Computer Organization & Assembly Language Programming

CSE 2312
Lecture 23 Assembler and Tracer

Junzhou Huang, Ph.D.
Department of Computer Science and Engineering
Reviewing (1): Registers

• **Registers in the 8088 Processor**
  – Every processor, including the 8088, has an internal state, where it keeps certain crucial information.
  – Thus, the processor has a set of registers where this information can be stored and processed.
  – Probably the most important of these is the PC (program counter), which contains the memory location, that is, the address, of the next instruction to be executed.
  – This register is also called IP (Instruction Pointer). This instruction is located in a part of the main memory, called the code segment.

• **Different Registers**
  – General registers
  – Segment registers
  – Pointer and index registers
  – Condition code register or Flag register
  – Instruction pointers, program counter
Reviewing (2): General Registers

<table>
<thead>
<tr>
<th>General registers</th>
<th>Segment registers</th>
</tr>
</thead>
<tbody>
<tr>
<td>AX</td>
<td>AH</td>
</tr>
<tr>
<td>BX</td>
<td>BH</td>
</tr>
<tr>
<td>CX</td>
<td>CH</td>
</tr>
<tr>
<td>DX</td>
<td>DH</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Pointer and index</th>
<th>Condition codes</th>
</tr>
</thead>
<tbody>
<tr>
<td>SP</td>
<td>Stack pointer</td>
</tr>
<tr>
<td>BP</td>
<td>Base pointer</td>
</tr>
<tr>
<td>SI</td>
<td>Source index</td>
</tr>
<tr>
<td>DI</td>
<td>Destination index</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction pointer</th>
<th>Program counter</th>
</tr>
</thead>
<tbody>
<tr>
<td>IP</td>
<td>PC</td>
</tr>
</tbody>
</table>

Spring 2015 CSE 2312 Computer Organization & Assembly Language Programming
Reviewing (3):

- What is the value in AX after executing the following instructions, where 1026 is a decimal number?

```
MOV AX, 1026
DIVB AH, AL
```

- Solution

- \((1026)_{10} = (0000 0100 0000 0010)_2\)
- \(\text{MOV } AX, 1026 \rightarrow AX = (0000 0100 0000 0010)_2\)
- \(\rightarrow AH = (0000 0100)_2 \ AL = (0000 0010)_2\)
- \(\text{DIVB } AH, AL \rightarrow AH = (00000010)_2\)
- \(\rightarrow AX = (0000 0010 0000 0010)_2 = (514)_{10}\)
Reviewing (4): Pointer Registers

- **Pointer or Index Registers**
  - Stack pointer, Base pointer, Source pointer, Destination pointer

- **Stack Pointer (SP)**
  - The stack is a segment of memory that holds certain context information about the running program. Usually, when a procedure is called, part of the stack is reserved for holding the procedure’s local variables, the address to return to when the procedure has finished, and other control information.
  - The portion of the stack relating to a procedure is called its stack frame. When a called procedure calls another procedure, an additional stack frame is allocated, usually just below the current one. Additional calls allocate additional stack frames below the current ones.
  - Stacks almost always grow downward, from high addresses to low addresses. But the lowest numerical address occupied on the stack is always called the top of the stack.
  - **PUSH** puts a 16-bit word on top of the stack. This instruction first decrements SP by 2, then stores its operand at the address SP is now pointing to. **POP** removes a 16-bit word from the top of the stack by fetching the value on top of the stack and then incrementing SP by 2.
  - The **SP** register points to the top of the stack and is modified by **PUSH**, **POP**, and **CALL** instructions, being decremented by **PUSH**, incremented by **POP**, and decremented by **CALL**.
Reviewing (5): Pointer Registers

• Base Pointer (BP)
  – It usually contains an address in the stack. Whereas SP always points to the top of the stack, BP can point to any location within the stack.
  – In practice, a common use for BP is to point to the beginning of the current procedure’s stack frame, in order to make it easy to find the procedure’s local variables.
  – Thus, BP often points to the bottom of the current stack frame (the stack frame word with the highest numerical value) and SP points to the top (the stack frame word with the lowest numerical value).
  – The current stack frame is thus delimited by BP and SP.

• Source Index (SI) and Destination Index (DI)
  – SI, the source index, and DI, the destination index.
  – These registers are often used in combination with BP to address data in the stack, or with BX to compute the addresses of data memory locations.
Reviewing (6): Instruction Pointer & Segment Register

- **Instruction Pointer (PC)**
  - The instruction pointer has the Intel’s name: program counter (PC).
  - This register is not addressed directly by the instructions, but contains an address in the program code segment of the memory.
  - The instruction cycle starts by fetching the instruction pointed to by PC. This register is then incremented before the rest of the instruction is executed.
  - So, it points to the first instruction beyond the current one.

