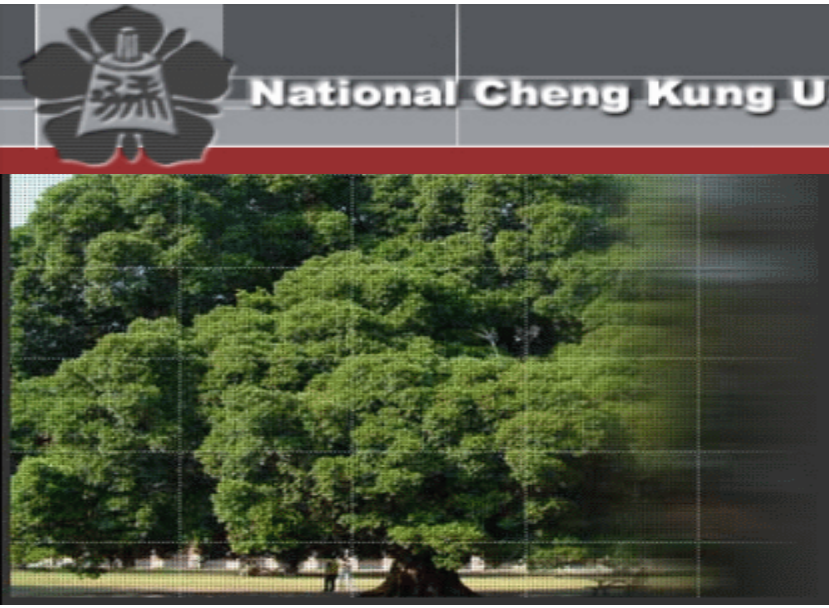

Homework 3 explanation

NCKU CSIE DICLAB

Introduction

- Design an circuit with addition, subtraction, and multiplication functions
- The input only consists of numbers 0 to 15, four operators: +, -, *, =, and paratheses
- The inputs are represented in ASCII codes
- The input expression string will not exceed 16 characters
- It's difficult for computers to comprehensively process and handle the entire equation
  - Converting the notation of the expression from infix to postfix

Represent	ASCII code	Represent	ASCII code	Represent	ASCII code
0	48	8	56	(	40
1	49	9	57	)	41
2	50	a (number 10)	97	*	42
3	51	b (number 11)	98	+	43
4	52	c (number 12)	99	-	45
5	53	d (number 13)	100	=	61
6	54	e (number 14)	101		
7	55	f (number 15)	102		

Finite State Machine

1. BUFFER：Read input string and store it into buffer
2. IN2POS：Convert the input string to postfix
3. POP：Pop out remaining operators in the buffer
4. CALCULATE：Calculate results according to the postfix expression
5. RESULT：Pull up ‘valid’ signal and output calculation result
6. RESET：Pull down ‘valid’ signal and go back to ‘BUFFER’ state

Data Registers

- dataBuffer: used to store input string
- OpStack: store operators while converting expression to postfix
- OutBuffer: store converted postfix output string
- sum: buffer for calculation

```
16    reg [2:0] nowState, nextState;
17    reg [6:0] dataBuffer [15:0];
18
19    reg [4:0] len;
20    reg [4:0] arrPt, stackPt, outPt;
21
22    reg [6:0] OpStack   [15:0];
23    reg [6:0] OutBuffer [15:0];
24
25    reg [6:0] sum [15:0];
26    reg [3:0] sumPt ;
```

