This exam permits one sheet of notes: NO OTHER BOOKS, NOTES OR EQUIPMENT (besides manual writing instruments, non-data storage watches and refreshments) are permitted. There are 7 parts and 8 pages. Make sure that you have them all before you begin. The duration is 2 hours.

PART 1 (8 points, clarity and simplicity will count)

(a) (4 points) Write a C function that will print a Java Virtual Machine 32-bit integer constant as required in Project 4, with given index, standard tag of 03 and the 4 bytes in Big Endian (most significant byte first) order. For example,

```c
printJavaIntConst( 10, 0xab34cd29 );
```

should print

```
000a: 03 ab 34 cd 29
```

(including a trailing newline).

Specifications and hints: Shift and mask operations must be used so the code is byte order independent and unconditional. Printf format "%04x:" will print the index. Format " %02x" (note the space) will print the integer value n in the range 0 ≤ n ≤ 255 in hexadecimal format that includes a leading space and a leading zero for values less than 16.

```c
#include <stdio.h>
void printJavaIntConst( unsigned short index, unsigned long value )
{
    /* You write --- */
}
```

(b) (4 points) Redo part (a) so the byte order is Little Endian (least significant first). For example, the output from `printNonJavaIntConst( 10, 0xab34cd29 );` should be

```
000a: 03 29 cd 34 ab
```

```c
#include <stdio.h>
void printNonJavaIntConst( unsigned short index, unsigned long value )
{
    /* You write --- */
}
```
PART 2 (25 points) Demonstrate the operation of two pass assemblers, using the Java assembly language of this course.

**First:** On the first pass side, mark each and every *forward reference* by drawing an arrow (→) from the place where the symbol’s value is needed to the place where it is defined when its value would be unknown in the first pass.

**Second:** Perform the first pass: Make use of the Java reference material on the next page. (1) Fill in the blanks of the code area Program Counter (PC) column and the local variable counter column with the values of these counters at the start of processing each input line. (2) Write in the symbol table column any modification to symbol table data that occurs when each input line is processed.

**Third:** Do the second pass by filling in the code emission column. Write each emitted byte in 2 digit hexadecimal form. **Write all calculations that the assembler does below the table.** Again, the reference material is needed for this job.

<table>
<thead>
<tr>
<th>First Pass</th>
<th>Second Pass</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Current code area PC</strong></td>
<td><strong>Current local var. area counter</strong></td>
</tr>
<tr>
<td>0 0</td>
<td>.locals</td>
</tr>
<tr>
<td>0 1</td>
<td>.code</td>
</tr>
<tr>
<td>0 1</td>
<td>A: bipush 3</td>
</tr>
<tr>
<td>2 1</td>
<td>iconst_1</td>
</tr>
<tr>
<td>3 1</td>
<td>goto STRT</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Mnemonic</td>
<td>Length (bytes)</td>
</tr>
<tr>
<td>----------</td>
<td>---------------</td>
</tr>
<tr>
<td>bipush</td>
<td>2</td>
</tr>
<tr>
<td>iconst_1</td>
<td>1</td>
</tr>
<tr>
<td>goto</td>
<td>3</td>
</tr>
<tr>
<td>iload</td>
<td>2</td>
</tr>
<tr>
<td>istore</td>
<td>2</td>
</tr>
<tr>
<td>iconst_m1</td>
<td>1</td>
</tr>
<tr>
<td>iadd</td>
<td>1</td>
</tr>
<tr>
<td>dup</td>
<td>1</td>
</tr>
<tr>
<td>ifgt</td>
<td>3</td>
</tr>
</tbody>
</table>

PART 3 (10 points) Simulate the Java Virtual Machine on the example of the previous problem.

When the instructions given are all finished, the value of local variable BILL is _____ and the value of local variable HLRY is _____.

The data computations that the Java Virtual Machine would make to get these answers must be shown below:
PART 4 (20 points) This problem is to construct the state and action table for a scanner that processes a simplified form of the variable=value strings that can appear inside of html “anchors”. For example, the anchor

<br>
</<applet codebase="1.0.2/" code=CardTest.class width=400 height=300 >

contains 4 different variable=value strings. Your scanner will be used to process exactly ONE variable=value string each time it is called.

CAUTION: The language is different from the languages in your projects. Write your answer in pencil (so you can erase) on the next page.

- Don’t worry about illegal characters. The characters have been classified into 5 disjoint classes: REGULAR, EQUAL, QUOTE, SPECIAL, and WS.
- Only REGULAR characters can appear in a variable or in a non-quoted value.
- Neither the variable nor the a non-quoted value can be the empty string.
- When a value begins and ends with QUOTE characters, there can be any number (including 0) of any kind of characters between the QUOTEs except for QUOTEs. Thus, the empty string can be expressed as a value for example by options=""
- The quotes are considered to be part of the value.
- Assume the scanner is started on the first character of the variable.
- Assume that each variable=value string will be followed by a WS kind of character. (This is not a realistic assumption but it is made for the purposes of your final examination.)
- On scanning the WS character following a legal variable=value string, the scanner should go to the state done. As soon as an error is detected, the scanner should go to the state error.
- Actions:
  1. On scanning the first character of each variable and value, mark_act should be called.
  2. At the character after each variable or value, variable_act or value_act should be called, respectively.