- **Segment Register**
  - The stack, the data and the instruction codes all reside in main memory, but usually in different parts of it. The segment registers govern these different parts of the memory, which are called segments.
  - There are four registers in the segment register group.
    - CS for the code segment register;
    - DS for the data segment register,
    - SS for the stack segment register;
    - ES for the extra segment register.
  - Most of the time, their values are not changed.
  - In practice, the data segment and stack segment use the same piece of memory, with the data being at the bottom of the segment and the stack being at the top.
Reviewing (7): Flag Register

- **Flag Register**
  - The flag register (condition code register) is a set of single-bit registers.
  - Some of the bits are set by arithmetic instructions and relate to the codes:
    - Z - result is zero
    - S - result is negative (sign bit)
    - V - result generated an overflow
    - C - result generated a carry
    - A - Auxillary carry (out of bit 3)
    - P - parity of the result
  - Other bits in this register control operation of certain aspects of the processor.
    - The I bit enables interrupts.
    - The T bit enables tracing mode, which is used for debugging.
    - Finally, the D bit controls the direction of the string operations.
  - Not all 16 bits of this flag register are used;
  - The unused ones are hardwired to zero.
Assembler
Assembler (1)

- **Assembly Language Program**
  - A program written using symbolic names for instructions and registers

- **Assembler**
  - To run an assembly language program, it is first necessary to translate it into the binary numbers that the CPU actually understands.
  - The program that converts an assembly language program into binary numbers is the assembler.
  - The output of the assembler is called an object file.

- **Linker**
  - Many programs make calls to subroutines that have been previously assembled and stored in libraries.
  - To run these programs, the newly assembled object file and the library subroutines it uses (also object files) must be combined into a single executable binary file by another program called a linker.
  - Only when the linker has built the executable binary file from one or more object files is the translation fully completed.
  - The OS can then read the executable binary file into memory and execute it.
Assembler (2)

- **Task**
  - The first task of the assembler is to build a symbol table, which is used to map the names of symbolic constants and labels directly to the binary numbers that they represent.
  - The defined constants can be put in the symbol table without any processing.
  - However, labels represent addresses whose values are not immediately obvious. To compute their values, the assembler scans the program line by line in what is called the first pass.

- **First Pass**
  - During this pass, it keeps track of a location counter usually indicated by the symbol “.”, pronounced dot. For every instruction and memory reservation that is found in this pass, the location counter is increased by the size of the memory necessary to contain the scanned item.

- **Linker**
  - At the start of the second pass, the numerical value of every symbol is known. Since the numerical values of the instruction mnemonics are constants, code generation can now begin.
  - One at a time, instructions are read again and their binary values are written into the object file. When the last instruction has been assembled, the object file is complete.
as 88 (1)

- **Task**
  - This assembler is Amsterdam Compiler Kit (ACK) and is patterned after UNIX assemblers rather than MS-DOS or Windows assemblers.
  - The comment symbol in this assembler is the exclamation mark (!). Anything following an exclamation mark until the end of the line is a comment and does not affect the object file produced.
  - This assembler uses three different sections, in which the translated code and data will be stored. Those sections are related to the memory segments of the machine.
- **Sections**
  - TEXT section, for the processor instructions.
  - DATA section for the initialization of the memory in the data segment, which is known at the start of the process.
  - BSS (Block Started by Symbol), section, for the reservation of memory in the data segment that is not initialized (i.e., initialized to 0).
  - Each of these sections has its own location counter.
  - The purpose of having sections is to allow the assembler to generate some instructions, then some data, then some instructions, then more data, and so on.
  - To have the linker rearrange the pieces so that all the instructions are together in the text segment and all the data words are together in the data segment.
  - Each line of assembly code produces output for only one section, but code lines and data lines can be interleaved.
  - At run time, the TEXT section is stored in the text segment and the data and BSS sections are stored (consecutively) in the data segment.
The *as88* pseudoinstructions.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>.SECT .TEXT</td>
<td>Assemble the following lines in the TEXT section</td>
</tr>
<tr>
<td>.SECT .DATA</td>
<td>Assemble the following lines in the DATA section</td>
</tr>
<tr>
<td>.SECT .BSS</td>
<td>Assemble the following lines in the BSS section</td>
</tr>
<tr>
<td>.BYTE</td>
<td>Assemble the arguments as a sequence of bytes</td>
</tr>
<tr>
<td>.WORD</td>
<td>Assemble the arguments as a sequence of words</td>
</tr>
<tr>
<td>.LONG</td>
<td>Assemble the arguments as a sequence of longs</td>
</tr>
<tr>
<td>.ASCII &quot;str&quot;</td>
<td>Store str as ascii an string without a trailing zero byte</td>
</tr>
<tr>
<td>.ASCIZ &quot;str&quot;</td>
<td>Store str as ascii an string with a trailing zero byte</td>
</tr>
<tr>
<td>.SPACE n</td>
<td>Advance the location counter n positions</td>
</tr>
<tr>
<td>.ALIGN n</td>
<td>Advance the location counter up to an n-byte boundary</td>
</tr>
<tr>
<td>.EXTERN</td>
<td>Identifier is an external name</td>
</tr>
</tbody>
</table>
as 88 (3)