BUFFER

- Decode ASCII code to numbers
  - The codes for operators are directly stored into ‘dataBuffer’

```
61          BUFFER:begin
62              if(ready) begin
63                  readEn <= 1;
64              end
65              if(ascii_in!=61 && (ready||readEn)) begin
66                  len <= len + 1;
67                  case(ascii_in)      // Mapping
68                      // number(0~9)
69                      48:  dataBuffer[len] <= 4'd0 ;  49: dataBuffer[len] <= 4'd1 ;  50: dataBuffer[len] <= 4'd2 ;
70                      51:  dataBuffer[len] <= 4'd3 ;  52: dataBuffer[len] <= 4'd4 ;  53: dataBuffer[len] <= 4'd5 ;
71                      54:  dataBuffer[len] <= 4'd6 ;  55: dataBuffer[len] <= 4'd7 ;  56: dataBuffer[len] <= 4'd8 ;
72                      57:  dataBuffer[len] <= 4'd9 ;
73                      // number(10~15)
74                      97:  dataBuffer[len] <= 4'd10;  98: dataBuffer[len] <= 4'd11;
75                      99:  dataBuffer[len] <= 4'd12; 100: dataBuffer[len] <= 4'd13; 101: dataBuffer[len] <= 4'd14;
76                      102: dataBuffer[len] <= 4'd15;
77                      // operation
78                      default : dataBuffer[len] <=  ascii_in;
79                  endcase
80              end
81          end
```

IN2POS

- arrPt：index used to scan ‘dataBuffer’
- stackPt：index of ‘OpStack’

- If the current token of ‘dataBuffer’ is 40 (‘(’), push it to ‘OpStack’
  - Store the value of ‘dataBuffer[arrPt]’ to OpStack[stackPt]
  - Increase the value of ‘stackPt’ by 1 (Push operation)
  - Increase the value of ‘arrPt’ by 1 (Scan the next token)

```
82              IN2POS:begin
83                  case(dataBuffer[arrPt])
84                      40:begin    // (     Put into stack
85                          OpStack[stackPt] <= dataBuffer[arrPt];
86                          stackPt <= stackPt + 1;
87                          arrPt <= arrPt + 1;
88                      end
```

IN2POS

- If the current token of ‘dataBuffer’ is 41 (‘)’), pop tokens from ‘OpStack’ until a left parenthesis is found
  - If the top token of ‘OpStack’ is not parenthesis, pop out it from stack and append it to output string
    - Line 91 & 92: append a value to ‘OutBuffer’ and increase its index (outPt) by 1
  - No matter whether the top token of ‘OpStack’ is a left parenthesis, it should be popped out (Line 94)
  - If the top token of ‘OpStack’ is a left parenthesis, scan the next token from the input string (Line 95)

```
89                  41:begin    // )     Put into stack
90                      if(OpStack[stackPt-1]!=40 && OpStack[stackPt-1]!=41)begin
91                          OutBuffer[outPt] <= OpStack[stackPt-1];
92                          outPt <= outPt + 1;
93                      end
94                      stackPt <= stackPt - 1;
95                      if(OpStack[stackPt-1]==40) arrPt <= arrPt + 1;
96                  end
```

IN2POS

- If the current token of ‘dataBuffer’ is 42 (‘*’), pop tokens from ‘OpStack’ until the precedence of the top token is lower than that of the current token
  - If the top token of ‘OpStack’ is ‘*’, pop out the top token and append it to the output string (Line 98 ~ 102)
  - Otherwise, push the current token onto ‘OpStack’ (Line 103 ~ 107), and scan the next token from the input string (Line 106)

```
97                  42:begin    // *
98                      if(OpStack[stackPt-1]==42 && stackPt!=0) begin
99                          OutBuffer[outPt] <= OpStack[stackPt-1];
100                         stackPt <= stackPt -1 ;
101                         outPt <= outPt + 1;
102                     end
103                     else begin
104                         OpStack[stackPt] <= dataBuffer[arrPt];
105                         stackPt <= stackPt + 1;
106                         arrPt <= arrPt + 1;
107                     end
108                 end
```

