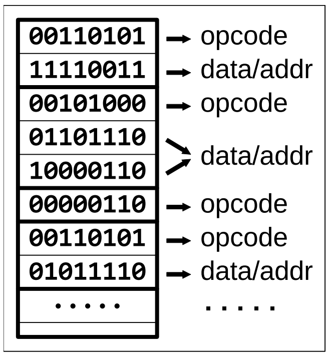
Lecture 3
Assembly Language

This lecture describes the machine instruction set, assembly language usage and the assembly process.

Examples given in the lecture do not use any particular assembly language or machine instruction set. Simple and common English language words are used to represent machine instructions of a generic microprocessor. This easy to understand language uses a few more letters but removes the burden of decyphering bunch of abbreviations. Assembly languages developed for real-world microprocessors have very similar operation names and syntax rules.

Mnemonic Code



Mnemonic = Memory aid

Memorizing hundreds or at least tens of binary numbers for all opcodes is not practical. Even if you can do so, most people cannot easily read a program written in numbers.

Microprocessor datasheets or programming manuals define easy to remember abbreviations for machine instructions. These abbreviations are referred to as **mnemonic code**.

How Many Operation Codes?

A microprocessor having hundreds of opcodes does not necessarily mean that you need to memorize the same number of mnemonic codes. Remember that an opcode specifies the method of accessing the data or address as well as the operation type:

$$\begin{tikzcd}
 & & \texttt{LOAD A,\#35;} & \text{immediate} \\
 & \texttt{LOAD A} \arrow[ur] \arrow[r] \arrow[dr] & \texttt{LOAD A,<addr>;} & \text{direct} \\
\texttt{LOAD} \arrow[ur] \arrow[ddr] & & \texttt{LOAD A,@XR;} & \text{indirect} \\
 & & \texttt{LOAD B,\#35;} & \text{immediate} \\
 & \texttt{LOAD B} \arrow[ur] \arrow[r] \arrow[dr] & \texttt{LOAD B,<addr>;} & \text{direct} \\
 & & \texttt{LOAD B,@XR;} & \text{indirect}
\end{tikzcd}$$

An assembler can resolve the numeric opcode not just by the operation type, but also the addressing mode you specify. So, the number of mnemonic codes you should remember is a fraction of the number of numeric codes.

Back to The μP Functions

- **Execute** a series of commands in memory.
- **Access memory** to move data.
- **Calculate** arithmetic and logic functions.
- **Compare:** Check for equality, less-than, greater-than.
- **Branch:** Make decisions and call subprograms.
- **Communicate** with peripheral devices.
- **Respond** to external events.

The first item in this list is nothing but what a μP does all the time. Other items require a group of instructions to perform the related tasks. The collection of all instructions form the **instruction set** of a μP.

Operation Types

Following are a typical list of mnemonic instruction codes:

Access Memory: `LOAD, STORE, MOVE`

Calculate: `ADD, SUB, MULT, AND, OR, XOR`

Branch: `JUMP, BRANCH, CALL, RETURN`

Compare + Branch: `JMPIZ, BRNIZ` (jump/branch if zero)
`JMPIN, BRNIN` (jump/branch if negative)
`JMPIP, BRNIP` (jump/branch if positive)

Communicate: `IN, OUT`

Respond: `ENINT, DISINT` (enable disable interrupts)

Some assembly languages may include a target register in the mnemonic code:

`LOAD -> LDA, LDB` or `MOVE -> MVA, MVB`

Some may include the access method as well:

`LDA -> LDAI` (load A immediate)

Assembly Language

Assembly language is the easy way of writing programs using machine instructions. At the minimum, assemly language helps us in two ways:

1. We can use mnemonic codes of instructions instead of numeric operation codes.

2. We can define alphanumeric symbols replacing memory addresses for branching targets (labels) and data locations (variables).

Why assembly language?

- Assembly language gives us direct access to machine instructions that we cannot use in high-level languages.
- Assembly language can be the only way to generate efficient code in terms of speed and memory usage.
- You may end up using assembly language just because you cannot afford to buy a high-level compiler like C.

