Lecture 1

Introduction to Computer Architecture

1.1 Computer and Abstraction

A *computer* is a device that performs a sequence of *instructions*. The instructions can be arithmetic and logical operations, e.g., addition, subtraction, comparison, etc. Typically, those operations are performed by the computer to solve a problem. Here are *the levels of transformation* from a problem to electrons.

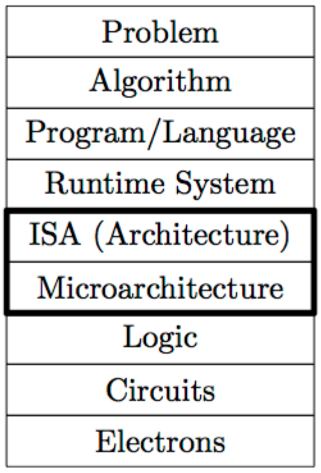

Figure 1.1: Levels of Transformation for Solving Problems by Electrons [Mut14]

Given a *problem*, we design an *algorithm* which is a precisely-defined sequence of steps to solve the problem. Then, the algorithm is implemented into a *program* using a *programming language*. The program is translated by a *compiler* to be run on a *runtime system*. Here, the runtime system can be an *operating system*, a *virtual machine*, a *memory manager* or a combination of them.

An *ISA* or *Instruction Set Architecture* provides a way for the program to interact with the computer hardware. The ISA defines the specification of the *machine language*. Different processors may implement the same ISA. A *microarchitecture* is used to refer to an implementation of the ISA. For example, "IA32" (Intel Architecture, 32-bit) is an ISA with multiple microarchitectures, e.g., 80386, 486, Pentium, etc.

An *microarchitecture* is implemented by *digital circuits* using a huge number of *logic gates*. Each type of gates is a *circuit* that controls the flow of *electrons*.

Example 1.1 Suppose we want to order a sequence of integers into the ascending order. For example,

$$12, 5, 2, 22, 7, 9, 10, 1 \quad \rightarrow \quad 1, 2, 5, 7, 9, 10, 12, 22$$

1. *Algorithm* is a sequence of steps to compare and exchange the order of a pair of numbers to make the sequence arrange in the ascending order. Figure 1.2 shows how to solve this problem by conducting *insertion sort*.

2. *Program* can be implemented based on programming languages. Here, we implement two programs in Python and C from the algorithm shown in Figure 1.3 and 1.4. Though, both programs work equivalently, they look different because of the syntax of the languages.

3. *Runtime System* provides tools and facilities to execute the programs. Here, we run the Python program under the Python interpreter working under the operating system. Here, each statement in the Python program is translated into a sequence of machine instructions before being executed.

   For the C program, we need a compiler to generate a complete executable program in the machine language. Then, we can run the program under the operation system.

4. *Instruction Set Architecture (ISA)* defines specification of the process as well as the interface to control the processor. ISA also sets the syntax of the machine language. Figure 1.5 and 1.6 show the machine and assembly instructions generated from the C program.

