TINYC Compiler

1 Today's goal

- Learn how to compile if, while, and do statements.
- Learn the design of TINYC Compiler.
- Learn how to extend the TINYC Compiler.
- Learn the TINYC programing.

2 Today's contents

Step 1 Read carefully Section 3 to understand how if, while, and do statements are compiled.

Step 2 Write Flex grammar file dtinyc.y (List 1) and Bison grammar file dtiny.y (List 2), and generate DTINYC compiler dtinyc.

Step 3 *Check 1* Compile the following program:

```
while(A){if(B){C=D;}else{if(E){F=G;}}}
```

Show the compiling process of the above program as shown in Table 1 to confirm that the compiling process is correct.

Step 4 Write Flex grammar file tinyc.y (List 3) and Bison grammar file tiny.y (List 4) for TINYC compiler. Compile the program wrote for Step 7 of the last week, and check if it is compiled correctly.

Step 5 Add the conditional operator (`?:`) to the TINYC compiler. Note that, the conditional operator takes form "`expr1 ? expr2 : expr3`". If expr1 is non-zero, then its value is expr2, and otherwise, expr3.
Hint: you need to assign a unique ID to terminal symbol `'?'` in tinyc.l. Also, `'?'` and `':'` are right associative and the lowest precedence.

Step 6 Rewrite the program wrote for Step 7 of the last week to use the conditional operator. Compile it to confirm if it is correct.

Step 7 *Check 2* Write the TINYC program of 2-digit decimal counter that outputs 00, 01, 02, ... 99. Perform the simulation to confirm that the program works correctly.

Step 8 *Check 3* Implement the 2-digit decimal counter in the FPGA to check if it works correctly.

3 If, while, and do statements

We first consider how if, while, and do statements are compiled. For this purpose, we assume a simple and fake programming language DTINYC, which supports if, while, and do statements, and simple assignment as follows:

if It takes forms "if(formula){ statements }" or "if(formula){ statements }else{ statements }" . The behavior is the same as C language.

while It takes form "while(formula){ statements }". The behavior is the same as C language.

do It takes form "do{ statements } while (formula)". The behavior is the same as C language.

assignment It takes form "variable = variable;" or "variable = integer;"

Lists 1 and 2 are Flex grammar file dtinyc.l and Bison grammar file dtinyc.l for DTINYC. All blank characters are ignored. Reserved words if, else, while, and do are converted to tokens IF, ELSE, WHILE, and DO. Also, integers and names are converted to tokens NUMBER and NAME. Tokens IF, WHILE, and DO takes an integer as an semantic value. An integer n defined in the C declarations section, is used to assign an sequential unique numbers for these tokens as semantic values. When reserved words if, while, and do are converted to the tokens, this integer n is incremented, and assigned to variable yylval.n. Note that yylval is a union variable that takes one of an integer value or a string. More specifically, yylval.n is an integer while yylval.s is a string. In the Bison declaration section of Bison grammar file "`%union {char s[17]; int n;}`" defines that the

semantic value can be one of the a string `s` of length 17 or an integer `n`. Also, from `%token <s> NAME NUMBER` tokens `NAME` and `NUBMER` take a string as a semantic value. From `<n> IF WHILE DO`, tokens `IF`, `WHILE`, and `DO` take an integer as a semantic value. This semantic value is a sequential number assigned by Flex grammar file, which are used to generate unique labels such as `_001T` or `_001F`. Further, from `%type <n> if0`, non-terminal symbol `if0` takes an integer.

List 1: Flex grammer file dtinyc.l for DTINYC

```
%{
 #include <string.h>
 #include "y.tab.h"
 int n=0;
%}
%%
[ \t\n\r]
if {yylval.n=++n;return(IF);}
else {return(ELSE);}
while {yylval.n=++n;return(WHILE);}
do {yylval.n=++n;return(DO);}
[0-9]+ {strncpy(yylval.s,yytext,16);return(NUMBER);}
[a-zA-Z][a-zA-Z0-9]* {strncpy(yylval.s,yytext,16);return(
    NAME);}
. {return(yytext[0]);}
%%
int yywrap(){ return(1);}
```

List 2: Bison grammer file dtinyc.y for DTINYC

```
%{
#include <stdio.h>
%}
%union {char s[17]; int n;}
%token <s> NAME NUMBER
%token <n> IF WHILE DO
%token ELSE
%type <n> if0
%%
statements : statement | statements statement
;
statement : assign | if | while | do
;
while: WHILE {printf("_%03dT:\n",$1);} '(' expr ')' {printf("
    \tJZ _%03dF\n",$1);} '{' statements '}' {printf("\tJMP
    _%03dT\n_%03dF:\n",$1,$1);}
;
do: DO {printf("_%03dT:\n",$1);} '{' statements '}' WHILE '
    (' expr ')' ';' {printf("\tJNZ _%03dT\n",$1);}
;
if: if0 {printf("_%03dF:\n",$1);}
  | if0 {printf("\tJMP _%03dT\n_%03dF:\n",$1,$1);} ELSE '
      {' statements '}' {printf("_%03dT:\n",$1);}
;
if0: IF '(' expr ')' {printf("\tJZ _%03dF\n",$1);} '{'
    statements '}' {$$=$1;}
;
assign: NAME '=' expr ';' {printf("\tPOP %s\n",$1);}
;
expr: NAME {printf("\tPUSH %s\n",$1);}
    | NUMBER {printf("\tPUSHI %s\n",$1);}
;

%%
int yyerror(char *s){ printf("%s\n",s); }
int main(){ yyparse(); }
```

