ICT106 Fundamentals of Computer Systems
Topic 8
Procedures, Calling and Exit conventions, Run-time Stack
Ref: Irvine, Ch 5 & 8

Objectives

• To understand how HLL procedures/functions are actually implemented at the assembly level
• To understand how the run-time stack is used.

1. Explain the meaning of the following concepts
   - run-time stack
   - stack top and base
   - SS:SP registers
   - push and pop
   - return address and procedure calls
   - near and far calls
   - actual and formal parameters (or arguments and parameters)

- passing parameters to a routine (pushing parameters)
- values and value parameters
- pointers and variable parameters
- (de)allocating local variables
- returning function values
- mixed language programming
- interfacing languages and the external directive
• 2. *Know how to perform the following operations:*
  – compile time allocation of data memory (i.e., heap & stack)
  – differences in passing information: value and variable parameters, function values
  – parameter passing and local variable allocation
  – parameter and local variable usage
  – differences between HLL and assembler procedures

• Example Program:
The following program finds the minimum and maximum values in an array - study it.

```c
/* File: find.c
Example of assembly language routine using parameters and local variables. */
#include <stdio.h> /* function prototype */
void Find( int *Address, int Number, int *Minimum, int *Maximum);

void main(void)
{
  int Numbers[ ] = {1, 6, 7, -10, 100, 101, 0};
  int items = 7;
  int Min, Max;
  Find (Numbers, items, &Min, &Max);
  printf("Min: %d   Max: %d \n", Min, Max);
  printf("End of Program - bye.\n");
} /* end main */
```

```
// The Find routine assumes small model. In particular, that
// the array is within the data segment such that only the
// offsets of the pointers are being passed to this routine
void Find( int *Address, int Number, int *Minimum, int *Maximum)
{
  int Min, Max;
  asm{
     mov Min, 0x7FFF   // Assign largest possible value
     mov Max, 0x8000   // Assign smallest possible value
     mov bx, word ptr Address  // Offset address of
     // array to search Number of
     mov cx, word ptr Number  // elements in array.
    } // end asm
```
In C, labels within the ASM statement are not available outside. So sometimes, although cumbersome, it may be necessary to have separate ASM statements for appropriate assembly instructions. For example,

```
asm SUB AX, AX
```

```
asm ADD AX, [BX]
asm CMP DX, BX
asm JNS Loc_1
```

```
loc_1:
    // Loop around the array looking for Min and Max
    mov ax, [bx]       // Get next array element
    add bx, 2          // Point to next element, ready for
                        // next time around loop
    cmp ax, Min         // Replace Min, if element smaller than current Min
    jge MaxTest
    mov Min, ax         // Replace Max if element greater than current Max
MaxTest:
    cmp ax, Max
    jle Finish
    mov Max, ax
Finish:
    // Loop back if more elements left
    dec cx
    jnz loc_1
```

// Put Min answer into contents pointed to by
// Minimum
mov bx, word ptr Minimum
mov ax, Min
mov [bx], ax

// Put Max answer into contents pointed to by
// Maximum
mov bx, word ptr Maximum
mov ax, Max
mov [bx], ax

Stack

- Concept: A data structure analogous to a "spring loaded" mechanism to hold plates in a cafeteria.
- It is a last-in-first-out (LIFO) structure in which items are inserted and deleted from the top of the stack.
- Insertion is referred to as "pushing" and deletion is referred to as "popping" the item to/from the stack.
At any time, only the element at the top of stack is directly accessible.
Can be used for a variety of applications: recursion, reversing, ...

Stack Implementation (assembly language/hardware)
- A stack is a contiguous area of memory used for temporary storage.
- All operations are performed at one end only – the top of the stack.
- The base of the stack is fixed at the address of the highest offset which is determined by how much stack space has been allocated. This means the stack grows towards offset zero.