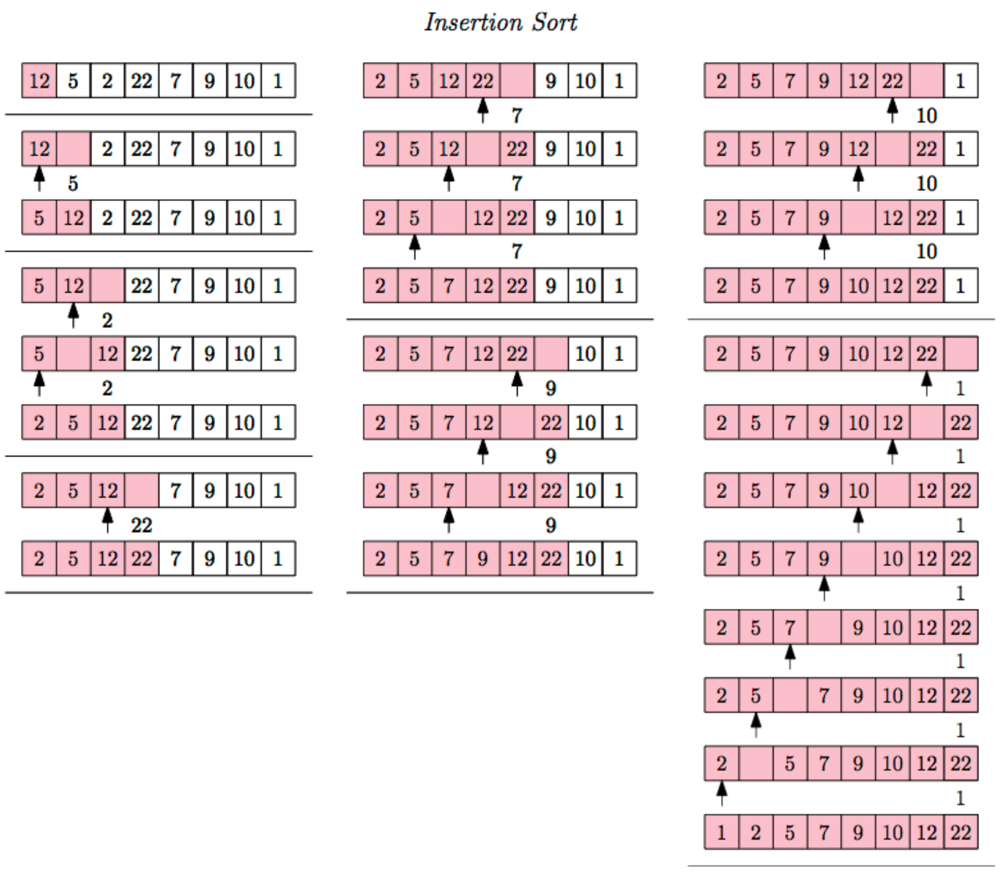

Figure 1.2: Insertion sort algorithm [Wik14]

5. *Microarchitecture* is an implementation of the ISA. For example, how to perform an addition operation (`addl %eax, %edx`). Different techniques may be used to implement the same ISA.

```
def insertion_sort(S):
    L = list(S)  # Make a copy of S
    for i in range(len(L)):
        j = i
        while j > 0 and L[j-1] > L[j]:
            tmp = L[j]
            L[j] = L[j-1]
            L[j-1] = tmp
            j -= 1
    return L

A = [12, 5, 2, 22, 7, 9, 10, 1]
print("Unsorted =", A)
print("Sorted   =", insertion_sort(A))
```

Figure 1.3: Insertion sort implemented in Python

```c
#include<stdio.h>
#include<stdlib.h>
#include<string.h>

int *insertion_sort(int *S, int n) {
    int i, j;
    int *L;

    L = (int *)malloc(sizeof(int)*n);
    memcpy(L, S, sizeof(int)*n);
    for(i=0; i<n; i++) {
        for(j=i; j>0 && L[j-1] > L[j]; j--) {
            int tmp = L[j];
            L[j] = L[j-1];
            L[j-1] = tmp;
        }
    }
    return L;
}

void print_list(int *A, int n) {
    int i;
    for(i=0; i<n-1; i++) {
        printf("%d", A[i]);
        printf(", ");
    }
    printf("%d", A[n-1]);
}

int main() {
    const int n = 8;
    int A[] = {12, 5, 2, 22, 7, 9, 10, 1};
    int *L;

    printf("Unsorted = [");
    print_list(A, n);
    printf("]\n");

    L = insertion_sort(A, n);
    printf("Sorted   = [");
    print_list(L, n);
    printf("]\n");
}
```

