CS 221 Computer Architecture

Week 5: PEP/6 Assembly

Fall 2001

Course Schedule

W1	Sep 11- Sep 14	Introduction	
W2	Sep 18- Sep 21	Information Representation (1)	(Chapter 3)
W3	Sep 25- Sep 28	Information Representation (2)	(Chapter 3)
W4	Oct 2- Oct 5	Computer Architecture Pep/6	(Chapter 4)
W5	Oct 9- Oct 12	Assembly Language	(Chapter 5)
W6*	**Oct 16- Oct 19**	**High-Level Languages (1)**	**(Chapter 6)**
W7*	**Oct 23- Oct 26**	**High-Level Languages (2)**	**(Chapter 6)**
	Midterm*		
W8	Oct 30- Nov 2	Instruction Sets	(Tanenbaum)
W9	Nov 6 - Nov 9	Storage Management	(Chapter 9)
W10*	**Nov 13- Nov 16**	**Devices and Interrupts**	**(Chapter 8)**
W11	Nov 20- Nov 23	Combinational Logic	(Chapter 10)
W12	Nov 27- Nov 30	Sequential Logic & Review	(Chapter 11)

**Marked classes will be taught by Dave Dove, same place same slot*

Plan for Week 5

- **Introductory Assembly Language**
  - The Assembler
  - Symbol Tables
  - Mnemonics, Pseudo-Operations and Symbols
- **Control structures and branching**
- **Operating System I/O Services**
- **First Look at Higher-Level Languages**

- **Assignment 2 will be posted**
- **Readings**
  - This week: Chapter 5
  - Next week: Chapter 6

I.
Introduction to Assembly

What is Assembly?

- **Assembly language (level 5) is just an alternative representation for machine language**
  - Represents machine code with words that people can decipher more easily
- **The operating system level (level 4) is interposed between the assembly language level and the machine language level**
  - We will use OS facilities in our assembly programs
- **So assembly is "a symbolic representation of the machine language of a computer"**
  - Most instructions in assembly language correspond directly to individual instructions in machine language.

Elements of Assembly

- **The items in an assembly language program fall into four categories**
  - Comments,
  - Mnemonics,
  - Pseudo-operations,
  - Symbols.
- **Let's discuss these using a sample program.**
  - The program reads two characters, then outputs the characters whose ASCII codes are 32 greater than the input characters.
  - If the input characters are upper-case letters, this has the effect of converting to lower-case.
  - You create this source code file using any text editor.

Assembly Language Source File

```
;
; An example program which reads two characters from
; the input device, changes upper-case letters to lower-case
; letters, then writes the resulting two
; characters to output.  The program does not check that the
; input actually does consist of upper-case letters.
;
chari h#0025,d  ; Read first character into Mem[h#0025].
chari h#0026,d  ; Read second character into Mem[h#0026].
loada d#0,i     ; A := 0
                ; (Clears the accumulator so its
                ; top half will be 0 after
                ; the ldbyta instruction , which only affects
                ; the 8 least-significant bits)
ldbyta h#0025,d ; A := Mem[h#0025], the first character.
loadx d#0,i     ; X := 0 (Clear the index register).
ldbytx h#0026,d ; X := Mem[h#0026], the second character.
adda   d#32,i   ; A := A + gapBetweenCases (uncapitalize).
addx   d#32,i   ; X := X + gapBetweenCases (uncapitalize).
stbyta h#0025,d ; Replace the character at Mem[h#0025] with
                ; the corresponding uncapitalized character.
stbytx h#0026,d ; Replace the character at Mem[h#0026].
charo h#0025,d  ; Print the first character.
charo h#0026,d  ; Print the second character.
stop            ; Halt the computer.
.block d#2      ; Space for the two characters.
.end            ; End of the assembly language program.
```


Assembler

- **Because we now have an intermediate representation, we need to translate our code.**
  - This is done by the assembler.
  - Translates symbolic representations into machine code.
  - The command for this is "***asem [input-file]***" at the unix command line
- **The assembler generates two things:**
  - output file that contains the machine code (.exe in DOS)
  - and a listing of that machine code in readable text
- **The machine code generated is called object code**
- **The object code can then be run as a PEP/6 program**