Assembly to Machine Code

1) Find all opcodes, 2) Fill in immediate data and address bytes

```
; Modulo M/N
        ORG     0x1A00;
        LOAD    B, #0;
        LOAD    A, M;
Loop:   SUB     A, N;
        JMPIN   Done;
        ADD     B, #1;
        JUMP    Loop;
Done:   STORE   B, Qtn;
        ADD     A, N;
        STORE   A, Rmd;
        STOP;
        VAR     M;
        VAR     N;
        VAR     Qtn;
        VAR     Rmd;
```

	addr	data			
	1A00	37	??		→ Load B immediate #0
	1A02	5A	??	??	→ Load A <addr M>
Loop:	1A05	62	??	??	→ Sub A <addr N>
	1A08	A5	??	??	→ Jump if neg <addr
	1A0B	77	??		→ Add B immediate #1
	1A0D	A1	??	??	→ Jump <addr Loop>
Done:	1A10	16	??	??	→ Store B <addr Qtn>
	1A13	7E	??	??	→ Add A <addr N>
	1A16	15	??	??	→ Store A <addr Rmd>
	1A19	B0			→ Stop
	1A1A	XX			→ Storage for M
	1A1B	XX			→ Storage for N
	1A1C	XX			→ Storage for Qtn
	1A1D	XX			→ Storage for Rmd
	....	..	..	..	

How To Make Machine Code

- Go through all instructions and find opcodes and instruction lengths referring to the microprocessor data sheet.
- Knowing the length of your code, organize executable part of your program in the memory.
- Organize all the data space you need for constants and variables in the remaining memory section.
- Find the target memory addresses for all branch instructions (`JUMP`, `CALL`, ...) and insert them into the machine code.
- Find the target memory addresses for all data references (`LOAD`, `STORE`, ...) and insert them into the machine code.
- Now you are ready to debug.

Making Machine Code (1)

First Pass:

• Read through the assembly source code

• Find opcodes and instruction lengths

• Organize the executable part

• Organize the data space

Immediate data and backward address references can be filled in during the first pass.

Other addresses are not known until the entire executable code and data space are organized in the memory.

	addr	data			
	1A00	37	00		➞Load B immediate **#0**
	1A02	5A	??	??	➞Load A <addr **M**>
Loop:	1A05	62	??	??	➞Sub A <addr **N**>
	1A08	A5	??	??	➞Jump if neg <addr
	1A0B	77	01		➞Add B immediate **#1**
	1A0D	A1	1A	05	➞Jump <addr **Loop**>
Done:	1A10	16	??	??	➞Store B <addr **Qtn**>
	1A13	7E	??	??	➞Add A <addr **N**>
	1A16	15	??	??	➞Store A <addr **Rmd**>
	1A19	B0			➞Stop
	1A1A	XX			➞**Storage for M**
	1A1B	XX			➞**Storage for N**
	1A1C	XX			➞**Storage for Qtn**
	1A1D	XX			➞**Storage for Rmd**
	....	..	..	..	

Making Machine Code (2)

Second Pass:

• Read through the assembly source code one more time.

• Fill in the address references for branch instructions.

• Fill in the address references for the operands in the data space.

All addresses are known after the entire executable code and data space are organized in the memory.

	addr	data			
	1A00	37	00		➞Load B immediate **#0**
	1A02	5A	1A	1A	➞Load A <addr **M**>
Loop:	1A05	62	1A	1B	➞Sub A <addr **N**>
	1A08	A5	1A	10	➞Jump if neg <addr
	1A0B	77	01		➞Add B immediate **#1**
	1A0D	A1	1A	05	➞Jump <addr **Loop**>
Done:	1A10	16	1A	1C	➞Store B <addr **Qtn**>
	1A13	7E	1A	1B	➞Add A <addr **N**>
	1A16	15	1A	1D	➞Store A <addr **Rmd**>
	1A19	B0			➞Stop
	1A1A	XX			➞**Storage for M**
	1A1B	XX			➞**Storage for N**
	1A1C	XX			➞**Storage for Qtn**
	1A1D	XX			➞**Storage for Rmd**
	....	..	..	..	

Assembly: Parsing

Assembler reads through your program analyzing every opcode, constant, identifier, and syntax notation (**`: , ; # @`**) following the syntax rules. These are the assembler operations in the parsing step:

- Identify directives (**`ORG`**, **`VAR`**) that are non-executable statements.
- Identify mnemonic operation codes (**`LOAD`**, **`ADD`**, ...) and build a draft machine code without any address specifications.
- Identify constants (**`#0`**, **`#1`**).
- Identify all symbols, that are user-defined labels (Loop, Done) and memory references (**`M`**, **`N`**, **`Qtn`**, **`Rmd`**).

