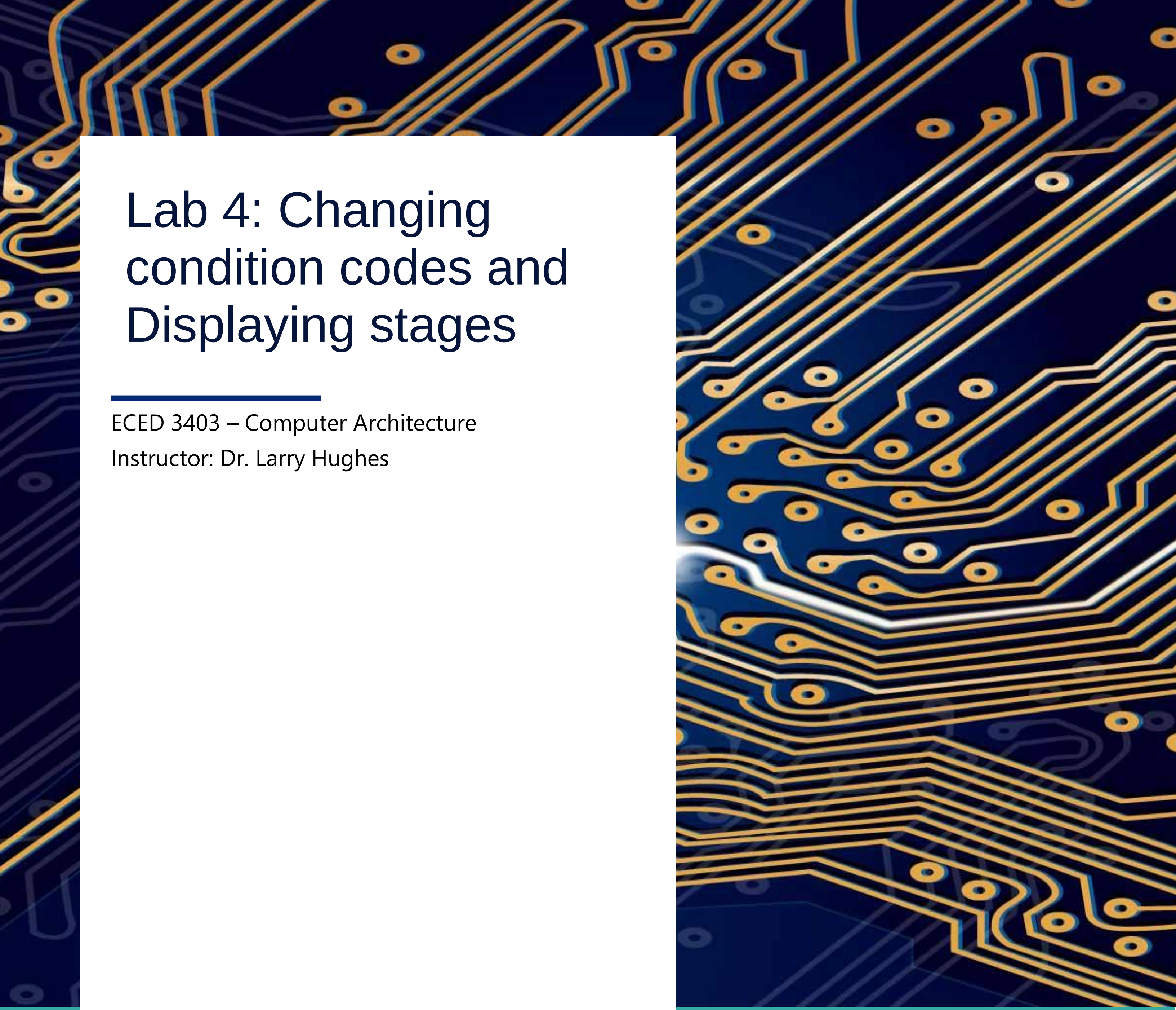


Lab 4: Changing condition codes and Displaying stages

ECED 3403 – Computer Architecture
Instructor: Dr. Larry Hughes

JULY 24

Name: MD Iftekhar Hossain Rafi
Student ID: B00871031

Table of Contents

Objective

Implement SETCC and CLRCC Instructions:

- Design, implement, and test the SETCC (set condition codes) and CLRCC (clear condition codes) instructions in the emulator.
- Ensure accurate manipulation of condition code bits (V, N, Z, C) in the Processor Status Word (PSW).

Enhance Debugger with Stage Display:

- Extend the debugger to display the clock cycle and the state of the Fetch, Decode, and Execute stages during program execution in debug mode.
- Provide detailed visibility into the execution flow of instructions for easier debugging and testing.

Design

The design section for this report will improve upon the existing design and implementation of the XM23p emulator from assignment 2. As requested, major structures for decoding and storing will be shown as flowcharts and a data dictionary. A short description of all files are also provided.