3.1 If and if-else statements

The structure of if statement is as follows:

```
if(expr){
   statements
}
```

This if statement is converted to:

```
  expr
  JZ F:
  statements
F:
```

If the value of epxr is 0 (false), then statements are not executed. For example,

```
if(A){
  B = 1;
}
```

is converted to:

```
        PUSH A
        JZ _001F
        PUSHI 1
        POP B
_001F:
```

Note that, 001 is a semantic value assigned to if. IF A is 0, PUSHI 1 and POP B are not executed.

The structure of if-else statement is as follows:

```
  if(expr){
     statements0
  } else {
     statements1
  }
```

This if-else statement is converted to:

```
  expr
  JZ F:
  statements0
  JMP T:
F:
  statements1
T:
```

Thus, if expr is 0 then statements1 is executed. Otherwise, statement0 is executed. For example, the following if-else statement

```
if(A){
  B = 1;
} else {
  B = 2;
}
```

is converted to:

```
        PUSH A
        JZ _001F
        PUSHI 1
        POP B
        JMP _001T
_001F:
        PUSHI 2
        POP B
_001T:
```

Note that, two labels _001F and _001T are used.

Table 1 is an example that shows how if-else statement is compiled.

Table 1: Compiling process of an if-else statement

input	output
if(A){B=1;}else{B=2;}	
IF(A){B=1;}else{B=2;}	
IF(NAME){B=1;}else{B=2;}	
IF(expr){B=1;}else{B=2;}	PUSH A
	JZ _001F
IF(expr){NAME=1;}else{B=2;}	
IF(expr){NAME=NUMBER;}else{B=2;}	
IF(expr){NAME=expr;}else{B=2;}	PUSHI 1
IF(expr){assign}else{B=2;}	POP B
IF(expr){statement}else{B=2;}	
IF(expr){statements}else{B=2;}	
if0 else{B=2;}	JMP _001T _001F:
if0 ELSE{B=2;}	
if0 ELSE{NAME=2;} if0 ELSE{NAME=NUMBER;}	
if0 ELSE{NAME=expr;}	PUSHI 2
if0 ELSE{assign}	POP B
if0 ELSE{statement}	
if0 ELSE{statements}	
if	_001T:
statement	
statements	

3.2 While and Do statements

The structure of if statement is as follows:

```
while(expr){
   statements
}
```

The while statement is converted to:

```
T:
  expr
  JZ F:
  statements
  JMP T:
F:
```

The structure of if statement is as follows:

```
do{
   statements
  } while(expr);
```

The do statement is converted to:

```
T:
  statements
  expr
  JNZ T:
```

Tables 2 and 3 show the compiling process of while and do statements.

Table 2: Compiling process of a while statement

input	output
while(A){B=1;}	
WHILE(A){B=1;}	_001T:
WHILE(NAME){B=1;}	
WHILE(expr){B=1;}	PUSH A JZ _001F
WHILE(expr){NAME=1;}	
WHILE(expr){NAME=NUMBER;}	
WHILE(expr){NAME=expr;}	PUSHI 1
WHILE(expr){statement}	POP B
WHILE(expr){statements}	
while	JMP _001T _001F:
statement	
statements	

Table 3: Compiling process of a do statement

input	output
do{B=1;}while(A);	
DO{B=1;}while(A);	_001T
DO{NAME=1;}while(A);	
DO{NAME=NUMBER;}while(A);	
DO{NAME=expr;}while(A);	PUSHI 1
DO{statement}while(A);	POP B
DO{statements}while(A);	
DO{statements}WHILE(A);	
DO{statements}WHILE(NAME);	
DO{statements}WHILE(expr);	PUSH A
do	JNZ _001T
statement	
statements	

4 TINYC Compiler

Lists 3 and 4 shows grammar files of TINYC compiler.u

List 4: Bison grammer file tinyc.y for TINYC

```
1  %{
2  #include <stdio.h>
3  %}
4  %union {char s[17]; int n;}
5  %token <s> NAME NUMBER
6  %token <n> IF WHILE DO
7  %type <n> if0
8  %token GOTO ELSE INT IN OUT HALT
9  %left OR
10 %left AND
11 %left '|'
12 %left '^'
13 %left '&'
14 %left EQ NE
15 %left GE LE '<' '>'
16 %left SHL SHR
17 %left '+' '-'
18 %left '*'
19 %right '!' '~' NEG
20 %%
21 statements : statement | statements statement
22 ;
23 statement : label | intdef | goto | if | while | do | halt | out |
       assign
24 ;
25 label : NAME ':' {printf("%s:\n",$1);}
26 ;
27 intdef: INT intlist ';'
28 ;
29 intlist: integer
30       | intlist ',' integer
```

