CPG 5 Coding and Programming

Introduction to Game Programming

Paul Sinnett, London South Bank University <paul.sinnett@gmail.com>

Table of Contents

Week 7

The Assembler

This week's subject areas are:

- assembly language
- mnemonics
- directives
- two pass assembly
- macros, comments, and expressions

Recap

We've seen how electrical circuits can be made to encode logical rules, store numbers, pictures, and audio, etc.. And we've seen how to construct loops of such circuits to evaluate the rules of a game in regular intervals.

Can you think of any game rule that could not be encoded in this way?

Last week we looked at the impracticalities of building a game from such circuits. And we designed a single programmable circuit (the ALU) capable of evaluating any single simple operation on 1 or 2 inputs. By constructing a circuit to cycle through instructions (machine code) in the computer's memory to automatically program the ALU we could duplicate the output of any hardware circuit.

Can you think of any circuit that could not be duplicated like this?

As an exercise I asked you to design a machine code program to duplicate the behaviour of any circuit you designed in a previous week. In performing that task you probably noticed some of the difficulties inherent in machine code programming languages:

- laborious to construct
- difficult to read
- hard to modify

These difficulties are what prompted the development of the second generation of programming language, known as assembly language.

Assembly language

An assembler is a program that automates the production of machine code. Typically you supply the assembler program with one or more text files of assembly source code and the program converts each one into a machine code file known as an object. Finally, the objects are linked together to create the an executable program.

Figure 1. Pong code as assembler

This is what an assembly language for our simple 16 bit processor might look like. The naming conventions and mnemonics are based on the x86 processor assembly code.

```
     screenWidth  = 640           ; these symbols are simply replaced by the
     screenHeight = 480           ; assembler in the code

.data
     ballPositionX word 320       ; these values are allocated addresses in RAM
     ballPositionY word 240       ; and initialised to the values given here
     ballVelocityX word 0         ; the assembler adds instructions
     ballVelocityY word 1         ; to do this before the program starts


.code
updateBallPositionY:              ; ballPositionY += ballVelocityY
     lea  ax, ballVelocityY       ; load address of "ballVelocityY" to "a"
     mov  dx, [ax]                ; move value at address "a" to "d"
     lea  ax, ballPositionY       ; load address of "ballPositionY" to "a"
     add  [ax], dx                ; add value "d" to the value at address "a"

ballOffTop:                       ; (ballPositionY < 0)
     lea  ax, ballPositionY       ; load address of "ballPositionY" to "a"
     mov  dx, [ax]                ; move value at address "a" to "d"
     lea  ax, invertBallVelocity  ; load address "invertBallVelocity" to "a"
     jl   ax                      ; jump to address "a" if "d" is > 0

ballOffBottom:                    ; (ballPositionY - screenHeight >= 0)
     lea  ax, ballPositionY       ; load address "ballPositionY" to "a"
     mov  dx, [ax]                ; move value at address "a" to "d"
     mov  ax, screenHeight        ; move constant value "screenHeight" to "a"
     sub  dx, ax                  ; subtract value of "a" from "d"
     lea  ax, endOfLoop           ; load address of "endOfLoop" to "a"
     jl   ax                      ; jump to "a" if "d" is < 0

invertBallVelocityY:              ; ballVelocityY = -ballVelocityY
     lea  ax, ballVelocityY       ; load address "ballVelocityY" to "a"
     neg  [ax]                    ; negate the value at address "a"

endOfLoop:
     lea  ax, updateBallPositionY ; load address "updateBallPositionY" to "a"
     jmp  ax                      ; jump unconditionally to address "a"
```

Compare the example assembly language program to the machine code version from last week. What are the similarities? What are the differences?

Mnemonics

Notice that the assembly language doesn't use numerical codes for instructions, instead it uses small abbreviated identifiers known as mnemonics.

When you were creating the machine language instructions last week you probably noticed that many of the instruction codes were doing variations of the same thing. For example, you might be adding the A register to the D register, or adding the D register to memory at the address of the A register, etc.. The mnemonic groupings recognise all of these codes as set with the same identifier: in this case, the `add` instruction.