Header Files:

1. debugger_mode.h
2. decode_instructions.h
3. execute_instructions.h
4. instructions.h
5. loader.h
6. pipeline.h

Source Files:

1. debugger_mode.c
2. decode_instructions.c
3. execute_instructions.c
4. instructions.c
5. loader.c
6. manager.c
7. memory.c
8. pipeline.c
9. srecord.c

loader.c

Purpose: Handles the loading of instructions and data into memory from external files. This file typically includes functions to read files, parse their contents, and store the parsed instructions and data in the appropriate memory locations.

manager.c

Purpose: Manages the overall control and execution flow of the program. This file coordinates the various components of the emulator, such as loading instructions, executing them, and handling input/output operations. It often acts as the main control loop for the program.

runmode.c

Purpose: Implements the different modes of execution for the emulator, such as running in normal mode, debugging mode, or stepping through instructions one by one. This file includes functions to initialize and manage the program counter (PC) and handle breakpoints.

pipeline.c

Purpose: Manages the instruction pipeline stages of the emulator, such as fetching, decoding, and executing instructions. This file includes functions to handle the different stages of the pipeline and ensure instructions flow correctly through them.

decode_instructions.c

Purpose: Handles the decoding of instructions fetched from memory. This file includes functions to interpret the binary representation of instructions and extract relevant fields, such as opcode, source and destination registers, and immediate values.

execute_instructions.c

Purpose: Contains the implementation of the specific instructions that the emulator supports. This file includes functions to execute each type of instruction, such as arithmetic operations, logical operations, and data movement instructions.

instructions.c

Purpose: Implements the detailed execution logic for each supported instruction, updating the program status word (PSW) and register values as needed. This file contains the core functions for performing arithmetic operations, logical operations, and other instructions.

debugger_mode.c

Purpose: Implements the debugging functionality for the emulator, allowing users to step through instructions, set breakpoints, and inspect register and memory values. This file includes functions to manage the debugging state and handle user inputs for debugging commands.

memory.c

Purpose: Manages the memory subsystem of the emulator, including allocation, initialization, reading, and writing memory. This file includes functions to handle memory operations and ensure the memory state is correctly maintained during program execution.

srecord.c

Purpose: Handles the loading and parsing of S-record formatted files, which are a common format for representing binary data. This file includes functions to read S-record files, extract the data, and load it into the appropriate memory locations.

pipeline.h

Purpose: Provides function declarations and macro definitions related to the instruction pipeline stages. This header file defines the interface for pipeline-related functions used in the emulator.

decode_instructions.h

Purpose: Provides function declarations and macro definitions related to instruction decoding. This header file defines the interface for functions that interpret and extract fields from binary instructions.

execute_instructions.h

Purpose: Provides function declarations and macro definitions related to instruction execution. This header file defines the interface for functions that perform the actual execution of instructions.

instructions.h

Purpose: Provides function declarations and macro definitions related to updating the program status word (PSW) and executing specific instructions. This header file defines the interface for functions that handle arithmetic, logical, and other operations.

loader.h

Purpose: Provides function declarations and macro definitions related to loading instructions and data into memory. This header file defines the interface for functions that read and parse input files and store their contents in memory.

manager.h

Purpose: Provides function declarations and macro definitions related to managing the overall control and execution flow of the program. This header file defines the interface for functions that coordinate the various components of the emulator.

memory.h

Purpose: Provides function declarations and macro definitions related to memory management. This header file defines the interface for functions that handle memory operations such as allocation, initialization, reading, and writing.

debugger_mode.h

Purpose: Provides function declarations and macro definitions related to debugging functionality. This header file defines the interface for functions that manage the debugging state and handle user inputs for debugging commands.

Flowchart

Flowcharts have been created for all source code files except for instructions.c- which is shown through pseudocode.