• **Label**
  – An instruction or data word in the assembly language program can begin with a label.
  – A label may also appear all by itself on a line, in which case it is as though it appeared on the next instruction or data word.
  – For example, L is a label that refers to the instruction of data word following it.

• **Different Kinds of Labels**
  – Global labels: alphanumeric identifiers followed by a colon (:). These must all be unique, and cannot match any keyword or instruction mnemonic.
  – Local labels: in the TEXT section only, we can have local labels, each of which consists of a single digit followed by a colon (:). A local label may occur multiple times.
• Constants
  – The assembler allows constants to be given a symbolic name using the syntax identifier
    expression identifier = expression
  – The identifier is an alphanumeric string, as in BLOCKSIZE = 1024
  – Like all identifiers in this assembly language, only the first eight characters are significant, so BLOCKSIZE and BLOCKSIZZ are the same symbol, namely, BLOCKSIZ.
  – Expressions can be constructed from constants, numerical values, and operators.
  – Labels are considered to be constants because at the end of the first pass their numerical values are known.

• Pseudoinstructions
  – Those influence the assembly process itself but which are not translated into binary code.
as 88 (5)

• **First Block of Pseudoinstructions**
  – It determines the section in which the following lines should be processed by the assembler.
  – Usually such a section requirement is made on a separate line and can be put anywhere in the code.
  – For implementation reasons, the first section to be used must be the TEXT section, then the DATA section, then the BSS section. The first line of a section should have a global label.

• **Second Block of Pseudoinstructions**
  – It contains the data type indications for the data segment. There are four types: .BYTE, .WORD, .LONG, and string.
  – After an optional label and the pseudoinstruction keyword, the first three types expect a comma-separated list of constant expressions on the remainder of the line.
  – For strings there are two keywords, ASCII, and ASCIZ, with the only difference being that the second keyword adds a zero byte to the end of the string.
  – Both require a string between double quotes. Several escapes are allowed in string definitions.
as88 (6)

<table>
<thead>
<tr>
<th>Escape symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\n</td>
<td>New line (line feed)</td>
</tr>
<tr>
<td>\t</td>
<td>Tab</td>
</tr>
<tr>
<td>\</td>
<td>Backslash</td>
</tr>
<tr>
<td>\b</td>
<td>Back space</td>
</tr>
<tr>
<td>\f</td>
<td>Form feed</td>
</tr>
<tr>
<td>\r</td>
<td>Carriage return</td>
</tr>
<tr>
<td>&quot;</td>
<td>Double quote</td>
</tr>
</tbody>
</table>

Some of the escapes allowed by as88.
as 88 (7)

• **The SPACE Pseudoinstructions**
  – The SPACE pseudoinstruction simply requires the location pointer to be incremented by the number of bytes given in the arguments.
  – This keyword is especially useful following a label in the BSS segment to reserve memory for a variable.

• **The ALIGN Pseudoinstructions**
  – The ALIGN keyword is used to advance the location pointer to the first 2-, 4-, or 8-byte boundary in memory to facilitate the assembly of words, longs, etc. at a suitable memory location.

• **The EXTERN Pseudoinstructions**
  – EXTERN announces that the routine or memory location mentioned will be made available to the linker for external references.
  – The definition need not be in the current file; it can also be somewhere else, as long as the linker can handle the reference.
Different from Others

• **First**
  – Other assemblers do not have a .BSS section, and initialize memory only in the DATA sections.
  – Usually the assembler file starts with some header information, then the DATA section, which is indicated by the keyword .data, followed by the program text after the keyword .code

• **Second**
  – Instead of the .WORD .BYTE and ASCIZ directives, these assemblers have keywords DW for define word and DB for define byte.
  – After the DB directive, a string can be defined inside a pair of double quotes. Labels for data definitions are not followed by a colon.

