Computer Organization & Assembly Language Programming

CSE 2312
Lecture 13 Flow of Control

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

• Different Instructions
  – Data Movement Instructions: copy data from one place to another place, =
  – Dyadic Instructions: combine two operands to produce a result, +, -
  – Monadic Instructions: have one operand and produce a result
  – Instructions for Comparisons and Conditional Branches: test some condition and branch to a particular memory address if the condition is met
  – Procedure Call Instructions: call subroutine
  – Loop Control Instructions: execute a group of instruction a fixed number of times
  – Input/Output Instructions:
    1) programmed I/O;
    2) Interrupt-driven I/O;
    3) DMA I/O
Reviewing (2): Dyadic Instruction

- **Dyadic Instructions**
  - Combine two operands to produce a result, +, -

- **A AND B**
  - The result of this operation is that the unwanted bits are all changed into zeros—that is, masked out

- **A OR B**
  - While the AND tends to remove 1s, because there are never more 1s in the result than in either of the operands. The OR operation tends to insert 1s,

\[
\begin{array}{cccccccccccc}
10110111 & 10111100 & 11011011 & 10001011 & A \\
11111111 & 11111111 & 11111111 & 00000000 & B \text{(mask)} \\
10110111 & 10111100 & 1101101100000000 & \text{AANDB} \\
00000000 & 00000000 & 00000000 & 01010111 & C \\
10110111 & 10111100 & 11011011 & 01010111 & (A \text{ AND B}) \text{ OR C}
\end{array}
\]
Reviewing (3): Monadic Instructions (1)

• **Monadic Instructions**
  – Have one operand and produce a result

• **Shift**
  – Operations in which the bits are moved to the left or right, with bits shifted off the end of the word being lost.
  – An important use is multiplication and division by powers of 2
  – Example: left shift k bits: the original number multiplied by $2^k$
  – Example: right shift k bits: the original number divided by $2^k$
  – Example: $18 \times n = 16 \times n + 2 \times n$; Then, a move, two shift and an addition instead of the multiplication

```
00000000 00000000 00000000 01110011 A
00000000 00000000 00000000 00011100 A shifted right 2 bits
11000000 00000000 00000000 00011100 A rotated right 2 bits
```
Reviewing (4): Monadic Instructions (2)

- **Rotate**
  - Shifts in which bits pushed off one end reappear on the other end.
  - Useful for packing and unpacking bit sequence from words
  - For example, test all bits in a word. Restore the original word after test all bits

- **Signed Number Representations**
  - One's complement negative number: BITWISE NOT applied to it — the "complement" of its positive counterpart.
  - Example: one's complement of 00101011 (43) becomes 11010100 (−43).
  - Two's complement negative number: The bit pattern which is one greater (in an unsigned sense) than the one's complement of the positive value.
  - Example: An easier method to get the negation of a number in two's complement is as follows:

<table>
<thead>
<tr>
<th>Example 1</th>
<th>Example 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0101001</td>
<td>0101100</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>1010111</td>
<td>1010100</td>
</tr>
</tbody>
</table>
Reviewing (5): Monadic Instructions (3)

- Shift Signed Number
  - Left shifting one’s complement negative numbers does not multiply by 2.
  - Right shifting does simulate division correctly, however.

```
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 A
0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 A shifted without sign extension
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 A shifted with sign extension
```

Right Shift are often performed with signed extension

```
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1 in one’s complement
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 -1 shifted left 1 bit = -3
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 -1 shifted left 2 bits = -7
```

Shifting negative number
Reviewing (6): Loop Control

- Loop
  - Execute a group of instructions a fixed number of times
  - A counter that is increased or decreased by some constant once each time through the loop. The counter is also tested once each time through the loop. If a certain condition holds, the loop is terminated.
  - Test-at-the-end looping has the property that the loop will always be executed at least once, even if \( n \) is less than or equal to 0.

```
L1: first-statement;
   
   
   last-statement;
i = i + 1;
if (i < n) goto L1;
```

(a) Test-at-the-end loop.

```
L1: if (i > n) goto L2;
first-statement;
   
   
   last-statement
i = i + 1;
goto L1;
```

(b) Test-at-the-beginning loop.
Reviewing (7): Programming I/O

- **Programming I/O**
  - The simplest possible I/O method, which is commonly used in low-end microprocessors, for example, in embedded systems or in systems that must respond quickly to external changes (real-time systems).
  - These CPUs usually have a single input instruction and a single output instruction.
  - Each of these instructions selects one of the I/O devices.
  - A single character is transferred between a fixed register in CPU and the selected I/O device.
  - The processor must execute an explicit sequence of instructions for each and every character read or written.
Reviewing (8): DMA I/O

**DMA I/O**
- To write a block of 32 bytes from memory address 100 to a terminal (device 4),
- CPU writes 32, 100, 4 and 1 (WRITE) into four DMA registers
- DMA controller makes a bus request to read byte 100 from the memory, the same way the CPU would read from the memory.
- DMA controller then makes an I/O request to device 4 to write the byte to it.
- DMA controller increments its address register by 1 and decrements its count register by 1.
- If the count register is still greater than 0, another byte is read from memory and then written to the device.

