Still More Assembler Programming

COMP375 Computer Architecture and Organization

Goals for Today

• Introduce assembler language constructs to:
  – If statements
  – Loops
  – Index an array

• Write simple assembler programs.

No Operation

• The NOP instruction is a one byte instruction that does nothing.
• Executing a NOP does not change any registers or status bits.
• When patching a machine language program, it is sometimes useful to be able to add a few extra instructions that don’t change the program.

Intel Status Register

• The status register records the results of executing the instruction.
• Performing arithmetic sets the status register.
• The compare instruction does a subtraction, but doesn’t store the results. It just sets the status flags.
• All jump instructions are based on the status register.
Intel Status Register

Flag Setting During Execution

```c
int dog=-3, cat=3, bird=5, cow;
__asm { // cow = dog + cat - bird;
    mov eax,dog
    add  eax,cat
    sub eax,bird
    mov cow,eax
}
```

Compare Instruction

- The `cmp` instruction compares two values and sets the status flags appropriately.
  ```c
  cmp  register, operand
  ```
- where the operand can be a memory location or a register
- The compare instruction subtracts the operand from the register value, but does not save the results.

Jump statements

- A `JMP` instruction (sometimes called `branch`) causes the flow of execution to go to a specified location.
- A `JMP` instruction loads the Program Counter with the specified address.
- An unconditional jump always jumps.
- A conditional jump will do nothing if the condition is not met.
- Some architectures have a separate compare instruction.
Labels in Assembler

- You can attach a name to a memory location in assembler. This allows you to use the name instead of numerical address.
- Labels start in first column and end with a colon:

  ```
  jmp rabbit
  // some other stuff here
  rabbit: mov eax, dog
  ```

Jumps Based on Status Flags

- **JE** Jump if equal $ZF=1$
- **JZ** Jump if zero $ZF=1$
- **JNE** Jump if not equal $ZF=0$
- **JNZ** Jump if not zero $ZF=0$
- **JLE** Jump if less or equal $ZF=1$ or $SF\neq OF$
- **JL** Jump if less $SF\neq OF$
- **JNS** Jump if not sign $SF=0$
- **JS** Jump if sign $SF=1$
- **JGE** Jump if greater or equal $SF=OF$
- **JG** Jump if greater $ZF=0$ and $SF=OF$

Program Counter

- The Program Counter or Instruction Pointer Register contains the address of the next instruction to be executed.
- At the beginning of the fetch/execute cycle, the CPU fetches the instruction whose address is in the program counter.
- A jump instruction is just a load of the program counter register.

Software Controls

- Assembler only has a simple compare instruction. Jumps are based on the compare.
- Assembler does not have:
  - `for`
  - `while`
  - `do while`
  - `switch`
  - `break`
  - `else` portion of an `if`
### If statements

- The high level language IF statement is easily implemented by a conditional jump.

```assembly
if (cat == dog)
    cow = goat;
else
    cow = bull;
```

```assembly
MOV eax, cat
CMP eax, dog
JNE noteq
MOV edx, goat
JMP after

noteq: MOV edx, bull
after: MOV cow, edx
```

### Loops

- There are usually no hardware instructions that directly implement loops (i.e. for, while, do)

```assembly
while (what == ever) {
    again: mov eax, what
    cmp eax, ever
    jne endloop
    // something
    jmp again
endloop:
```

### Try It

- Complete this program in assembler

```assembly
int cow=0, dog=0, cat=3;
_asm{
    do { // convert this to assembler
        cow++;
        dog = dog + cat;
    } while (dog < 12);
}
```

### Possible Solution

```assembly
cow=0; dog=0; cat=3;
_asm{
    mov eax, dog ; put dog in eax
    again: inc cow ; cow++
    add eax, cat ; add cat to dog
    cmp eax, 12 ; < 12 ?
    jl again ; repeat if not
    mov dog, eax ; save result to dog
}
```
**Addresses**

- Assembler programs often have to manipulate addresses.
- A pointer in C++ represents an address in assembler.
- You may need to use addresses to follow links in a data structure or to get an element from an array.

**Intel Assembler Addresses**

- You can load a register with the address of a memory location by using the Load Effective Address, lea, instruction.
  
  `lea eax, dog ; eax = addr of dog`

- If the memory location is based on indexing, the lea instruction will compute the correct address.
  
  `lea eax, dog[esi] ; effective addr`

**Indexing**

- To specify that the address of the data is in a register in Intel assembler, you put the register in the operand field in [brackets].

  ```
  //    char cat[47], goat;
  //    goat = cat[10];
  lea  ebx, cat  ; ebx = addr of cat
  add  ebx, 10  ; add 10 to address
  mov  al, [ebx] ; al = cat[10]
  mov  goat, al  ; save in goat
  ```

**Assembler Pointers**

```
Indexing Arrays

• An array is a sequential collection of data values at consecutive addresses.
• The first element of an array (index 0) is at the start address of the array.
• The second element’s address is the start address of the array plus the size of each element.

Program to Sum 5 Numbers

```c
int main(){
    int sum = 0;
    /* Sum in C++ */
    int i;
    for (i = 0; i < 5; i++) {
        sum = sum + arrayA[i];
    }
}
```

/* Sum in Assembler */

```assembly
_asm{
    push  esi ; save value of esi pointer
    lea  esi, arrayA ; esi = start addr of arrayA
    mov  eax, 0 ; eax = 0, initialize sum
    mov  ebx, 5 ; ebx = 5, loop counter
    forloop:
    add  eax, [esi] ; add next value of array to eax
    add  esi, 4 ; increment esi to next element
    sub  ebx, 1 ; decrement loop counter
    jnz  ebx, forloop ; repeat if not zero
    mov  sum, eax ; move result to sum
    pop  esi ; restore esi pointer
}
```
Two Dimensional Arrays

- Consider the two dimensional array
  
  ```c
  int aray[2][3];
  ```
- This is allocated in memory as:
  
  | 0,0 | 0,1 | 0,2 | 1,0 | 1,1 | 1,2 |
- To set \( x = \text{aray}[i][j] \):
  
  - temp = size of int * (i*3 + j)
  - get the value at the address computed as the address of the start of aray + temp

2D Array in Assembler

```assembly
int aray[2][3], i, j, rat;
_asm {
    // rat = aray[i][j];
    mov ebx, i ; ebx = i
    imul ebx, 3 ; ebx = i*3
    add ebx, j ; ebx = i*3 + j
    mov eax, aray[ebx] ; eax = aray[i][j]
    mov rat, eax ; store in rat
}
```