List 3: Flex grammar file tinyc.l for TINYC

```
%{
 #include <string.h>
 #include "y.tab.h"
 int n=0;
%}
%%
[ \t\n\r]
&& {return(AND);}
\|\| {return(OR);}
== {return(EQ);}
!= {return(NE);}
\>= {return(GE);}
\<= {return(LE);}
\<\< {return(SHL);}
\>\> {return(SHR);}
do {yylval.n=++n;return(DO);}
else {return(ELSE);}
goto {return(GOTO);}
halt {return(HALT);}
if {yylval.n=++n;return(IF);}
in {return(IN);}
int {return(INT);}
out {return(OUT);}
while {yylval.n=++n;return(WHILE);}
[0-9]+ {strncpy(yylval.s,yytext,16);return(NUMBER);}
[a-zA-Z][a-zA-Z0-9]* {strncpy(yylval.s,yytext,16);return(
    NAME);}
. {return(yytext[0]);}
%%
int yywrap(){ return(1);}
```

```
;
integer: NAME {printf("%s: 0\n",$1);}
       | NAME '=' NUMBER {printf("%s: %s\n",$1,$3);}
       | NAME '=' '-' NUMBER {printf("%s: -%s\n",$1,$4);}
;
goto: GOTO NAME ';' {printf("\tJMP %s\n",$2);}
;
if: if0 {printf("_%03dF:\n",$1);}
  | if0 {printf("\tJMP _%03dT\n_%03dF:\n",$1,$1);} ELSE
      '{' statements '}' {printf("_%03dT:\n",$1);}
;
if0: IF '(' expr ')' {printf("\tJZ _%03dF\n",$1);} '{'
    statements '}' {$$=$1;}
;
while: WHILE {printf("_%03dT:\n",$1);} '(' expr ')' {printf("\
    tJZ _%03dF\n",$1);} '{' statements '}' {printf("\tJMP _
    %03dT\n_%03dF:\n",$1,$1);}
;
do: DO {printf("_%03dT:\n",$1);} '{' statements '}' WHILE
    '(' expr ')' ';' {printf("\tJNZ _%03dT\n",$1);}
;
assign: NAME '=' expr ';' {printf("\tPOP %s\n",$1);}
;
halt : HALT ';' {printf("\tHALT\n");}
;
out: OUT '(' expr ')' ';' {printf("\tOUT\n");}
```

```
;
expr: NAME {printf("\tPUSH %s\n",$1);}
    | NUMBER {printf("\tPUSHI %s\n",$1);}
    | IN {printf("\tIN\n");}
    | '!' expr {printf("\tNOT\n");}
    | '~' expr {printf("\tBNOT\n");}
    | '-' expr %prec NEG {printf("\tNEG\n");}
    | expr '+' expr {printf("\tADD\n");}
    | expr '-' expr {printf("\tSUB\n");}
    | expr '*' expr {printf("\tMUL\n");}
    | expr AND expr {printf("\tAND\n");}
    | expr OR expr {printf("\tOR\n");}
    | expr '&' expr {printf("\tBAND\n");}
    | expr '|' expr {printf("\tBOR\n");}
    | expr '^' expr {printf("\tBXOR\n");}
    | expr SHL expr {printf("\tSHL\n");}
    | expr SHR expr {printf("\tSHR\n");}
    | expr EQ expr {printf("\tEQ\n");}
    | expr NE expr {printf("\tNE\n");}
    | expr GE expr {printf("\tGE\n");}
    | expr LE expr {printf("\tLE\n");}
    | expr '<' expr {printf("\tLT\n");}
    | expr '>' expr {printf("\tGT\n");}
    | '(' expr ')'
;

%%
int yyerror(char *s){ printf("%s\n",s); }
int main(){ yyparse(); }
```

5 Homework

Homework 1 Let us support `++` and `--` statements for TINYCPU, TINYASM, and TINYC.

1. Modify ALU module alu.v to support unary operators INC and DEC such that they outputs a+1 and a-1, respectively.
2. Modify assembler tinyasm to support instruction INC and DEC, which increments and decrements the top of the stack, qtop, respectively
3. Modify Flex grammar file tinyc.l and bison grammar file tinyc.y to support `++` and `--` sentenses. For example, `++n;` and `n++;` increment the value of variable n.
4. Write the countdown TINYC program using `--`. Compile and assemble, and then perform the simulation to confirm that it works correctly.

Homework 2 Write the 4-digit decimal counter using TINYC. More specifically, it outputs 0000, 0001, 0002, ..., 9999, and terminates. Perform the simulation and check if it works correctly.