Figure 1.4: Insertion sort implemented in C

```
00000000 <insertion_sort>:
   0:  55                      push   %ebp
   1:  89 e5                   mov    %esp,%ebp
   3:  53                      push   %ebx
   4:  57                      push   %edi
   5:  56                      push   %esi
                                 ...
  30:  85 ff                   test   %edi,%edi
  32:  7e 37                   jle    6b <insertion_sort+0x6b>
  34:  31 c0                   xor    %eax,%eax
  36:  8d 76 00                lea    0x0(%esi),%esi
  39:  8d bc 27 00 00 00 00    lea    0x0(%edi,%eiz,1),%edi
  40:  85 c0                   test   %eax,%eax
  42:  7e 22                   jle    66 <insertion_sort+0x66>
  44:  8b 0c 86                mov    (%esi,%eax,4),%ecx
  47:  89 c2                   mov    %eax,%edx
  49:  8d b4 26 00 00 00 00    lea    0x0(%esi,%eiz,1),%esi
  50:  8b 5c 96 fc             mov    -0x4(%esi,%edx,4),%ebx
  54:  39 cb                   cmp    %ecx,%ebx
  56:  7e 0e                   jle    66 <insertion_sort+0x66>
  58:  89 1c 96                mov    %ebx,(%esi,%edx,4)
  5b:  89 4c 96 fc             mov    %ecx,-0x4(%esi,%edx,4)
  5f:  8d 52 ff                lea    -0x1(%edx),%edx
  62:  85 d2                   test   %edx,%edx
  64:  7f ea                   jg     50 <insertion_sort+0x50>
  66:  40                      inc    %eax
  67:  39 f8                   cmp    %edi,%eax
  69:  75 d5                   jne    40 <insertion_sort+0x40>
  6b:  89 f0                   mov    %esi,%eax
  6d:  83 c4 0c                add    $0xc,%esp
  70:  5e                      pop    %esi
  71:  5f                      pop    %edi
  72:  5b                      pop    %ebx
  73:  5d                      pop    %ebp
  74:  c3                      ret
```

Figure 1.5: Assembly for insertion sort for 32-bit x86 ISA

```
00000000 <insertion_sort>:
   0:  e92d41f0   push  {r4, r5, r6, r7, r8, lr}
   4:  e1a04101   lsl  r4, r1, #2
   8:  e1a05000   mov  r5, r0
   c:  e1a00004   mov  r0, r4
  10:  e1a07001   mov  r7, r1
                        ...
  28:  e3570000   cmp  r7, #0
  2c:  da00001a   ble  9c <insertion_sort+0x9c>
  30:  e3a06000   mov  r6, #0
  34:  e2866001   add  r6, r6, #1
  38:  e1560007   cmp  r6, r7
  3c:  e1a03008   mov  r3, r8
  40:  0a000015   beq  9c <insertion_sort+0x9c>
  44:  e3560000   cmp  r6, #0
  48:  da000016   ble  a8 <insertion_sort+0xa8>
  4c:  e8931004   ldm  r3, {r2, ip}
  50:  e152000c   cmp  r2, ip
  54:  c2831004   addgt  r1, r3, #4
  58:  c1a05001   movgt  r5, r1
  5c:  c2834008   addgt  r4, r3, #8
  60:  c1a0e006   movgt  lr, r6
  64:  ca000004   bgt  7c <insertion_sort+0x7c>
  68:  ea00000e   b  a8 <insertion_sort+0xa8>
  6c:  e5332004   ldr  r2, [r3, #-4]!
  70:  e593c004   ldr  ip, [r3, #4]
  74:  e152000c   cmp  r2, ip
  78:  da000003   ble  8c <insertion_sort+0x8c>
  7c:  e25ee001   subs  lr, lr, #1
  80:  e5242004   str  r2, [r4, #-4]!
  84:  e525c004   str  ip, [r5, #-4]!
  88:  1afffff7   bne  6c <insertion_sort+0x6c>
  8c:  e1a03001   mov  r3, r1
  90:  e2866001   add  r6, r6, #1
  94:  e1560007   cmp  r6, r7
  98:  1affffe9   bne  44 <insertion_sort+0x44>
  9c:  e1a00008   mov  r0, r8
  a0:  e8bd41f0   pop  {r4, r5, r6, r7, r8, lr}
  a4:  e12fff1e   bx  lr
  a8:  e2831004   add  r1, r3, #4
  ac:  e1a03001   mov  r3, r1
  b0:  eafffff6   b  90 <insertion_sort+0x90>
```