DMA always has a higher bus priority than the CPU
5.6 Flow Control

• **What?**
  – Flow of control refers to the sequence in which instructions are executed dynamically during program execution.

• **Related Factors**
  – Successively-executed instructions are fetched from consecutive memory locations, in the absence of branches and procedure calls.
  – Procedure calls cause the flow of control to be altered, stopping the procedure currently executing and starting the called procedure.
  – Coroutines are related to procedures and cause similar alterations in the flow of control. They are useful for simulating parallel processes.
  – Traps and interrupts also cause the flow of control to be altered when special conditions occur.
Sequential Flow of Control and Branches

- **Most instructions do not alter the flow of control.**
  - After each instruction, the program counter is increased by the instruction length.
  - If observed over an interval of time that is long compared to the average instruction time, the program counter is approximately a linear function of time.

- **Containing Branches**
  - This relation is not true
  - Program Counter is no longer a monotonically increasing function of time
  - It becomes difficult to visualize the instruction execution sequence from the program listing.

Program counter as a function of time (smoothed)

(a) Without branches. (b) With branches.
Procedures

• What?
  – The most important technique for structuring programs

• Procedures and Flow of Control
  – A procedure call alters the flow of control just as a branch does
  – Unlike the branch, when finished performing its task, it returns control to the statement or instruction following the call.
  – A procedure body can be regarded as defining a new instruction on a higher level.
  – Procedure call can be thought of as a single instruction, even though the procedure may be quite complicated.
  – To understand a piece of code containing a procedure call, it is only necessary to know what it does, not how it does it.

• Recursive Procedure
  – A procedure that calls itself, either directly or indirectly via a chain of other procedures.
  – Tower of Hanoi: an example of a recursive procedure.
Recursive Procedures (1)

- **Towers of Hanoi**
  - There are three gold pegs. Around the first one were a series of 64 concentric gold disks, each with a hole in the middle for the peg.
  - Each disk is slightly smaller in diameter than the disk directly below it. The second and third pegs were initially empty.
  - How to transfer all the disks to peg 3, one disk at a time, but at no time may a larger disk rest on a smaller one?
Recursive Procedures (2)

The steps required to solve the Towers of Hanoi for three disks.
Recursive Procedures (3)

The steps required to solve the Towers of Hanoi for three disks.
Recursive Procedures (4)

The number of moves required to solve a Tower of Hanoi is $2^n - 1$, where $n$ is the number of disks.

A procedure for solving the Towers of Hanoi:

```java
public void towers(int n, int i, int j) {
    int k;
    if (n == 1)
        System.out.println("Move a disk from "+i+" to "+j);
    else {
        k = 6 - i - j;
        towers(n - 1, i, k);
        towers(1, i, j);
        towers(n - 1, k, j);
    }
}
```
Recursive Procedures (5)

The stack at several points during the execution
Exercise

- 4 disks. **How? How many moving-operations?**
The Calling/Called Procedures

• Distinction
  – Procedure A calls a procedure B
  – Symmetric situation: neither A nor B is a main program, both being procedures; first control is transferred from A to B—the call—and later control is transferred from B to A—the return.
  – Asymmetry: 1) when control passes from A to B, procedure B begins executing at the beginning; when B returns to A, execution starts not at the beginning of A but at the statement following the call.
  – If A calls B many times, B starts at the beginning all over again each and every time, whereas A never starts over again.
Coroutines

• **What?**
  – Two procedures, each of which calls the other as a procedure.
  – When B returns to A, it branches to the statement following the call to B.
  – When A transfers control to B, it does not go to the beginning (except the first time) but to the statement following the most recent “return”
  – Two procedures that work this way are called coroutines.

• **Usage?**
  – Simulate parallel processing on a single CPU. Each coroutine runs in pseudo-parallel with the others, as though it had its own CPU.
  – It also is useful for testing software that will later actually run on a multiprocessor.

When a coroutine is resumed, execution begins at the statement where it left off the previous time, not at the beginning.
Trap

- Trap
  - An automatic procedure call initiated by some condition caused by the program, usually an important but rarely occurring condition.
  - A good example is overflow. If the result of an arithmetic operation exceeds the largest number that can be represented, a trap occurs to switch the flow of control to some fixed memory location instead of continuing in sequence.
  - At that fixed location is a branch to a procedure called the trap handler, which performs some appropriate action, such as printing an error message.
  - If the result of an operation is within range, no trap occurs.