Code Listing

```
      Object
Addr  code   Mnemon  Operand    Comment
-------------------------------------------------------------------------------
             ;
             ; An example program which reads two characters from
             ; the input device, changes upper-case letters to lower-case
             ; letters , then writes the resulting two
             ; characters to output.  The program does not check that the
             ; input consists of capital letters.
             ;
0000  D90025 CHARI   h#0025,d   ; Read first character into Mem[h#0025].
0003  D90026 CHARI   h#0026,d   ; Read second character into Mem[h#0026].
0006  080000 LOADA   d#0,i      ; A := 0
             ; (Clears the accumulator so its
             ; top half will be 0 after
             ; the ldbyta instruction , which only affects
             ; the 8 least-significant bits)
0009  510025 LDBYTA  h#0025,d   ; A := Mem[h#0025], the first character.
000C  0C0000 LOADX   d#0,i      ; X := 0 (Clear the index register).
000F  550026 LDBYTX  h#0026,d   ; X :=  Mem[h#0026], the second character.
0012  180020 ADDA    d#32,i     ; A := A + gapBetweenCases (uncapitalize).
0015  1C0020 ADDX    d#32,i     ; X := X + gapBetweenCases (uncapitalize).
0018  590025 STBYTA  h#0025,d   ; Replace the character at Mem[h#0025] with
             ; the corresponding uncapitalized character.
001B  5D0026 STBYTX  h#0026,d   ; Replace the character at Mem[h#0026].
001E  E10025 CHARO   h#0025,d   ; Print the first character.
0021  E10026 CHARO   h#0026,d   ; Print the second character.
0024  00     STOP               ; Halt the computer.
0025  0000   .BLOCK  d#2        ; Space for the two characters.
0027         .END               ; End of the assembly language program.
```


Elements of Assembly

(1) Comments

- **Observe that assembly language follows a rigid format**
  - Only one statement per line
- **The semicolon is the comment marker in PEP/6 assembly language.**
  - Everything from a semicolon to the end of the line is a comment.
- **Lots of comments are necessary to make program readable!**

(2) Mnemonics

- **In assembly, a mnemonic is a word that stands for the instruction, to help you remember it.**

In our example:

ADDA d#32, i Generates the machine code 180020 (h).
- ADDA is mnemonic for instruction "add to the accumulator".
- It corresponds to an opcode of 00011 and a register specifier of 0 (A or the accumulator) in machine language.
- d#32 says that the operand specifier for this instruction is decimal 32
- The "i" means that the operand specifier is to be interpreted using the immediate addressing mode.

ADDX d#32, i. Generates machine code 1C0020 (h)
- Uses index register (X)
- Generated object code differs only in having a register specifier of 1 instead of 0 -> 1C instead of 18 hex.

Note: Immediate Addressing Mode

- **Remember we so far have only used direct addressing mode**
  - Operand indicates address in memory of value to be used
- **In immediate addressing mode, the operand specifier is the value to be used in the instruction**
  - So the code generated by "ADDA d#32,i" adds the value 32 directly to the accumulator.
- **Not all addressing modes work with all instructions.**
  - It doesn't make sense to specify the immediate mode with a STORE instruction, since you cannot store anything in an immediate constant; you can only store into memory.
  - The assembler will complain about illegal addressing modes at assembly time.

(3) Pseudo Operations

- **In our example, ".BLOCK d#2" and ".END" are pseudo operations.**
- **In PEP/6, mnemonics generate instruction bits in the machine code, while pseudo-operations generate data bits.**
  - Like .BLOCK leaves space for data in the object (machine) code. It generates two bytes of storage, initialized to zero in the object code.
  - We use it to put two bytes worth of data space into the program, into which we can read our two characters.
  - The operand to the .BLOCK pseudo-operation is the number of bytes of storage to allocate.
- **Pseudo-ops can also provide directions to the assembler.**
  - Like .END is a pseudo-operation that provides overall direction to the assembler. In this case, it marks the end of the assembly language source file, and tells the assembler to stop translating the program.
  - Is different from STOP, which only tells processor to stop.
- **Pseudo-operations are also known as assembler directives, or dot commands (because they start with a dot).**

More Pseudo Operations

.BLOCK value
- Allocates the number of bytes indicated by value and initializes them with zero.

.BYTE value
- Reserves only one byte of data space, and initializes it with value.

.WORD value
- Reserves one word (16 bits) of data space, and initializes it with value.