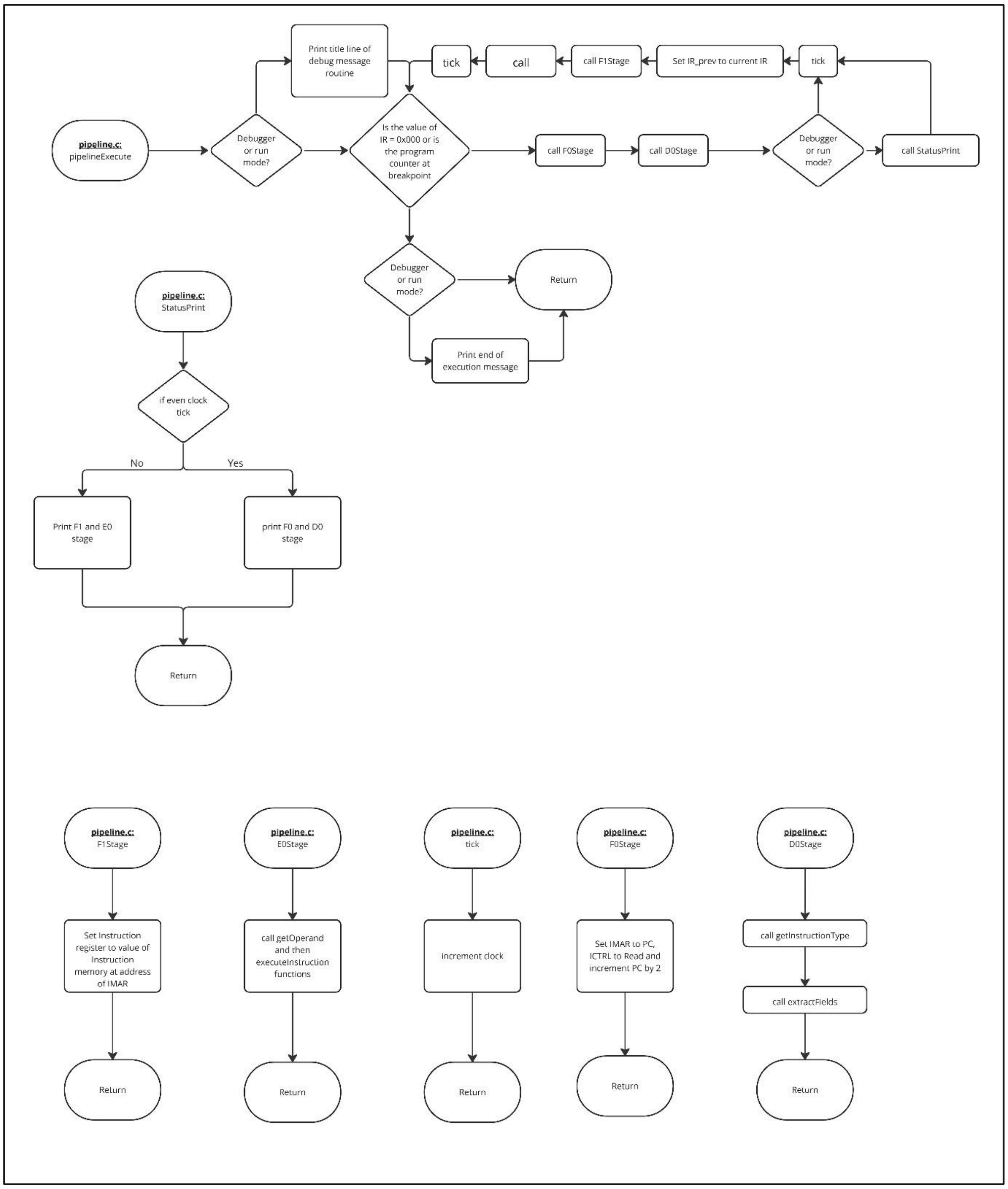


Figure 1 Flowchart for pipeline.c

Pseudocode for `executeSETCC`

```
Function executeSETCC(v, c, slp, n, z):
    If c is true:
        Set Carry Flag (CF) to 1
    If v is true:
        Set Overflow Flag (OF) to 1
    If n is true:
        Set Sign Flag (SF) to 1
    If z is true:
        Set Zero Flag (ZF) to 1
    If slp is true:
        Set Sleep Flag (slp) to 1
End Function
```

Pseudocode for `executeCLRCC`

```
Function executeCLRCC(v, c, slp, n, z):
    If c is true:
        Clear Carry Flag (CF) to 0
    If v is true:
        Clear Overflow Flag (OF) to 0
    If n is true:
        Clear Sign Flag (SF) to 0
    If z is true:
        Clear Zero Flag (ZF) to 0
    If slp is true:
        Clear Sleep Flag (slp) to 0
End Function
```

Implementation

The implementation section will show the updated code files to achieve the requirements for lab 4.

Full code for pipeline.c shown here:

```
/*
Name: Iftekhar Rafi
ID: B00871031
Course: ECED 3403 Computer Architecture
Instructor: Larry Hughes

File Name: pipeline.c
File Purpose: This file contains functions to fetch, decode, and execute instructions
from IMEM.
*/

#include <stdio.h>
#include "loader.h"
#include "pipeline.h"
#include "execute_instructions.h"

void pipelineExecute(unsigned short* PC, int display) {
    InstructionType type;
    int instruction_count = 1;
    unsigned short IR_prev = 0;
    unsigned char rc, wb, src, dst, con, bb, v = psw.OF, c = psw.CF, slp = psw.slp, z =
psw.ZF, n = psw.SF;
    IR = 0x6800;


    printf("Clock  PC   Instruction      Fetch      Decode      Execute    Z N V C\n");
    printf("---------------------------------------------------------------------------
\n");

    while (!(IR == 0x0000 || *PC == breakpoint)) {

        F0Stage(PC);
        D0Stage(&type, &rc, &wb, &src, &dst, &con, &bb, display, PC, &v, &c, &slp, &n,
&z);

        if (display) StatusPrint(PC, IR_prev);

        tick();

        IR_prev = IR;

        F1Stage();
        E0Stage(type, rc, wb, src, dst, con, bb, v, c, slp, n, z);


        if (display) StatusPrint(PC, IR_prev);

        tick();
    }
    if (display) {
        if (*PC == breakpoint) {
            printf("Breakpoint reached at %04X. Execution stopped.\n\n", *PC);
        }
        else {
            printf("End of program reached. Execution stopped.\n\n");
        }
    }
    return;
}

void StatusPrint(unsigned short* PC, unsigned short IR_prev) {
      if(clock_ticks % 2 == 0) printf("  %-3d %-9X %-10X F0: %-7X D0: %-5X \n",
clock_ticks, IMAR, IMEM[IMAR / 2], IMAR, IR);
    else printf("  %-24d F1: %-19X E0: %-7X %d %d %d %d\n", clock_ticks, IR, IR_prev,
psw.ZF, psw.SF, psw.OF, psw.CF);
}

void F0Stage(unsigned short* PC) {
    IMAR = *PC;
    ICTRL = READ;
    *PC += 2;
}

void D0Stage(InstructionType* type, unsigned char* rc, unsigned char* wb, unsigned char*
src, unsigned char* dst, unsigned char* con, unsigned char* bb, int display, unsigned
short* PC, unsigned char* v, unsigned char* c, unsigned char* slp, unsigned char* n,
unsigned char* z) {
    *type = getInstructionType(IR);
    extractFields(IR, *type, rc, wb, src, dst, con, bb, v, c, slp, n, z);
}

void F1Stage() {
    IR = IMEM[IMAR / 2];
}

void E0Stage(InstructionType type, unsigned char rc, unsigned char wb, unsigned char src,
unsigned char dst, unsigned char con, unsigned char bb, unsigned char* v, unsigned char*
c, unsigned char* slp, unsigned char* n, unsigned char* z) {
    if (type != INVALID) {
        unsigned short operand = getOperand(rc, src);
        executeInstruction(type, operand, rc, wb, src, dst, con, bb, v, c, slp, n, z);
    }
}

void tick() {
    clock_ticks++;
}
```