Figure 1.6: Assembly for insertion sort for ARM ISA

1.2 Von Neumann Architecture

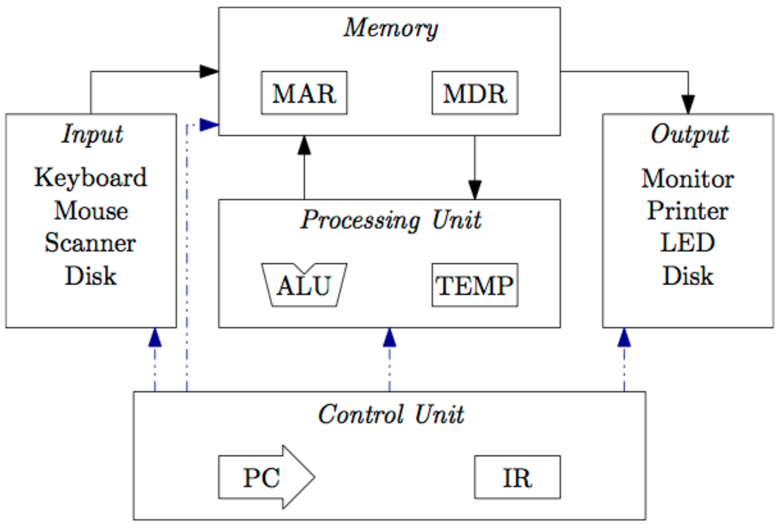

Figure 1.7: Von Neumann Architecture [Gan09]

The *Von Neumann architecture* is a model for computing devices proposed by John von Neumann in 1945. The computer composes of

1. *memory* storing both *data* and *program*,

    a) each byte of data can be referred as a *memory address*,

    b) the unit provides two basic memory operations: *load*, and *store*,

    c) *memory address register (MAR)* holds the address of data to be operated, and *memory data register (MDR)* holds the data to be loaded or stored,

    d) To load a block of data, (i) write the address to MAR, (ii) send a "read" signal, and (iii) get the data from MDR,

    e) To store a block of data, (i) write the data to MDR, (ii) write the address to MAR, and (iii) send a "write" signal,

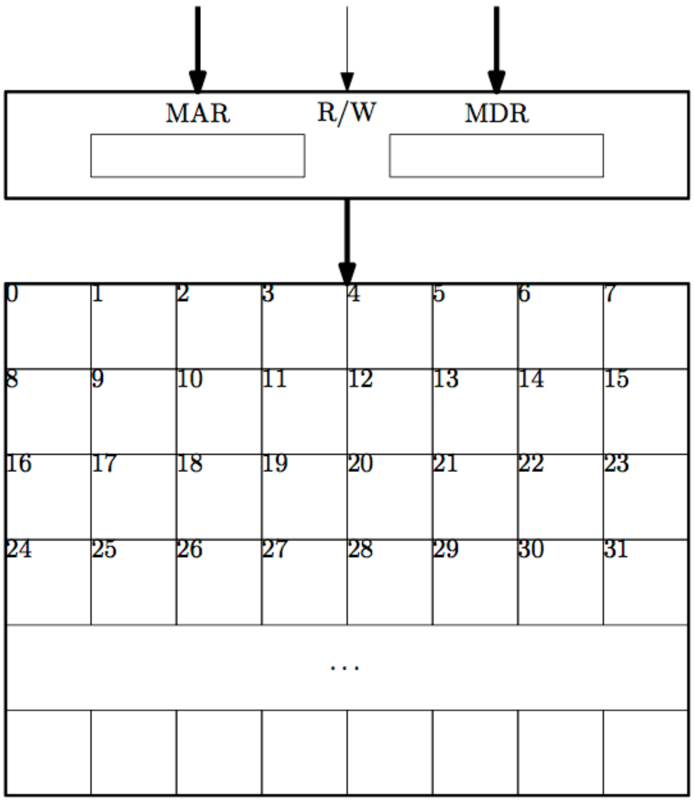

Figure 1.8: Memory controller

2. *processing unit* containing

    a) *arithmetic logic unit (ALU)* for performing all the computational operations,

