I. Stack Concept
A. LIFO = Last In, First Out
B. Top of stack is the only part of the data structure that the user can see and use.
C. Pop = Remove data from top of stack.
D. Push = Add data to top of stack.
E. Top = Copy data from top of stack, but do not remove it from the stack.
F. Good for storing information that is processed in a nested manner.
   1. Ex: Store order of parts from a disassembly using Push, then Pop will provide the proper order of reassembly.
   2. Ex: Store the letters of a word using Push, then printing Popped letters will provide the word in reverse order.
   3. Ex: Procedure parameter storage for procedure calls. Recursion is a good example of this.

II. Computer Stacks
A. Stack pointer, SP, used to store the address of the top-of-stack.
   1. 12 bits in CUSP.
B. Stack is composed of consecutive memory words.
C. Stack moves down from upper memory.
D. Stack overflow occurs when a Push would overwrite memory that is already allotted for other purposes. This is a real problem with recursive programs.
E. Push = Decrement SP, then store data at address pointed to by SP. Combines two instructions to save time and space.
F. Pop = Retrieve data from address pointed to by SP, and then increment SP. Does not destroy the retrieved stack data. Combines two instructions to save time and space.
G. CUSP Instructions
   1. PSH = Push data at addressed memory on stack. Used with any addressing mode.
   2. PSHA = Push accumulator on stack.
   3. PSHX = Push XR on stack.
   4. PSHF = Push FR on stack.
   5. POP = Pop data from stack into addressed memory. Used with all but immediate mode.
   6. POPA = Pop from stack into accumulator.
   7. POPX = Pop from stack into XR. OV affected.
   8. POPF = Pop from stack into FR. OV affected.
   9. LDS = Load SP. OV affected.
   10. STS = Store SP.
   11. ADS = Add to SP. Used to create space for variables without initialization. OV and EQ affected.
   12. SBS = Subtract from SP. Used to delete temporary variables. OV and EQ affected.
   13. CMS = Compare SP. EQ and LT affected.
   14. TAS = Transfer accumulator to SP. OV effected.
   15. TSA = Transfer SP to accumulator.
   16. TXS = Transfer XR to SP.
   17. TSX = Transfer SP to XR.
   18. TFS = Transfer FP to SP
   19. TSF = Transfer SP to FP.

III. Frame Register (FP)
A. Frame addressing mode.
   1. Contents of FP are added to the operand before the rest of the address computation is done.
   2. Indicated by placing the "!" character next to the operand. "!" must be preceded by a space, and may be followed by space[s].
   3. Used in conjunction with other modes.
      a) LDX# ! 23 means XR := 23 + FP.
b) $LDX ! 100$ means $XR := Memory[100 + FP]$.

c) $LDX+ ! 101$ means $XR := Memory[(101 + FP) + XR]$.

4. Relative addressing value = addressing value of an instruction without (not before as the book indicates) the addition of the FP. For example, the relative addressing value of $LDA+ ! 105$ is $105 + XR$, despite the fact that the XR will be added after FP.

5. Absolute addressing value = final addressing value of an instruction.

6. Provides more flexible access to data than does the SP.

7. Permits relocateable code.

8. Useful to access temporary variables frozen within the stack.

B. CUSP Instructions

1. $LDF = Load FP$. OV affected.
2. $STF = Store FP$.
3. $ADF = Add to FP$. OV and EQ affected.
4. $SBF = Subtract from FP$. OV and EQ affected.
5. $CMF = Compare with FP$. EQ and LT affected.
6. $TAF = Transfer accumulator to FP$. OV affected.
7. $TFA = Transfer FP to accumulator$.
8. $TSF = Transfer SP to FP$.
9. $TFS = Transfer FP to SP$.
10. $PSHF = Push FP on stack$.
11. $POPF = Pop FP from stack$. OV affected.

IV. Subroutine Calls

A. Primary use for stack.

1. To pass parameters.
2. To hold return address.
3. To hold local variables.

B. $JSR = Jump to SubRoutine$

1. After the PC is automatically incremented, PUSH PC onto stack.
2. Jump to addressed location, i.e., set PC to addressed location.
3. For example, if current PC is $100$ with instruction $JSR 223$ in location $100$, then the execution of $JSR 223$ will cause $101$ to be PUSHed onto the stack and the PC will be set to $223$.

C. $RTN = Return from subroutine$.

1. After the PC is automatically incremented, POP the PC from the stack.

D. Stack should also be used to save any registers that will have unwanted changes made to them in the subroutine.

1. Two possible register saving conventions.
   a) 1. Caller saves its own registers.
   b) 2. Subroutine saves registers it will change. This is typical, why?
   c) Each can be inefficient. How?
2. SPARC's have a sliding register window to minimize this.

