This is an open book examination. You are free to consult any texts or notes you have with you while completing the exam. Please write your answers to the questions in the spaces provided in the exam booklet. Extra paper is available for scrap work.

1) Write 68000 assembly language code for the following fragment of Pascal code assuming that the arrays x and y are declared as global variables by a declaration of the form:

```
var x,y : array [ 1 .. 100 ] of integer;
```

If possible, use auto-increment/decrement addressing to speed accesses to the array elements.

```
for i := 1 to cur_size do
  x [ i ] := y [ cur_size + 1 - i ];
```

2) In class, I have shown you that the code for a procedure call on the 68000 typically looks like:

```
  < code to push actual parameter values>
JSR  proc-name
ADD.L    #4*number-of-parameters,A7
```

and that a typical procedure looks like:

```
LINK      A6,# - number- of-bytes-needed -for-locals
  ...
    < code for procedure's body >

UNLK    A6
RTS
```

a) Suppose that the LINK and UNLK instructions were not included in the 68000's instruction set. Show what the typical call sequence and procedure code would look like.

b) Herb Gritz, a mythical CS student, finds it disconcerting that negative displacements must be used to reference local variables stored in a procedure's activation record. He would like to replace the "standard" 68000 procedure prologues and eplilogues shown above with new code that would leave the frame pointer register (A6) pointing to the word in the frame with the lowest address. Of course, he would like the code to be as efficient as possible. He is willing to change the layout of the activation record if that will help.

Please show me (and Herb) a "typical" procedure call sequence and a "typical" procedure body that would meet his requriements. If the format of the activation record produced by your code is different from the standard format, show the layout your code assumes. You may use the LINK and UNLK instructions in your code if you find them helpful.

3) We have seen in class, that 1-bit adders are designed so that n of them can be grouped together to form a circuit capable of adding two n-bit numbers together. In such a circuit, all of the adders take one bit from each of the two n-bit numbers being added as input. It is also possible (though probably not reasonable) to design an n-bit adder using only one 1-bit adder. Such a circuit would have to add the way humans do --- i.e. it would add one pair of digits at a

time starting with the rightmost digits and working to the left. This algorithm is described more precisely by the following pseudo-code:

```
carry := 0;
result := 0;
readln (x,y);

for p := 1 to number-of-bits  do
     begin
      result := result div 2;

      high order bit of result :=
        sum produced by adding together carry and the low order bits of x and y ;
      carry := carry produced by adding together carry and the low order bits of x and y ;

      x := x div 2;
      y := y div 2;
     end;
```

Obviously, the result of 'adding together carry and the low order bits of x and y' may be a two digit binary number. In the algorithm, the phrase 'sum produced by adding ...' refers to the low order bit of this result and the 'carry produced...' refers to the high order bit.

What I would like you to do is design a circuit that performs addition based on this algorithm. Your complete circuit should take two numbers to be added as input and produce an output and a carry. In addition there should be a 'set' control line telling the circuit when to input new x and y values. You need not include any control signal that indicates when the addition is complete. You can instead assume that any user of your circuit will know that the correct output will appear n full cycles after the addition begins (where n is the number of bits). In fact, you need not even leave the correct result on the output line for more than one cycle.

In some portions of the circuit it may be critical to show exactly how the lines in a bus are connected to some component. In such cases, assume that you are building a 4-bit version of the circuit. That is show 4 wires in each data bus and assume that registers hold 4-bits.

HINTS: 1) The use of 'div' in the algorithm suggests the use of a shifter (as in the multiply circuit shown in class). You can do without a shifter if you connect the inputs and output of registers in the appropriate way. Doing this will make the circuit much simpler.

2) To avoid making this an all-or-nothing problem, let me give you a few options to ensure I have plenty of room to give partial credit. If you have trouble solving the complete problem make sure that you at least show me the data flow diagram for the circuit. You can get another step closer to full credit by showing the circuitry needed to execute the loop shown in the algortihm while ignoring the question of how the x and y values get initialized (this is how I presented the multiplier circuit in class). Finally, at the other extreme, to give a perfect answer, you should not only produce a correct circuit but you should use as few components as possible. After all, the only possible reason for using such an adder instead of chaining together a collection of 1-bit adders would be a desire to use fewer components.