PIPPIN MACHINE LANGUAGE AND ASSEMBLY

Previous Page

Basic Terms

Hypothetical Computer	In the study of computer architecture - we must create for the sake of learning - a hypothetical computer (a make believe computer). This hypothetical computer allows us to simplify the complexities of the real computer architecture so that we can learn the important basic theories behind the design of computers.
Operation Code	This is the code (binary code) that the machine can recognize for each instruction to carry out. Also referred to as the "*Opcode*". With the PIPPIN - the opcode is 8 bits long. The actual opcode looks like this 01000111 - this is machine language.
Accumulator	This is a special location (called a register) for storing intermediate results from the execution of the opcode. Often this register is used in the performance of the opcode - therefore often it must first be loaded with a value or afterward the resulting value must be store in memory.
Assembler	First program translator - it translates the mnemonic coding used into machine language. The assembler takes the source code (assembly language program) and outputs the "object code" or machine language program. This language is only preferred if - the resulting machine code must extra efficient and extra compact...otherwise it takes much more time to create than other languages.
Object Code	A translator such as a compiler or assembler outputs an object code file. This file is one step away from being actually executed by a computer. For a computer to execute it - it must load it into memory - this means "fixing" all of the symbolic variable addresses to actual physical memory addresses. This allows programs to be placed into memory where ever the computer has space.
Symbolic References	Instead of having the programmer select references to actual memory locations - the assembler allows the programmer to access "symbolic references". Symbolic references are much easier to work with and are necessary for the flexible loading of programs mentioned above (see object code). Also the assembler allows "*Statement Labels*" - the purpose of these are for the same reason. However in this case - the programmer is allow to refer to a label instead of a physical location in memory. (Note: The programmer is not able to know where the program will eventually be loaded into memory.
Machine Language (code)	Like all other information (number and characters) - the computer only can work with binary - ones and zeros. The actual instructions that the computer understands is machine language - it looks like this 01010110110010010011001 1001 (this could be 2 different commands that the computer understands and can execute)
Loader	See above: Object Code. This is a piece of software that places the machine code (object file) into memory.
High-level Language	These are commands that are easily understood by programmers (but never understood by machines). They look like this "IF (X>34) THEN..." There are hundreds of languages used by programmers - however the most popular languages today are C++ and Java.
Translator	A program that translates the "high-level code" to machine language. This process is necessary - because computers can only read binary numbers - they can't read high-level "english-like" statements.
Interpreter	A type of *translator* - that translates the "high-level coded program " to machine language. This program translates a line of source code into one or more lines of object code and then instructs the computer to perform those instructions until another line has to be translated. These programs execute slower - however they are much easier to "debug" because the interpreter will stop on the line that is causing execution errors. Example languages: Javascript, BASIC, LISP (other scripting languages like Perl, TCL/TK, PHP, ASP)
Compiler	A type of *translator* - that translates the "high-level coded program " to machine language. This program translates the high-level source code once and for all, producing a complete machine language program. These are executable files (extension EXE). These programs execute faster - but they tend to b more difficult to debug. Example languages: C, C++, Fortran, COBOL
Virtual Machine (VM)	A programmer can write programmer - as if the high-level language is a computer's machine language. That computer is referred to as a Virtual Machine - again not a real machine, but a "make believe machine". This allows the programmer to forget about the internal workings of the computer (binary coded instructions) and concentrate on the high-level language. The most famous virtual machine is the Java Virtual Machine (JVM).
	4 Major Language Groups Programming Language Examples
Imperative	fundamental unit of abstraction is a procedure (C, Pascal, Fortran, Ada)
Functional	everything is defined as a function which has certain parameters and returns a value. (LISP)
Declarative	emphasis is on describing the information being processed by a program (COBOL, Prolog)
Object-Oriented	organized in terms of "objects" which have certain properties and abilities. (C++, Java, Smalltalk)

The PIPPIN assembly language is a simple language for programming a simple machine which has 256 bytes of memory, labelled byte 0 to byte 255. These bytes are divided into three categories:

Instructions (opcodes)

Operands

Variables (data)

The first 128 bytes are used for the "program space" of a PIPPIN program. These are where the statements for the programs we write will be found. Each statement requires two bytes. The first byte is for the actual instruction to be performed, for example `ADD`. Note that each instruction is represented by an 8-bit (1-byte) opcode. This means that the PIPPIN machine can have up to 256 different instructions. In reality, the machine has far fewer, but the choice of 1-byte opcodes allows that many. The second byte describes an operand which the instruction will need to perform its task, for example "add *what*?" In PIPPIN, the operand will either be an actual value (like `42`) or it will be an address for a memory location in the "variable space" of the machine (like the location of variable `X`).