- The top of stack is dynamic in that it changes as PUSH and POP operations are performed.
- Plays a central role in function/procedure calls
- In this context it is called the run-time stack and known as the stack frame
- During program execution, the SS:SP register pair (stack segment register & stack pointer register) is used to maintain the stack.
- SS points to the base of the stack (i.e., beginning of the stack segment). SP points to the top of the stack.
- Base of stack:
  - c(SS) + initial value of SP
- Top of stack:
  - c(SS) + current value of SP
- The SS and SP are maintained by the hardware.
Example:

- Suppose SS is initialized to $5600_{16}$ and SP to $2000_{16}$. Suppose we push the values 12, 29, 51, 4, 75, 37 onto stack, in that order. Then the state of the stack would be as follows:

<table>
<thead>
<tr>
<th>20-bit memory address</th>
<th>Memory contents</th>
<th>comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>57FF4</td>
<td>37</td>
<td>top sp=1FF4</td>
</tr>
<tr>
<td>57FF6</td>
<td>75</td>
<td>ss=5600; sp=1FF6</td>
</tr>
<tr>
<td>57FF8</td>
<td>4</td>
<td>ss=5600; sp=1FF8</td>
</tr>
<tr>
<td>57FFA</td>
<td>51</td>
<td>ss=5600; sp=1FFA</td>
</tr>
<tr>
<td>57FFC</td>
<td>29</td>
<td>ss=5600; sp=1FFC</td>
</tr>
<tr>
<td>57FFE</td>
<td>12</td>
<td>ss=5600; sp=1FFE</td>
</tr>
<tr>
<td>58000</td>
<td>---</td>
<td>base (not used for stack storage) ss=5600; sp=2000</td>
</tr>
</tbody>
</table>

- Each of the above addresses is calculated as follows (using hex arithmetic):
  - Contents of SS $\times$ 16 + contents of SP
  - When an item is pushed on to the stack, SP decreases by 2 (i.e., by the no. of bytes the item occupies)
  - To remove the top item from the stack, we pop the stack – this results in SP being incremented by 2

- Thus a push operation automatically decreases the SP, and a pop operation automatically increases the SP.
- For example, if the two operations POP AX POP BX were performed on the stack given in the above diagram, the top of the stack then would be as shown below:
### 20-bit memory

<table>
<thead>
<tr>
<th>Address</th>
<th>Contents</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>57FF4</td>
<td>37</td>
<td></td>
</tr>
<tr>
<td>57FF6</td>
<td>75</td>
<td></td>
</tr>
<tr>
<td>57FF8</td>
<td>4</td>
<td>top sp=1FF8</td>
</tr>
<tr>
<td>57FFA</td>
<td>51</td>
<td>ss=5600; sp=1FFA</td>
</tr>
<tr>
<td>57FFC</td>
<td>29</td>
<td>ss=5600; sp=1FFC</td>
</tr>
<tr>
<td>57FFE</td>
<td>12</td>
<td>ss=5600; sp=1FFE</td>
</tr>
<tr>
<td>58000</td>
<td>---</td>
<td>base (not used for stack storage) ss=5600; sp=2000</td>
</tr>
</tbody>
</table>

---

### Individual instructions available:

**PUSH, POP, PUSHF, POPF**

- **PUSH instruction**
  
  General form:
  
  `[label] PUSH source [comment]`
  
  `source` is used to identify the location of a 16-bit or a 32-bit value to be pushed onto the stack.

- **The stack operates as follows:**
  1. Decrement the SP by 2 (in case of a 16-bit value), or by 4 (in case of a 32-bit value)
  2. Store the value at the location identified by SS:SP

- As a result of the above pop operations, the registers AX and BX will contain the popped values 37 and 75, respectively.

- Note that the values 37 and 75, which were the top two items of the stack before, are still there, i.e., c(57FF4) is still 37, and c(57FF6) is still 75 – but these values are no longer considered to be part of the stack.

- The source operand can be one of:
  - general register
  - segment register
  - memory location
  - immediate value
• Note that the source operand cannot be a byte.
• Examples:
  - PUSH AX ; push a 16-bit register
  - PUSH Num ; push a 16-bit mem operand
  - PUSH 1234H ; push 16-bit immediate value
  - PUSH EBX ; push a 32-bit register

• PUSH is a single instruction in its own right although combines actions of SUB and MOV
• Space efficient, eg, the following operation uses only 4 bytes
  MOV AX, 22h
  PUSH AX
• To achieve the same effect without the PUSH instruction:
  SUB SP, 2
  MOV BP, SP ; cannot write mov word ptr [sp], 22h
  MOV WORD PTR [BP], 22h
  would need 10 bytes of memory
• Time efficient, eg, only 2 instructions are fetched and decoded in the above example

