Computer Organization

COMP2120

Qi Zhao

April 15, 2024

Assembly Language Programming

High Level vs. Assembly Language

High Level Languages

- More programmer friendly
- More ISA (instruction set architecture) independent
- Each high-level statement translates to several instructions in the ISA of the computer

Assembly Languages

- Lower level, closer to ISA
- Very ISA-dependent
- One assembly language instruction is translated into one machine instruction by the assembler
- Makes low level programming more user friendly
- More efficient code

Assembly Language Programming

- Each line has four fields:

```
[label]   mnemonic   operand list   comment
```

- Example:

```
a:      .word 0          # create storage, a=0
loop:   add r8,r10,r8    # r8+=r10
```

- Mnemonic Field: Instruction, Assembler directive
- Assembler directives are directions to the assembler to take some action or change a setting. Assembler directives do not represent instructions, and are not translated into machine code.
- Similar to compiler directives (`#define, #include` in C++)
- The following tables show some useful assembler directives.

Assembler Directives

Directives	Description
`.data`	Tells the assembler to add all subsequent data to the data section.
`.text`	Tells the assembler to add subsequent code to the text section (i.e. program section)
`.globl` name	Makes name external to other files, for multiple files in the program.
`.space expression`	Reserves spaces, amount specified by the value of expression in bytes. The assembler fills the space with zeros.
`.word value1 [,value2],`	Put the values in successive memory locations.

Assembly Language Programming

- We will not focus on any CPU. Instead, we invent a simple instruction set for a hypothetical machine.
- In our instruction, capital letter and small letter is equivalent.
- The destination operand is the last operand. (some instruction sets have the destination operand as the first operand)
- Comments — Anything after # will be comments. (Some assemblers use ;)

Control Structures (If-then-else construct)

Example:

```
if (a[0] >= a[1]) x=a[0];
else x=a[1];
```

ALP

```
        .data    # data segment
 a:     .word 1  # create storage containing a[0]=1
        .word 3  # create storage containing a[1]=3
 x:     .word 4  # create storage containing 4, x=4
        .text    # program segment
main:
        ld #a, r8       # r8 = address of a (#a)
        ld 0(r8),r9     # r9 = a[0]=1
        ld 4(r8),r10    # r10= a[1]=3
        bgt r9,r10,f1   # branch if r9>r10, goto f1
        st r10,x        # x=r10
        br f2           # goto f2
f1:     st r9,x         # x=r9
f2:     ret             # return to OS (same as return in C++)
```

Repetition construct

Example:

```
a=0;
for (i=0; i<10; i++) a+=i;
```

ALP

```
        .data
a:      .word 0
        .text
main:
        sub r8,r8,r8       # r8=0, or xor r8,r8,r8
        ld #0xa,r9         # r9=0xa=10, no.  of iterations
        sub r10,r10,r10    # r10=0, loop counter
        ld #1,r11          # r11=1
f1:     add r8,r10,r8      # r8+=r10
        add r10,r11,r10    # r10++, increment counter
        blt r9,r10,f1      # if (r9<r10) goto f1
        st r8,a            # a=r8
        ret                # return
```

While-construct

Example:

```
temp=0;
a=1;
while (temp < 100){
  temp+=a;
  a++;
}
```

ALP

```
     # fill in the .data, .text part as in
     # previous examples
     sub r8,r8,r8      # r8 is temp, r8=0
     ld #1,r9          # r9 is a, r9=1
     mv r9,r10         # r10=r9
     ld #0x64,r11      # r11=100
f1:  add r8,r9,r8      # temp+=a
     add r9,r10,r9     # a++
     blt r8,r11,f1     # if (r8<r11) goto f1
```

Function Calls

- Use `call` and `ret` for function calls. Return address stored in the stack.
- You need to specify the input parameters and the output parameters for your function. They may be put in registers or memory.
- For example, two input parameters stored in `r8` and `r9`, and return parameter is in `r10`.
- If you change any values of the registers in the function, you need either to spell it out in the program specification, or push the original value, and pop it out at the end of the function call

Example:

Complex Number Addition and Multiplication

```
        .data
ar:     .word 1    # real part of a
ai:     .word 5    # imaginary part of a
br:     .word 2    # b
bi:     .word 3
cr:     .word 0    # c
ci:     .word 0
dr:     .word 0    # d
di:     .word 0
```

We will calculate $(1 + 5i) + (2 + 3i)$ and $(1 + 5i) \times (2 + 3i)$

Example:

```
        .text
main    ld ar,r8        # setup parameters for cadd
        ld ai,r9
        ld br,r10
        ld bi,r11
        call cadd       # call cadd
        st r12,cr       # store result
        st r13,ci
        call cmult      # call cmult
        st r12,dr       # store result
        st r13,di
        ret             # return
```

Example:

```
cadd:  #complex add subroutine
       #input parameter (r8,r9) (r10,r11)
       #output parameter(r12,r13)

       add r8,r10,r12
       add r9,r11,r13
       ret              #return
```

Example:

```
cmult:  #complex multiplication
        #input parameter (r8,r9) (r10,r11)
        #output parameter (r12, r13)
        #push r14, save register 14
        push r14           #r14 used in this function
        mul r8,r10,r12     #r12=mul real part
        mul r9,r11,r14     #r14=mul img part
        sub r12,r14,r12    #result.real
        mul r8,r11,r13     #imaginary parts
        mul r9,r10,r14
        add r13,r14,r13
        pop r14            #restore r14
        ret                # return
```

Example

- Assuming that there is no `MUL` (integer multiply) instruction for your CPU. Write a function `MYMUL` to compute input `R8` $\times$ input `R10` and return the result in `R12`.

Example

Convert all characters into upper case letters.

```
        .data
a:      .asciiz "This is a test"
        # zero-terminated string

        .text
main:   sub r9,r9,r9      # r9=0
loop:   lb a(r9),r10      # load byte
        beq r10,#0,exit   # r10==0?  end of string
        call capitalize   # call capitalize
        sb r10, a(r9)     # store result back
        add r9, #1,r9     # incr r9, next char
        br loop           # goto loop
exit:   ret               # return
```

Example

Suppose that characters `a` to `z` are represented by bytes `0x61` to `0x7a`, and capital characters `A` to `Z` are represented by `0x81` to `0x9a`.

```
Capitalize:
#input is r10 ,output is r10, if r10 is lower
#case letter, change to upper case
       push r8
       push r9
       ld #0x61,r8          #r8='a'
       ld #0x7a,r9          #r9='z'
       blt r10,r8,ret1
       bgt r10,r9,ret1
       sub r10,#0x20,r10    #0x20='a'-'A'
ret1:  pop r9
       pop r8
       ret                  # return
```