THE UNIVERSITY OF AUCKLAND

First Semester, 2007

City Campus

Computer Science

Language Implementation

(Time allowed TWO hours)

FAMILY NAME:
PERSONAL NAMES:
STUDENT ID NUMBER:
LOGIN NAME:
SIGNATURE:

This Examination is out of 100 Marks. Attempt **ALL** questions. Write your answers in the spaces provided in this question and answer booklet. Do not remove the staples from the question and answer booklet. However, you may detach and remove the staples from the appendices.

1		10
2(a)		15
2(b)		5
3(a)		15
3(b)		5
4		17
5		18
6		15
Total		100

Continued ...

Print Name and Student ID _____

1. JFlex **[10 Marks]**

Indicate at least 10 different kinds of errors in the following fragment of JFlex code. ("..." just means omitted code). Give a reason or appropriate correction for each one. Assume line breaks and spaces are not syntactically important. Assume comments **can** be nested.

```
%{
     int commentNest = 0;
     int lineCount = 0;
     ...
%}
%init{
     yybegin( NORMAL );
%init}
newline        =    \r|\n|\r\n
space          =    [\ |\t]
ident          =    [A-Za-z][A-Za-z0-9]+
number         =    [1-9][0-9]+
%state NORMAL, COMMENT
%%
<NORMAL> {
     ident     { return token( sym.IDENT ); }
     if        { return token( sym.IF ); }
     else      { return token( sym.ELSE ); }
     "("       { return token( sym.LEFT ); }
     ")"       { return token( sym.RIGHT ); }
     .         { return token( sym.DOT ); }
     ...
     "/*"      { commentNest++; return token( sym.COMMENT ); }
     {number}  { return token( sym.NUMBER ); }
     .         { }
     {space}   { }
     {newline} { lineCount++; }
     }
<COMMENT> {
     "/*"      { commentNest++; }
     "*/"      { --commentNest; if ( commentNest == 0 ) yyend( COMMENT ); }
     {newline} { }
     .         { return token( sym.LEXERROR ); }
     }
<<EOF>>        { return token( sym.EOF ); }
```

Continued ...

Print Name and Student ID _____

(10 marks)

Continued ...

Print Name and Student ID _____

2. Bottom Up LALR(1) Parsing [20 Marks]

Consider the CUP grammar in the **Appendix For Question 2**. Note that the rules for `RHSList` are left recursive, while the rules for `SymbolList` are right recursive. Also, `SymbolList` can expand to empty.

(a) Using the information provided in the appendix, perform a shift-reduce LALR(1) parse of the input

```
Body ::= BEGIN StmtList:stmtList END | ;
```

Assume "`Body`", "`BEGIN`", "`StmtList`", "`stmtList`", and "`END`" match `IDENT`, and "`::=`", "`:`", "`|`" and "`;`" match `EXPANDSTO`, `COLON`, `OR` and `SEMICOLON`, respectively.

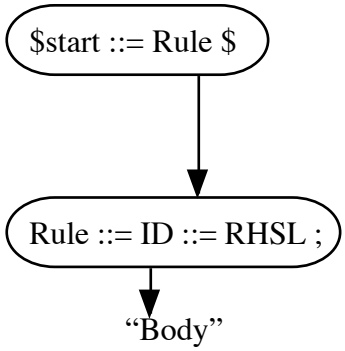
Stack							Token	Reduce	Shift
$0							ID Body		Shift 2
$0	ID 2						::=		Shift 3
$0	ID 2	::= 3					ID BGN		Shift 7
$0	ID 2	::= 3	ID 7				ID StmtL	Sym ::= ID	Shift
$0									
$0									
$0									
$0									
$0									
$0									
$0									
$0									
$0									
$0									
$0									
$0									
$0									
$0									
$0									
$0									
$0	Rule1								Shift 14
$0	Rule1	$14					$	$start ::= Rule $	Shift -1
$0	$strt -1							Accept	

(15 marks)

Continued ...

Print Name and Student ID _____

(b) Draw the parse tree corresponding to the grammar rules used to parse this input



(5 marks)

Continued ...

Print Name and Student ID _____

3. Write a grammar for component invocation statements, etc [20 Marks]

(a) Consider the language of assignment 2, used to specify logic circuits.