Updates to instructions.c shown here:

```c
/*
Name: Iftekhar Rafi
ID: B00871031
Course: ECED 3403 Computer Architecture
Instructor: Larry Hughes

File Name: instructions.c
File Purpose: This file contains the functions to handle execution of specific
instructions for the XM23P emulator.
*/


void executeSETCC(unsigned char v, unsigned char c, unsigned char slp, unsigned char n,
unsigned char z) {
   if (c) psw.CF = 1;
   if (v) psw.OF = 1;
   if (n) psw.SF = 1;
   if (z) psw.ZF = 1;
   if (slp) psw.slp = 1;
}
void executeCLRCC(unsigned char v, unsigned char c, unsigned char slp, unsigned char n,
unsigned char z) {
   if (c) psw.CF = 0;
   if (v) psw.OF = 0;
   if (n) psw.SF = 0;
   if (z) psw.ZF = 0;
   if (slp) psw.slp = 0;
}
```

Full execute_instructions.c shown here:

```c
/*
Name: Iftekhar Rafi
ID: B00871031
Course: ECED 3403 Computer Architecture
Instructor: Larry Hughes

File Name: execute_instructions.c
File Purpose: This file contains the function to execute instructions.
*/

#include "loader.h"
#include "execute_instructions.h"
#include "instructions.h"

void executeInstruction(InstructionType type, unsigned short operand, unsigned char rc,
unsigned char wb, unsigned char src, unsigned char dst, unsigned char con, unsigned char
bb, unsigned char* v, unsigned char* c, unsigned char* slp, unsigned char* n, unsigned
char* z) {

    operand = getOperand(rc, src);

    // Execute the instruction based on the decoded type
    switch (type) {
    case ADD:
        executeADD(dst, operand);
        return;
    case ADDC:
        executeADDC(dst, operand);
        return;
    case SUB:
        executeSUB(dst, operand);
        return;
    case SUBC:
        executeSUBC(dst, operand);
        return;
    case DADD:
        executeDADD(dst, operand);
        return;
    case CMP:
        executeCMP(dst, operand);
        return;
    case XOR:
        executeXOR(dst, operand);
        return;
    case AND:
        executeAND(dst, operand);
        return;
    case OR:
        executeOR(dst, operand);
        return;
    case BIT:
        executeBIT(dst, operand);
        return;
    case BIC:
        executeBIC(dst, operand);
        return;
    case BIS:
        executeBIS(dst, operand);
        return;
    case MOV:
        executeMOV(dst, operand);
        return;
    case SWAP:
        executeSWAP(src, dst);
        return;
    case SRA:
        executeSRA(dst);
        return;
    case RRC:
        executeRRC(dst);
        return;
    case SWPB:
        executeSWPB(dst);
        return;
    case SXT:
        executeSXT(dst);
        return;
    case MOVLZ:
        executeMOVLZ(dst, bb);
        return;
    case MOVL:
        executeMOVL(dst, bb);
        return;
    case MOVLS:
        executeMOVLS(dst, bb);
        return;
    case MOVH:
        executeMOVH(dst, bb);
        return;
    case SETCC:
        executeSETCC(v, c, slp, n, z);
            return;
    case CLRCC:
            executeCLRCC(v, c, slp, n, z);
          return;
    default:
        // Handle invalid instruction or not implemented
        return;
    }
}
```

