Lecture 17: Recursion and MUL/DIV

- Recursion
- Multiply
- Division

Recursion

- Recursion is when an algorithm is defined in terms of itself.
- Example: Factorial

\[ n! = n \times (n-1) \times (n-2) \times (n-3) \ldots (1) \]

\[ \text{fact}(0) = 1 \]

\[ 4! = 4 \times 3 \times 2 \times 1 = 24 \]

- Defined in terms of itself:
  \[ \text{fact}(n) = n \times \text{fact}(n-1) \]
  \[ \text{fact}(0) = 1 \]

Issues with Recursion

- Recursive algorithms can be implemented by having a procedure call itself, but care must be taken to assure that each successive call does not destroy the parameters and results generated by the previous one.
- Assembly variables (defined using data directives db, dw, etc.) have a fixed location and are will be overwritten if used in a recursive program.

Solution

- Parameters, registers, and temporary results need to be stored in a different place in memory for each invocation of the recursive function.
- How? The stack!
- Using the stack will guarantee that separate areas of memory are used.
- This is done by setting up a stack frame for each procedure.
Stack Frames in Recursion

data pushed on the stack can be retrieved in reverse order as the function returns from its nesting.

BP can be used to permit access to items in a frame.

Multiplication and Division

- Instructions for integer multiplication on 8, 16, and 32 bit operands
- MUL, DIV – unsigned binary numbers
- IMUL, IDIV – signed binary numbers
- For floating point? Special floating point instructions (Ch 15 in Irvine)

MUL

- Multiplies an 8, 16, or 32 bit operand by AL, AX, or EAX respectively.
- Format:
  MUL multiplier
  multiplier – register or memory (not immediate!)
- Registers used:

<table>
<thead>
<tr>
<th>Multiplier</th>
<th>Product</th>
</tr>
</thead>
<tbody>
<tr>
<td>AL</td>
<td>AX</td>
</tr>
<tr>
<td>AX</td>
<td>AX</td>
</tr>
<tr>
<td>AX</td>
<td>DX:AX</td>
</tr>
</tbody>
</table>
MUL Examples

bval db 10
...
mov al, 100
mul bval

Before MUL:

<table>
<thead>
<tr>
<th>AH</th>
<th>AL</th>
</tr>
</thead>
<tbody>
<tr>
<td>?</td>
<td>64h</td>
</tr>
</tbody>
</table>

After MUL:

<table>
<thead>
<tr>
<th>AH</th>
<th>AL</th>
</tr>
</thead>
<tbody>
<tr>
<td>03</td>
<td>E8h</td>
</tr>
</tbody>
</table>

MUL Examples

wval db 1000
...
mov ax, 55555h
mul wval

Before MUL:

<table>
<thead>
<tr>
<th>DX</th>
<th>AX</th>
</tr>
</thead>
<tbody>
<tr>
<td>?</td>
<td>D903</td>
</tr>
</tbody>
</table>

After MUL:

<table>
<thead>
<tr>
<th>AH</th>
<th>AL</th>
</tr>
</thead>
<tbody>
<tr>
<td>03F</td>
<td>B3B8</td>
</tr>
</tbody>
</table>

• Multiplication result might need a high-order byte or word.
• We’ll be using numbers that are small enough for the result to fit in AX so we can ignore DX.
• How do we know when we can do this?
  – MUL sets CF and OF
  – If result is small enough to fit into AL (bytes) or AX (words), CF = 0 and OF = 0
  – If result is big, CF = 1, OF = 1

Example: Checking Size

; multiply CX by AX. If result extends into DX, copy 2 words to result locations, else copy one word.
data
ResultLo dw ?
ResultHi dw ?
...
mul cx
mov ResultLo, AX
jnc L1 ;jump if carry not set
mov ResultHi, DX
L1:
IMUL

- IMUL multiplies signed binary numbers.
- Why is this different? It sign extends the result when needed.
- Formats and use of registers are the same as in MUL.
- Carry and overflow flags are set the same.

IMUL Examples

```
bval    db    4
...
    mov al, 48
    imul bval
```

Before IMUL:

<table>
<thead>
<tr>
<th>AH</th>
<th>AL</th>
</tr>
</thead>
<tbody>
<tr>
<td>?</td>
<td>FC</td>
</tr>
</tbody>
</table>

After IMUL:

<table>
<thead>
<tr>
<th>AH</th>
<th>AL</th>
</tr>
</thead>
<tbody>
<tr>
<td>FF</td>
<td>F0</td>
</tr>
</tbody>
</table>

CF = 0, OF = 0: Result fits into 8 bits, AH holds sign extension.

```
wval    dw    4
...
    mov ax, 48
    imul wval
```

Before IMUL:

<table>
<thead>
<tr>
<th>DX</th>
<th>AX</th>
</tr>
</thead>
<tbody>
<tr>
<td>?</td>
<td>0030</td>
</tr>
</tbody>
</table>

After IMUL:

<table>
<thead>
<tr>
<th>DX</th>
<th>AX</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>00C0</td>
</tr>
</tbody>
</table>

CF = 0, OF = 0: Result fits into 16 bits.
Warning

- Sixteen bit division will wipe out whatever is in DX!
- It’s easy to forget this if you are only using the result returned in AX.