Some typical component invocation statements in this language are

```
input( "opd", 10, 100, base, n ) { out opd };

{ in result } output( "result", 10, 200, base, n );

{ in opd1 opd2 } xor( 2 ) { out sum };

{ in value1, value2, carryIn } add( n ) { out result, carryOut };

{ in clkOpd1 result1 } or( 2 ).not( 1 ) { out result2 };

{ in opd1[ i ], opd2[ i ], carry[ i ] } fullAdder
     { out sum[ i ], carry[ i + 1 ] };

{ in opd1[ n / 2 @ 0 ], opd2[ n / 2 @ 0 ] }
     compare( n / 2 )
     { out lessLow, equalLow, greaterLow };

{ in cond, opd[ n - p @ 0 ] zero opd } select( 1, n ) { out result };
```

The "." separated list of "IDENT (ExprList)" represents a "pipelined" list of invocations, where the output paths from a preceding invocation become the input paths of the following invocation. Normally there is only one invocation.

The input path parameters "{ in ... }", value parameters "(...)", and output path parameters "{ out ... }" are omitted if there are no parameters of the appropriate type.

The actual value parameters are a "," separated list of one or more value expressions ("Expr"s).

The actual input and output parameters are a "," separated list of one or more path array expressions.

A path array expression is a list of zero or more path names, side by side, with nothing in between.

A path name is a simple path name ("IDENT"), indexed path name ("IDENT[Expr]"), or subarray path name ("IDENT[Expr @ Expr]").

Write a grammar for component invocation statements **in general**.

You do not have to define the grammar for a value expression ("Expr").

You do not have to include any actions ("{...}") or attribute names (":ident"). You may write terminal symbols such as keywords ("in", "out", etc), and special symbols (",", "(", ")", etc) directly by enclosing them in double quotes ("...").

Continued ...

Print Name and Student ID _____

Continued ...

Print Name and Student ID _____

(15 marks)

Continued ...

Print Name and Student ID _____

(b) Write a component declaration

```
component { in opd1, opd2 } compareBit { out less, equal, greater }
     begin
          ...
     end
```

to set the value of the paths “less”, “equal”, and “greater” to indicate whether “opd1 < opd2”, “opd1 == opd2”, and “opd1 > opd2”.

(5 marks)

Continued ...

Print Name and Student ID _____

4. Show the run time stack for a B-- program [17 Marks]

Use the program written in the B-- language in the **Appendix For Question 4**.

Complete the drawing of the data structure built for the global variables "`source1`", "`source2`" and "`dest`".

Use multiple colours (other than red) for arrows, etc., to make your diagram clearer.

Display the activation records (stack frames) for all methods in the process of being invoked when the maximum level of nesting of method invocations occurs when the statement

```
merge( 0, source1, source2, &dest );  // Inside this invocation
```

on line 74 is invoked, and the process is almost ready to return. At this stage the process should be executing the method "`merge`" at line 58.

Indicate the appropriate values for each activation record (stack frame) you draw. The line numbers on the left-hand side of the program should be used to represent the return address. Draw appropriate arrows for the var parameters, and pointers to objects. For var parameters, make sure you indicate the exact variable pointed to in an object very clearly. Represent `List` nodes as shown in the sample entries.

