Perl programming, and the design of assembler

1 Today's goal

- Learn Perl programming including lists, associative arrays, regular expressions, pattern matching, and substitution.
- Learn the design of assembler using Perl language.

2 Today's contents

Step 1 *Check 1* Change List 1 to list all prime factors of n. For example if $n = 120$, the output must be 2, 2, 2, 3, 5.

Step 2 *Check 2* Change List 2 to compute the correlation coefficient $\rho(x, y)$, which is given by

$$\rho(x,y) = \frac{E[(X - E[X])(Y - E[Y])]}{\sqrt{(X - E[X])^2}\sqrt{(Y - E[Y])^2}},$$

where $E[X]$ is the expected value (average) of X.

Step 3 *Check 3* Change List 3 to apply the discount rates which are stored in a file as follows:

```
strawberry 30
potato 15
```

In this case, strawberry and potato are 30% and 15% off, respectively. The items that are not in the discount rate list are no discount. Compute the total using the discount rate.

Step 4 *Check 4* Change List 4 to perform the inverse conversion. In other words, "`Jan 2, 2008`" must be converted to "`2008/1/2`."

Step 5 Write tinyasm (List 5) and execute it to check if it assemble countdown program (List 6) correctly.

Step 6 *Check 5* Let n be a binary number, and $g(n)$ be its gray code. Table 1 shows 4-bit binary numbers and their gray code numbers. It should be clear that, for every i, $g(i)$ and $g(i + 1)$ differs only in one bit.

Table 1: Binary and gray code numbers

n	$g(n)$
0000	0000
0001	0001
0010	0011
0011	0010
0100	0110
0101	0111
0110	0101
0111	0100
1000	1100
1001	1101
1010	1111
1011	1110
1100	1110
1101	1011
1110	1001
1111	1000

Let $n = n_3n_2n_1n_0$ and $g(n) = g_3g_2g_1g_0$ be a 4-bit number and its gray code number. $g(n)$ can be computed from n by

$$\begin{aligned} g_3 &= n_3 \\ g_2 &= n_3 \hat{} n_2 \\ g_1 &= n_2 \hat{} n_1 \\ g_0 &= n_1 \hat{} n_0, \end{aligned}$$

where $\hat{}$ denotes XOR operator. Write an assembly language program as follows:

- 16-bit binary number n is given from input port `in`
- Gray code numbers $g(0)$, $g(1)$, $g(2)$, ..., $g(n - 1)$ are written in the output buffer in turn.

Convert the assembly language program into Verilog HDL, and perform the simulation to confirm that

it works properly.

Step 6 *Check 6* Implement tinycpu with gray code program (in Step 5) in the FPGA, and confirm that it works properly.

3 Compiler and Assembler

Figure 1 illustrates the tasks of compiler and assembler. *Compiler* converts a C-like language program into an assembly language program. *Assembler* converts the assembly language program into a machine program. The machine program is converted to a Verilog HDL source codes which should be included in ram.v. We will design assembler using *Perl language*, and compiler using *compiler compiler tools* including *flex* (lexical analysis) and *bison* (context analysis).

4 Introduction to Perl language

Perl language borrows features from a variety of other languages including C language, Lisp, AWK, and sed. Perl takes lists from Lisp, associative arrays (hashes) from AWK, and regular expressions from sed. These simplify and facilitate parsing, text handling, and data management tasks.

4.1 Scalar variables and basic structure

List 1 is a Perl program prime that determines if n is prime. The first argument of this program is stored in $ARGV[0]. $n is a scalar variable. A scalar variable can store a *number* (integer or real number), or a *string* (text). Using for-loop it is checked if $n is divisible by $i for $i=2, 3, ..., $\sqrt{\text{n}}$. If it is divisible for all $i, $n is not prime.

List 1: Perl program prime to determine if n is prime

```
1  #!/usr/bin/perl -W
2
3  $n=$ARGV[0];
4  for($i=2;$i<=sqrt($n);++$i){
5      if($n%$i==0){
6          print "$n is not prime.\n";
7          exit(0);
8      }
9  }
10 print "$n is prime.\n";
```

4.2 Reading a file and handling a list

List 2 is a Perl program sum that compute the sum of two numbers in a file. It works as follows: Suppose that file data stores the following values.

```
1 2
3 4
5 6
```

If we execute

```
$ ./sum data
```

then we have the output

```
1+2=3
3+4=7
5+6=11
```

In this program "<>" means that a line of a file specified by the first argument is read and it is stored in a special variable $_. Since "<>" returns false if there is no more lines to be read, while loop is repeated for every line in the file.