The next 128 bytes are reserved for the "variable space" which will be used by the program to hold all of its data. All of the values to be processed will be stored in these locations, which can be referred to by names.

```
                                1 1 1 1 1 1 1 1 1 1    2
                          1 1   2 2 2 2 3 3 3 3 3 3    5
0 1 2 3 4 5 6 7 8 9 0 1 ... 6 7 8 9 0 1 2 3 4 5 ... 5
                        ...                       ...
Program space                   Variable space
```

It may be more useful to take the linear array of computer memory locations and organize them in a table that shows their relationships more clearly:

Contents	Memory Location	OPCODE/ DATA	OPERAND
STMT[1]	byte 0 & byte 1		
STMT[2]	byte 2 & byte 3		
. . .	. . .		
STMT[64]	byte 126 & byte 127		
VAR W	byte 128		
VAR X	byte 129		
VAR Y	byte 130		
VAR Z	byte 131		
VAR T1	byte 132		
VAR T2	byte 133		
VAR T3	byte 134		
VAR T4	byte 135		
. . .	. . .		
128th VAR	byte 255		

Finally, there is a very special byte of memory found on the CPU itself, not in the midst of the other memory bytes. This byte is referred to as the ACCUMULATOR and is the place where CPU results will be stored.

ACCUMULATOR

PIPPIN statements

Conventions used: **[n]** means "value stored in memory location n." However, **n** means "the literal value of n." **ACC** refers to the accumulator register.

1. Data flow:

```
LOD   n       copies value from location n into ACC.          ACC = [n]
LOD  #n       places the literal number n into ACC.           ACC = n
STO   n       copies the value from ACC into location n.      [n] = ACC
```

2. Control structures:

```
JMP   n       continue with instruction at byte n.
JMZ   n       if (ACC == 0) continue with instruction at byte n.
NOP           do nothing.
HLT           stop running the program.
```

3. Arithmetic-Logic:

```
ADD   n (or #n)     ACC += [n]  (or ACC += n); error halt on overflow or underflow
SUB   n (or #n)     ACC -= [n]; error halt on overflow or underflow
MUL   n (or #n)     ACC *= [n]; error halt on overflow or underflow
DIV   n (or #n)     ACC /= [n]; integer division no remainder;
                                error halt on divide by zero; overflow or underflow
AND   n (or #n)     if (ACC != 0) && ([n] != 0) ACC = 1 else ACC = 0
NOT                 if (ACC == 0) ACC = 1 else ACC = 0
CPZ n               if ([n] == 0) ACC = 1 else ACC = 0
CPL n               if ([n] < 0)  ACC = 1 else ACC = 0
```

What people do

We can use this simple language, and a dash of creativity, to solve many problems that the system is not explicitly designed to solve. For example, suppose we need to determine whether a number is even or not. There is no built in facility to do this, but we can create a solution by following the algorithm:

algorithm even(X):

```
LOD X

DIV #2  ; note that even numbers are divided exactly, odd numbers are not
MUL #2  ; if X was even, then the ACC holds the same value as X.  If it was
STO T1  ; odd, then the answer is 1 less than the original value
SUB X   ; if the number was even, the ACC holds 0; otherwise it is negative
STO T1
CPL T1  ; if the number was odd, the ACC is 1; otherwise it is 0
NOT     ; reverse the ACC.  If the number was odd, the ACC is 0; otherwise it is 1
```

Note that there are many ways to solve this problem, all of which are correct but some of which are more "elegant" in that they require much less work in the computer. For example, the same result is obtained from the following algorithm.

algorithm betterEven(X):

```
LOD X   ; solution by Nathan A Walker, CS111, Fall 2003
DIV #2
MUL #2
SUB X   ; if the number was even, this leaves 0 in ACC, otherwise -1
ADD #1  ; if the number was even, this leaves 1 in ACC, otherwise 0
```

What machines do

"Mindlessly" following an algorithm with no element of creativity...

For example, define the method to determine odd(X) as

algorithm odd(X):

code for even(X) NOT

Note that this is could be inefficient. The result of mindlessly following translation rules is that we will not find clever shortcuts that can make the code more efficient. This stage of translation is only interested in the generation of correct code. An optimizing compiler would next try to make the code more efficient. We will not discuss optimization further in this course.

Translation of a JavaScript-like language into PIPPIN

Expressions

Note: for simplicity's sake, this version of a JavaScript-like language will not include algebraic precedence rules. All expressions will be parenthesized to show explicit precedence. Within parentheses, operations are assumed to occur from left to right.