.ASCII value *Example: .ascii / + 3 = /*
- reserves enough space for the characters between the delimiters /, and initializes with those chars.
- The delimiter can be any character, not just "/".

(4) Symbols

- **In our example, how did the programmer know that the addresses of the bytes used to store the two characters would be h#0025 and h#0026?**
  - Well, by counting the number of bytes that would be occupied by the machine code
  - This is a bad idea.
- **By using symbols, the assembler to does the counting for you, and makes program easier to read**
  - Here is an example of the same program using symbols char00, char01, and casegap
  - Instead of the explicit addresses h#0025, h#0026, and the integer constant d#32.


```
        ; An example program which reads two characters from
        ; the input device, changes upper-case letters to lower-case
        ; letters , then writes the resulting two characters to output.
        ;
        casegap: .equate h#0020
        ;
        chari char00,d   ; Read the first character.
        chari char01,d   ; Read the second.
        loada d#0,i      ; Clear the accumulator
                         ; (so top half is 0 after
                         ; ldbyta)
        ldbyta char00,d ; A := first character.
        loadx d#0,i      ; Clear the index register.
        ldbytx char01,d ; X := second character.
        adda   casegap,i ; A := A + gapBetweenCases.
        addx   casegap,i ; X := X + gapBetweenCases.
        stbyta char00,d ; Store the new first char.
        stbytx char01,d ; Store the new second char.
        charo char00,d  ; Print the first character.
        charo char01,d  ; Print the second character.
        stop
        ;
        char00: .byte d#0
        char01: .byte d#0
        .end
```


Symbols: Label Definitions

- **A symbol is an identifier used as a name for an address or another numerical constant.**
  - Two types: label definition and constant definition.
- **Label Definitions**
  - When you put symbol at the left of a source line and followed by a colon you are defining a label for the address of the object code that the line generates.
  - Here, char00 is defined as a symbol for the address of the byte we are reserving to store the first input character.
  - This can then be used earlier in the code to refer to that address.
- **How does this work?**
  - At the same time as the assembler generates object code, it also maintains a symbol table

The Symbol Table

- **A Symbol Table is a mapping from identifiers (or symbols) to numerical values.**
  - As the assembler reads the source code, whenever it sees a label definition, it stores that symbol in the symbol table.
  - It associates it with the address it generates for that line of source code.
- **So what happens when you put a symbol into the operand position of a mnemonic or pseudo-op?**
  - The assembler substitutes the numeric value it has stored for that symbol when it generates the machine code for that line.

Assembled Code

```
      Object
Addr  code   Symbol    Mnemon  Operand    Comment
-------------------------------------------------------------------------
             casegap:  .EQUATE h#0020
0000  D90025           CHARI   char00,d   ; Read the first character.
0003  D90026           CHARI   char01,d   ; Read the second.
0006  080000           LOADA   d#0,i      ; Clear the accumulator
                       ; (so top half is 0 after
                       ; ldbyta)
0009  510025           LDBYTA  char00,d   ; A := first character.
000C  0C0000           LOADX   d#0,i      ; Clear the index register.
000F  550026           LDBYTX  char01,d   ; X := second character.
0012  180020           ADDA    casegap,i  ; A := A + gapBetweenCases.
0015  1C0020           ADDX    casegap,i  ; X := X + gapBetweenCases.
0018  590025           STBYTA  char00,d   ; Store the new first char.
001B  5D0026           STBYTX  char01,d   ; Store the new second char.
001E  E10025           CHARO   char00,d   ; Print the first character.
0021  E10026           CHARO   char01,d   ; Print the second character.
0024  00               STOP
                       ;
0025  00      char00:  .BYTE   d#0
0026  00      char01:  .BYTE   d#0
0027                   .END
```


After assembling the code:

- **The assembler prints out the symbol table it used.**

```
Symbol table
---------------------------------------
Symbol    Value        Symbol    Value
---------------------------------------
casegap   0020         char00    0025
char01    0026
---------------------------------------
No errors.  Successful assembly.
```

  - The lines labeled by char00 and char01 generate the object code bytes at addresses h#0025 and h#0026
  - For the assembly statements where char00 and char01 are used as operands
  - The assembler puts h#0025 and h#0026 into the operand specifier portions of the output machine language.