You get most of the syntax errors as a result of the code analysis in parsing step. If you forget a colon (**`:`**) after a label then the assembler tries to analyze it as an opcode. You are likely to get the message "Error - Unknown opcode: Loop".

Assembly: Symbol Table

Assembler makes a symbol table that contains all identifiers or the "symbols" you reference in the program. The first column of symbol table is a list of labels and memory references in the program. The address column of the table is left blank initially.

```
; Modulo M/N
        ORG     0x1A00;
        LOAD    B, #0;
        LOAD    A, M;
Loop:   SUB     A, N;
        JMPIN   Done;
        ADD     B, #1;
        JUMP    Loop;
Done:   STORE   B, Qtn;
        ADD     A, N;
        STORE   A, Rmd;
        STOP;
        VAR     M;
        VAR     N;
        VAR     Qtn;
        VAR     Rmd;
```

SYMBOL TABLE

symbol	addr
M	????
Loop	????
N	????
Done	????
Qtn	????
Rmd	????

Filling in The Symbol Table

Once the draft code is ready, assembler can find the addresses corrosponding to labels and variables defined in the program.

SYMBOL TABLE

symbol	addr
M	1A1A
Loop	1A05
N	1A1B
Done	1A10
Qtn	1A1C
Rmd	1A1D

	addr	data			
	1A00	37	00		→ Load B immediate **#0**
	1A02	5A	1A	1A	→ Load A <addr **M**>
Loop:	1A05	62	1A	1B	→ Sub A <addr **N**>
	1A08	A5	1A	10	→ Jump if neg <addr
	1A0B	77	01		→ Add B immediate **#1**
	1A0D	A1	1A	05	→ Jump <addr **Loop**>
Done:	1A10	16	1A	1C	→ Store B <addr **Qtn**>
	1A13	7E	1A	1B	→ Add A <addr **N**>
	1A16	15	1A	1D	→ Store A <addr **Rmd**>
	1A19	B0			→ Stop
	1A1A	XX			→ **Storage for M**
	1A1B	XX			→ **Storage for N**
	1A1C	XX			→ **Storage for Qtn**
	1A1D	XX			→ **Storage for Rmd**
	....	..	..	..	

Assembly: Making The Code

Assembler outputs produced until now:

- Prepared a draft machine code to find out the the address of machine instructions and data locations in the memory.
- Made a symbol table showing the address of all label and variable definitions.

In the final stage, the assembler reads through your program again, but this time it knows the address of every label and every variable in the symbol table. The assembler completes the draft machine code filling in the address field of machine instructions for all memory references in your program.

If the assembler cannot find the address of a label or variable in the symbol table, then it gives an error message. This means either you forgot to define an address or made a typo:

"Error - Undefined reference: Rmd"
"Error - Undefined reference: Loopp"

Absolute / Relative Addressing

Machine code example for addressing relative to PC:

	Absolute Addressing				Relative Addressing				
	addr	data			addr	data			
	1A00	37	00		1A00	37	00		→ Load B immediate **#0**
	1A02	5A	1A	1A	1A02	5B	11		→ Load A <rel addr **M**>
Loop:	1A05	62	1A	1B	1A04	63	10		→ Sub A <rel addr **N**>
	1A08	A5	1A	10	1A06	A6	06		→ Jump if neg <rel addr **Done**>
	1A0B	77	01		1A08	77	01		→ Add B immediate **#1**
	1A0D	A1	1A	05	1A0A	A2	FA		→ Jump <rel addr **Loop**>
Done:	1A10	16	1A	1C	1A0C	17	09		→ Store B <rel addr **Qtn**>
	1A13	7E	1A	1B	1A0E	7F	06		→ Add A <rel addr **N**>
	1A16	15	1A	1D	1A10	16	06		→ Store A <rel addr
	1A19	B0			1A12	B0			→ Stop
	1A1A	XX			1A13	XX			→ **Storage for** M
	1A1B	XX			1A14	XX			→ **Storage for** N
	1A1C	XX			1A15	XX			→ **Storage for** Qtn
	1A1D	XX			1A16	XX			→ **Storage for** Rmd
	....	..	..	..	....	..	..	..	

Assembler Directives

Assembler directives are the messages you place in your program to tell the assembler about the code generation options you need. These directives do not correspond to any machine instructions. So far, we have seen two directives:

- **ORG** directive tells assembler where to place your program in the memory.
- **VAR** directive reserves a memory location for data storage.