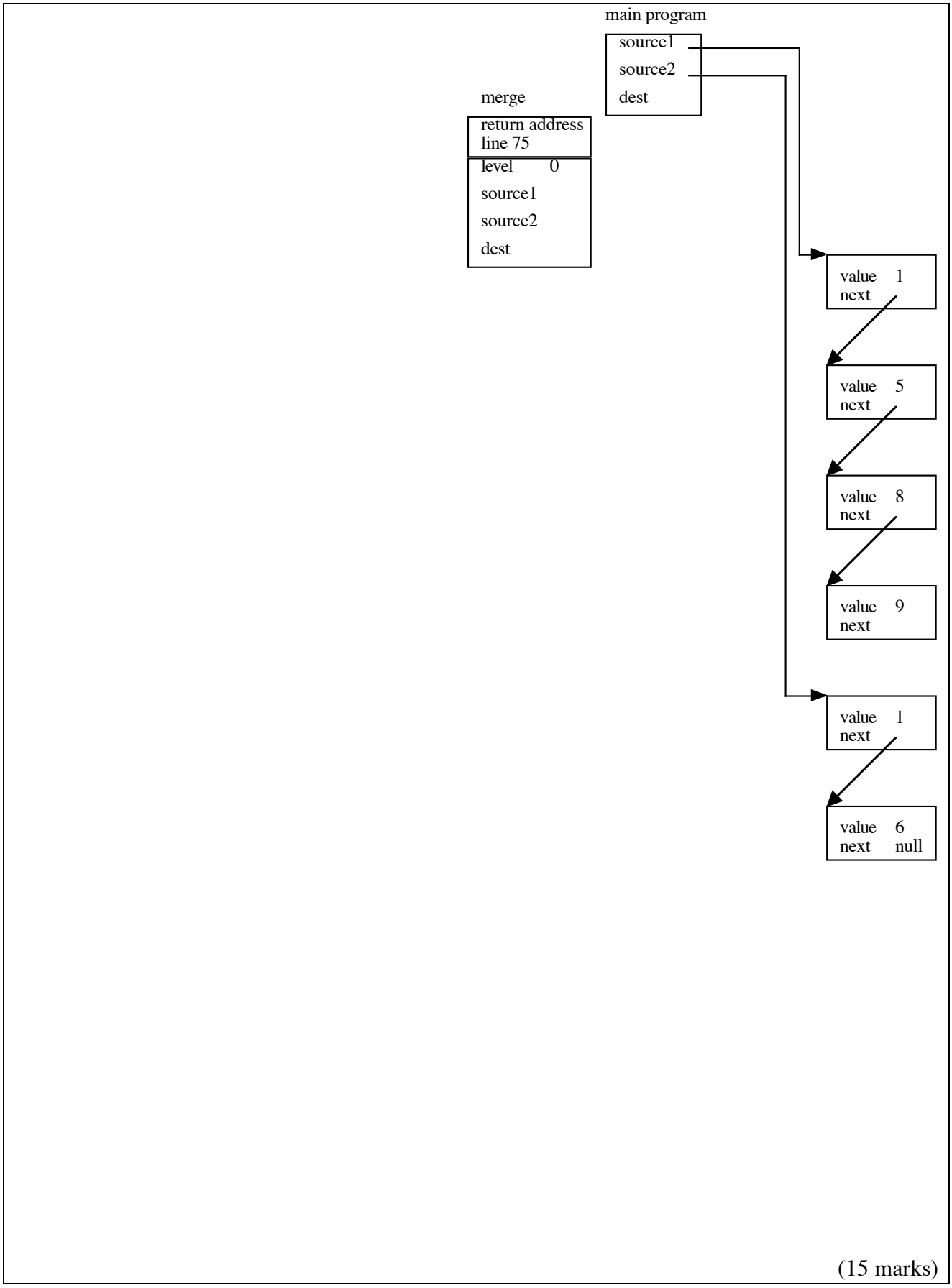
Also indicate the output generated by the complete execution of the program.

Output generated:

(2 marks)

Continued ...

Print Name and Student ID _____



(15 marks)

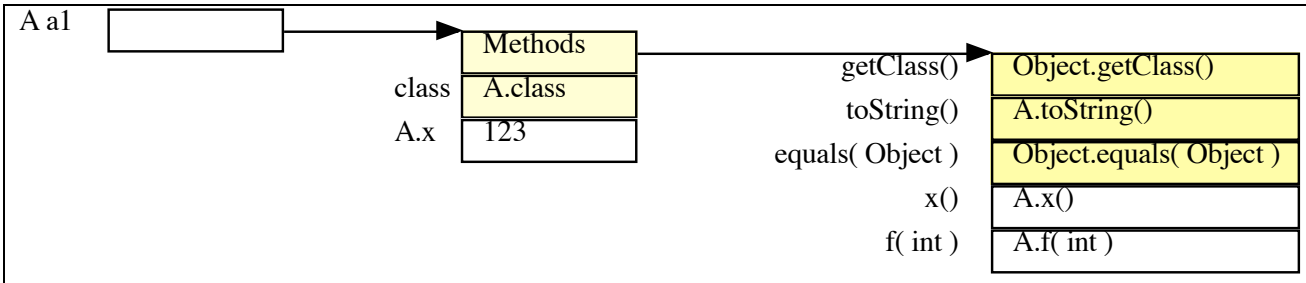
Continued ...