DIV

- DIV divides unsigned 8-bit, 16-bit, and 32-bit numbers.
- Format:
  DIV \textit{divisor}
  \textit{divisor} - register or memory (not immediate!)
- Registers used:

  \begin{tabular}{|c|c|c|c|}
  \hline
  Dividend & Divisor & Quotient & Remainder \\
  AX & op-8 & AL & AH \\
  DX:AX & op-16 & AX & DX \\
  \hline
  \end{tabular}

DIV Examples

\begin{verbatim}
  bval db 2 ;divisor
  ...  
  mov ax, 0083h ;dividend
  div bval

  Before DIV:
  \begin{tabular}{|c|c|}
  \hline
  AH & AL \\
  \hline
  00 & 83h \\
  \hline
  \end{tabular}

  After DIV:
  \begin{tabular}{|c|c|}
  \hline
  AH & AL \\
  \hline
  01 & 41h \\
  \hline
  \end{tabular}

  83h / 02h = 41h, R=1
\end{verbatim}

DIV Examples

\begin{verbatim}
  wval dw 100h ;divisor
  ...  
  mov dx, 0 ;clear dividend high!
  mov ax, 8003h ;dividend
  div wval

  Before DIV:
  \begin{tabular}{|c|c|}
  \hline
  DX & AX \\
  \hline
  0000 & 8003h \\
  \hline
  \end{tabular}

  After DIV:
  \begin{tabular}{|c|c|}
  \hline
  DX & AX \\
  \hline
  0003 & 0080h \\
  \hline
  \end{tabular}

  8003h / 0100h = 80h, R=3
\end{verbatim}
Warning

• If you don’t remember to clear DX you will get unexpected results!!!
• If you had something you were using in DX, it will get destroyed by the division.

IDIV

• IDIV works like DIV except it uses signed numbers.
• In 8-bit division, the dividend is in AX, so the sign is determined by bit 15.
• In 16-bit division, the sign is determined by bit 15 in DX.

IDIV Examples

\[\text{bval db } 5 \quad ;\text{divisor}\]
\[...\]
\[\text{mov ax, -48} \quad ;\text{dividend}\]
\[\text{idiv bval}\]

Before IDIV:

\[
\begin{array}{c|c}
\text{AH} & \text{AL} \\
\hline
\text{FF} & \text{D0}
\end{array}
\]

After DIV:

\[
\begin{array}{c|c}
\text{AH} & \text{AL} \\
\hline
\text{FD} & \text{F7}
\end{array}
\]

\[
\text{FFD0/5 = F7, R FD}
\]
\[
\text{F7 = -9, FD = -3}
\]

What NOT to Do!

\[\text{bval db } 5 \quad ;\text{divisor}\]
\[...\]
\[\text{mov ah, 0}\]
\[\text{mov ah, -48} \quad ;\text{dividend}\]
\[\text{idiv bval}\]

Before IDIV:

\[
\begin{array}{c|c}
\text{AH} & \text{AL} \\
\hline
\text{00} & \text{D0}
\end{array}
\]

After DIV:

\[
\begin{array}{c|c}
\text{AH} & \text{AL} \\
\hline
\text{03} & \text{29}
\end{array}
\]

\[
\text{00D0/5 = 29, R 3}
\]
\[
\text{29 = 41}
\]
Sign Extending

- You will need to sign extend your dividend.
- Intel provides instructions for this:
  - CBW – convert byte to word extends AL into AX
  - CWD – convert word to double word extends AX into DX:AX

IDIV Examples

```assembly
wval  dw  256  ;divisor
...
mov  ax, -5000  ;DX:AX = ???EC78h
cdw
FFFFEC78h
idiv  wval

Before IDIV:

<table>
<thead>
<tr>
<th>DX</th>
<th>AX</th>
</tr>
</thead>
<tbody>
<tr>
<td>FFFF</td>
<td>EC78</td>
</tr>
</tbody>
</table>

After DIV:

<table>
<thead>
<tr>
<th>DX</th>
<th>AX</th>
</tr>
</thead>
<tbody>
<tr>
<td>FF78</td>
<td>FFED</td>
</tr>
</tbody>
</table>

FFED = -19 (quotient)
FF78 = -136 (remainder)
```

Division Problems

- Divide Overflow is produced by:
  - a division result that is too large
  - dividing by zero
- This is not handled gracefully by the processor: your program will die a horrible death.
- You’ll need to prevent it:
  - for large numbers, use larger operands (registers).
  - for divide by zero, check for the zero yourself before you divide.
  - or, you can write a special interrupt handler (see chapter 15 if you’re interested.)