IN2POS

- If the current token of ‘dataBuffer’ is 43 or 45 (‘+’ or ‘-’), pop tokens from ‘OpStack’ until the precedence of the top token is lower than that of the current token
  - If the top token of ‘OpStack’ is ‘+’、‘-’、‘*’, pop out the top token and append it to the output string (Line 110 ~ 114)
  - Otherwise, push the current token onto ‘OpStack’ (Line 115 ~ 119), and scan the next token from the input string (Line 118)
  - The pop operation will stop only when a left parenthesis is found or the stack becomes empty

```
109                 43, 45:begin  // + -
110                     if((OpStack[stackPt-1]==42 || OpStack[stackPt-1]==43 || OpStack[stackPt-1]==45) && stackPt!=0) begin
111                         OutBuffer[outPt] <= OpStack[stackPt-1];
112                         stackPt <= stackPt -1 ;
113                         outPt <= outPt + 1;
114                     end
115                     else begin
116                         OpStack[stackPt] <= dataBuffer[arrPt];
117                         stackPt <= stackPt + 1;
118                         arrPt <= arrPt + 1;
119                     end
120                 end
```

IN2POS

- If the current token of ‘dataBuffer’ is a number (0~15), append it to the output string

```
121                 default:begin  // Normal number
122                     OutBuffer[outPt] <= dataBuffer[arrPt];
123                     outPt <= outPt + 1;
124                     arrPt <= arrPt + 1;
125                 end
```

POP

- If ‘OpStack’ has remaining tokens, pop them out append them to the output string
  - Line 129：Judge if ‘OpStack’ is empty
  - Line 130：Pop operation
  - Line 131：Ignore parenthesis while appending
  - Line 132 & 133：append the top token of ‘OpStack’ to ‘OutBuffer’, and increase the index of ‘OutBuffer’ by 1

```
128         POP:begin
129             if(stackPt!=0) begin
130                 stackPt <= stackPt - 1;
131                 if(OpStack[stackPt-1]!=40 && OpStack[stackPt-1]!=41)begin
132                     OutBuffer[outPt] <= OpStack[stackPt-1];
133                     outPt <= outPt + 1;
134                 end
135             end
136         end
```

CALCULATE

- Use stack ‘sum’ for calculation
  - If the current token of ‘OutBuffer’ is a number, push it onto the stack (Line 152 ~ 155)
  - Otherwise, pop out two tokens from the stack and calculate, store the calculation results back into the stack (Line 140 ~ 151)

```
137             CACULATE:begin
138                 stackPt <= stackPt + 1;
139                 case(OutBuffer[stackPt])
140                     42:begin
141                         sum[sumPt-2] <= sum[sumPt-2] * sum[sumPt-1];
142                         sumPt <= sumPt -1;
143                     end
144                     43:begin
145                         sum[sumPt-2] <= sum[sumPt-2] + sum[sumPt-1];
146                         sumPt <= sumPt -1;
147                     end
148                     45:begin
149                         sum[sumPt-2] <= sum[sumPt-2] - sum[sumPt-1];
150                         sumPt <= sumPt -1;
151                     end
152                     default:begin
153                         sum[sumPt] <= OutBuffer[stackPt];
154                         sumPt <= sumPt +1;
155                     end
156                 endcase
157             end
```

RESULT & RESET

- Pull up the ‘valid’ signal (Line 159), and reset registers (Line 160 ~ 172)
- Pull down the ‘valid’ signal at the next cycle (Line 175)

```
158                 RESULT:begin
159                     valid <= 1;
160                     result <= sum[sumPt-1];
161                     arrPt <= 0;
162                     stackPt <= 0;
163                     outPt <= 0;
164                     sumPt <= 0;
165                     readEn <= 0;
166                     len <= 0;
167                     for(i=0;i<16;i=i+1)begin
168                         OutBuffer[i]<=0;
169                         OpStack[i]<=0;
170                         dataBuffer[i]<=0;
171                         sum[i] <= 0;
172                     end
173                 end
174                 RESET:begin
175                     valid <= 0;
176                 end
```

