This is an open book and notes exam. There are 6 parts and 8 pages. Make sure that you have them all before you begin. The duration is 1 hour.

PART 1 (10 points)
The program

```c
int main( int argc, char *argv[], char *env[] )
{
    ... }
```

is compiled and linked with “gcc main.c” and run with

```bash
> a.out main.c -nodebug
```

Assume that the environment (as listed by “printenv”) is

ED=joe
UNAME=sdc

Complete the “box and arrow” diagram that shows how the command line arguments and environment are made accessible to `main()` under Solaris UNIX. Show all relevant null pointers.
PART 2 (15 points)
An array of function pointers can be used to bind “action functions” to “events” so that which action function is called when a particular event occurs can vary during runtime.
Assume that a legal event number is an integer in the range 0 to MAX_EVENT.
Assume that an action function should be called with the event number as its one argument and it should return nothing. One action function is defined by

```c
void default_Action( int event )
{ /* Do nothing */ }
```

The type “pointer to action function” is declared by

```c
typedef void ( * pAction_type ) ( int );
```

Assume that the current bindings of events to action functions is stored in the array of function pointers

```c
extern pAction_type pActions[MAX_EVENT];
```

1. Write the function below that will take an event number and do the following:

(a) Check if the event number is legal. If it is not, call bad_event (declared below) with the event number as its one argument. Then call exit(1).

```c
extern void bad_event( int event );
```

(b) Call the action function whose pointer has been stored in the position of pActions given by the event number. Then return.

```c
void dispatch_event( int event )
{
    if (event<0 || event>= MAX_EVENT) /* check for legality of input */
    {
        bad_event(event);
        exit(1);
    }
    else {
        (*pAction[event])(event); /* one solution */
        /* alternate solution : pAction[event](event); */
        /* alternate solution : (pAction[event])(event); */
        /* alternate solution : (*pAction[event])(event); */
    }
}
```

This problem is continued on the next page
2. Write the function below that will replace the old binding for an event with a new binding. It should:

(a) Check for event legality (as dispatch_event did.)
(b) Replace the old action function pointer for event number event with the given new one.
(c) If the pointer to the new action function is NULL then put in default_Action as the new binding.
(d) Function bind should return the old pointer value that used to be stored in the indicated position of the array pActions. (This way, the caller can save that pointer so it may be restored by a later call to bind.)

```c
pAction_type bind( int event, pAction_type pNewAction )
{
    pAction_type old_action; /* new variable to hold the old pointer value */

    if (event < 0 || event >= MAX_EVENT) /* check for legality */
    {
        bad_event(event);
        exit(1);
    }
    else
    {
        old_action = pAction[event]; /* save old pointer value */

        if (pNewAction==NULL)
            pAction[event] = &default_Action;
        else pAction[event] = pNewAction;

        return(old_action);
    }
}
```
PART 3 (15 points) Consider the following Makefile:
(What `cat` and `echo` do is described in the comments.)

```
SRCS = G H K
RESULTS = A B C D E F

all : A B C D
      echo Hello  #Print Hello when run.
A : E
   cat E > A   #Copy file E to file A.
B : F E
   cat F E > B #Concatenate files F, E, write result to B.
C : F
   cat F > C
E : G H
   cat G H > E
F : H K
   cat H K > F
D :
   lpr $(SRCS)
clean :
   rm -f $(RESULTS)
```

1. Locate the first `rule` in this Makefile. Circle and label (a) the `target`, (b) the `dependencies`, and (c) the `command` of this rule.
   **This problem is continued on the next page**

   (a) all
   (b) A B C D
   (c) echo Hello
2. The following (obtained by `ls -l`) shows all the files in the working directory together with their modification times.

```
-rw-r--r-- 1 sdc 38 Mar 12 12:19 A
-rw-r--r-- 1 sdc 76 Mar 12 12:19 B
-rw-r--r-- 1 sdc 38 Mar 12 12:17 C
-rw-r--r-- 1 sdc 38 Mar 12 12:19 E
-rw-r--r-- 1 sdc 38 Mar 12 12:17 F
-rw-r--r-- 1 sdc 19 Mar 12 12:22 G
-rw-r--r-- 1 sdc 19 Mar 12 12:11 H
-rw-r--r-- 1 sdc 19 Mar 12 12:11 K
-rw-r--r-- 1 sdc 370 Mar 12 12:23 makefile
```

Based on this information, write here exactly what is printed to the terminal when the command “make all” is given:

```
cat G H > E
cat E > A
cat F E > B
cat F > C (or nothing, depending on exact times of F and C)
lpr G H K
```

Hello

PART 4 (20 points)
Show how the binary form of these dlx402 instructions is decoded. Write after each instruction a precise description of the changes of register and/or memory values that it causes. Assume the instructions are loaded beginning at address 0 and are executed sequentially, and the memory and registers are initialized as in project3.

```
0x2003ffff
001000 00000 00011 1111 1111 1111 1111
  08 rs rd  16 bit immediate
addi r0  r3  0xffff
```

Makes value in r3 be 0xffffffff (by adding r0 + 0xffff with sign extension)

```
0x00632821
000000 00011 00011 00101 000 00 10 0001
  00 rs1 rs2 rd  2 1
special r3  r3  r5  addu
```

Makes value in r5 be 0xffffffff (by adding r3 + r3 )

```
0xac630011
101011 00011 00011 0000 0000 0001 0001
  20+8+3 rs rd  16 bit displacement
sw r3  r3  0x0011
```

Stores value 0xffffffff (from r3) into memory

at address 0x00000010, since r3+0x0011=0xffffffff+0x0011

5
PART 5 (20 points) This problem is to construct the state and action table for a line scanner that
processes lines of a simplified generic assembly language. CAUTION: The language is different from
the language in your project. 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 4 disjoint classes:
R, WS, COLON and NL.