A string matching operation is performed for /([0-9]+)\s+([0-9]+)/. In this string matching, [0-9] matches one of the characters in 0, 1, ..., 9, and + means the repetition for 1 or more times. Hence, [0-9]+ matches a number. Also, \s is a meta character (Talbe 2) of a blank. Therefore, /([0-9]+)\s+([0-9]+)/ matches a line with "number blanks number". If a line is matched, /([0-9]+)\s+([0-9]+)/ returns true, and returns false, otherwise. Also, a substring corresponding to the first [0-9]+ is stored in a special variable $1, since brackets "( )" includes it. In the same way, the second one is stored in $2.

A list starts with @, while a scalar variable starts from $. For example, @X and @Y denote lists. A scalar variable $1 is appended to @X by push(@X,$1). A list is an array of scalar variables. List @X can be accessed as an array, such that $X[0], $X[1], ..., $X[$#X] are scalar variables, where $#X denotes the index of the last element. Thus, @X has $#X+1 elements. It follows that push(@X,$1) increments $#X and then stores $1 in $X[$#X].

A loop foreach $x (@X) is used to execute some routine for each element in @X. Every element in @X is stored in $x in turn, and the following routine is executed.

Note that, if $x is omitted in foreach $x (@X), then each element in @X is stored in special variable $_.

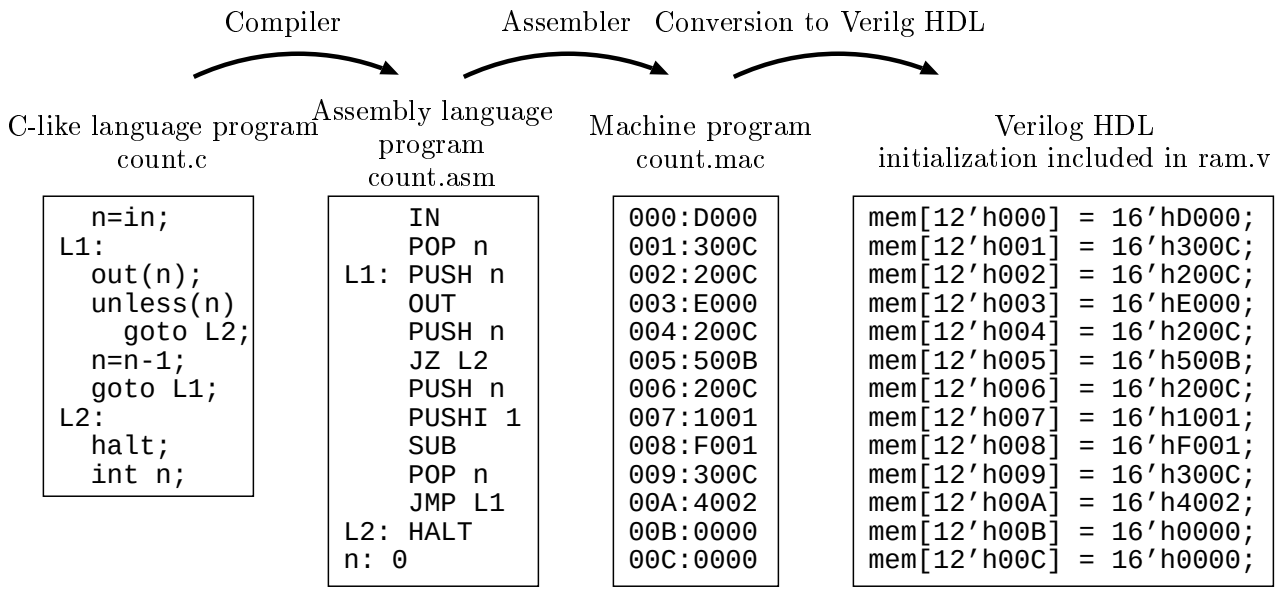

Figure 1: Compiler and Assembler

Table 2: Meta characters of Perl language

Meta character	its value
`[ ]`	Any character in the bracket.
.	Any one character
`\d`	a number. Equivalent to `[0-9]`.
`\w`	a number or "_". Equivalent to `[a-zA-Z0-9_]` と同じ.
`\s`	blank character(space, tab, or newline).