Example

- Given an expression in infix notation “3 + 4 * (2 - 1)”, convert it to postfix and calculate the result

len = 9

dataBuffer

arrPt	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
data	3	+	4	*	(	2	-	1	)							

OpStack

stack Pt	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
data																

OutBuffer

outPt	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
data																

Example

- dataBuffer[0] = 3, append it to the output string (Line 121 ~ 125)

len = 9

dataBuffer

arrPt	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
data	3	+	4	*	(	2	-	1	)							

OpStack

stack Pt	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
data																

OutBuffer

outPt	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
data	3															

Example

- dataBuffer[1] = '+'. 'OpStack' is empty, push '+' onto 'Opstack' (Line 115 ~ 119)

len = 9

dataBuffer

arrPt	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
data	3	+	4	*	(	2	-	1	)							

OpStack

stack Pt	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
data	+															

OutBuffer

outPt	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
data	3															

Example

- dataBuffer[2] = 4, append it to the output string (Line 121 ~ 125)

len = 9

dataBuffer

arrPt	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
data	3	+	4	*	(	2	-	1	)							

OpStack

stack Pt	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
data	+															

OutBuffer

outPt	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
data	3	4														

Example

- dataBuffer[3] = '*', the precedence of the top token of 'OpStack' is lower, push '*' onto 'OpStack' (Line 103 ~ 107)

len = 9

dataBuffer

arrPt	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
data	3	+	4	*	(	2	-	1	)							

OpStack

stack Pt	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
data	+	*														

OutBuffer

outPt	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
data	3	4														

Example

- dataBuffer[4] = '(', push it onto 'OpStack' (Line 84~88)

len = 9

dataBuffer

arrPt	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
data	3	+	4	*	(	2	-	1	)							

OpStack

stack Pt	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
data	+	*	(													

OutBuffer

outPt	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
data	3	4														

Example

- dataBuffer[5] = 2, append it to the output string (Line 121 ~ 125)

len = 9

dataBuffer

arrPt	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
data	3	+	4	*	(	2	-	1	)							

OpStack

stack Pt	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
data	+	*	(													

OutBuffer

outPt	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
data	3	4	2													

Example

- dataBuffer[6] = '-', the top token of 'OpStack' is a left parenthesis, push '-' onto 'OpStack' (Line 115 ~ 119)

len = 9

dataBuffer

arrPt	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
data	3	+	4	*	(	2	-	1	)							

OpStack

stack Pt	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
data	+	*	(	-												

OutBuffer

outPt	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
data	3	4	2													

Example

- dataBuffer[7] = 1, append it to the output string (Line 121 ~ 125)

len = 9

dataBuffer

arrPt	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
data	3	+	4	*	(	2	-	1	)							

OpStack

stack Pt	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
data	+	*	(	-												

OutBuffer

outPt	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
data	3	4	2	1												

Example

- dataBuffer[8] = ')', the top token of 'OpStack is '-', pop '-' from 'OpStack' and append it to the output string (Line 90 ~ 94). 'arrPt' = len – 1, next state is 'POP'

len = 9

dataBuffer

arrPt	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
data	3	+	4	*	(	2	-	1	)							

OpStack

stack Pt	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
data	+	*	(													

OutBuffer

outPt	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
data	3	4	2	1	-											

Example

- stackPt != 0, pop a token from OpStack (Line 130). The token is '(', so it will not be append to the output string. (Line 131)

len = 9

dataBuffer

arrPt	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
data	3	+	4	*	(	2	-	1	)							

OpStack

stack Pt	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
data	+	*														

OutBuffer

outPt	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
data	3	4	2	1	-											

Example

- stackPt != 0, pop a token from OpStack (Line 130). The token is '*', so it will be append to the output string. (Line 132 ~ 133)

len = 9

dataBuffer

arrPt	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
data	3	+	4	*	(	2	-	1	)							

