Pseudo-Ops

- `.ORIG  x3000` ;;the first instruction should be at x3000
- `.END` ;;indicate this is the end of the program
- `.FILL    #3` ;;#-3, #5, #0, xFFC0, xABCD, etc.
- `.BLKW#3` ;;how many memory location you want to put
- `.STRINGZ “Hello, World!”` ;;Null-terminated
- `.TRAP    x25` ;;same as HALT
Examples

- How to clear R0?
  - AND R0, R0, #0

- How to do R0 ← R1?
  - ADD R0, R1, #0 ;;remember: -16 ≤ immediate value and ≤ 15

- How to get –R0?
  - NOT R0, R0
  - ADD R0, R0, #1
Tips

- .asm → (PASS 1) → symbol table → (PASS2) → .obj (the executable)
- Use LABELS
- Use semicolon to comment
- BR = BRnzp
- Set breakpoint when debugging
- Draw a flow chart if necessary
- Try to remember what kind of numbers are in the registers that you are using. Write them down when calculation gets complicated.
- Assign different registers to specific functionality when the task is complex (R1 for row count, R2 for column count, etc)
- Don’t get frustrated, breathe and start over.
LC-3 Review: I/O

I/O Interactions

• Polling vs Interrupts
  • Polling
    • Loop indefinitely until data is available by checking status register
  • Interrupts
    • Allows program to perform other work while no data is available
    • Upon reception of interrupt, pause current code execution and execute special interrupt handling functions
    • Return to interrupted code once interrupt has been handled
    • Will be covered in depth in ECE 391!
LC-3 Review: I/O

Memory Mapped I/O

• Map I/O to specific memory addresses
  • Removes the need for dedicated I/O channels
• Accessing the mapped memory address gives access to the input or output device
  • Reading from xFE02 (KBDR) returns a char of what key was pressed on the keyboard
  • Writing ‘a’ to xFE06 (DDR) will display ‘a’ on the display
  • Check the status register (KBSR, DSR) of the respective input/output before reading or writing
LC-3 Review: Keyboard Input

Reading from the keyboard

- Poll KBSR until ready bit is set then access input data stored in lower 8 bits of KBDR

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>POLL</td>
<td>Check status register</td>
</tr>
<tr>
<td>BRzp POLL</td>
<td>Loop while ready bit not set</td>
</tr>
<tr>
<td>LDI R0, KBDR</td>
<td>Get keyboard input</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Address</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>KBSR</td>
<td>KBSR address</td>
</tr>
<tr>
<td>KBDR</td>
<td>KBDR address</td>
</tr>
</tbody>
</table>
LC-3 Review: Display Output

Writing to the display

- Poll DSR until ready bit is set then write display data to DDR

```
LDI    R1, DSR ; Check status register
BRzp   POLL ; Loop while ready bit not set
STI    R0, DDR ; Write display data

.FILL   xFE04 ; DSR address
.FILL   xFE06 ; DDR address
```
TRAPS

TRAP function
- Passes control to operating system
- Programmers can use complex operations without specialized knowledge

<table>
<thead>
<tr>
<th>Trap Vector</th>
<th>Assembler Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>x20</td>
<td>GETC</td>
<td>Read single character from keyboard into R0</td>
</tr>
<tr>
<td>x21</td>
<td>OUT</td>
<td>Write character from R0 to display</td>
</tr>
<tr>
<td>x22</td>
<td>PUTS</td>
<td>Write null terminated string of characters to display starting from memory location at R0</td>
</tr>
<tr>
<td>x23</td>
<td>IN</td>
<td>Prompts for input; Reading char from keyboard and echo input to console</td>
</tr>
<tr>
<td>x24</td>
<td>PUTSP</td>
<td>Same as puts but use characters from both lower and upper 8 bits</td>
</tr>
<tr>
<td>x25</td>
<td>HALT</td>
<td>Halts program execution</td>
</tr>
</tbody>
</table>
TRAPS: How they work

- TRAP function is called by the user
- The 8-bit trap vector is used as the index of the service routine’s address in the trap vector table
  - the table is stored in memory at 0x0000 – 0x00FF
- The PC is loaded with the address of the service routine
- After executing the service routine, control returns to the user program

\[
\begin{align*}
\text{MAR} & \leftarrow \text{ZEXT}(\text{trapvector}) \\
\text{MDR} & \leftarrow \text{MEM}[\text{MAR}] \\
\text{R7} & \leftarrow \text{PC} \\
\text{PC} & \leftarrow \text{MDR}
\end{align*}
\]
Subroutines

- Similar to service routines but not part of the OS
- Useful if there is a code segment that needs to be executed multiple times
- Subroutines can be invoked by JSR or JSRR
- Return is implemented with RET instruction

\[
\begin{align*}
\text{TEMP} & \leftarrow \text{PC} \\
\text{If}(IR[11] = 0) & \\
& \quad \text{- PC} \leftarrow \text{BaseR} \\
\text{Else} & \\
& \quad \text{- PC} \leftarrow \text{PC} + \text{SEXT(PCoffset11)} \\
R7 & \leftarrow \text{TEMP}
\end{align*}
\]
Subroutines: Callee and Caller Save

- Subroutine will save and restore registers that it modifies except for the return values
  - The only visible change should be the return value (if any) upon return

- Caller should save registers that could be modified by the subroutine if they contain important data
  - R7 would need to be saved since JSR and JSRR overwrite its value
Stacks

- Last-In-First-Out (LIFO)
- Stack operations
  - Push: puts a new thing on top of the stack
  - Pop: removes whatever is on the top of the stack
  - IsEmpty: checks if the stack is empty
  - IsFull: checks if the stack is full
- Example:
Stacks (continued)

- Implementation
  - Keep elements stationary, just move the pointer
  - More efficient than moving everything

- Example: Calculator
- Questions?