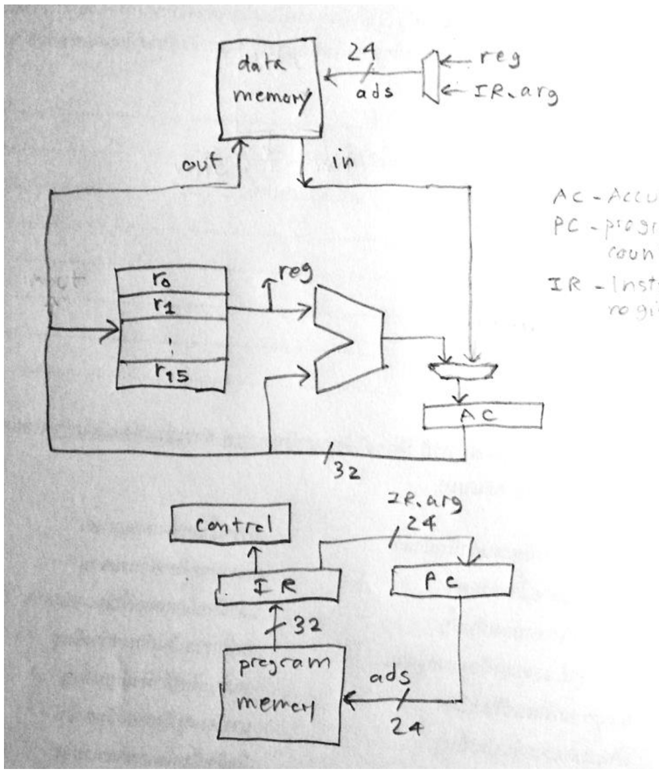
Simple Machine Z0

This is a hypothetical machine. It is used to illustrate a structure of processor. We can define and program this machine on paper. It has a minimal set of instructions that have simple form. First, we define the machine itself. Z0 has fixed width instructions. Its natural size is 32 bits. It has 16 registers.



<picture of architecture of Z0>

Instruction Set

Z0 has what is called "single address" instruction format. That is, an instruction has only one argument. An instruction is 32-bit, with opcode 8 bits and operand 24 bits.

```
    Opcode:8    Operand:24
```

We define the first group of instruction <operator> or <op>

Assembly language: `op reg`

Where reg is the register r0..r15

op is `add, sub, inc, dec`

The result is mostly stored in a special register called “accumulator”. Most operators store results there.

```
add r1       means    ac = ac + r1
sub r1                ac = ac – r1
inc r1                r1 = r1 + 1
dec r1                r1 = r1 – 1
```

Another group of instructions perform data movement between registers.

```
mov r1                ac = r1
mvi n                 ac  = n     where n is a 24-bit constant, called immediate
put r1                r1 = ac     that is a compliment operation of mov r1
```

The next group of instructions is the logic group. They are used for comparison of two values. They affect “flag” in the machine. This flag is one bit storage (True/False) and it is used in the transfer of control instruction (explain next).

<logic op> `eq ne lt le gt ge z` equal, not equal, less that, less than or equal … zero

```
eq r1                 flag = (ac == r1)
ne r1                 flag = (ac != r1)
lt r1                 flag = (ac < r1)
. . .
z r1                  flag = (r1 == 0)
```

The next group of instructions affects the transfer of control of program (or jump).

```
jmp ads               unconditional jump to ads, where ads is the location of the instruction
jt ads                jump to ads if flag is True
jf ads                jump to ads if flag is False
stop                  this is a pseudo instruction to tell us that the program has ended.
```

Please note that Z0 as defined right now, does not have “external” memory (we will add that later). All it has is the internal registers (16 of them). Each can hold a 32-bit value.

Now we have enough instructions to perform some computation. Let us try some simple program in the assembly language of Z0.

Example 1 A simple arithmetic statement. B = C + D

We assign r1, r2, r3 to B, C and D. (Notice that we preserve r0)

```
mvi 0                 ;  ac = 0
add r2                ;  ac = ac + r2
add r3                ;  ac = ac + r3
put r1                ;  r1 = ac
stop
```

Encoding of instruction

Each instruction has a 8-bit code. We assign them as follows:

```
1 add,  2 sub,  3 inc,  4 dec,  5 mov,  6 mvi,  7 -,    8 put,  9 eq,
10 ne, 11 lt,  12 le,  13 gt,  14 ge,  15 z,   16 jmp, 17 jt,  18 jf,
19 stop
```

We can write the above program in "machine code" as follows. Assume we start the program at location 0 and we write down each instruction as opcode,operand.

```
Address  instruction
   0     6,0          mvi 0
   1     1,2          add r2
   2     1,3          add r3
   3     8,1          put r1
   4     19,0         stop
```

Let us try some other program that contains loop.