    b) *temporary storage* (TEMP) for storing values,

3. *control unit* for conducting the instruction execution, containing

    a) *instruction register (IR)* for storing an instruction loaded from the memory,

    b) *program counter (PC)* that holds the *memory address* of the next instruction,

4. *input* and *output* units interacting with a user.

1.3 Instruction and Instruction Set Architecture

1.3.1 Instruction

An *instruction* is used to refer to a primitive computation operation performed by the *CPU*. For example, addition, subtraction, data copy, etc. The combination of these primitive operations can be a sophisticated program to solve problems.

Due to the complexity in the hardware design, the number of operands for each instruction is limited. At most two operands are typically allowed.

The computers can only interpret the instructions in form of *binary numbers* called *machine code*. For example, an instruction to add 1 to a storage location in ARM architecture is `1110 0010 1000 0110 0110 0000 0000 0001`. The similar instruction in x86 architecture is `0100 0000`.

Instructions can be written in form of *assembly code* to make it easier to comprehend by programmers. For example,

ARM	`1110 0010 1000 0110 0110 0000 0000 0001`	=	`add r6, r6, #1`
IA-32	`0100 0000`	=	`incl %eax`

Note that **`r6`** and **`eax`** are called 'register'. They are the storage location in *TEMP*.

1.3.2 Instruction Set Architecture (ISA)

instruction set architecture—the portion of the computer visible to the programmer or compiler writer

Hennessy and Patterson [HP07, Appendix B]

An ISA typically defines

- *instructions* to perform various computing operations,
- *temporary storages* to store values in the processor,
- *addressing modes* specifying memory locations.

However, ISA **does not** define,

- how to implement the operations,
- the computing speed of the operations,
- the power consumption of the operations.

1.3.3 Classification of ISAs

We can classify ISAs based on the structure of the temporary storage (*TEMP*) [HP07, Appendix B].

1. *Accumulator architecture* has a temporary storage storing only one value called *accumulator*. Then, all the operations are performed against the accumulator.