1. Arithmetic expressions (results in ACC)

a. number

```
LOD #n
```

---

b. variable

```
LOD var
```

---

c. (expr)

```
code for expr
```

---

d. -expr

```
code for expr
MUL #-1
```

---

e. $expr_1$ + $expr_2$

```
code for expr2
STO T1
code for expr1
ADD T1
```

---

f. $expr_1$ - $expr_2$

```
code for expr2
STO T1
code for expr1
SUB T1
```

---

g. $expr_1$ * $expr_2$

```
code for expr2
STO T1
code for expr1
MUL T1
```

---

h. Math.floor($expr_1$ / $expr_2$)

```
     code for expr2
     STO T1
     code for expr1
     DIV T1          ; PIPPIN only supports integer division
```

(**Math.floor()** - a method that returns the largest whole number
less than or equal to the specified number.)

---

i. $expr_1$ % $expr_2$

```
     code for expr2
     STO T2          ; T2 = expr2
     code for expr1
     STO T1          ; T1 = expr1
     DIV T2          ; ACC = quotient
     MUL T2          ; ACC = quotient * expr2
     STO T2
     LOD T1
     SUB T2
```

2. Boolean expressions (results in ACC; 1 = true, 0 = false)

a. Compute a "less-than" relation

($expr_1$ < $expr_2$)

translates to

```
     code for expr2
     STO T1
     code for expr1
     SUB T1              ; ACC = expr1 - expr2
     STO T1              ; T1  = expr1 - expr2
     CPL T1
```

---

b. Compute a "less-than-or-equal-to" relation

($expr_1$ <= $expr_2$)

translates to

```
           code for expr2
           STO T1
           code for expr1
           SUB T1
           STO T1            ; T1 = expr1 - expr2
           CPZ T1            ; ACC = 1, if expr1==expr2;
otherwise ACC = 0
           JMZ notEq         ; 0 in ACC means (expr != 0)
           JMP done          ; this instruction executes when (expr == 0)
    notEq: CPL T1            ; now test to see if expr1 <
expr2
     done: NOP
```

---

c. Compute a "greater-than" relation

(expr$_1$ > expr$_2$)

translates to

code for (expr$_2$ < expr$_1$)

---

d. Compute a "greater-than-or-equal-to" relation

(expr$_1$ >= expr$_2$)

translates to

code for (expr$_2$ <= expr$_1$)

---

e. Compute an "equal-to" relation

(expr$_1$ == expr$_2$)

translates to

```
code for expr2
STO T1
code for expr1
SUB T1
STO T1
CPZ T1
```

---

f. Compute a "not-equal-to" relation

(expr$_1$!= expr$_2$) <==> !(expr$_1$ == expr$_2$)

translates to

```
code for (expr1 == expr2)
NOT
```

---

g. True

```
LOD #1
```

---

h. False

```
LOD #0
```

---

i. the "NOT-operator"

!boolExpr

translates to

```
code for boolExpr
NOT
```

---

j. the "AND-operator"

$boolExpr_1$ && $boolExpr_2$

translates to

```
code for boolExpr1
STO T1
code for boolExpr2
AND T1
```

---

k. the "OR-operator"

$boolExpr_1$ || $boolExpr_2$

The OR operator must be constructed from the available AND and NOT operators. Following the rules of Boolean algebra, $boolExpr_1$ OR $boolExpr_2$ is equivalent to NOT (NOT $boolExpr_1$ AND NOT $boolExpr_2$) as demonstrated in the truth table:

b1	b2	b1 OR b2	!b1	!b2	!b1 AND !b2	!(!b1 AND !b2)
T	T	T	F	F	F	T
T	F	T	F	T	F	T
F	T	T	T	F	F	T
F	F	F	T	T	T	F

Thus, $boolExpr_1$ || $boolExpr_2$ is equivalent to

!((!$boolExpr_1$) && (!$boolExpr_2$))

which translates to

```
code for !boolExpr1
STO T1
code for !boolExpr2
AND T1
NOT                 ; this follows from def'n of NOT and AND above
```

Statements

Statements are executed sequentially unless order is interrupted by a control structure statement.

1. Compound statements

```
{
   stmt-1;
   stmt-2;
     ...
   stmt-n
}
```

translates to

```
code for stmt-1
code for stmt-2
...
code for stmt-n
```

2. Assignment statements

a. var = expr

translates to

```
code for expr
STO var
```

---

b. var++

translates to

```
LOD #1
ADD var
STO var
```

---

c. var--

translates to

```
LOD #-1
ADD var
STO var
```

---

d. var += expr

translates to

```
code for expr
ADD var
STO var
```

---

e. var -= expr

translates to

```
code for expr
MUL #-1
ADD var
STO var
```

---

f. var *= expr

translates to

```
code for expr
MUL var
STO var
```

---

g. var /= expr // reminder - this is integer result division

translates to

```
     code for expr
     STO T1
     LOD var
     DIV T1
     STO var
```