- Common conditions that can cause traps
  - floating-point overflow, floating-point underflow, integer overflow
  - protection violation, undefined opcode, stack overflow,
  - attempt to start nonexistent I/O device,
  - attempt to fetch a word from an odd-numbered address
  - division by zero.
Interrupt

• **Interrupt**
  – Interrupts are changes in the flow of control caused not by the running program, but by something else, usually related to I/O.
  – When finished, the interrupt handler returns control to the interrupted program.
  – It must restart the interrupted process in exactly the same state that it was in when the interrupt occurred, which means restoring all the internal registers to their preinterrupt state.

• **Trap vs. Interrupt**
  – Traps are synchronous with the program and interrupts are asynchronous.
  – If the program is rerun a million times with the same input, the traps will reoccur in the same place each time but the interrupts may vary, depending, for example, user I/O operations.
  – The reason for the reproducibility of traps and irreproducibility of interrupts is that traps are caused directly by the program and interrupts are indirectly caused by the program.
Interrupt Priority

• Interrupt Priority
  – Assign each I/O device a priority, high for very critical devices and low for less critical devices.
  – Similarly, the CPU should also have priorities, typically determined by a field in the PSW.
  – When a priority n device interrupts, the interrupt routine should also run at priority n.
  – While a priority n interrupt routine is running, any attempt by a device with a lower priority to cause an interrupt is ignored until the interrupt routine is finished and the CPU goes back to running lower priority code.
  – On the other hand, interrupts from higher-priority devices should be allowed to happen with no delay.
Interrupt Example

Time sequence of multiple interrupt example.
Towers of Hanoi in Pentium 4 Assembly (1)

```assembly
.MODEL FLAT
PUBLIC _towers
EXTERN _printf:NEAR
.CODE
_towers:  PUSH EBP
          MOV EBP, ESP
          CMP [EBP+8], 1
          JNE L1
          MOV EAX, [EBP+16]
          PUSH EAX
          MOV EAX, [EBP+12]
          PUSH EAX
          PUSH OFFSET FLAT:format
          CALL _printf
          ADD ESP, 12
          JMP Done
```

; compile for Pentium (as opposed to 8088 etc.)
; export 'towers'
; import printf
; save EBP (frame pointer) and decrement ESP
; set new frame pointer above ESP
; if (n == 1)
; branch if n is not 1
; printf(" ...", i, j);
; note that parameters i, j and the format
; string are pushed onto the stack
; in reverse order. This is the C calling convention
; offset flat means the address of format
; call printf
; remove params from the stack
; we are finished
## Towers of Hanoi in Pentium 4 Assembly (2)

| L1: | MOV EAX, 6 | ; start k = 6 – i – j |
|     | SUB EAX, [EBP+12] | ; EAX = 6 – i |
|     | SUB EAX, [EBP+16] | ; EAX = 6 – i – j |
|     | MOV [EBP+20], EAX | ; k = EAX |
|     | PUSH EAX | ; start towers(n – 1, i, k) |
|     | MOV EAX, [EBP+12] | ; EAX = i |
|     | PUSH EAX | ; push i |
|     | MOV EAX, [EBP+8] | ; EAX = n |
|     | DEC EAX | ; EAX = n – 1 |
|     | PUSH EAX | ; push n – 1 |
|     | CALL _towers | ; call towers(n – 1, i, 6 – i – j) |
|     | ADD ESP, 12 | ; remove params from the stack |
|     | MOV EAX, [EBP+16] | ; start towers(1, i, j) |
|     | PUSH EAX | ; push j |
|     | MOV EAX, [EBP+12] | ; EAX = i |
|     | PUSH EAX | ; push i |
|     | PUSH 1 | ; push 1 |
|     | CALL _towers | ; call towers(1, i, j) |
Towers of Hanoi in Pentium 4 Assembly (3)

```
ADD ESP, 12 ; remove params from the stack
MOV EAX, [EBP+12] ; start towers(n – 1, 6 – i – j, i)
PUSH EAX ; push i
MOV EAX, [EBP+20] ; EAX = k
PUSH EAX ; push k
MOV EAX, [EBP+8] ; EAX = n
DEC EAX ; EAX = n-1
PUSH EAX ; push n – 1
CALL _towers ; call towers(n – 1, 6 – i – j, i)
ADD ESP, 12 ; adjust stack pointer

Done:
LEAVE ; prepare to exit
RET 0 ; return to the caller

.DATA
format DB "Move disk from %d to %d\n" ; format string
END
```