Experiment 7
Assembly Language Programming

• Objectives:
  – Use the OMTC program, a computer simulator, and its relatively simple instruction set for assembly language programming
  – Reinforce the relationship between hardware and software; recall that “instructions” serve as control codes for digital circuits
A Hardware Specific Language

• Assembly Languages are *hardware specific* -- different microprocessors (e.g., a Pentium processor or a PowerPC processor) may have different instruction sets

• Even when the machines have the same operations (e.g., an “ADD” operation), the syntax of the instruction may be different and the associated machine code (the *control codes*) will almost certainly be different
Macro Assembler Programming

Microsoft Macro Assembler 6.1 Programming

By Len Dorfman and Marc J. Neuberger

Learn how to write easy-to-read, structured assembly code with this exciting, next-generation program.

Includes a complete subroutine library on a 3.5" disk.

```assembly
set inverse bar row location variables
mov old_row, 1

relocate Window structure ddps to locals
IF mdl eq 1
mov bx,ddptr
ELSE
lea bx,ddptr
ENDIF

INVOKE displayWind, ADDB (DDP[bx]).ddwind

set ax:[di] to point to M_ITEM structure
IF mdl eq 1
push ds
pop es
mov di,mitpr
ELSE
lea di,mitpr
ENDIF

write menu items to the drop down window
mov al,(DDP[bx]).lnum
```
Addition Machine

**Control Buttons** (2-state)

**Address Buttons** (2-state)

**Accumulator Register**

**Memory Locations 0 - 7**

**Execute Button** (temporary contact)

**Control Button Settings (1 means button is pushed in):**

00 – Load contents of Box “Address” into Box A
01 – Store contents of Box A into Box “Address”
10 – Add contents of Box “Address” to contents of Box A
11 – Do nothing
Addition Machine

Control Buttons (2-state)  Address Buttons (2-state)  Button set to “1”  Execute Button (temporary contact)

Control Button Settings (1 means button is pushed in):
00 – Load contents of Box “Address” into Box A
01 – Store contents of Box A into Box “Address”
10 – Add contents of Box “Address” to contents of Box A
11 – Do nothing
Addition Machine

Control Button Settings (1 means button is pushed in):
00 – Load contents of Box “Address” into Box A
01 – Store contents of Box A into Box “Address”
10 – Add contents of Box “Address” to contents of Box A
11 – Do nothing
Addition Machine

Control Buttons
(2-state)

Address Buttons
(2-state)

Execute Button
(temporary contact)

Control Button Settings (1 means button is pushed in):
00 – Load contents of Box “Address” into Box A
01 – Store contents of Box A into Box “Address”
10 – Add contents of Box “Address” to contents of Box A
11 – Do nothing
Ole Miss Teaching Computer (OMTC)

• The OMTC program is a computer simulator.

• It simulates a small teaching computer with only 256 bytes of memory.

• The simulated machine has a built-in assembler program and can step through program execution one instruction at a time.

• The contents of the registers and the memory of the simulated machine are easily accessed through a visual interface.
OMTC Specifications

• 8-bit machine
• 256 bytes of memory (0-255, or 00-FF)
• 8 registers (temporary memory used during execution)
• 4 addressing modes
• 48 Instructions
• Execution by
  – machine cycle
  – instruction
• Save/Restore memory
OMTC Main Window
OMTC Main Window

Binary and Hexadecimal values shown

Background color change indicates that the value *could* have been changed by the last instruction executed
OMTC Main Window

Contents of two portions of memory
OMTC Registers

Accumulator (A): A general-purpose register; virtually all calculations involve this register.

Index Register (X): This register is used as an index register, or counter, in some instructions. It is also used in combination with the A register to hold 16-bit numbers in multiply and divide operations.

Instruction (INSTR): During execution this register displays the operation code portion of an instruction (which tells the computer what type of operation to perform).
OMTC Registers (cont.)

Memory Address Register (ADDR): Often shows the address in memory of the data to be used by the instruction. The value displayed, however, depends on the operand address portion of the instruction, as well as on the addressing mode of the instruction.

Program Counter (PC): Shows the location in memory of the next instruction to be executed.

Flag Register (C+N+Z): Individual bits representing flags. The three flags are the Carry (C), the Negative (N), and the Zero (Z) flags.
Addressing Modes

Implied Address

In this form the address of the data to be operated on is implied by the instruction. The operand address (the second byte of the instruction) must be present as part of the instruction, but it is not used.

Immediate Mode Addressing

In this form of addressing, the operand address represents the data to be used by the instruction directly. For example, in the instruction LDA #05, the number 5 is to be loaded into the A register.

Absolute Mode Addressing

In this form of addressing, the operand address represents a memory address, rather than numerical data. LDA 05, for example, means to load the accumulator (A) with the data that is stored in memory location 05.