Testing

Test 1: Test SETCC and CLRCC

```
; Program to test SETCC and CLRCC instructions

          org     #1000
START
       CLRCC   VNZC
       SETCC   C              ; Set Carry bit
       CLRCC   C              ; Clear Carry bit
       SETCC   V              ; Set Overflow bit
       CLRCC   V              ; Clear Overflow bit
       SETCC   N              ; Set Negative bit
       CLRCC   N              ; Clear Negative bit
       SETCC   Z              ; Set Zero bit
       CLRCC   Z              ; Clear Zero bit
       SETCC   ZNVC              ; Set all bits
       CLRCC   ZNVC              ; Clear all bits
       HALT                 ; Halt the program

       END     START
```

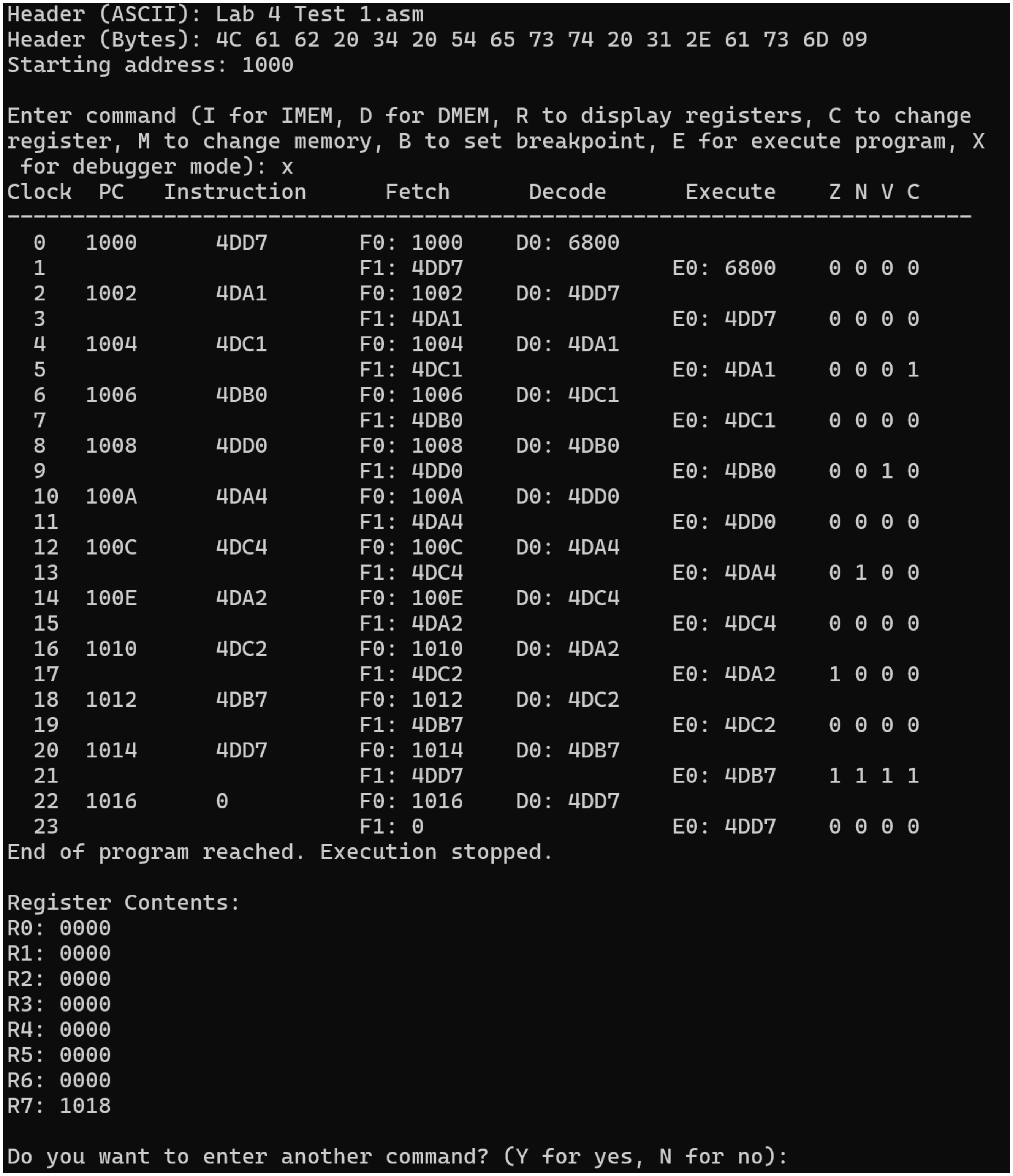
Expected Output

1. **Instruction: CLRCC VNZC**
   - Operation: Clear Overflow, Negative, Zero, and Carry flags.
   - Expected PSW State: `V = 0, N = 0, Z = 0, C = 0`
2. **Instruction: SETCC C**
   - Operation: Set Carry flag.
   - Expected PSW State: `V = 0, N = 0, Z = 0, C = 1`
3. **Instruction: CLRCC C**
   - Operation: Clear Carry flag.
   - Expected PSW State: `V = 0, N = 0, Z = 0, C = 0`
4. **Instruction: SETCC V**
   - Operation: Set Overflow flag.
   - Expected PSW State: `V = 1, N = 0, Z = 0, C = 0`
5. **Instruction: CLRCC V**
   - Operation: Clear Overflow flag.
   - Expected PSW State: `V = 0, N = 0, Z = 0, C = 0`
6. **Instruction: SETCC N**
   - Operation: Set Negative flag.
   - Expected PSW State: `V = 0, N = 1, Z = 0, C = 0`
7. **Instruction: CLRCC N**
   - Operation: Clear Negative flag.
   - Expected PSW State: `V = 0, N = 0, Z = 0, C = 0`
8. **Instruction: SETCC Z**
   - Operation: Set Zero flag.
   - Expected PSW State: `V = 0, N = 0, Z = 1, C = 0`
9. **Instruction: CLRCC Z**
   - Operation: Clear Zero flag.
   - Expected PSW State: `V = 0, N = 0, Z = 0, C = 0`
10. **Instruction: SETCC ZNVC**
    - Operation: Set Zero, Negative, Overflow, and Carry flags.
    - Expected PSW State: `V = 1, N = 1, Z = 1, C = 1`
11. **Instruction: CLRCC ZNVC**
    - Operation: Clear Zero, Negative, Overflow, and Carry flags.
    - Expected PSW State: `V = 0, N = 0, Z = 0, C = 0`
12. **Instruction: HALT**
    - Operation: Halt the program.
    - Expected PSW State: The PSW flags remain unchanged from the previous state.
    - Final PSW State: `V = 0, N = 0, Z = 0, C = 0`

Summary of Expected PSW States After Each Instruction:

1. `CLRCC VNZC`: `V = 0, N = 0, Z = 0, C = 0`
2. `SETCC C`: `V = 0, N = 0, Z = 0, C = 1`
3. `CLRCC C`: `V = 0, N = 0, Z = 0, C = 0`