Symbols: Constant Definitions

- **In the Symbol Table we also see a Constant Definitions**
  - casegap doesn't correspond to an address in the object code. Instead, it is a name that we are giving to the difference between the ASCII codes.
  - It's just the name of a constant value, like in high-level language.
- **The .EQUATE pseudo-operation lets us define the constant.**

  name*: .EQUATE *value
  - tells the assembler to add name to the symbol table, associating it with the number value.
- **You can use symbols before defining them. This works because conceptually assemblers make two passes over the source file.**
  - In the first, they find all the address labels and .EQUATEs and use those definitions to build the symbol table.
  - In the second pass, they replace uses of the symbols with the values now stored in the symbol table.

Symbols Use

- **Symbols are precursor of variable and constant definitions in high-level languages.**
  - However, assembly language symbols are not associated with any notion of type.
  - The symbols are just a way of associating numbers with names, and of making it easy to refer to addresses in the generated code.
- **The rules for forming symbol names are more restrictive than those for variable names in C or Pascal.**
  - In the PEP/6 assembler, symbols may be no more than 8 characters long.
  - They can contain digits and characters, but must begin with a character.
  - Symbols are case-sensitive.

II. Control Structures: Branching

Assignment 2

- **Posted on the web**
- **Due October 25th, 11:00 in box 221 Goodwin 2nd**

Assignment 2: A simple PEP/6 Assembly program

Write a program for the Pep/6 which reads two ascii words (i.e., the words you find in a book, not integer words), of 4 lowercase 8-bit characters each, from the input, and that outputs the two words again in the correct alphabetical order.

Input will be restricted to 4 character words only, so you do not need to handle any longer or shorter words! The alphabetical order is of course indicated by the first character. If the first two characters are the same, you compare the next two characters. You do not need to go beyond this.

Test your program with inputs which execute all the instructions.

Make sure you include a USER guide this time!

Branching

- **So what if we want to base the actions our program takes on a value of say, input?**
  - Or want to iterate part of the program?
  - Or jump to a subroutine that can be reused?
- **For this we need to have control structures**
  - Like if, else in Higher-level languages
  - And while and for loops
- **In assembly, this is implemented by branching.**
  - Next instruction indicated by Program Counter
  - Change the PC, and another part of code is executed.

Straight Branch

`BR` *`operand`*

Sets the Program Counter to *`operand`*

- If you make operand a label, you can branch to a subroutine this way
- A branch is usually to an unvarying address, so if you leave out the addressing mode it will be assumed to be immediate mode (,i).
- Branching is why in the Von Neumann Cycle is not

1. Fetch the next instruction.
2. Execute the instruction. (Changes PC to the branch address)
3. Increment the PC. (Oops: added 3, we're shooting past it now!)

Conditional Branch

- **Conditional branches can execute code on the basis of comparing the values of variables.**
  - If input == 'Q' then do the quit subroutine.
- **Pep/6 has 8 conditional branches: `BRXX`**

`if(a` *`XX`* `b) then` where XX is

BRLE	**Branch less then or equal to**	**(<=)**
BRLT	**Branch on less than**	**(<)**
BREQ	**Branch on equal to**	**(==)**
BRNE	**Branch on not equal to**	**(!=)**
BRGE	**Branch on greater than or equal to**	**(>=)**
BRGT	**Branch on greater than**	**(>)**
BRV	**Branch on oVerflow**	**(branches if V bit set)**
BRC	**Branch on Carry**	**(branches if C bit set)**

Conditional Branch (2)

- **So how does the comparison work?**
  - The branches test one or two of the status bits NZVC
  - If the condition is true, the operand is placed in PC, causing the branch.
  - If not, the operand is not placed in PC, causing normal flow.
- **To make the comparison that sets the status bits we use the `COMPr operand` instruction.**
  - Compares an operand with the value of a register `r`.
  - Works just like a subtract but does not affect registers.
  - Does set the status bits accordingly.

Conditional Branch (3)

- The ***`COMPr`*** instruction compares operand as follows:

```
if(register XX operand)
COMPr = register - operand
```

negative: `o>r` positive: `o<r` zero: `r==o`

- Result is tested by BRXX using status bits
- The branch instruction will typically jump to the code that needs to be executed when the comparison holds not true: the else.
- So use the branch that is the opposite of the comparison you want to make.
- If the comparison is greater than, branch on less than or equal to.