List 2: Perl program sum to compute the sum of pairs

```
1  #!/usr/bin/perl -W
2
3  while(<>){
4      if(/([0-9]+) ([0-9]+)/){
5          push(@X,$1);
6          push(@Y,$2);
7      }
8  }
9
10 printf "%d pairs are stored.\n",$#X+1;
11
12 $i=0;
13 foreach $x (@X){
14     $y=$Y[$i];
15     $sum=$x+$y;
16     print "$x+$y=$sum\n";
17     ++$i;
18 }
```

4.3 Associative Array

List 3 is a Perl program that computes the total of purchased goods.

An associative array starts with %, while a list start with @. For example, %pricelist is an associative array. While the index of a list is a number, while that of an associative array is a scalar value. In this example, the associative array is initialized such that $pricelist{apple}=100, etc.

Using the while loop, the quantity of each item is read from a file, and stored in associative array %num. For example, if the file contents is

```
orange 3
apple 2
strawberry 4
potato 5
bread 6
```

then %num stores the quantities such that $num{orange}=3, etc.

keys(%num) is the list of keys of %num, that is, (orange, apple, strawberry, potato, bread) sort sort them in alphabetical order. Thus, foreach statement stores keys of this list in n in the alphabetical order.

Using defined, you can check if an element of an associative array is defined. For example defined($pricelist{apple}) is true, but defined($pricelist{banana}) if false.

List 3: Perl program price to compute the total price of purchased product

```
1  #! /usr/bin/perl -W
2
3  %pricelist=(
4   apple=>100,
5   bread=>150,
6   orange=>120,
7   strawberry=>380,
8   potato=>80
9  );
10
11 while(<>){
12   if(/(\w+)\s+(\d+)/){
13       $num{$1}=$2;
14   }
15 }
16
17 foreach $n (sort(keys(%num))){
18     $product=$pricelist{$n}*$num{$n};
19     $total+=$product;
20     printf "%10s %4d %4d %6d\n",$n,$pricelist{$n},$num{
           $n},$product;
21 }
22 print "total=$total\n";
```

4.4 Substitution

List 4 is a Perl program that converts the formats of date. For example, if the following file is given as an input,

```
2008/5/31
2011/10/20
```

then we have the output as follows:

```
May 31, 2008
Oct 20, 2011
```

In this program, associative array @month stores the names of months, such that $month[0]=Jan. In while loop, each line of the input stored in special variable $_. s/(\d+)\/(\d+)\/(\d+)/$month[$2-1] $3, $1/ perform the substitution on $_. Note that "\" is an escape character. Hence, "\/" corresponds to "/". Since \d+ matches a number, (\d+)\/(\d+)\/(\d+) matches "number/number/number". Also, if matched, 1st, 2nd, and 3rd numbers are stored in special variables $1, $2, and $3. After that, a matched substring is substituted with "$month[$2-1] $3, $1". In this way, variable $_ is substituted. Finally, print; writes $_ to the standard output.

List 4: Perl program conv to convert the formats of date

```
1  #!/usr/bin/perl -W
2
3  @month=(Jan,Feb,Mar,Apr,May,Jun,Jul,Aug,Sep,Oct,Nov,Dec
       );
4
5  while(<>){
6      s/(\d+)\/(\d+)\/(\d+)/$month[$2-1] $3, $1/;
7      print;
8  }
```

5 Assembler

List 5 is an assembler program written by Perl, which converts an assembler language program into a machine program. The first while loop makes label list `%label`. The keys of `%label` are labels, and their values are corresponding addresses. Note that substitution "`s/\w+://`" removes "`label:`" from `$_`. The first foreach loop outputs the label list in alphabetical order. The second foreach loop mainly converts mnemonics and labels into machine code.