X-Indexed Mode Addressing

In this form of addressing, the operand address again represents a memory address. The memory address of the data, however, is obtained by adding the operand address to the contents of the X register. For example, if the X register contains the value 02, then the instruction LDA 05,X means to load the accumulator with the data that is stored in memory location 07.
### Extract from Instruction Set Description

<table>
<thead>
<tr>
<th>Hex Code</th>
<th>Mnemonic</th>
<th>Instruction and RTL Representation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>LDA #nn</td>
<td>Load A</td>
<td>Load the A register with the number (nn).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(A \leftarrow nn)</td>
<td>Flags affected: N Z</td>
</tr>
<tr>
<td>12</td>
<td>LDX #nn</td>
<td>Load X</td>
<td>Load the X register with the number (nn).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(X \leftarrow nn)</td>
<td>Flags affected: N Z</td>
</tr>
<tr>
<td>14</td>
<td>ADD #nn</td>
<td>ADD</td>
<td>Add the number (nn) to the A register and leave the result in A</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(A \leftarrow A+nn)</td>
<td>Flags affected: C N Z</td>
</tr>
</tbody>
</table>
A Simple Example: Add two numbers and store the result

Step 1. Decide what variables are needed for data and (mentally) reserve memory locations to store them.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Memory Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A$</td>
<td>00</td>
</tr>
<tr>
<td>$B$</td>
<td>01</td>
</tr>
<tr>
<td>$C$</td>
<td>02</td>
</tr>
</tbody>
</table>

The *program* can then be put anywhere *else* in memory -- for example, we may choose to start the program at memory location 20.
Example (cont.)

Step 2. Review the OMTC *Instructions* available to see what instructions can be used to accomplish the task.

You will note, for example, that there is no single instruction capable of adding together two numbers that are currently stored in memory.

There is an instruction, however, to add a number stored in memory to a number in the Accumulator, and there is an instruction to load the Accumulator with a number stored in memory.
Step 3. Write the assembly language program.

<table>
<thead>
<tr>
<th>Memory Address</th>
<th>Operation Code</th>
<th>Operand Address</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>LDA</td>
<td>A</td>
<td>; Load A into the Accumulator</td>
</tr>
<tr>
<td>22</td>
<td>ADD</td>
<td>B</td>
<td>; Add B to the Accumulator</td>
</tr>
<tr>
<td>24</td>
<td>STA</td>
<td>C</td>
<td>; Store Accumulator contents to C</td>
</tr>
<tr>
<td>26</td>
<td>STP</td>
<td>00</td>
<td>; Stop the program (the operand address 00 is not used)</td>
</tr>
</tbody>
</table>

Note that *Instructions always* occupy *two bytes* of memory (16 bits)
Example (cont.)

Step 4. Store the *program* in the OMTC memory:

- Hand-assemble the program into Hex codes, or
- Use the built-in assembler

Step 5. Store the *data* in the OMTC memory

Step 6. Set the Program Counter (PC) register to the location in memory of the first executable statement of your program

Step 7. Run the program
### Hand Assembly

<table>
<thead>
<tr>
<th>Memory Address</th>
<th>Contents</th>
<th>Why?</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>20</td>
<td>(20 is the LDA $nn$ op code)</td>
</tr>
<tr>
<td>21</td>
<td>00</td>
<td>($A$ is associated with memory address 00)</td>
</tr>
<tr>
<td>22</td>
<td>24</td>
<td>(24 is the ADD $nn$ op code)</td>
</tr>
<tr>
<td>23</td>
<td>01</td>
<td>($B$ is stored in memory address 01)</td>
</tr>
<tr>
<td>24</td>
<td>21</td>
<td>(21 is the STA $nn$ op code)</td>
</tr>
<tr>
<td>25</td>
<td>02</td>
<td>($C$ is stored in memory address 02)</td>
</tr>
<tr>
<td>26</td>
<td>0F</td>
<td>(0F is the STP op code)</td>
</tr>
<tr>
<td>27</td>
<td>00</td>
<td>(operand address is required, but is not used)</td>
</tr>
</tbody>
</table>

Hex codes as entered into the OMTC
Using the Built-In Assembler

<table>
<thead>
<tr>
<th>Memory Address</th>
<th>Instruction</th>
<th>Why?</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>LDA 00</td>
<td>Load A into the Accumulator</td>
</tr>
<tr>
<td>22</td>
<td>ADD 01</td>
<td>Add B to the Accumulator</td>
</tr>
<tr>
<td>24</td>
<td>STA 02</td>
<td>Store Accumulator contents to C</td>
</tr>
<tr>
<td>26</td>
<td>STP</td>
<td>Stop the program (the operand address 00 is not used)</td>
</tr>
</tbody>
</table>

Instructions as entered into the OMTC
# Tracking Program Operation (Example1.dat)

<table>
<thead>
<tr>
<th>Contents of Register or Memory Address</th>
<th>Register Name or Memory Address</th>
<th>comments</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>( X )</td>
<td></td>
</tr>
<tr>
<td></td>
<td>04</td>
<td>04 04 00 00</td>
</tr>
<tr>
<td></td>
<td>03</td>
<td>01 00 01 01</td>
</tr>
<tr>
<td></td>
<td>02</td>
<td>01 01 01 01</td>
</tr>
<tr>
<td></td>
<td>01</td>
<td>03 01 02 03</td>
</tr>
</tbody>
</table>

- \( X \)
  - 04: \( X \) with [66]
  - 03: decrement \( X \)

- \( A \)
  - 00: load \( A \) from [66]
  - 01: add 1 to \( A \)
  - 02: store \( A \) to mem. add. 67
  - 03: add [68] to \( A \)
  - 04: store \( A \) to mem. add. 68
  - 05: decrement \( X \)

(initial data)
(load \( X \) with [66])
(decrement \( X \))
(load \( A \) with [67])
(add 1 to \( A \))
(store \( A \) to mem. add. 67)
(add [68] to \( A \))
(store \( A \) to mem. add. 68)
(decrement \( X \))

etc.
Additional Notes

• The second example program (Example2.dat) is an example of a program that *modifies its own code*

• Remember that negative numbers must be entered in and are stored in two’s complement form