POP instruction

• General form:
  [label] POP destination [comment]
• Complementary instruction to PUSH, gets the top element of the stack, removes it and puts it some where. Eg,
  POP BX ; pop top item of stack into 16-bit register
  POP Num ; pop top item of stack into 16-bit memory operand
  POP ECX ; pop top item of stack into 16-bit register

PUSHF instruction

• General form:
  [label] PUSHF [comment]
• PUSHF causes the SP to be decremented by 2 and then pushes the flags register onto the stack.
• Question: Why would we want to do this?
**POPF instruction**

- General form:
  
  [label] POPF [comment]

- POPF puts the 16-bit values pointed to by the SS:SP into the flags register and then increments the SP by 2.

- Note: The flags register is not affected by PUSH, POP, or PUSHF. Each of the flag bits is potentially modifiable by the POPF instruction.

**Additional instructions**

- PUSHA and POPA instructions can be used to save and restore the eight general-purpose registers: AX, BX, CX, DX, SP, BP, SI, and DI.

- PUSHA pushes these 8 registers in the above order (AX first and DI last)

- POPA restores these registers except that SP value is not loaded into the SP register.

- Use PUSHAD and POPAD for saving and restoring 32-bit registers.

**Further Comments on the Stack**

- Allocation of space for stack
  - SS & SP allocated automatically (by hardware) - not by programmer
  - Size of stack is compile time option

- Stack is like other constructs/segments contained in same physical memory

- Too many pushes or pops can cause the stack to collide with other objects in memory - stack overflow, underflow

- If stack exceeds its boundaries - things may be overwritten
  - Operations on stack may overwrite something
  - Other memory operations may overwrite stack

- Note the difference in generated code with stack checking option switched on and off. When on, extra code is generated upon entry and exit - some registers are used
  - Select Options/Project/Compiler/Debugging
• When programming, be aware of stack size and if needed check for overflow/underflow – using either stack checking compiler option or user’s explicit code

Uses of the Stack

• Three main uses
  – Temporary storage of data
  – Transfer of control
  – Parameter passing

Example 1:

• Exchanging the values of var1 and var2 can be done by using the stack to temporarily hold data:
  push var1
  push var2
  pop var1
  pop var2

Example 2

• Often used to free a set of registers, eg
  ; save EAX and EBX registers on stack
  push EAX
  push EBX ; EAX and EBX can now be used
  ; Do something
  ; restore EAX and EBX from the stack
  pop EBX
  pop EAX
Transfer of control

- In procedure/subroutine/function calls and interrupts, the return address is stored on the stack (see below for details)
- **Parameter Passing:**
  - Stack is extensively used for parameter passing (see below for details)

Calling and Exit Conventions

- **Subroutine**
  - A subroutine/procedure (in assembly language) is a block of instructions that can be called from another place. A subroutine usually does one specific task — similar to functions in C
  - Each subroutine should begin with a comment which identifies what that subroutine does. The comment **must** specify what and how data is coming in/leaving the subroutine.

CALL and RET instructions

- Calling a routine implies that a return (from the routine) must take place
- It is not the same as a *jump* (eg, JMP instruction) — which does not return
- It is equivalent to HLL function/procedure calls. Its syntax is:
  - CALL <name of procedure/routine>
- To return from a subroutine: RET
- It is important to understand how CALL/RET work

- Each subroutine must save contents of any registers that are going to be used as temporary storage within the subroutine. The subroutine must restore the contents of the registers just before leaving the subroutine.
- To save the contents of a register - **push**, to restore – **pop**
- Pentium architecture provides two instructions for transfer of control: CALL and RET
• How does the machine know where to return to?
• The call instruction saves (pushes) the return address on the run-time stack.
• The ret instruction pops the return address off the stack.
• What is the return address?
• The address of the next instruction after the CALL instruction.

CALL instruction thus performs the following steps:
– Return address is pushed onto stack
– jump to start of routine is made, that results in executing code within the routine

• Actions taken during a (near) subroutine call:
  \[
  \begin{align*}
  \text{SP} &= \text{SP} - 2 \quad ; \text{push return address on stack} \\
  \text{SS:SP} &= \text{IP} \quad ; \text{push return address on stack} \\
  \text{IP} &= \text{IP + relative displacement} \\
  \end{align*}
  \]
  ; update IP to point to the subroutine.