Every assembler program has its own set of directives and syntax rules. MPASM assembler for PIC processsors supports **ORG** directive in absolute code, but requires **CODE** directive in relocatable code for the same purpose. Similarly, **RES** directive replaces **VAR** to reserve memory.

A frequently used assembler/compiler directive is the "**#include <filename>**" directive that allows several source code files share the common definitions in a single header file.

Conditional Assembly

You can select alternate program segments using conditional assembly directives. Conditional assembly is useful in case you need similar versions of a program with some small differences. You can include optional code segments in a single file, so that you don't had to maintain bulk of the program in two different files.

```
#define UseOption1
...
#ifdef UseOption1
  <code option 1>
#else
  <code option 2>
#endif
```

Assembler generates `<code option 1>` and ignores `<code option 2>` when the directive, "`#define UseOption1`", is included at the beginning. If you remove the line, "`#define UseOption1`", then `<code option 2>` will be used.

You can use the same "`#ifdef UseOption1 ... #else ...`" structure as many times as you need to select several alternate code segments in your program.

Conditional Assembly Example

You wrote 2000 lines of assembly code for the first revision of an MCU. You need to modify your code for a second MCU revision that uses Timer2 instead of Timer1 and PortB instead of PortA.

```
#define MCUrev2
...
...
#ifdef MCUrev2
  <code using Timer2>
#else
  <code using Timer1>
#endif
...
...
#ifdef MCUrev2
  <code using PortB>
#else
  <code using PortA>
#endif
```

In this example, the assembler generates executable code for the parts that use **Timer2** and **PortB**, and ignores the code written for **Timer1** and **PortA**.

If you remove or comment-out the line, "`#define MCUrev2`", then the code written for **Timer1** and **PortA** will be used.

A single source code can be used for the two MCU versions by changing a single line. Otherwise, you will need to maintain duplicate copies of the 2000-line source code for all corrections and updates.

Macro Assembler

The macro utility of assembler allows us to duplicate code without actually repeating code. Assembler first **expands** the macro calls and then starts the assembly process. When a macro is expanded, macro statements are inserted into the program replacing macro arguments with the parameters listed in the macro call.

```
; Define the macro:
AddUp macro Sum, N1, N2
LOAD  A, N1;
ADD   A, N2;
STORE A, Sum;
endmacro
...
; Use the macro
AddUp SumX, X1, X2;
AddUp SumY, Y1, Y2;
AddUp SumZ, Z1, Z2;
```

```
;After macro expansion:
...
; Use the macro
LOAD  A, X1;
ADD   A, X2;
STORE A, SumX;
LOAD  A, Y1;
ADD   A, Y2;
STORE A, SumY;
LOAD  A, Z1;
ADD   A, Z2;
STORE A, SumZ;
```

Macros in C Language

To define a macro that uses arguments, specify parameters between parentheses in the macro definition. Following is a macro example that computes the minimum of two values:

```
// Define the macro:
#define min(X, Y)  ((X) < (Y) ? (X) : (Y))
```

This macro can be invoked (called) as follows:

```
Nmin = min(A, B);
Mmin = min(C + D, E);
```

Macro expansion produces the code given below:

```
Nmin = ((A) < (B) ? (A) : (B));
Mmin = ((C + D) < (E) ? (C + D) : (E));
```

The parantheses used before and after the parameters in the macro definition (i.e. `(X)` and `(Y)`) may seem redundant, but they are required to avoid problems resulting from **operator precedence** in C language.

Code Portability

Assembler or compiler directives help us write portable code that can be used with different development environments. The following example defines Dynamic Link Library (DLL) function declerations for three C compilers. You can change a single line instead of all function declerations to switch between compilers.

```
// Use following definition for Watcom C compiler:
#define Fexport  __export __cdecl
// Use following definition for LabWindows CVI compiler:
//#define Fexport  __stdcall
// Use following definition for Microsoft C compiler:
//#define Fexport  __declspec(dllexport)
.....
// Use Fexport in all DLL export function declerations:
void Fexport SetDefaultPath(char *Path);
void Fexport GetGUImsg(char *Message);
.....
```

Similar portability problems may exist between the hardware revisions of a microprocessor, or between the software versions of an assembler or compiler.

Macros Versus Subprograms

- Expansion of macro calls is completed during assembly or compilation process. Subprogram calls are executed when the processor runs the machine code
- A subprogram has only one copy in the program memory. Macro code is duplicated when every macro call is expanded just as you copy-paste the same statements in your program.
- A subprogram can declare its own variables and it can store private data. A macro must use the variables that are declared in the program module where the macro call occurs.
- Subprogram calls can be arranged dynamically during the program execution. Macro calls cannot be changed after assembly or compilation.