Conditional Branch (4)

Here's an overview of what the conditional branches test:

BRLE	**if N=1 or Z=1 then PC := Operand**
BRLT	**if N=1 then PC := Operand**
BREQ	**if Z=1 then PC := Operand**
BRNE	**if Z=0 then PC := Operand**
BRGE	**if N=0 then PC := Operand**
BRGT	**if N=0 or Z=0 then PC := Operand**
BRV	**if V=1 then PC := Operand**
BRC	**if C=1 then PC := Operand**

Conditional Branch Example

Assembly Language

```
0000 700005          BR      Main
               limit: .EQUATE d#100
0003 0000      num:   .BLOCK  d#2
               ;
0005 E90003 Main:   DECI    num,d       ;cin >> num
0008 090003 If:     LOADA   num,d       ;if (num >= limit)
000B B80064         COMPA   limit,i
000E 800020         BRLT    Else
0011 E00068         CHARO   c#/h/,i     ;cout << "high"
0014 E00069         CHARO   c#/i/,i
0017 E00067         CHARO   c#/g/,i
001A E00068         CHARO   c#/h/,i
001D 700029         BR      EndIf       ;else
0020 E0006C Else:   CHARO   c#/l/,i     ;cout << "low"
0023 E0006F         CHARO   c#/o/,i
0026 E00077         CHARO   c#/w/,i
0029 00      EndIf: STOP
002A                .END
```


Operating System I/O Services

- **PEP/6 can only input and output characters**
  - This is not very useful for numbers
- **The operating system provides a couple of useful extra instructions: DECI, DECO, HEXO.**
  - DECI read decimal number from input.
  - DECO outputs decimal number from memory.
  - HEXO outputs hexadecimal number from memory.
- **The opcode specifier which the assembler generates for DECO, 11110, is unimplemented at Level 3.**
  - When the PEP/6 encounters it, it traps into the operating system.
  - The interrupt service routine runs a program that reads in a decimal number.

An Example: Converting Decimals

```
      Object
Addr  code   Symbol   Mnemon  Operand    Comment
0000  700005         BR      start      ; Branch to start label.
              tab:    .EQUATE d#9        ; ASCII code for a tab.
0003  00      value:  .BLOCK  d#1        ; Space for one word, but with
0004  00      value1: .BLOCK  d#1        ;   a label that lets us get at
              ;   the second byte in the word.
0005  E10003 start:  DECI    value,d    ; Read a decimal number into value.
0008  D80064         CHARO   c#/d/,i    ; Print value in decimal (including.
000B  D80023         CHARO   c#/#/,i    ;   "d#" prefix).
000E  E90003         DECO    value,d
0011  D80009         CHARO   tab,i      ; Tab to next column.
              ;
0014  D80068         CHARO   c#/h/,i    ; Print value in hexadecimal
0017  D80023         CHARO   c#/#/,i    ;   (including "h#" prefix).
001A  F10003         HEXO    value,d
001D  D80009         CHARO   tab,i      ; Tab to next column.
              ;
0020  D80063         CHARO   c#/c/,i    ; Print value as two characters
0023  D80023         CHARO   c#/#/,i    ;   (including "c#" prefix and
0026  D8002F         CHARO   c#'/',i    ;   "/" delimiters).
0029  D90003         CHARO   value,d
002C  D90004         CHARO   value1,d
002F  D8002F         CHARO   c#'/',i
0032  00             STOP
0033                 .END
```


Example: Converting Decimals

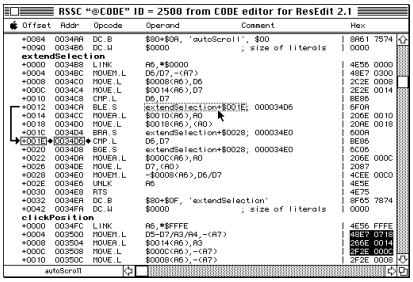
- **The program uses the traps to do what you did in the first assignment:**
  - reading decimal numbers with DECI instruction,
  - writing them in decimal format with DECO
  - writing them in hexadecimal format with HEXO
  - then using the machine instruction CHARO to print the number as two characters.
- **Here's a run of the program:**

```
20299
d#20299 h#4F4B  c#/OK/
```