• RET instruction uses the return address (pushed onto stack as part of executing the CALL instruction) to transfer control back to the calling subroutine

  Important: note that the top of stack points to this return address when RET instruction is executed.

• RET instruction thus performs the following steps:
  
  • Actions taken during the execution of RET:
    \[
    \begin{align*}
    \text{IP} &= (\text{SS:SP}) \quad ; \text{pop return address} \\
    \text{SP} &= \text{SP} + 2 \quad ; \text{from the stack} \\
    \end{align*}
    \]

  • We can specify an optional integer in the RET instruction:
    \[
    \text{RET} \quad \text{optional-integer}
    \]

  • Eg,
    \[
    \text{RET} \quad 6
    \]

  • Actions taken during the execution of RET with optional-integer:
    \[
    \begin{align*}
    \text{IP} &= (\text{SS:SP}) \\
    \text{SP} &= \text{SP} + 2 + \text{optional-integer} \\
    \end{align*}
    \]
• Important note: By the time the CALL instruction is executing, IP will contain the address of the instruction after the CALL ==> So value in IP (CS) is simply pushed onto stack by CALL
• Within the called routine, the return to the calling routine is performed by the RET instruction
  – Top of stack popped and placed into IP
  – Next instruction will then automatically be instruction after the original CALL instruction

• You should mechanically work through the sequences involved in calling a routine/function and returning from it.
• Trace with single step in debugger, looking at the content of stack - convince yourself of the operations and changes.
• Saving information on stack is perfect for nested CALL’s due to Last In- First Out property of stack

Nested Call

• Nested call means, a routine may itself call other routines (eg, Main calling F1, and F1 calling F2, and so on)
• In this case, the stack holds the list of return addresses so the CPU can find its way back to the original call.

Near and Far Subroutines

• NEAR subroutine
  – the subroutine and the instruction that CALLs it are both in the same code segment, i.e., both the calling and the called routines are in the same segment
  – In a call to near subroutine, only IP of the next instruction is pushed onto the stack
• Thus, the CALL instruction
  – saves the current value of IP on the stack
  – loads the called routine’s offset into IP, causing the CPU to jump to the first instruction in the routine
• The PROC statement identifies the routine as a near subroutine, eg
  testsubn proc near
  ;place the code here
  testsubn endp

FAR subroutine
• the calling routine and the called routine are in different code segments
• In a call to far subroutine, both the CS and IP of the next instruction in the caller are pushed onto the stack

• A return from a far subroutine has to be a far return - both the CS and IP has to be removed from the stack
• The PROC statement identifies the routine as a FAR subroutine
  testsubf proc far
  ;place the code here
  testsubf endp

The differences between NEAR and FAR CALL instructions
• Why/when use different CALLs
• Need FAR to call a routine in another code segment
• Defaults - beware of them, or at least check upon problems
• Size of instruction (number of bytes) – are different
• FAR calls have 2 extra bytes
• Type of information encoded in machine instruction
  • Near: offset; Far: segment & offset
• Type of information pushed onto stack
  • Near: offset - Far: segment & offset
• Extra steps upon call - changing of CS

The differences in NEAR & FAR RET instructions

  • Return within same segment - pops only offset
  • Return back to another segment - pops segment & offset

Passing information (parameters) to/from a subroutine from/to the calling routine

  • Subroutine call achieved via CALL
  • Extensive use of parameters is made in HLL
  • How is this information passed/exchanged in assembly language subroutines?

• There are two possible methods:
  – Passing information via registers, eg MOV AX, result
  – Passing information using the stack
// Example: program fragment of a "main"
// assembly routine - parameters are passed
// in via stack, i.e., the start address & the end
// address of array are pushed onto stack
// before the subroutine SUM is called.
MOV AX, OFFSET array           // Start address
PUSH AX
MOVE AX, OFFSET array+9       // equivalent to
                             // array[9] End address
PUSH AX
CALL SUM                     // Result comes back in AX