When to Use Macros

Every subprogram call requires some overhead processing during execution

```
Pass parameters (store them in a reserved area)
Save the return PC address (part of CALL instruction)
Jump to target address (first instruction of subprog.)
Save registers if necessary
- - -
<do the actual task>
- - -
Pass the return value (if any)
Retrieve the saved register contents
Retrieve the saved PC address (RETURN instruction)
```

Macro calls have no overhead processing, but expanding long macros use up code memory. If you need to make a choice, you should consider the length of code that performs the actual task and efficiency of the microprocessor in handling subprogram calls.

Absolute or Relocatable Code

If you are writing **absolute code** you have complete control over the memory organization for your executable machine code and data space. As usual, this control power comes with responsibility. Organizing several complex program modules can be a difficult job.

You can avoid this difficult job developing relocatable code:

- Assembler generates **object code** instead of absolute machine code. Object code address fields are kept as offset values that will be added to an absolute starting address.
- Another program, **linker**, takes the object codes of your program modules and positions them in the available memory according to code length and data space requirements. Once the linker decides on starting address of a module, it can fill in the absolute address fields and generate the executable machine code. Your responsibility is to tell the linker about address range of the memory available to the microprocessor.

Code Example: Expression

```
SUM = K + (M - N);
```

Assembly code:

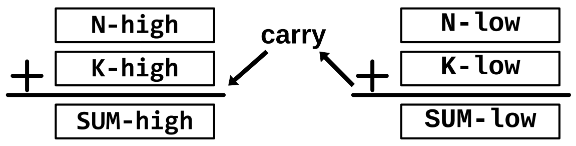
```
LOAD   A, M;    Acc = M
SUB    A, N;    Acc = M-N
ADD    A, K;    Acc = (M-N)+K
STORE  A, SUM;
```

Problem: What if your data is bigger than the accumulator or ALU size? For example, you need to work on 16-bit numbers when you have an 8-bit accumulator / ALU.

Data Bigger Than Accumulator

```
N += k;  // 16-bit addition
```

Break each 16-bit number into two 8-bit numbers:



Assembly code:

```
LOAD   A, Nlow;   load low byte
ADD    A, Klow;   carry flag updated
STORE  A, Nlow;   store low byte
LOAD   A, Nhigh;  load high byte
ADDwC  A, Khigh;  ADD-with-CARRY: include
                  carry flag from low byte
STORE  A, Nhigh;  store high byte
```

A high-level compiler generates this code automatically, but you should be prepared for a longer and slower machine code.

Example: Conditional Statement

```
if  ( M > N )
{  /* code executed for TRUE */  }
else
{  /* code executed for FALSE */  }
```

Assembly code:

```
       LOAD    A, M;   Acc = M
       SUB     A, N;   Acc = M-N, flags set
       JUMPiP  MGTN;   jump if positive
       ......
       ; code executed for FALSE
       ......
       JUMP    CONT;   unconditional jump
MGTN:  ......
       ; code executed for TRUE
       ......
CONT:  ......
```

Integrated Development Environment

An Integrated Development Environment or IDE provides access to several development tools from a single source:

- **Text editor** to write your programs. An IDE can have powerful search tools that allow you to jump specific definitions and references in your program.
- Access to device-specific **configuration and header files** that give you an easy head start.
- **Compiler/Assembler** execution and configuration.
- **Linker** execution and configuration.
- **Programmer** to download machine code into a test hardware.
- **Debugger** to analyze and debug your code.
- **File management utility** to organize your files.

You will need to learn an IDE interface whether you are developing a firmware for a microcontroller, a Windows application in C, or a Verilog code for an ASIC or FPGA.

File Management In IDE

If you will end up writing 5000 lines of code you don't want to scroll up and down through a single big file. Furthermore, other engineers may need to work on the same project which is not possible if everything is in a single file. File management utility of an IDE help you organize the files that are parts of a project:

- **Source files** of the program modules developed by you and others working on the project. You need to tell the assembler or compiler where to find all necessary components of the project. The file management utility makes it easy.
- **Include files** or **header files** shared between the program modules.
- **Configuration files** for assembler/compiler, linker, and possibly other tools available in the IDE.
- **Output files** generated by assembler and linker.