4. `SETCC V`: `V = 1, N = 0, Z = 0, C = 0`
5. `CLRCC V`: `V = 0, N = 0, Z = 0, C = 0`
6. `SETCC N`: `V = 0, N = 1, Z = 0, C = 0`
7. `CLRCC N`: `V = 0, N = 0, Z = 0, C = 0`
8. `SETCC Z`: `V = 0, N = 0, Z = 1, C = 0`
9. `CLRCC Z`: `V = 0, N = 0, Z = 0, C = 0`
10. `SETCC ZNVC`: `V = 1, N = 1, Z = 1, C = 1`
11. `CLRCC ZNVC`: `V = 0, N = 0, Z = 0, C = 0`
12. `HALT`: `V = 0, N = 0, Z = 0, C = 0`

Results



```
Header (ASCII): Lab 4 Test 1.asm
Header (Bytes): 4C 61 62 20 34 20 54 65 73 74 20 31 2E 61 73 6D 09
Starting address: 1000

Enter command (I for IMEM, D for DMEM, R to display registers, C to change
register, M to change memory, B to set breakpoint, E for execute program, X
 for debugger mode): x
Clock  PC   Instruction      Fetch      Decode      Execute    Z N V C
---------------------------------------------------------------------------
  0   1000       4DD7       F0: 1000    D0: 6800
  1                         F1: 4DD7                E0: 6800    0 0 0 0
  2   1002       4DA1       F0: 1002    D0: 4DD7
  3                         F1: 4DA1                E0: 4DD7    0 0 0 0
  4   1004       4DC1       F0: 1004    D0: 4DA1
  5                         F1: 4DC1                E0: 4DA1    0 0 0 1
  6   1006       4DB0       F0: 1006    D0: 4DC1
  7                         F1: 4DB0                E0: 4DC1    0 0 0 0
  8   1008       4DD0       F0: 1008    D0: 4DB0
  9                         F1: 4DD0                E0: 4DB0    0 0 1 0
  10  100A       4DA4       F0: 100A    D0: 4DD0
  11                        F1: 4DA4                E0: 4DD0    0 0 0 0
  12  100C       4DC4       F0: 100C    D0: 4DA4
  13                        F1: 4DC4                E0: 4DA4    0 1 0 0
  14  100E       4DA2       F0: 100E    D0: 4DC4
  15                        F1: 4DA2                E0: 4DC4    0 0 0 0
  16  1010       4DC2       F0: 1010    D0: 4DA2
  17                        F1: 4DC2                E0: 4DA2    1 0 0 0
  18  1012       4DB7       F0: 1012    D0: 4DC2
  19                        F1: 4DB7                E0: 4DC2    0 0 0 0
  20  1014       4DD7       F0: 1014    D0: 4DB7
  21                        F1: 4DD7                E0: 4DB7    1 1 1 1
  22  1016       0          F0: 1016    D0: 4DD7
  23                        F1: 0                   E0: 4DD7    0 0 0 0
End of program reached. Execution stopped.

Register Contents:
R0: 0000
R1: 0000
R2: 0000
R3: 0000
R4: 0000
R5: 0000
R6: 0000
R7: 1018

Do you want to enter another command? (Y for yes, N for no):
```

The test is successful!

