23. Example of Interrupt, Subroutines, Recursion

Chapter 10

- Review
- Interrupt processing
- Subroutines
- Recursion
- Example: Towers of Hanoi
;; Main program which can be interrupted
;; Number in R1 is continuously incremented in a loop
;; When it gets above 3 digits, the number is reset to 0
;;
.ORIG x3000
LEA   R6, #.-1 ; initializing R6, to 3000 (empty stack)
LD    R0, IEnable ; Interrupt enable vector
STI   R0, KBSR   ; Stored in Keyboard Status register

; R1 keeps numbers up to 999
AND   R1, R1, #0
LEA   R0, Welcome
PUTS
LD    R0, NL
TRAP  x21

NEXTLN ADD   R1, R1, #1 ; Increment the value in R1
LD    R0, MAX
ADD   R0, R0, R1
BRn   NEXTLN
AND   R1, R1, #0 ; Reset number if more that 3 digits
BRnzp NEXTLN
HALT  

MAX   .FILL  #.-999 ; negative of maximum number
IEnable .FILL  x4000 ; Interrupt enable
KBSR   .FILL  xFE00 ; Keyboard status register address
Welcome .STRINGZ "Pick your lucky number: Press any key"
NL   .FILL  #13 ; new line

.END
;; Interrupt vector table
;; Location x0180 is filled with the starting address of the
;; interrupt routine
.ORIG x0100
.BLKW x80
.FILL x1000 ; starting address of the keyboard interrupt routine
.END

**INTERRUPT VECTOR FOR KBD**
;; Interrupt routine
;; Prints the number in R1
;; Calls subroutine BIN2ASC which converts binary number in R0
to ASCII
;; Number is at most 3 digits
;; The string of characters representing the number is stored
;; in ASCIIBUFF
.ORIG x1000
;
; Push registers on stack. Might not need that, included as a reference.
;; R6 is initialized in the main code
ADD R6, R6, #-$1
STR R0, R6, #0
ADD R6, R6, #-$1
STR R1, R6, #0
ADD R6, R6, #-$1
STR R7, R6, #0
;
GETC ; Read input to clear keyboard
;
; Don't need to do anything with this char.
; rest of code goes here
ADD R0, R1, #0
JSR BIN2ASC
LEA R0, ASCIIBUFF
PULL
LD R0, NL1
TRAP x21
LDR R7, R6, #0
ADD R6, R6, #1
LDR R1, R6, #0
ADD R6, R6, #1
LDR R0, R6, #0
ADD R6, R6, #1
RTI
LD    R2, Save2
LD    R3, Save3
RET

; Save1    .FILL  #0
Save2    .FILL  #0
Save3    .FILL  #0
ASCIIplus .FILL  x002B
ASCIIoffset .FILL  x0030
Neg100    .FILL  xFF9C
Pos100    .FILL  x0064
Neg10     .FILL  xFFF6
NL1       .FILL  xa
ASCIIBUFF .BLKW  #3
            .FILL  #0

.END
BIN2ASC

ST  R1, Save1
ST  R2, Save2
ST  R3, Save3
LEA  R1, ASCIIUFF  ; R1 points to string being generated
ADD  R0, R0,#0  ; R0 contains the binary value

LD  R2,ASCIIoffset  ; Prepare for "hundreds" digit

LD  R3, Neg100  ; Determine the hundreds digit
Loop100
ADD  R0, R0, R3
BRn  End100
ADD  R2, R2, #1
BRnzp  Loop100

End100
STR  R2, R1, #0  ; Store ASCII code for hundreds digit
LD  R3, Pos100
ADD  R0, R0, R3  ; Correct R0 for one-too-many subtractions

LD  R2, ASCIIoffset  ; Prepare for "tens" digit

Begin10
LD  R3, Neg10  ; Determine the tens digit
Loop10
ADD  R0, R0, R3
BRn  End10
ADD  R2, R2, #1
BRnzp  Loop10