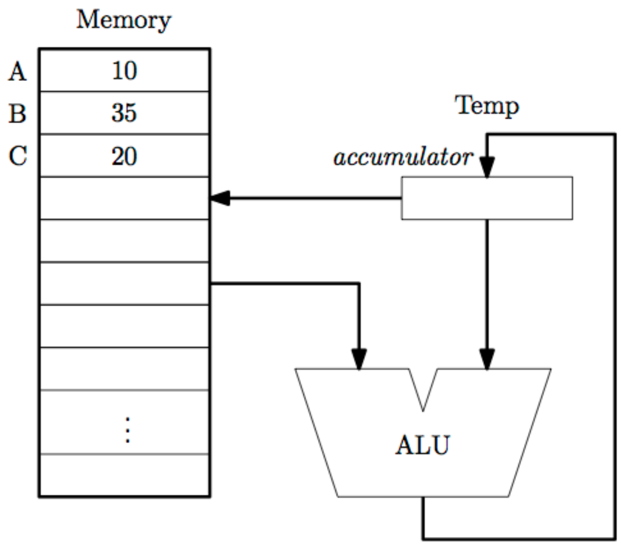

Figure 1.9: Accumulator architecture

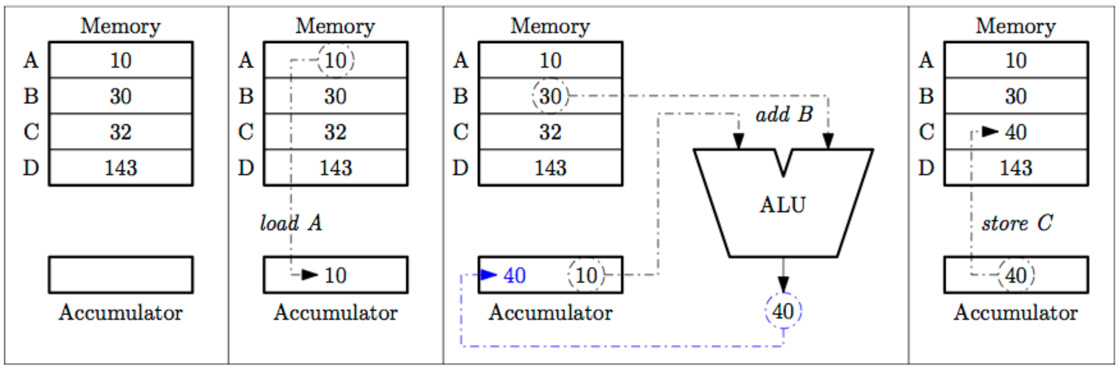

Figure 1.10: Execute $C = A + B$ in the Accumulator architecture

2. *Stack architecture* keeps all the temporary values in a stack. *ALU* uses the value(s) from the top of the stack for the computation. This architecture provides **push** and **pop** operations to access data in the memory. The **push** operation loads the data at a specified address from the memory and append to the stack at the *top of stack (TOS)*. The **pop** operation saves the data at the *top of stack* to a specified address.

   This architecture can be found in *Java Virtual Machine*.

