results for “Output Trace File” Module:**

Test Description:

Test correct implementation of Output Trace File module.

Test Procedure:

Insert an input object file, and verify the correct output trace file.

Expected and Simulated Results:

Input Object File:

102

000

012

000

012

001

Output Trace File:

2 200

0 200

2 201

0 201

# PDP-8 ISA Simulation Source Code

Please see attached for all source codes for the PDP-8 ISA (attached respectively).

1. makefile

2. constants.h

3. main.cpp

4. memory.h

5. memory.cpp

6. parser.h

7. parser.cpp

8. bits.h

9. bits.cpp