List 6 is an assembly language program for countdown. To assemble this program, execute the following command:

```
$ ./tinyasm count.asm
```

Also, you can use a Perl program List 7 to convert machine program to Verilog HDL source code. To obtain the Verilog HDL source code for count.asm, execute the following command:

```
$ ./tinyasm count.asm | ./mac2mem
```

The Verilog HDL source code thus obtained can be inserted in ram.v to initialize memory.

List 6: assembly language program count.asm for countdown

```
1          IN
2          POP n
3  L1: PUSH n
4          OUT
5          PUSH n
6          JZ L2
7          PUSH n
8          PUSHI 1
9          SUB
10         POP n
11         JMP L1
12 L2: HALT
13 n: 0
```

List 5: Assembler programtinyasm

```
1  #!/usr/bin/perl -W
2
3  %MCODE = (HALT=>0x0000,
4          PUSHI=>0x1000,
5          PUSH=>0x2000,
6          POP=>0x3000,
7          JMP=>0x4000,
8          JZ=>0x5000,
9          JNZ=>0x6000,
10         IN=>0xD000,
11         OUT=>0xE000,
12         ADD=>0xF000,
13         SUB=>0xF001,
14         MUL=>0xF002,
15         SHL=>0xF003,
16         SHR=>0xF004,
17         BAND=>0xF005,
18         BOR=>0xF006,
19         BXOR=>0xF007,
20         AND=>0xF008,
21         OR=>0xF009,
22         EQ=>0xF00A,
23         NE=>0xF00B,
24         GE=>0xF00c,
25         LE=>0xF00D,
26         GT=>0xF00E,
27         LT=>0xF00F,
28         NEG=>0xF010,
29         BNOT=>0xF011,
30         NOT=>0xF012);
31
32 $addr=0;
33 while(<>){
34     push(@source,$_);
35     if(/(\w+):/){
36         $label{$1}=$addr;
37         s/\w+://;
38     }
39     if(/-?\d+|[A-Z]+/){
40         $addr++;
41     }
42 }
43
44 print "*** LABEL LIST ***\n";
45 foreach $l (sort(keys(%label))){
46     printf "%-8s%03X\n",$l,$label{$l};
47 }
48
49 $addr=0;
50 print "\n*** MACHINE PROGRAM ***\n";
51 foreach (@source){
52     $line = $_;
53     s/\w+://;
54     if(/PUSHI\s+(-?\d+)/){
55         printf "%03X:%04X\t$line",$addr++,$MCODE{
       PUSHI}+($1&0xfff);
56     } elsif(/(PUSH|POP|JMP|JZ|JNZ)\s+(\w+)/){
57         printf "%03X:%04X\t$line",$addr++,$MCODE{$1
       }+$label{$2};
58     }elsif(/(-?\d+)/){
59         printf "%03X:%04X\t$line",$addr++,$1&0xffff;
60     } elsif(/([A-Z]+)/){
61         printf "%03X:%04X\t$line",$addr++,$MCODE{$1};
62     } else {
63         print "\t\t$line";
64     }
65 }
```

List 7: conversion program mac2mem from machine program to ram.v

```
1  #!/usr/bin/perl -W
2
3  while(<>){
4      if(/([0-9A-F]+):([0-9A-F]+)(.+)/){
5          print "mem[12'h$1] = 16'h$2; \/\/$3\n";
6      }
7  }
```

6 Homework

Homework 1 Write an assembly language program such that

- 16-bit gray code number n is given from input port in
- Binary numbers $g^{-1}(0)$, $g^{-1}(1)$, $g^{-1}(2)$, ..., $g^{-1}(n-1)$ are written in the output buffer in turn.

Perform the simulation to confirm that it works correctly.

Homework 2 Assembler (List 5) for tinyasm does not handle mistakes in assembly language programs. The mistakes includes:

undefined mnemonic undefined mnemonic is used.

undefined label undefined label is used.

multiply defined label the same label is defined twice or more.

immediate operand is out of range the immediate value in the operand is not in the range of 12-bit 2's complement.

initial value is out of range the initial value of the variables is not in the range of 16-bit 2's complement.

Modify the assembler program to find these errors and output appropriate error messages. Write intentionally incorrect assembly language programs and show that the modified assembler program handles them appropriately.