Example 2 Add 1..10

Pseudo code

```
     i = 1
     s = 0
     while i <= 10
           s = s + i
           i = i + 1
```

Z0 assembly language: Let r1 be i, r2 s, r3 10

```
     mvi 10
     put r3
     mvi 1
     put r1
     mvi 0
     put r2
:loop
     mov r2
     le r3                   ;  i <= 10
     jf  exit
     mov r2
     add r1
     put r2                  ;  s = s + i
     inc r1                  ;   i = i + 1
     jmp loop
:exit
     stop
```

Notice that we use "label" such as "loop" and "exit" to designate the location. We use the prefix ":" to signify the labels but no prefix when they are used in the instructions. They are used with jump instructions. Again we write down machine code here:

Address   Instruction

```
0     6,10  mvi 10
1     8,3   put r3
2     6,1   mvi 1
3     8,1   put r1
4     6,0   mvi 0
5     8,2   put r2
  :loop
6     5,2   mov r2
7     12,3  le r3                     ;  i <= 10
8     18,14 jf  exit
9     5,2   mov r2
10    1,1   add r1
11    8,2   put r2                    ;  s = s + i
12    3,1   inc r1                    ;   i = i + 1
13    16,6  jmp loop
  :exit
14    19,0  stop
```

Note that the labels become the real address. "loop" is the address 6. "exit" is the address 14. The "assembler" is the program that performs the translation of "assembly" language into machine codes. The assembler can handle "label" automatically.

Example 3   Max of three numbers. We use comparisons that form a decision tree.
Let numbers be B, C, D

```
                      B > C
         Y                                N
       B > D                            B > D
 Y                N               Y               N
max is B      max is D        max is C       max is D
```

Let r1 be B, r2 C, r3 D, r4 max

```
     mov r1
     gt r2           ; B > C
     jf   no1
     mov r1
     gt r3           ; B > D
     jf  no2
     mov r1
     put r4            ;  max = B
     jmp exit

:no2
     mov  r3
     put r4         ; max = D
     jmp exit
:no1
     mov r2
```

```
      gt r3           ; C > D
      jf no3
      mov r2
      put r4        ; max = C
      jmp exit
:no3
      mov r3
      put r4       ; max = D
:exit
      stop
```

You will probably notice by now that programming in machine language (assembly language) is simple but it is tedious. In fact, machine language is the simplest computer language (you opinion may be different from mine).

To extend the storage to main memory, we need additional instructions to move data between registers and memory.

```
ld ads                  ;   ac = M[ads]
st ads                  ;   M[ads] = ac
ldd r1                  ;   ac = M[r1]
std r1                  ;   M[r1] = ac
```

with these encoding

```
20 ld, 21 st,  22 ldd,  23 std
```

Now it is time to talk about "addressing space". The amount of memory that can be "directly" access by an instruction depends on the number of bit of argument in the instruction. In our machine, it is 24 bits. Therefore Z0 has 16 M words memory (each word is 32 bits). This is comparable to a 64 Mbytes of conventional RAM we have in our present laptop machine. We have "direct" addressing and "indirect" addressing. In direct address, the argument is used to locate the "address" directly, for example, ld 100 will access the memory at location 100. The indirect address, the value of a register is used as the address (this is called an effective address). In fact, a register has 32-bit value so theoretically we can use indirect address of our machine to address the main memory as large as 16G bytes.

Now let us program with some data structure. The most frequently used data structure is Array. Indirect addressing is used to perform indexing of an array. We can access to the value of an element as follows. Let ax[.] be an array. An index is in r1. The base address (the first location where ax resides) is in r2.

C = ax[i] can be written as ; let r2 be &ax, r1 be i, r3 be C

```
mov r2            ;  ac = i
add r1            ;  ac = &ax + i, compute effective address
put r1
ldd r1            ; get ax[i]  into ac
put r3            ; C = ax[i]
```

Using this access method, you can write a program to sum all elements in an array. Let us see one more example of storing a value to an array. How to do this   ax[i] = C

Let r2 be &ax, r1 be i, r3 be C

```
mov r2          ;  ac = i
add r1          ;  ac = &ax + i, compute effective address
put r1
mov r3          ;  get C
std r1          ;  ax[i] = C
```

Creating function

In a high level language, we can write a modular program that reuses its component. This is achieved by packaging the often use part into a "function" (or "procedure" or "method"). In order for a function to return to its caller, we need to store the "return address" in a special data structure, usually implemented as a "stack". The Z0 has an implicit stack for this purpose (with the depth of 8). This stack is not accessible to programmers. We need two more instructions to do a function call:

```
call ads        ;  next pc -> stack; jump to ads
ret             ;  stack -> pc
```

with encoding 24 call, 25 ret.

Example 4 Calling a function, passing parameters on registers.

```
double(x)
   return x + x

main()
   a = 3
   b = double(a)
```

Let r2 x, r3 b, we pass a to x via r2 and keep return value in r0

```
:double
  mov r2
  add r2           ; result in r0
  ret
:main
  mvi 3
  put r2           ; pass 3 to x
  call double
  put r3           ; store result in b
  stop
```

Exercises

1) Write a program to sum all elements in an array
2) Write a program to multiply two numbers using “repeat” addition. Suppose you want to multiply A by B. You can do that by adding A to itself B times, A + A + A ….
3) How can you “exchange” the values of two registers?
4) When you want loop for x times, you can do it two ways. One way is to count up to x. The other way is to count down from x to 0. Write some loop in both ways. Which way is shorter?

Enjoy simple machine coding!

Prabhas Chongstitvatana 18 September 2014

(Scotland for independence?)

update 25 Sept 2014