Assume the input line will end with NL.

Every label, operator and operand is one or more R (“ordinary”) characters.

A legal input line is either null (zero or more WS characters, of course terminated by NL), or is
described by the following sequence:

1. An optional label that must begin (if present) at the first column. If there is no label the first
character must be WS. The label is delimited by a COLON at the end.

2. A mandatory operator. The operator must begin after the first column or after the COLON
of a label. There are zero or more WS characters between the first column or the COLON
and the beginning of the operator.

3. Zero or more operands. They are separated from the operator and from each other by one
or more WS characters. There might (or might not) be WS characters after the last operand
(before the NL).

Actions:

1. At the beginning of each label, operator and operand, mark_act should be called.

2. At the character after each label, operator and operand, label_act, operator_act, and
operand_act should be called, respectively.

Ending:

1. If the line is “null”, the state machine should go to state “null” when the NL is processed.

2. For non-null, correct input, the ending state should be “done.”

3. As soon as the scanner detects a violation of the above rules, the state machine should go to
state “error.”

It may help some of you to see an (extended) grammar for the language:

<table>
<thead>
<tr>
<th>line</th>
<th>Examples of legal lines</th>
<th>Examples of erroneous lines</th>
</tr>
</thead>
<tbody>
<tr>
<td>line</td>
<td>null_line</td>
<td>ordinary_line</td>
</tr>
<tr>
<td>null_line</td>
<td>WS* NL</td>
<td></td>
</tr>
<tr>
<td>ordinary_line</td>
<td>( label:</td>
<td>WS* operator (WS* operand)*</td>
</tr>
<tr>
<td></td>
<td>WS* NL</td>
<td></td>
</tr>
<tr>
<td>label, operator, operand</td>
<td>R*</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
typedef enum { WS, R, COLON, NL } ch_type;
typedef enum { start, in_label, bef_operator, in_operator, 
    bef_operand, in_operand, done, null, error } state_type;
typedef ... (*pAction) ( ... );
extern pAction no_act, mark_act, label_act, operator_act, operand_act;
typedef struct {ch_type symb; state_type new_state; pAction action;} transition;

transition start_row[] = {
    { WS, bef_operator , no_act },
    { R, in_label , mark_act },
    { COLON, error , no_act }, /*actions in cases of*/
    { NL, null , no_act }}, /*error or null*/
    /*were not graded*/

transition in_label_row[] = {
    { WS, error , no_act },
    { R, in_label , no_act },
    { COLON, bef_operator , label_act },
    { NL, error , no_act }};

transition bef_operator_row[] = {
    { WS, bef_operator , no_act },
    { R, in_operator , mark_act },
    { COLON, error , no_act },
    { NL, null or error , no_act }}; /*bug in problem*/
    /*not graded*/

transition in_operator_row[] = {
    { WS, bef_operand , operator_act },
    { R, in_operator , no_act },
    { COLON, error , no_act },
    { NL, done , no_act });

transition bef_operand_row[] = {
    { WS, bef_operand , no_act },
    { R, in_operand , mark_act },
    { COLON, error , no_act },
    { NL, done , no_act }};

transition in_operand_row[] = {
    { WS, bef_operand , operand_act },
    { R, in_operand , no_act },
    { COLON, error , no_act },
    { NL, done , operand_act }};
PART 6 (20 points)

(a.) Write a function that returns the first source operand for dlx402 instruction to be simulated. Remember that this operand is designated by bits 6-10 of the instruction. Remember also the special properties of register 0.

Solution 1:

```c
unsigned long get_rs1( unsigned long instrucion, unsigned long regs[] )
{
    /* Assume register i is implemented by regs[i] */
    /* r0 is implemented by keeping regs[0] == 0 */
    int index;
    index = ( instrucion >> 21 ) & 0x0000001f;
    return regs[index];
}
```

Solution 2:

```c
unsigned long get_rs1( unsigned long instrucion, unsigned long regs[] )
{
    /* Assume register i is implemented by regs[i] */
    /* regs[0] is not used for r0 */
    int index;
    index = ( instrucion >> 21 ) & 0x0000001f;
    if (index != 0)
        return regs[index];
    return 0;
}
```

(b.) Write a function that writes back the result of an immediate (or memory load) dlx402 instruction into a register. Remember that the destination register for this type of instruction is designated by bits 11-15. Be sure your answer here is consistent with your answer above.

Solution 1:

```c
void write_back_imm_type( unsigned int instruction,
                unsigned long value,    /*value to write back*/
                unsigned long regs[] )    /*simulated reg. array*/
{
    /* Assume register i is implemented by regs[i] */
    /* r0 is implemented by keeping regs[0] == 0 */
    int index;
    index = ( instruction >> 16 ) & 0x0000001f;
    if (index != 0)
        regs[index] = value;
}
```

Solution 2:

```c
void write_back_imm_type( unsigned int instruction,
                unsigned long value,    /*value to write back*/
                unsigned long regs[] )    /*simulated reg. array*/
```
{ /* Assume register i is implemented by regs[i] */
/* regs[0] is not used for r0 */
int index;
index = ( instruction >> 16 ) & 0x000000ff;
regs[index] = value;
}