End10
STR  R2, R1, #1  ; Store ASCII code for tens digit
ADD  R0, R0, #10  ; Correct R0 for one-too-many subtractions
Begin1
LD  R2, ASCIIoffset  ; Prepare for "ones" digit
ADD  R2, R2, R0
STR  R2, R1, #2
LD  R1, Save1
Subroutines

A subroutine is a program fragment that:
- lives in user space
- performs a well-defined task
- is invoked (called) by another user program
- returns control to the calling program when finished

Like a service routine, but not part of the OS
- not concerned with protecting hardware resources
- no special privilege required

Reasons for subroutines:
- reuse useful (and debugged!) code without having to keep typing it in
- divide task among multiple programmers
- use vendor-supplied library of useful routines
**JSR Instruction**

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>JSR</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>PC</td>
<td>offset</td>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Jumps to a location (like a branch but unconditional), and saves current PC (addr of next instruction) in R7.

- saving the return address is called "linking"
- target address is PC-relative \((PC + \text{Sext(IR[10:0])})\)
- bit 11 specifies addressing mode
  - if \(=1\), PC-relative: target address = \(PC + \text{Sext(IR[10:0])}\)
  - if \(=0\), register: target address = contents of register IR[8:6]
NOTE: PC has already been incremented during instruction fetch stage.
JSRR Instruction

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>JSRR</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Base</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Just like JSR, except Register addressing mode.

- target address is Base Register
- bit 11 specifies addressing mode

What important feature does JSRR provide that JSR does not?
NOTE: PC has already been incremented during instruction fetch stage
Returning from a Subroutine

RET (JMP R7) gets us back to the calling routine.
  • just like TRAP

MAKE SURE DATA IN R7 IS SAVED
Can a Subroutine Call Other Subroutines?

Yes - but make sure RT (multiple) values are saved

Can a subroutine call itself?
Yes, but need to "stack" return values
What is Recursion?

A **recursive function** is one that solves its task by **calling itself** on smaller pieces of data.

- Similar to recurrence function in mathematics.
- Like iteration -- can be used interchangeably; sometimes recursion results in a simpler solution.

**Example: Running sum** ($\sum_{1}^{n} i$)

**Mathematical Definition:**

- $\text{RunningSum}(1) = 1$
- $\text{RunningSum}(n) = n + \text{RunningSum}(n-1)$

**Recursive Function:**

```c
int RunningSum(int n) {
    if (n == 1) {
        return 1;
    } else {
        return n + RunningSum(n-1);
    }
}
```
EXAMPLE: FACTORIAL(!)

\[ n! = n \times (n-1) \times (n-2) \times \ldots \times 1 \]

RECURRANCE:

```
CHECK IF
n IS A POSITIVE INTEGER

FACTORIAL = 1

FACTORIAL = FACTORIAL \times n

n \leq n-1

YES
h=0?

NO
STOP
```
RECURSIVE

CHECK IF \( n \) IS A POSITIVE INTEGER

IF \( n = 1 \), EXIT

OTHERWISE,

\[
\text{FACTORIAL}(n) = n \times \text{FACTORIAL}(n-1)
\]
High-Level Example: Towers of Hanoi

**Task:** Move all disks from current post to another post.

**Rules:**
1. Can only move one disk at a time.
2. A larger disk can never be placed on top of a smaller disk.
3. May use third post for temporary storage.
Task Decomposition

Suppose disks start on Post 1, and target is Post 3.

1. Move top n-1 disks to Post 2.

2. Move largest disk to Post 3.

3. Move n-1 disks from Post 2 to Post 3.
Task Decomposition (cont.)

Task 1 is really the same problem, with fewer disks and a different target post.
- "Move n-1 disks from Post 1 to Post 2."

And Task 3 is also the same problem, with fewer disks and different starting and target posts.
- "Move n-1 disks from Post 2 to Post 3."

So this is a recursive algorithm.
- The terminal case is moving the smallest disk -- can move directly without using third post.
- Number disks from 1 (smallest) to n (largest).