V. Parameter Passing

A. Increases modularity by minimizing data and variable dependencies.

B. Two types of parameters:

1. Variable parameters = subroutine can change caller's variable.
2. Value parameters = subroutine cannot change caller's variable.

C. Three calling conventions. Advantages and disadvantages.

1. Shared variables.
   a) Fast--no time taken at all!
   b) Tight coupling—bad programming practice because it is difficult to modify code without knowing side effects.
   c) Difficult to reuse and debug code.
2. Registers.
a) Fast.
b) Limited number and size of parameters.
c) Difficult to reuse and debug code.

3. Stack = Push actual parameters onto the stack.
   a) Slower.
   b) Unlimited size.
   c) Loose coupling.
   d) Easy to reuse and debug code.

VI. Stack calling conventions.
A. How are the two types of parameters each passed?
   1. Two techniques
      a) CBV = Call By Value; make a local copy. Insulates the caller's variable from changes made in the subroutine.
         (1) Costly for large data structures, such as arrays.
      b) CBA = Call By Address; pass the address of the variable to the subroutine so that the subroutine may change the caller's value of the variable.
         (1) Requires indirect addressing modes.
   B. In what order are the parameters passed?
      1. CUSP: first parameter is PUSHed first, then second parameter is PUSHED.
         a) Use ADS rather than POP to delete parameters after returns.
         b) For subroutine, location of kth parameter in stack is SP + n - k + 1, where n is the total number of formal parameters (assuming each parameter requires only one word to store). The "+ 1" accounts for PUSH PC that occurs during the execution of a JSR.
      2. Function calls return one non-parameter value in a well known location, i.e. a register.
   C. Use frame pointer to access data within the stack.
      1. By transferring the stack to the FP, TSF, at the beginning of a subroutine we have a snapshot of the stack. This allows us access the parameters within the subroutine despite using POP and PUSH within the subroutine (and the subroutines it calls).
      2. Since CUSP has the called procedure responsible for preserving changed registers, each subroutine that will use TSF will first have to PSHF first.
      3. Using stack to store local variables.
         a) To make room for local variables use SBS, use ADS to remove them. For example SBS# 3 would provide room on the stack for 3 one-word local variables, and ADS# 3 would "delete" the 3 local variables from the stack.
         b) BGN = Begin subroutine. Pushes FP onto stack, makes room for local variables and then transfers the stack to the FP
            (1) For example, BGN# 4
                (a) PSHF
                (b) SBS# 4
                (c) TSF
            c) FIN = Finish subroutine. "Deletes" the local variables from the stack, and then restores the FP by popping its value from the stack.
               (1) For example, FIN# 5
                  (a) ADS# 5
                  (b) POPF

VII. Indirect Addressing
A. Use asterisk, *, as the mnemonic.
B. Operand of the instruction is the address of a word that contains the address of the data.
C. Permits CBA of single word data structures.
D. For example, let Memory[$150] hold $22 and Memory[$22] hold $89 then
   1. LDX* $150 is XR := Memory[Memory[$150]] := Memory[$22] := $89
2. If FP hold $05 then LDA* ! $14B is ACCUM := Memory[Memory[$05 + $14B]] :=
   Memory[Memory[$150]] := Memory[$22] := $89
E. In C, think of the operand in immediate mode as normal data, in direct mode the operand is a pointer, and
   indirect addressing mode the operand is a pointer to a pointer.

VIII. Indirect Indexed Addressing
A. Use ampersand, &, as the mnemonic.
B. The operand of the instruction is the address of a word that contains the address of the start of a data
   structure; add to the data structure's address the XR.
C. Permits CBA of arrays and other multi-word data structures.
D. For example, let Memory[$176] hold $432 and Memory[$435] hold $93 and XR hold $3 then
   2. if FP holds $10 then LDA& ! $166 is ACCUM := Memory[Memory[$166 + FP] + XR] :=

IX. Development Method
A. 1. Determine the subroutine description in HLL, specifying CBA and CBV
   2. Write assembly language subroutine using normal variables; assigning distinct local variable names.
C. 3. Add BGN# Num to top of subroutine and FIN# Num to end of the subroutine, where Num is the
   number of local variables.
D. 4. Add a .EQU statement to the top of subroutine for each variable used.
   1. For local variables start at 0
   2. For parameters use the n - k + 2 + # of local variables formula.
E. 5. If registers are to be saved, then add appropriate pushes add the beginning of the subroutine and pops at
   the end.

   - Bubble sort
     o int Arr[N + 1];
     o for (i = N; i > 0; i--)
       * for (j = 0; j <= i - 1; j++)
         * if (Arr[j] > Arr[j + 1] ) {
           * Temp = Arr[j];
           * Arr[j] = Arr[j + 1];
           * Arr[j + 1] = Temp;
         * }

•