• Assembler Procedure Sum: the BP register is used
  to access parameters within the called procedure:
  SUM()
  
  asm{
    MOV BP, SP
    MOV BX, [BP+4]      // Start address
    MOV CX, [BP+2]      // End address
  }
  asm{
    SUB AX, AX
    top:
    ADD AX, [BX]
    ADD BX, 2
    CMP CX, BX
    JNS top
  }

• The diagram below shows the top three entries in the stack immediately after the
  CALL to SUM is executed

  ??
  address of array[0]
  address of array[9]

  SP-->  IP
  (top of stack)

• It is important to understand and know how, using the stack, to put and access
  information from the calling routine and within the subroutine
• Understand and know the consequences when using NEAR calls
• Understand and know how the details change with FAR calls
• Suppose SUM was in a different segment than the calling routine
• a far call instruction is used
• ==> 4 bytes of return information put on stack
• Accessing information from the stack within SUM changes
  [BP+6] contains start address
  [BP+4] contains end address
• a return far instruction is to be used to return at the end of SUM

• Some compilers/assemblers automatically recognize when calling routines and procedures are in a different segment such that CALL & RET instructions automatically changed to a far call & and a far return
• Look at actual machine code generated by your C compiler.

Cleaning up the stack

• Remove leftover information (parameters) from the stack - especially important for nested procedures
• Be careful of order of removal (parameters vs return information)

• Understand the consequences if these issues are not addressed

// Example fragment of a "main" routine

MOV AX, OFFSET array
PUSH AX       // parameter passed
MOVE AX, OFFSET array[9]    // other method
PUSH AX       // parameter passed
CALL SUM
ADD SP, 4     // Remove leftover parameters
.....         // Result left in AX
• Alternative stack cleanup mechanism
  – Get the subroutine to clean up the stack
  – Assumes the subroutine knows the number of bytes
    pushed onto the stack
  – Uses advanced version of RET instruction
    RET x
  – Performs ordinary RET instruction, but additionally,
    adds data value x to stack pointer
  – (Question: If the called routine cleans up the stack before
    returning, how are values passed out to the caller?)

Using stack for temporary storage of information

• HLL equivalent: local variables in procedures
• Reason for using stack same as the reason for using the stack to pass
  information to procedures
• For example, consider the following C in-line assembly routine needing 2 WORDs
  for local variables:

```c
Test ()
{
    asm {
        MOV BP, SP  // To access info. on stack
        SUB  SP, 4  // Create space on stack
        .............. // NOTE: Uninitialised !!
        MOV AX, [BP+2]  // Top data item passed
        MOV BX, [BP+4]  // 2nd Top ....
        ....
        MOV [BP-2], AX  // First local variable
        MOV [BP-4], BX  // Second local variable
        ....
        MOV SP, BP      // Restore stack pointer
        RET  n          // Remove parameter information
    }
}
```

• Information passed to function accessed with positive displacements
• Local information accessed with negative displacements
• Remember, [BP] contains the return information
• Note changes for FAR calls
Stack and temporary storage: Complications with nesting

- Higher level routines must maintain access to information on stack
- The routines themselves may have parameter and local variable information
- However, there is a need to save and restore BP in all routines/procedures

Saving Old BP

```c
asm {
    PUSH BP    // Stack perfect for saving
    MOV  BP, SP

    ....
    ....  // body of function .... //
    ....

    MOV  SP, BP  // Restore stack pointer
    POP BP      // Restore from stack

    RET n
}
```

Local variables

- Note that saving BP will cause offset changes in accessing parameter information
  - [BP+4] ==> First parameter
  - [BP+6] ==> Second parameter
- However, there will be no changes in accessing local variable information
- Local variables of a procedure are dynamic in nature - because they come into existence when a procedure is invoked and disappear when a procedure terminates
- space for local variables is reserved on the stack
- Example: assume that in an assembly language program, x and y are two parameters which are passed to a procedure, and var1 and var2 are two local variables, each requiring 2 bytes of storage
• The information stored in the stack
  – parameters,
  – the return address,
  – old BP value, and
  – local variables
• is collectively known as a Stack Frame

• In HLLs, stack frame is also referred to as the activation record because each procedure activation requires all this information.
• The BP value is also referred to as the Stack Frame Pointer.
  – once the BP value is known, we can access all the data in the stack frame.