OpStack

stack Pt	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
data	+															

OutBuffer

outPt	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
data	3	4	2	1	-	*										

Example

- stackPt != 0, pop a token from OpStack (Line 130). The token is '+', so it will be append to the output string. (Line 132 ~ 133)

len = 9

dataBuffer

arrPt	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
data	3	+	4	*	(	2	-	1	)							

OpStack

stack Pt	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
data																

OutBuffer

outPt	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
data	3	4	2	1	-	*	+									

Example

- stackPt = 0, no operation. Next state is ‘CALCULATE’

len = 9

dataBuffer

arrPt	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
data	3	+	4	*	(	2	-	1	)							

OpStack

stack Pt	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
data																

OutBuffer

outPt	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
data	3	4	2	1	-	*	+									

Example

- 'StackPt' is used as index for scanning 'OutBuffer', increase 'stackPt' by 1 (Line 138)
- OutBuffer[0] = 3, push it onto 'sum' (Line 152 ~ 155)

outPt = 7

OutBuffer

stack Pt	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
data	3	4	2	1	-	*	+									

sum

sumPt	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
data	3															

Example

- 'StackPt' is used as index for scanning 'OutBuffer', increase 'stackPt' by 1 (Line 138)
- OutBuffer[1] = 4, push it onto 'sum' (Line 152 ~ 155)

outPt = 7

OutBuffer

stack Pt	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
data	3	4	2	1	-	*	+									

sum

sumPt	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
data	3	4														

Example

- 'StackPt' is used as index for scanning 'OutBuffer', increase 'stackPt' by 1 (Line 138)
- OutBuffer[2] = 2, push it onto 'sum' (Line 152 ~ 155)

outPt = 7

OutBuffer

stack Pt	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
data	3	4	2	1	-	*	+									

sum

sumPt	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
data	3	4	2													

Example

- 'StackPt' is used as index for scanning 'OutBuffer', increase 'stackPt' by 1 (Line 138)
- OutBuffer[3] = 1, push it onto 'sum' (Line 152 ~ 155)

outPt = 7

OutBuffer

stack Pt	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
data	3	4	2	1	-	*	+									

sum

sumPt	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
data	3	4	2	1												

Example

- 'StackPt' is used as index for scanning 'OutBuffer', increase 'stackPt' by 1 (Line 138)
- OutBuffer[4] = '-'
  - Pop sum[2] and sum[3], push sum[2] – sum[3] back to 'sum' (Line 149)
  - Decrease sumPt by 1 (Line 150)

outPt = 7

OutBuffer

stackPt	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
data	3	4	2	1	-	*	+									

sum

sumPt	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
data	3	4	1													

Example

- 'StackPt' is used as index for scanning 'OutBuffer', increase 'stackPt' by 1 (Line 138)
- OutBuffer[5] = '*'
  - Pop sum[1] and sum[2], push sum[1] * sum[2] back to 'sum' (Line 141)
  - Decrease sumPt by 1 (Line 142)

outPt = 7

OutBuffer

stackPt	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
data	3	4	2	1	-	*	+									

sum

sumPt	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
data	3	4														

Example

- 'StackPt' is used as index for scanning 'OutBuffer', increase 'stackPt' by 1 (Line 138)
- OutBuffer[6] = '+'
  - Pop sum[0] and sum[1], push sum[0] + sum[1] back to 'sum' (Line 145)
  - Decrease sumPt by 1 (Line 146)
- stackPt = outPt – 1 (before positive edge of 'clk'), next state is 'RESULT'

outPt = 7

OutBuffer

stack Pt	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
data	3	4	2	1	-	*	+									

sum

sumPt	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
data	7															

Example

- In ‘RESULT’ state
  - Pull up ‘valid’ signal (Line 159)
  - Result is sum[0] (Line 160)
  - Initialized registers (Line 161 ~ 172)

sumPt = 1

sum

sumPt	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
data	7															