Print Name and Student ID _____

5. Implementation of object oriented languages [18 Marks]

Use the Java program in the **Appendix For Question 5**.

(a) Draw a diagram showing the data structures (variable, field table, method table, etc) created for the variables `a1`, `a2`, `c`, `a`, within the method `Main.main`. Shared data structures should be drawn only once.



(12 marks)

Continued ...

Print Name and Student ID _____

(b) Indicate the output generated by the method Main.main.

```
Constructor A( 123 )









a1.getClass() = class A

a2.getClass() =

a.getClass() =

c.getClass() =

a1 = A.toString()

a2 =

a =

c =

a1.x = 123

a2.x =

a.x =

c.x =

a1.x() = 123

a2.x() =

a.x() =

c.x() =

a1.f( 'A' ) = A.f( 65 )

a.f( 'A' ) =

c.f( 'A' ) =
```

(6 marks)

Print Name and Student ID _____

6. Code generation [15 Marks]

(a) Consider the the B-- language.

What action has to be performed at run time to initialise an object?

(2 marks)

What are the $pv, $ra, $fp, $ip, and $nip registers used for, and when are they used?

$pv

$ra

$fp

$ip

$nip

(5 marks)

(b) Consider the following program written in the B-- language.

```
void inc( ^int dest; int change; )
     begin
          dest^ = dest^ + change;
     end
int b = 5;
int c = 3;
inc( &b, c );
printf( "%d\n", b );
```

Print Name and Student ID _____

Indicate the Alpha assembly language likely to be generated for the lines

```
dest^ = dest^ + change;
```

(3 marks)

and

```
inc( &b, c );
```

(5 marks)

An appendix is provided describing common Alpha instructions.

_____End of Questions_____

Continued ...

Print Name and Student ID _____

This Page is left blank for questions that overflow

Continued ...

Print Name and Student ID _____

This Page is left blank for questions that overflow

Print Name and Student ID _____

This Page is left blank for questions that overflow

End of Questions, Appendices follow

Appendices

This Page is left blank for formatting purposes

Appendices Continued ...

Appendix For Question 2

Grammar

```
terminal String
          IDENT,    EXPANDSTO,     SEMICOLON,     OR,  COLON, ERROR;
          //        ::=            ;              |    :

non terminal Rule, RHSList, SymbolList, Symbol;

start with Rule;

Rule::=
          IDENT EXPANDSTO RHSList SEMICOLON          //   Rule 1
     ;

RHSList::=
          SymbolList                                 //   Rule 2
     |
          RHSList OR SymbolList                      //   Rule 3
     ;

SymbolList::=
          /* Empty */                                //   Rule 4
     |
          Symbol SymbolList                          //   Rule 5
     ;

Symbol::=
          IDENT                                      //   Rule 6
     |
          IDENT COLON IDENT                          //   Rule 7
     ;
```

Appendix For Question 2 Continued On Next Page ....

Appendix For Question 2 Continued ...

Action Table

```
From state #0
     IDENT:SHIFT(2)
From state #1
     EOF:SHIFT(14)
From state #2
     EXPANDSTO:SHIFT(3)
From state #3
     IDENT:SHIFT(7) SEMICOLON:REDUCE(4) OR:REDUCE(4)
From state #4
     SEMICOLON:SHIFT(11) OR:SHIFT(12)
From state #5
     SEMICOLON:REDUCE(2) OR:REDUCE(2)
From state #6
     IDENT:SHIFT(7) SEMICOLON:REDUCE(4) OR:REDUCE(4)
From state #7
     IDENT:REDUCE(6) SEMICOLON:REDUCE(6) OR:REDUCE(6)
     COLON:SHIFT(8)
From state #8
     IDENT:SHIFT(9)
From state #9
     IDENT:REDUCE(7) SEMICOLON:REDUCE(7) OR:REDUCE(7)
From state #10
     SEMICOLON:REDUCE(5) OR:REDUCE(5)
From state #11
     EOF:REDUCE(1)
From state #12
     IDENT:SHIFT(7) SEMICOLON:REDUCE(4) OR:REDUCE(4)
From state #13
     SEMICOLON:REDUCE(3) OR:REDUCE(3)
From state #14
     EOF:REDUCE(0)
```