• Tasks performed by the linker
  – A source program may consists of a number of files (eg, Main.c, F1.c, F2.c, etc).
  – The compiler compiles each source file to a corresponding object file (eg, Main.obj, F1.obj, F2.obj etc.)
  – The linker combines all .obj files to create one executable file.
• The object files may contain calls to routines (eg, operating system service functions) in an external link library. The linker then performs the following tasks:
  – copies the needed routines from the link library,
  – combines all object files into one file to form an executable program
  – creates a special header record at the beginning of the executable file.

• A program called loader decodes the header record of the executable program and loads it into memory so that it can be run.
• At the start of the compilation/translation process, a counter (which represents IP) is set to 0 for each source file (eg, Main.c, F1.c, F2.c). This assumes that each source module will start at (virtual) location 0 in main memory, as shown in the following diagram:

• The linker merges the separate address spaces of the object modules into a single linear address space as follows:
  – Construct a table of all the object modules and their lengths
  – Using this table, assign an address to each object module, which is taken as the start address of that module
  – Find all instructions that contain a memory address, and add to each a constant value (relocation constant) equal to the start address of the module in which the instruction is contained.
  – Find all instructions that reference other functions and insert the address of those functions in place.
• Using our example (see above diagram) a table produced during step 1 is:

<table>
<thead>
<tr>
<th>Module</th>
<th>Length</th>
<th>Start Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>Main</td>
<td>500</td>
<td>000</td>
</tr>
<tr>
<td>F1</td>
<td>700</td>
<td>500</td>
</tr>
<tr>
<td>F2</td>
<td>300</td>
<td>1200</td>
</tr>
</tbody>
</table>

The following diagram shows the modules after linking has been performed:

0 push ax Start of module Main
100 mov ax, a
500 call F1
700 move ax,5
call F2
1200 inc cx
1250 jmp 1290
1250 Start of module F1
1500 ret End of module F2

• Note that all addresses in the above diagram are **relocatable addresses** (relative to address 0). These addresses will be converted to **absolute (physical) addresses** when the program is loaded (by the loader) into memory.

• Eg, if the program is loaded starting at some memory location x, then x will be added to each of the above addresses (offsets) to determine the correct address at which each instruction will be loaded.

• If relocation is done by the linker, and the addresses in the code are actual physical addresses, then a problem arises if the program is not loaded starting at the address the linker designed it to load.

• In a multi-tasking environment, it is difficult to guarantee that a program will always load at the same position in memory every time.
• **Binding time** determines the time at which a symbolic name (used by the programmer) is bound to the actual physical address in memory.
  
• Take the example of an assembly program. In your program, you may have a line of code:
  
  ```assembly
  JMP Again ; jump to label Again
  ```

• The machine instruction corresponding to the above code would not have any reference to the label ‘Again’. Instead an address (or offset) is used.

• If an offset is used, then the code is relocatable because the jump is relative to current instruction and therefore does not depend on where the program is loaded.

• If an address is used, then the program will depend on where it is loaded if the address is to be interpreted as an actual physical address.

• In general, the relocation problem is intimately related to binding time. You would have gathered by now, that binding symbolic names to actual addresses very early would cause problems. So the problem can be solved in 2 parts:
  
  – First bind symbolic names to virtual addresses (this is not virtual memory).
  – Then at a later stage, bind the virtual addresses to actual addresses.

• The offsets mentioned above are **virtual addresses**. Relocation and linking binds symbolic names to virtual addresses. When the linker creates the load module (diagram shown earlier), the linear address space in the module is actually a binding of symbolic names to virtual addresses.

• At load time, the loader will carry out the binding of virtual addresses to actual addresses, i.e., setting the CS register. The general term for the CS register is relocation register.
To get the actual address, the content of the relocation register is added to the virtual addresses. That is, given a `segment:offset` we can determine the actual address (segment * 16 + offset).

There are other methods for performing virtual-actual binding but we won’t be looking at them.

Modular programming still important when considering assembly language

In its own right

Parts of large programs/libraries often written in different languages

Important to understand how HLL compilers translate to low level assembly/machine code

Important in mixed language programming

Low level aspects which affect HLL modular programming

- Stacks
- Procedures - calling/returning
- Machine level aspects of procedures in HLL - parameter passing
- MACROs - to make code easier to read as well as reduce typing (not covered in this unit)
- MAKE or Project: maintenance utility (not covered)