<table>
<thead>
<tr>
<th>Examples of legal inputs</th>
<th>Examples of erroneous inputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>size=90</td>
<td>Novall=</td>
</tr>
<tr>
<td>funstring=&quot;=&quot;</td>
<td>eqsymb=</td>
</tr>
<tr>
<td>dothis=&quot;var=987&quot;</td>
<td>ab=cd=e</td>
</tr>
<tr>
<td>EmTee=&quot;&quot;</td>
<td>a bee</td>
</tr>
<tr>
<td></td>
<td>leadchars=&quot;spaces&quot;</td>
</tr>
<tr>
<td></td>
<td>endjunk=&quot;good=part&quot; junk</td>
</tr>
<tr>
<td></td>
<td>=dontKnow</td>
</tr>
</tbody>
</table>
typedef enum { REGULAR, EQUAL, QUOTE, SPECIAL, WS } ch_type;
typedef enum { start, in_variable, bef_value, in_value, in_quoted, after_quoted
done, error } state_type;
typedef void (*pAction)( void );
extern pAction mark_act, variable_act, value_act;
typedef struct { ch_type kind; state_type next; pAction action; } transition;

transition start_row[] = {
    { REGULAR, },
    { EQUAL, },
    { QUOTE, },
    { SPECIAL, },
    { WS, });

transition in_variable_row[] = {
    { REGULAR, },
    { EQUAL, },
    { QUOTE, },
    { SPECIAL, },
    { WS, });

transition bef_value_row[] = {
    { REGULAR, },
    { EQUAL, },
    { QUOTE, },
    { SPECIAL, },
    { WS, });

transition in_value_row[] = {
    { REGULAR, },
    { EQUAL, },
    { QUOTE, },
    { SPECIAL, },
    { WS, });

transition in_quoted_row[] = {
    { REGULAR, },
    { EQUAL, },
    { QUOTE, },
    { SPECIAL, },
    { WS, });

transition after_quoted_row[] = {
    { REGULAR, },
    { EQUAL, },
    { QUOTE, },
    { SPECIAL, },
    { WS, });}
PART 5 (20 points)

typedef enum {OK, MULTIPLY_DEFINED} insert_results;
insert_results insert( char *symbol, long int value );
/* insert will make its own copy of each new symbol inserted */

typedef enum {OK, NOT_DEFINED} retrieve_results;
retrieve_results retrieve( char *symbol, long int *pvalue );
/* *pvalue is written only if the string symbol is a defined symbol. */

The above defines the interface to a minimal symbol table. It is used with the strategy that only defined symbols are inserted in the symbol table. Choose a simple enough data structure so you can completely and correctly implement a symbol table in C (or in C++). Write the code. Clearness, conciseness and simplicity will count as well as correctness. If you are unsure of the calling sequence or name of some standard library function, explain clearly; no credit will be deducted if your intention is clear enough. The two functions must be coordinated.

/* Global variable definitions here: */

insert_results insert( char *symbol, long int value )
{
    
}
retrieve_results retrieve( char *symbol, long int *pvalue )
{
    
}
PART 6 (9 points)

1. (3 points) Briefly describe one specific and non-trivial kind of data inside of /kernel/genunix that is similar to other ELF files that were in your Unix account or you worked with in this course, and explain what that data is for.

2. (3 points) The 402sparc simulator set up its simulated virtual memory mapping tables prior to the start of simulation. It was impossible for the Sparc instructions that you implemented to change them. On a real Sparc processor, under what conditions can the virtual memory mapping tables be modified?

3. (3 points) One Unix file abstraction is that a file appears to programmers as an array of bytes. The process running the program can access any byte in the file by calling fseek, fread or fwrite appropriately (or their lower level variants lseek, read and write). Why must system calls be issued from within these library routines rather than just ordinary instructions or subroutine calls, even though the file “is” (in some sense) just an array of bytes? (Hint: Can user mode processes access disks the way 402sparc test programs could access the Simple Output Device?)

4. (3 points, extra) What do you think the operating system programmers did in order to make the machine instructions that handle system calls get into the kernel?
PART 7 (8 points) The Sparc SAVE instruction both decrements CWP and functions as an ADD instruction. “The source registers ... are from the [old] CWP window, while the result is written into a register in the CWP-1 window.” Subroutines typically execute instructions like save %sp,-120,%sp soon after they are entered.

1. (4 points) What does the SAVE instruction save?

2. (2 points) What is the specific use of SAVE’s addition capability?

3. (2 points) How does the called subroutine get the proper value for its frame pointer %fp, which should point to the base (numerically highest) address of that subroutine’s stack frame?