• **Third**
  – The biggest difference between the MASM, TASM and as88 is in making system calls. The system is called in MASM and TASM by means of a system interrupt INT.
  – For different devices there are different interrupt vectors, and interrupt numbers, such as INT 16H for the BIOS keyboard functions and INT 10H for the display.
  – In order to program these functions, the programmer has to be aware of a great deal of device-dependent information.
  – In contrast, the UNIX system calls available in as88 are much easier to use.
Tracer
# The Tracer

<table>
<thead>
<tr>
<th>Processor with registers</th>
<th>Stack</th>
<th>Program text</th>
</tr>
</thead>
<tbody>
<tr>
<td>Subroutine call stack</td>
<td></td>
<td>Source file</td>
</tr>
<tr>
<td>Interpreter commands</td>
<td></td>
<td>Error output field</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Input field</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Output field</td>
</tr>
<tr>
<td></td>
<td>Values of global variables</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Data segment</td>
<td></td>
</tr>
</tbody>
</table>

The Tracer Window
Tracer: files

- **Tracer**
  - Usually, the tracer reads both its commands and its input from standard input. However, it is also possible to prepare a file of tracer commands and a file of input lines to be read before the control is passed to the standard input.
  - Tracer command files have extensions `.t` and input files `.i`. In the assembly language, both uppercase and lowercase characters can be used for keywords, system subroutines and pseudoinstructions.
  - During the assembly process, a file with extension `.s$` is made in which those lowercase keywords are translated into uppercase and carriage return characters are discarded.

- **Different Files**
  - For each project, say, pr we can have up to six different files:
    - 1. pr.s for the assembly source code.
    - 2. pr.$ for the composite source file.
    - 3. pr.88 for the load file.
    - 4. pr.i for preset standard input.
    - 5. pr.t for preset tracer commands.
    - 6. pr.# for linking the assembly code to the load file.
## Tracer Commands (1)

<table>
<thead>
<tr>
<th>Address</th>
<th>Command</th>
<th>Example</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>#</td>
<td>, ! , X</td>
<td>Execute one instruction</td>
</tr>
<tr>
<td>#</td>
<td>, ! , X</td>
<td>24</td>
<td>Execute # instructions</td>
</tr>
<tr>
<td>/T+##</td>
<td>g , ! ,</td>
<td>/start+5g</td>
<td>Run until line # after label T</td>
</tr>
<tr>
<td>/T+##</td>
<td>b</td>
<td>/start+5b</td>
<td>Put breakpoint on line # after label T</td>
</tr>
<tr>
<td>/T+##</td>
<td>c</td>
<td>/start+5c</td>
<td>Remove breakpoint on line # after label T</td>
</tr>
<tr>
<td>#</td>
<td>g</td>
<td>108g</td>
<td>Execute program until line #</td>
</tr>
<tr>
<td>#</td>
<td>b</td>
<td>g</td>
<td>Execute program until current line again</td>
</tr>
<tr>
<td>#</td>
<td>c</td>
<td>b</td>
<td>Put breakpoint on current line</td>
</tr>
<tr>
<td>c</td>
<td>c</td>
<td>c</td>
<td>Remove breakpoint on current line</td>
</tr>
</tbody>
</table>

The tracer commands. Each command must be followed by a carriage return (the Enter key). An empty box indicates that just a carriage return is needed. Commands with no Address field listed above have no address. The # symbol represents an integer offset.
Tracer Commands (2)

<table>
<thead>
<tr>
<th>Address</th>
<th>Command</th>
<th>Example</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>n</td>
<td>n</td>
<td>Execute program until next line</td>
</tr>
<tr>
<td></td>
<td>r</td>
<td>r</td>
<td>Execute until breakpoint or end</td>
</tr>
<tr>
<td></td>
<td>=</td>
<td>=</td>
<td>Run program until same subroutine level</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>-</td>
<td>Run until subroutine level minus 1</td>
</tr>
<tr>
<td></td>
<td>+</td>
<td>+</td>
<td>Run until subroutine level plus 1</td>
</tr>
<tr>
<td>/D+#</td>
<td>/buf+6</td>
<td></td>
<td>Display data segment on label+#</td>
</tr>
<tr>
<td>/D+#</td>
<td>d, !</td>
<td>/buf+6d</td>
<td>Display data segment on label+#</td>
</tr>
<tr>
<td>R, CTRL L</td>
<td>R</td>
<td></td>
<td>Refresh windows</td>
</tr>
<tr>
<td>q</td>
<td>q</td>
<td></td>
<td>Stop tracing, back to command shell</td>
</tr>
</tbody>
</table>