From machine code back to assembly language

- **This process is called disassembly**
  - It is very useful to look what's happening when you run an executable in Windows or MacOS.
- **Problem: 1-1 relationship between machine instructions and mnemonics**
  - but not between the pseudo-operations and the bits produced by them.
  - Data produced by pseudo-operations may even have the same values as the code generated by some instructions.
  - As a result you cannot reliably regenerate from an object file the assembly language source code which generated it.
- **Debuggers and disassemblers do attempt to map back to the instructions that produced a given program, but are not 100% successful**

Example: ResEdit CODE resource editor



III. A First Look at Compiling Higher-Level Languages

Compiling Higher-Level Code

- **What you run on your computer as a program, the executable, consists of machine or object code.**
- **The assembler translates your assembly source code into such code.**
- **Higher-level language compilers do exactly the same thing.**
  - But it is more difficult for the compiler because there is no one-to-one relationship with machine instructions
  - C statements will invariably result in multiple machine instructions
- **Java is different in that the object code does not run on a hardware CPU: its virtual like PEP/6**

Assignment Statements

- **Let's examine a simple java program to see how it might be compiled**

```
#define C_to_K 273

int degreesC, degreesK;
char chLower, chUpper;

scanf ("%d %c", &degreesC, &chLower);

degreesK = degreesC + C_to_K;
chUpper = chLower - ('a' - 'A');

printf ("%d %c", degreesK, chUpper);v
```


In assembly

```
0000 70000B            BR start         ;Branch around data.
               CtoK: .EQUATE   d#273    ;const CtoK   273
0003 0000   degreesC: .WORD      d#0          ;int degreesC
0005 0000   degreesK: .WORD      d#0          ;int degreesK
0007 00     chLower:  .BYTE      d#0          ;char chLower
0008 00     chUpper:  .BYTE      d#0          ;char chUpper
0009 0000   temp:        .WORD   d#0          ;int temp
            ;
000B E90003 start:      DECI     degreesC,d ;read in  degreesC
000E D90007                      CHARI    chLower,d ;read in chLower
            ;
0011 090003                      LOADA    degreesC,d ;degreesK = degreesC + CtoK
0014 180111                      ADDA     CtoK,i
0017 110005                      STOREA   degreesK,d
            ;
001A 080061                      LOADA    c#/a/,i      ; temp = 'a - 'A'
001D 200041                      SUBA     c#/A/,i
0020 110009                      STOREA   temp,d
            ;
0023 510007                      LDBYTA   chLower,d  ;chUpper := chLower - temp
0026 210009                      SUBA     temp,d
0029 590008                      STBYTA   chUpper,d
            ;
002C F10005                      DECO     degreesK,d  ;print out degreesK, " ", chUpper
002F E00020                      CHARO    c#/ /,i
0032 810008                      CHARO    chUpper,d
            ;
0035 00                          STOP
0036                             .END
```


Optimizations

- **Early compilers for high-level languages went through the source program line-by-line**
  - Generated a "literal translation".
  - For example, the assembly language above for computing the expression

    chUpper := chLower - ('a' - 'A')
  - follows directly from the way the C source is written.
  - To compute the parenthesized "('a' - 'A')" and then subtract it from "chLower", the result of the parenthesized expression is stored as a temporary value in memory.
- **Over the last two decades, optimizing compilers have been developed which use various tricks to produce better quality code.**

Optimizing Our Example

- **In our example, by rephrasing the right-hand side of the assignment, instead of translating directly the expression as written in C, we can:**
  - avoid the need for the temporary variable
  - remove an instruction
  - remove a reference to memory (remember that accessing memory is much slower than accessing registers)

Optimizing our Example (2)

```
0000  700009          BR      start       ; Branch around data.
            CtoK:     .EQUATE d#273       ; define CtoK 273
0003  0000    degreesC:.WORD   d#0         ; int degreesC;
0005  0000    degreesK:.WORD   d#0         ; int degreesK;
0007  00      chLower: .BYTE   d#0         ; char ch_Lower;
0008  00      chUpper: .BYTE   d#0         ; char ch_Upper;
                ;
0009  E90003 start:   DECI     degreesC,d ; read in degreesC, chLower
000C  D90007          CHARI    chLower,d
                ;
000F  090003          LOADA    degreesC,d ; degreesK = degreesC + CtoK
0012  180111          ADDA     CtoK,i
0015  110005          STOREA   degreesK,d
                ;
0018  080000          LOADA    d#0,i      ; Make sure accumulator's top
                ; half is 0 before loading
                ; a character into the bottom
                ; half.
                ;
001B  510007          LDBYTA   chLower,d  ; chUpper = chLower
001E  200061          SUBA     c#'a',i    ;   - 'a' + 'A'
0021  180041          ADDA     c#'A',i
0024  590008          STBYTA   chUpper,d
                ;
0027  F10005          DECO     degreesK,d ; print out degreesK, " ", chUpper
002A  E00020          CHARO    c#' ',i
002D  E10008          CHARO    chUpper,d
                ;
0030  00              STOP
0031                  .END
```