Here's a table of all the instructions:

Table 1. Assembly language mnemonics

Mnemonic	Meaning
`lea ax, expr`	load into "a" register, the effective address of the expression
`mov reg, expr`	copy to register the value of the given expression • any 16 bit integer value can be copied to "a" • "d" and "[a]" can only be assigned 1, 0, or -1
`mov reg, reg`	copy to the first register the value of the other
`add reg, reg`	add to the first register, the value of the second
`inc reg`	increment the register
`and reg, reg`	bitwise AND into the first register, the value of the second
`or reg, reg`	bitwise OR into the first register, the value of the second
`sub reg, reg`	subtract from the first register, the value of the second
`dec reg`	decrement the register
`neg reg`	negate the register
`not reg`	apply a bitwise NOT to the register
`jmp ax`	jump to the code address given by the value of register "a"
`je ax`	jump to "a" if the value of the "d" register is 0
`jne ax`	jump to "a" if the value of the "d" register is not 0
`jg ax`	jump to "a" if the value of the "d" register is > 0
`jl ax`	jump to "a" if the value of the "d" register is < 0
`jge ax`	jump to "a" if the value of the "d" register is >= 0
`jle ax`	jump to "a" if the value of the "d" register is <= 0

Instructions can apply to 1 or more registers listed after the mnemonic. These are known as the operands of the instruction. The convention for instructions with two operands is to list the destination register first. Registers are identified as follows:

Table 2. Assembly language operands

Operand	Meaning
`ax`	The address register
`dx`	The data register
`[ax]`	The register in memory at the address in register "a"
`number`	The literal value of the number
`expression`	The literal value of the expression or its effective address

Note

The convention to append an "x" after register names comes from x86 assembly language. Because the x86 language was first designed for 8 bit machines, the 16 bit versions adopted a convention to remain backwardly compatible with the earlier processors. A register appended with an "l" identified the lower 8 bits of the 16 bit register. One appended with an "h" identified the top bits. "x" was used to operate on the whole 16 bit value at once. Since our processor only operates on 16 bit values, I've adopted the "x" naming throughout.

Directives

In addition to a list of instructions for the computer, the assembler also accepts instructions to control its own behaviour. These instructions are known as assembler directives and are distinguished in the example above by being prefixed with a dot.

The two directives in the example (`.data` and `.code`) tell the assembler what follows the directive.

In the data segment, each identifier you provide is allocated an address in RAM such that the areas do not overlap. Usually this will start at some address and work down.

In the code segment, identifiers from the data section can be used as effective addresses to point to the actual values in RAM.

Two pass assembly

As you will have experienced while attempting to write machine code by hand. Sometimes it is not possible to fill in an address until for an instruction until you have written more of the program.

Assemblers resolve this problem by making two passes through your assembly language program.

On the first pass through, no code is generated. The assembler simply looks through the code for identifiers that it might need in the following pass. When the first pass is complete, it should have a final address in ROM or RAM for every identifier in your program.

The second pass through actually generates the machine code instructions. If the instruction refers to an identifier, the assembler can look up the address it generated in the first pass.

Macros, comments, and expressions

As assemblers and assembly languages evolved they added more features to make programming easier. Macros are simple text replacement definitions. In the code example, the identifiers `screenWidth` and `screenHeight` are macro definitions. After they have been defined, when the assembler program encounters the macro label it replaces it with the definition text. For example, when the assembler gets to the instruction `mov ax, screenHeight` it replaces it with `mov ax, 480`.

Tip

As your programs grow in size, you'll find that by defining your program constants as macros you'll make them easier and safer to modify. You'll also find that this helps to document your programs and reduce the need for comments.

Comments begin with a semi-colon. Whenever the assembler program encounters a semi-colon, it ignores the remainder of the line. You can use this to comment each line of assembly code or create a description block to be followed by a raw list of instructions.

Finally, most assemblers can evaluate simple expressions as they process your source files. This allows you to specify values in your program in terms of what they represent rather than their result.

Example 1. Using expression evaluation to help document a program

Suppose you wanted to write a loop to clear every pixel of the screen.

To do this you'd need to cycle through every pixel in the screen's memory. How many pixels is that? If the screen is 640 pixels by 480 pixels, the number of pixels you'd need to clear would be 640 times 480, that is 307,200. And if we were representing 1 bit per pixel that would require clearing 307,200 / 16 registers, that is 19,200.

You could put this value into the "a" register with the instruction:

```
mov ax, 19200             ; 16 bit registers for a screen 640 x 480
```

A better way to write this is:

```
mov ax, (640 * 480) / 16  ; 16 bit registers for a screen 640 x 480
```

Better:

```
mov ax, (screenWidth * screenHeight) / bitsPerPixel ; screen size
```

Better still:

```
screenSize = (screenWidth * screenHeight) / bitsPerPixel
mov ax, screenSize
```

Notice that the best way to document a program is to find a way to eliminate the need for a comment. If you feel the need to add a comment to a program in any language, first add the comment. Then ask yourself if it's possible to rewrite the code in such a way as to make the comment redundant. This practice is known as self-documenting code.

Exercise

For this week's exercise I'd like you to convert your machine code program from last week into an assembly language program.

Again, you can follow my example above as a guide.

Write out your assembler program code as in the example above in a plain text file. Submit your files as usual to source control before Sunday. And don't forget to include your work log with your submission.