---

h. var %= expr

translates to

```
     code for expr
     STO T1
     LOD var
     DIV T1
     MUL var
     STO T1
     LOD var
     SUB T1
```

3. Control structure statements

a. the "while-statement"

```
     while (boolExpr) stmt
```

translates to

```
     loop: code for boolExpr      ; result in ACC:  1=true, 0=false
           JMZ end
           code for stmt
           JMP loop
     end: NOP
```

---

b. the "if-statement"

```
     if (boolExpr) stmt
```

translates to

```
           code for boolExpr      ; result in ACC:  1=true, 0=false
           JMZ end
           code for stmt
     end: NOP
```

---

c. the "if-else-statement"

```
     if (boolExpr) stmt1 else stmt2
```

translates to

```
           code for boolExpr      ; result in ACC:  1=true, 0=false
           JMZ s2
           code for stmt1
           JMP end
      s2: code for stmt2
     end: NOP
```

---

d. the "for-statement"

```
     for (var = expr; boolExpr; assignment) stmt
```

translates to

```
          code for var = expr
    loop: code for boolExpr          ; result in ACC:  1=true, 0=false
          JMZ end
          code for stmt
          code for assignment
          JMP loop
    end:  NOP
```

Example of translation from a high-level statement to PIPPIN

EXAMPLE 1

```
          if ((x + y) > z) y = x * 2
```

translates to

```
          code for (x + y) > z    ; code for if-statement
          JMZ end
          code for y = x * 2
     end: NOP
```

which further translates to

```
          code for z < (x + y)    ; code for greater-than relation
          JMZ end
          code for y = x * 2
     end: NOP
```

which further translates to

```
          code for x + y          ; code for less-than relation
          STO T1
          code for z
          SUB T1
          STO T1
          CPL T1
          JMZ end
          code for y = x * 2
     end: NOP
```

which further translates to

```
          code for y              ; code for expr1 + expr2
          STO T1
          code for x
          ADD T1
          STO T1
          code for z
          SUB T1
          STO T1
          CPL T1
          JMZ end
          code for y = x * 2
```

```
     end: NOP
```

which further translates to

```
          LOD Y                  ; code for variable, y
          STO T1
          code for x
          ADD T1
          STO T1
          code for z
          SUB T1
          STO T1
          CPL T1
          JMZ end
          code for y = x * 2
     end: NOP
```

which further translates to

```
          LOD Y
          STO T1
          LOD X                  ; code for variable, x
          ADD T1
          STO T1
          code for z
          SUB T1
          STO T1
          CPL T1
          JMZ end
          code for y = x * 2
     end: NOP
```

which further translates to

```
          LOD Y
          STO T1
          LOD X
          ADD T1
          STO T1
          LOD Z                  ; code for variable, z
          SUB T1
          STO T1
          CPL T1
          JMZ end
          code for y = x * 2
     end: NOP
```

which further translates to

```
          LOD Y
          STO T1
          LOD X
          ADD T1
          STO T1
          LOD Z
          SUB T1
          STO T1
          CPL T1
          JMZ end
          code for x * 2         ; code for var =
expr
          STO Y
     end: NOP
```

which further translates to

```
          LOD Y
          STO T1
          LOD X
          ADD T1
          STO T1
          LOD Z
          SUB T1
          STO T1
          CPL T1
          JMZ end
          code for 2             ; code for expr1 * expr2
          STO T1
          code for x
```

```
          MUL T1
          STO Y
     end: NOP
```

which further translates to

```
          LOD Y
          STO T1
          LOD X
          ADD T1
          STO T1
          LOD Z
          SUB T1
          STO T1
          CPL T1
          JMZ end
          LOD #2                ; code for number
          STO T1
          code for x
          MUL T1
          STO Y
     end: NOP
```

which further translates to

```
          LOD Y
          STO T1
          LOD X
          ADD T1
          STO T1
          LOD Z
          SUB T1
          STO T1
          CPL T1
          JMZ end
          LOD #2
          STO T1
          LOD X                 ; code for variable, x
          MUL T1
          STO Y
     end: NOP
```

Homework Problem:

A famous problem in the history of mathematics is called the Greatest Common Divisor problem. One solution to the problem is called "Euclid's Method" which is shown below in a JavaScript algorithm.

Algorithm (Euclid's method):

```
W = initialValue1;    // for example, 48
X = initialValue2;    // for example, 18
while (X != 0)
    {
        Z = W % X;
        W = X;
        X = Z;
    }
// when finished, W = GCD(W,X)
```

ASSIGNMENT

- What computers do

  Use the mechanical algorithm demonstrated above to translate the JavaScript version of Euclid's Method into the corresponding PIPPIN code.

- What people do

  Extra assignment: Write PIPPIN code which will correctly round the result of a PIPPIN division to the nearest integer. Consider both positive and negative results.