No Typing In Assembly Language

- **It is tempting to think symbols in assembly language are equivalent to high-level language variables.**
  - assembly language lacks the notion of type.
- **When a compiler translates a high-level language program, it stores information about the variables in a symbol table.**
  - For each variable, there is an entry in the symbol table containing the variable's name and its address.
  - The difference is that the compiler's symbol table will also contain the type of the variable.
  - Type determines how much space will be required to store the variable, what operations are possible, and how those operations are to be implemented.
- **Types also allow the compiler to report compile time errors when a program does something incompatible with type.**
  - The assembler however has no notion of types at all. It will let programmers do whatever they want to whatever address.

Symbol Table Example

- **The symbol table for our example in C will contain entries for chUpper and degreesK like:**

Name	address	type
chUpper	0008	char
degreesK	0005	int

- **Because `chUpper` has the type `char` while `degreesK` has the type `int` the compiler**
  - reserves one byte for `chUpper` but two for degreesK,
  - uses `CHARI` to read `chUpper` but `DECI` to read degreesK,
  - uses `STBYTA` to store results to `chUpper` but `STOREA` to store to degreesK

Erroneous Typing in Assembly

```
0000      70000B            BR      start       ; Branch around data.
            CtoK:       .EQUATE d#273       ; define CtoK 273
0003      0000    degreesC: .WORD   d#0         ; int degreesC;
0005      0000    degreesK: .WORD   d#0         ; int degreesK;
0007      00      chLower:  .BYTE   d#0         ; char ch_lower;
0008      00      chUpper:  .BYTE   d#0         ; char ch_Upper;
0009 0000   temp:           .WORD   d#0     ; int temp
                ;
000B E90003 start:        DECI      degreesC,d ;read in  degreesC
000E D90007               CHARI     chLower,d ;read in chLower
                ;
0011 090003               LOADA     degreesC,d ;degreesK = degreesC + CtoK
0014 180111               ADDA        CtoK,i
0017 110008               STOREA    chUpper    ;this overwrites the first byte
                                      ;of temp
001A 080061               LOADA     c#'a',i     ; temp = 'a - 'A'
001D 200041               SUBA        c#'A',i
0020 110009               STOREA    temp,d
                ;
0023 510007               LDBYTA    chLower,d  ;chUpper := chLower - temp
0026 210009               SUBA        temp,d
0029 590005               STBYTA    degreesK,d
                ;
002C F10005               DECO        degreesK,d  ;print out degreesK, " ", chUpper
002F E00020               CHARO     c#' ',i
0032 E10008               CHARO     chUpper,d
                ;
0035 00                 STOP
0036                      .END
```


Erroneous Typing in Assembly (2)

- **Performing a `STOREA` to the address associated with `chUpper` overwrites the first byte in the word labeled by `temp`.**
  - In this program, that doesn't have any bad consequences, since the program has not yet stored a value it cares about into `temp`
  - Had temp already contained something that would be needed again, though, storing a word to `chUpper` would have overwritten it.
- **The answers produced by this program are still wrong, of course. since**
  - we stored the character result into `degreesK` and the integer result into `chUpper`,
  - but left the output statements as "`DECO degreesK,d`" and "`CHARO chUpper,d`".

Erroneous Typing in Assembly (3)

- **Compare the output from the right and wrong programs in this run:**

```
20      20
 a       a
293 A  16640
```

- **The incorrect output value, d#16640 is the same h#4100, which makes sense since h#41 is the ASCII code for the letter "A"**
  - Which we stored into the first byte of degreesK.
  - Although you can't see it the program is also printing h#01 as its character output.
  - This is because h#01 is the first byte of the value, h#0125 (or d#293) stored in the word at degreesK.
- **Lack of typing in assembly means that errors which a compiler would have discovered at compile time become logical errors which the programmer has to debug.**

Questions?