Reduce (Go To) Table

```
From state #0:
     Rule:GOTO(1)
From state #1:
From state #2:
From state #3:
     RHSList:GOTO(4)
     SymbolList:GOTO(5)
     Symbol:GOTO(6)
From state #4:
From state #5:
From state #6:
     SymbolList:GOTO(10)
     Symbol:GOTO(6)
From state #7:
From state #8:
From state #9:
From state #10:
From state #11:
From state #12:
     SymbolList:GOTO(13)
     Symbol:GOTO(6)
From state #13:
From state #14:
```

Appendices Continued ...

Appendix For Question 4

```
1 class List
2  begin
3      int value;
4      ^List next;
5  end
6
7 int freeNode = 0;
8 [ 20 ]List nodeHeap;
9
10 ^List new( int value; ^List next; )
11     begin
12          ^List node = nodeHeap[ freeNode++ ];
13          node.value = value;
14          node.next = next;
15          return node;
16     end
17
18 void printList( ^List a; )
19     begin
20          printf( "{ " );
21          while a != null do
22               printf( "%d", a.value );
23               a = a.next;
24               if a != null then
25                    printf( ", " );
26               end
27          end
28          printf( " }" );
29     end
30
31 void printlnList( ^List a; )
32     begin
33          printList( a );
34          printf( "\n" );
35     end
36
```

Appendix For Question 4 Continued On Next Page ....

Appendix For Question 4 Continued ...

```
37 void merge( int level; ^List source1, source2; ^^List dest; )
38    begin
39         if source1 == null then
40              dest^ = source2;
41
42         elif source2 == null then
43              dest^ = source1;
44
45         elif source1.value < source2.value then
46              dest^ = new( source1.value, null );
47              merge( level + 1, source1.next, source2, &dest^.next );
48
49         elif source1.value > source2.value then
50              dest^ = new( source2.value, null );
51              merge( level + 1, source1, source2.next, &dest^.next );
52
53         elif source1.value == source2.value then
54              dest^ = new( source1.value, null );
55              merge( level + 1, source1.next, source2.next, &dest^.next );
56
57         end
58    end
59
60 ^List source1 =
61    new( 1,
62    new( 5,
63    new( 8,
64    new( 9,
65         null ) ) ) );
66
67 ^List source2 =
68    new( 1,
69    new( 6,
70         null ) );
71
72 ^List dest = null;
73
74 merge( 0, source1, source2, &dest ); // Inside this invocation
75
76 printlnList( source1 );
77 printlnList( source2 );
78 printlnList( dest );
79
```

Appendix For Question 5

```
class A {
     public int x = 111;
     public A() { System.out.println( "Constructor A()" ); }
     public A( int x ) {
          System.out.println( "Constructor A( " + x + " )" );
          this.x = x;
          }
     public int x() { return x; }
     public String toString() { return "A.toString()"; }
     public String f( int z ) { return "A.f( " + z + " )"; }
     }

class B extends A {
     public int x = 222;
     public B() { System.out.println( "Constructor B()" ); }
     public B( int x ) {
          System.out.println( "Constructor B( " + x + " )" );
          this.x = x;
          }
     public String toString() { return "B.toString()"; }
     public String f( int z ) { return "B.f( " + z + " )"; }
     public int x() { return x; }
     }

class C extends B {
     public int x = 333;
     public C() { System.out.println( "Constructor C()" ); }
     public C( int x ) {
          System.out.println( "Constructor C( " + x + " )" );
          this.x = x;
          }
     public String f( char z ) { return "C.f( \'" + z + "\' )"; }
     }
```

Appendix For Question 5 Continued On Next Page ....