The tracer commands. Each command must be followed by a carriage return (the Enter key). An empty box indicates that just a carriage return is needed. Commands with no Address field listed above have no address. The # symbol represents an integer offset.
Tracer: Breakpoint and Subroutine

- **Breakpoint**
  - Done with the command `b`, which can be optionally preceded by an instruction label, possibly with an offset. If a line with a breakpoint is encountered during execution, the tracer stops.
  - To start again from a breakpoint, a return or run command is required. If the label/number are omitted, then the breakpoint is set at the current line.
  - The breakpoint can be cleared by a breakpoint clear command, `c`, which can be preceded by labels and numbers, like the command `b`.
  - There is a run command, `r`, in which the tracer executes until either a breakpoint, an exit call, or the end of the commands is encountered.
- **Subroutine**
  - The tracer also keeps track of the subroutine level at which the program is running. This is shown in the window below the processor window and can also be seen through the indication numbers in the stack window.
  - The `–` command causes the tracer to run until the subroutine level is one less than the current level. What this command does is execute instructions until the current subroutine is finished.
  - The converse is the `+` command, which runs the tracer until the next subroutine level is encountered.
  - The `=` command runs until the same level is encountered, and can be used to execute a subroutine at the CALL command. If `=` is used, the details of the subroutine are not shown in the tracer window.
Program: Div

```assembly
.EXIT      - 1            ! 1
_PRINTF   = 127           ! 2
.SECT .TEXT             ! 3 ===== Begin of code segment =====
start:                     ! 4 init AX and BX
    MOV     AX, 1028     ! 5 AX = 1028
    MOV     BX, 514      ! 6 BX = 514
    MOV     CX, AX       ! 7 CX = AX, backup value of AX for later division
    MOV     AX, 0        ! 8 AX = 0, clean up AH and AL
    MOVB    AL, CH       ! 9 AL = CH, CH is old value of AH, assigned in Line 40
    DIVB    BH           ! 10 AH:AL = AX/BH, AH is the remainder, AL is the quotient
    MOVB    CH, AL       ! 11 CH = AL, backup result
    MOV     AX, 0        ! 12 AX = 0, clean up AH and AL
    MOVB    AL, CL       ! 13 AL = CL, CH is old value of AL, assigned in Line 40
    DIVB    BL           ! 14 AH:AL = AX/BL, AH is the remainder, AL is the quotient
    MOVB    CL, AL       ! 15 CL = AL, backup result
    MOV     AX, CX       ! 16 AX = CX, store the result back to AX
output4:                      ! 17 printf("%d, %d, %d, %d\n", AH, AL, BH, BL);
    MOV     CX, 0        ! 18 output4
    MOVB    CL, BL       ! 19
    PUSH    CX           ! 20
    MOVB    CL, BH       ! 21
    PUSH    CX           ! 22
    MOVB    CL, AL       ! 23
    PUSH    CX           ! 24
    MOVB    CL, AH       ! 25
    PUSH    CX           ! 26
    PUSH    pfmf3        ! 27
    PUSH    _PRINTF      ! 28
    SY S                 ! 29 printf will modify AX as its return value
    ADD     SP, 4        ! 30
    POP     CX           ! 31
    MOVB    AH, CL       ! 32 BH = CL
    POP     CX           ! 33 CX = old BL, pushed in stack in Line 16
    MOVB    AL, CL       ! 34 BL = CL
    ADD     SP, 4        ! 35 clean up the stack
exit:                         ! 36 exit(0)
    PUSH    0            ! 37 Just return success(0)
    PUSH    _EXIT        ! 38 system call id exit
    SYSTEM             ! 39 make system call
.SECT .DATA                  ! 40 ===== Begin of data segment =====
pfmf3:                           ! 45 .ASCIZ "%d, %d, %d, %d\n"       ! 46 Define format string for print four 16bit integer
.SECT .BSS                    ! 47 ===== Begin of BSS segment =====
```

Spring 2015  CSE 2312 Computer Organization & Assembly Language Programming  26