Test 2: Test SETCC and CLRCC in an addition script

```
; Program to test SETCC and CLRCC instructions in an addition script

        org     #1000

START

      SETCC   C              ; Set Carry bit

      MOVLZ     $15,R1       ; Load R1 with 0x000F

      MOVLZ     $12,R0       ; Load R0 with 0x000C

      ADDC      R0,R1        ; R1 <--- R1 + R0 + CARRY

      CLRCC   C              ; Clear Carry bit

      SUBC      $1,R1        ; R1 <---- R1 - 1 - CARRY

      SETCC   CV             ; Set Carry and Overflow

      SUBC    R0,R1          ; R1 <---- R1 - R0

      CLRCC   C              ; Clear Carry bit

      SETCC   N

      HALT                   ; Halt the program


      END     START
```

Expected Output

1. **SETCC C**
   - Initial PSW: `C = 0`
   - Effect: `C = 1`
   - Final PSW: `C = 1`
2. **MOVLZ $15, R1**
   - Initial R1: `0x0000`
   - Effect: `R1 = 0x000F`
   - Final R1: `0x000F`
3. **MOVLZ $12, R0**
   - Initial R0: `0x0000`
   - Effect: `R0 = 0x000C`
   - Final R0: `0x000C`
4. **ADDC R0, R1**
   - Initial R1: `0x000F`
   - Initial R0: `0x000C`
   - Initial PSW: `C = 1`
   - Operation: `R1 = R1 + R0 + CARRY`
   - Calculation: `R1 = 0x000F + 0x000C + 1 = 0x001C`
   - Final R1: `0x001C`
   - Final PSW: `C = 0` (No carry generated)
5. **CLRCC C**
   - Initial PSW: `C = 0`
   - Effect: `C = 0`
   - Final PSW: `C = 0`
6. **SUBC $1, R1**
   - Initial R1: `0x001C`
   - Initial PSW: `C = 0`
   - Operation: `R1 = R1 - 1 - CARRY`
   - Calculation: `R1 = 0x001C - 1 - 0 = 0x001B`
   - Final R1: `0x001B`
   - Final PSW: `C = 0` (No carry/borrow generated)
7. **SETCC CV**
   - Initial PSW: `C = 0, V = 0`
   - Effect: `C = 1, V = 1`
   - Final PSW: `C = 1, V = 1`
8. **SUBC R0, R1**
   - Initial R1: `0x001B`
   - Initial R0: `0x000C`
   - Initial PSW: `C = 1, V = 1`
   - Operation: `R1 = R1 - R0 - CARRY`
   - Calculation: `R1 = 0x001B - 0x000C - 1 = 0x000E`

   - Final R1: `0x000E`
   - Final PSW: `C = 0, V = 1` (No borrow generated)
9. **CLRCC C**
   - Initial PSW: `C = 1`
   - Effect: `C = 0`
   - Final PSW: `C = 0, V = 1`
10. **SETCC N**
    - Initial PSW: `N = 0`
    - Effect: `N = 1`
    - Final PSW: `N = 1, V = 1`
11. **HALT**
    - Program execution halts

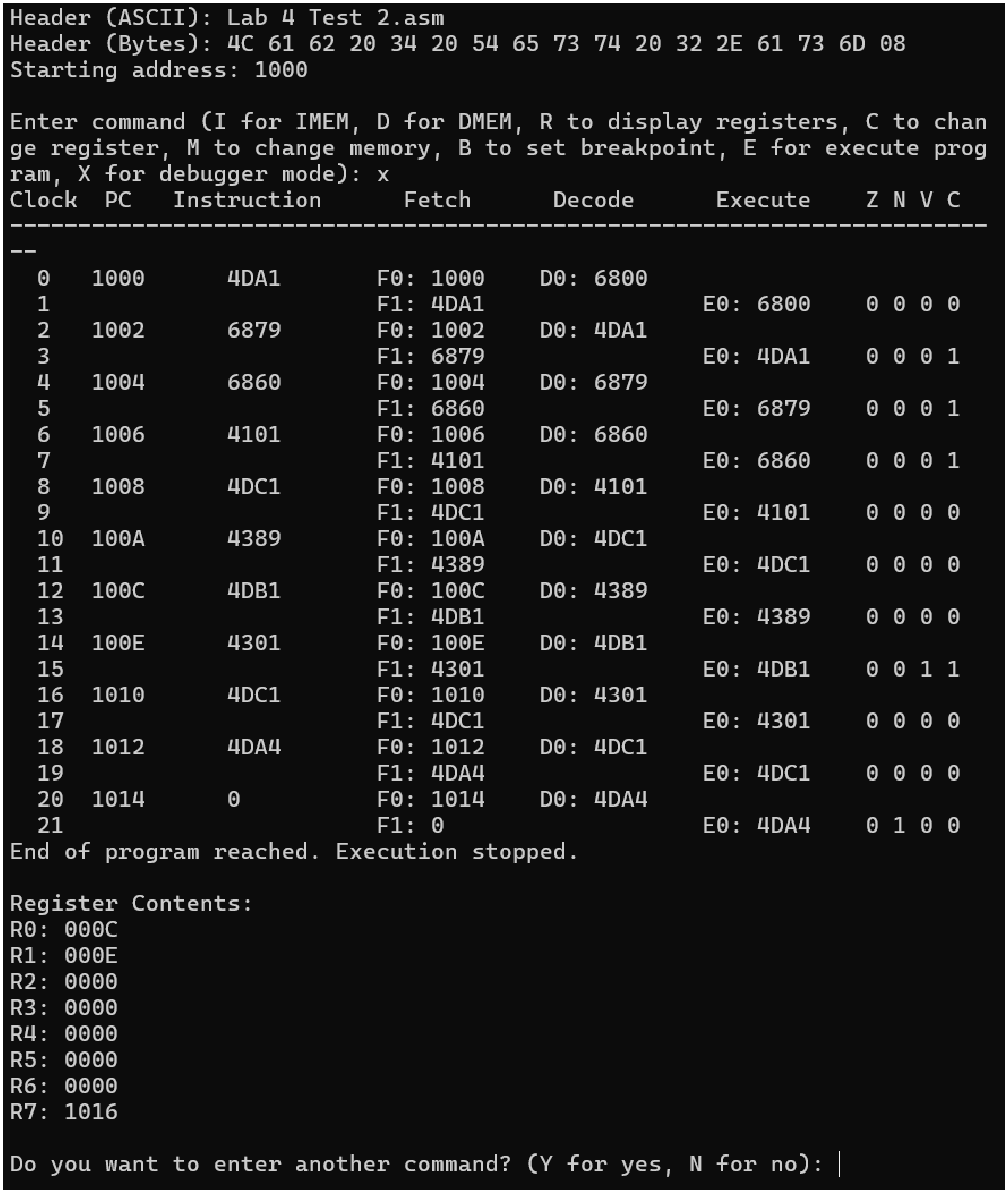
Final Register and PSW States:

- **R0**: `0x000C`
- **R1**: `0x000E`
- **R2**: `0x0000`
- **R3**: `0x0000`
- **R4**: `0x0000`
- **R5**: `0x0000`
- **R6**: `0x0000`
- **R7**: `0x1018` (Program Counter)
- **PSW**: `Z = 0, N = 1, V = 1, C = 0`

Expected PSW Changes Throughout the Program:

1. **SETCC C**: `C = 1`
2. **MOVLZ $15, R1**: No change
3. **MOVLZ $12, R0**: No change
4. **ADDC R0, R1**: `C = 0`
5. **CLRCC C**: `C = 0`
6. **SUBC $1, R1**: No change
7. **SETCC CV**: `C = 1, V = 1`
8. **SUBC R0, R1**: `C = 0`
9. **CLRCC C**: `C = 0`
10. **SETCC N**: `N = 1`
11. **HALT**: No change

Results



```
Header (ASCII): Lab 4 Test 2.asm
Header (Bytes): 4C 61 62 20 34 20 54 65 73 74 20 32 2E 61 73 6D 08
Starting address: 1000

Enter command (I for IMEM, D for DMEM, R to display registers, C to chan
ge register, M to change memory, B to set breakpoint, E for execute prog
ram, X for debugger mode): x
Clock  PC   Instruction      Fetch       Decode      Execute    Z N V C
------------------------------------------------------------------------
--
  0   1000       4DA1       F0: 1000    D0: 6800
  1                         F1: 4DA1                E0: 6800    0 0 0 0
  2   1002       6879       F0: 1002    D0: 4DA1
  3                         F1: 6879                E0: 4DA1    0 0 0 1
  4   1004       6860       F0: 1004    D0: 6879
  5                         F1: 6860                E0: 6879    0 0 0 1
  6   1006       4101       F0: 1006    D0: 6860
  7                         F1: 4101                E0: 6860    0 0 0 1
  8   1008       4DC1       F0: 1008    D0: 4101
  9                         F1: 4DC1                E0: 4101    0 0 0 0
  10  100A       4389       F0: 100A    D0: 4DC1
  11                        F1: 4389                E0: 4DC1    0 0 0 0
  12  100C       4DB1       F0: 100C    D0: 4389
  13                        F1: 4DB1                E0: 4389    0 0 0 0
  14  100E       4301       F0: 100E    D0: 4DB1
  15                        F1: 4301                E0: 4DB1    0 0 1 1
  16  1010       4DC1       F0: 1010    D0: 4301
  17                        F1: 4DC1                E0: 4301    0 0 0 0
  18  1012       4DA4       F0: 1012    D0: 4DC1
  19                        F1: 4DA4                E0: 4DC1    0 0 0 0
  20  1014       0          F0: 1014    D0: 4DA4
  21                        F1: 0                   E0: 4DA4    0 1 0 0
End of program reached. Execution stopped.

Register Contents:
R0: 000C
R1: 000E
R2: 0000
R3: 0000
R4: 0000
R5: 0000
R6: 0000
R7: 1016

Do you want to enter another command? (Y for yes, N for no):
```

The test is successful!