Appendix For Question 5 Continued ...

```
public class Main {
     public final static void main( String[] arg ) {
          A a1 = new A( 123 );
          A a2 = new B( 456 );
          C c = new C( 789 );
          A a = c;

          System.out.println( "a1.getClass() = " + a1.getClass() );
          System.out.println( "a2.getClass() = " + a2.getClass() );
          System.out.println( "a.getClass() = " + a.getClass() );
          System.out.println( "c.getClass() = " + c.getClass() );

          System.out.println( "a1 = " + a1 );
          System.out.println( "a2 = " + a2 );
          System.out.println( "a = " + a );
          System.out.println( "c = " + c );

          System.out.println( "a1.x = " + a1.x );
          System.out.println( "a2.x = " + a2.x );
          System.out.println( "a.x = " + a.x );
          System.out.println( "c.x = " + c.x );

          System.out.println( "a1.x() = " + a1.x() );
          System.out.println( "a2.x() = " + a2.x() );
          System.out.println( "a.x() = " + a.x() );
          System.out.println( "c.x() = " + c.x() );

          //   'A' is 65
          System.out.println( "a1.f( \'" + 'A' + "\' ) = " + a1.f( 'A' ) );
          System.out.println( "a.f( \'" + 'A' + "\' ) = " + a.f( 'A' ) );
          System.out.println( "c.f( \'" + 'A' + "\' ) = " + c.f( 'A' ) );
          }
     }
```

Commonly used Alpha instructions

Integer operate instructions

```
Opcode $regA, $regB, $regC
```

intReg[regC] = intReg[regA] op intReg[regB]

```
Opcode $regA, constantB, $regC
```

The constant is an 8 bit unsigned constant.
intReg[regC] = intReg[regA] op constantB

Arithmetic integer operate instructions

addq	add	+
subq	subtract	-
mulq	multiply	*
divq/divqu	divide, signed/unsigned	/
modq/modqu	modulo, signed/unsigned	%
s8addq	scaled 8 add	8*operandA+operandB

Shift integer operate instructions

sll	shift left logical	<<
srl	shift right logical	>>>
sra	shift right arithmetic	>>

Compare integer operate instructions

cmpeq	compare equal	==
cmplt/cmpult	compare less than signed/unsigned	<
cmple/cmpule	compare less than or equal signed/unsigned	<=

Logical integer operate instructions

and	and	&
bic	bit clear	& ~
bis/or	bit set/or	|
eqv/xornot	equivalent/exclusive or not	^ ~
ornot	or not	| ~
xor	exclusive or	^

Appendices Continued ...

Memory instructions

```
Opcode $regA, displacement($regB)
Opcode $regA, ($regB)
Opcode $regA, constant
```

The displacement or constant is a 16 bit signed constant.

Load address instruction

intReg[regA] = displacement + intReg[regB]

lda	load address

Load memory instructions

intReg[regA] = Memory[displacement + intReg[regB]]

ldq	load quadword
ldl	load longword
ldbu	load byte unsigned

Store memory instructions

Memory[displacement + intReg[regB]] = intReg[regA]

stq	store quadword
stl	store longword
stb	store byte

Branch instructions

Conditional branch instructions

```
Opcode $regA, destination
```

if (condition holds for intReg[regA])
    programCounter = destination

beq	branch equal
bne	branch not equal
blt	branch less than
ble	branch less than or equal
bgt	branch greater than
bge	branch greater than or equal
blbs	branch low bit set
blbc	branch low bit clear

Unconditional branch instructions

```
Opcode destination;
```

programCounter = destination // br

intReg[ra] = programCounter // bsr
programCounter = destination

br	branch
bsr	branch to subroutime

Appendices Continued ...

Jump instruction

```
Opcode ($regA);
```

programCounter = intReg[regA] // jmp

intReg[ra] = programCounter // jsr
programCounter = intReg[regA]

jmp	jump
jsr	jump to subroutine

Return instruction

programCounter = intReg[ra]

ret	return

Callpal instruction

```
call_pal constant;
```

The constant is a 26 bit constant.

call_pal	call PALcode

Pseudoinstructions

Load immediate

```
ldiq $regA, constant
```

The constant is a 64 bit constant.
intReg[regA] = constant

ldiq	load immediate quadword

Clear

```
clr $regA
```

intReg[regA] = 0

clr	clear

Unary pseudoinstructions

```
Opcode $regB, $regC
```

intReg[regC] = op intReg[regB]

```
Opcode constantB, $regC
```

The constant is an 8 bit unsigned constant.
intReg[regC] = op constantB

mov	move
negq	negate

_____End of Appendices_____