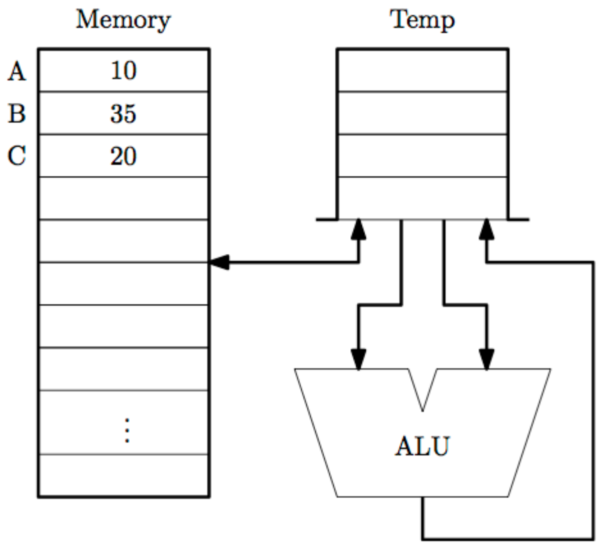

Figure 1.11: Stack architecture

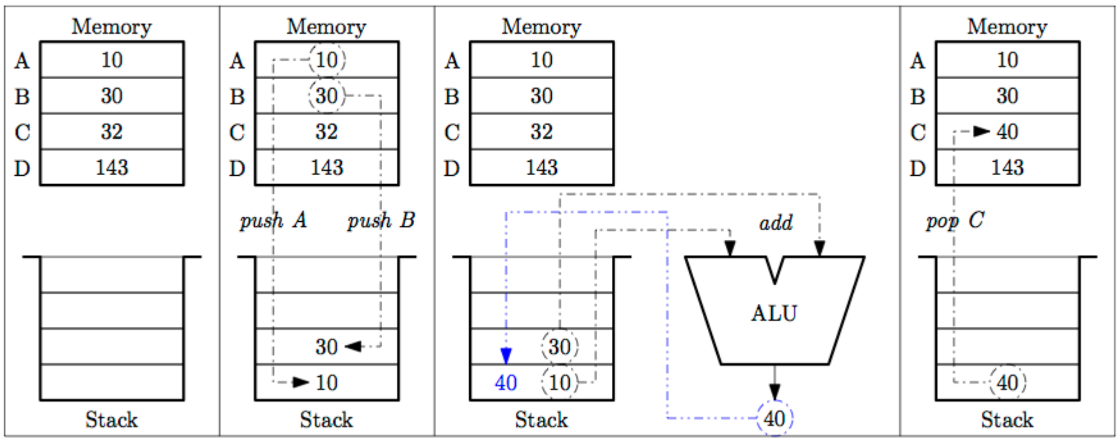

Figure 1.12: Execute $C = A + B$ in the Stack architecture

3. *General-purpose register architecture* use a temporary storage that can hold multiple values. Each value is called a 'register' which can be referred in the code by its number or name. This is the most common architecture for modern processors. This architecture can be separated into two subtypes i.e.

   a) *Register-Register architecture* allows only the registers to be the operands of the instructions. Then, it provides two special instructions (`load` and `store`) for accessing data in the memory. This architecture can be found in ARM processors.

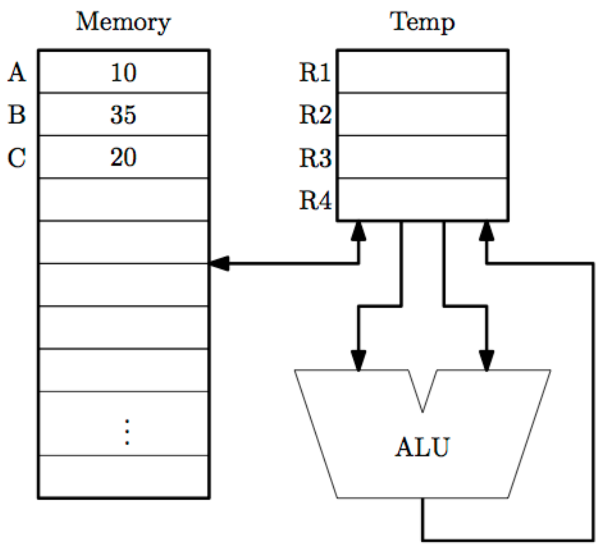

Figure 1.13: Register-Register architecture

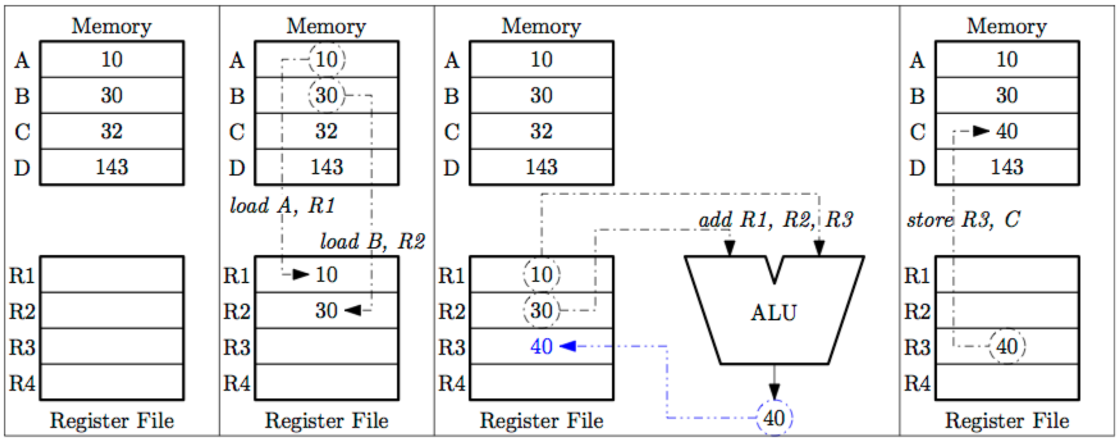

Figure 1.14: Execute $C = A + B$ in the Register-Register architecture

b) *Register-Memory architecture* allows at most one operand to be a memory location. This architecture is more flexible. This architecture can be found in Intel processors.

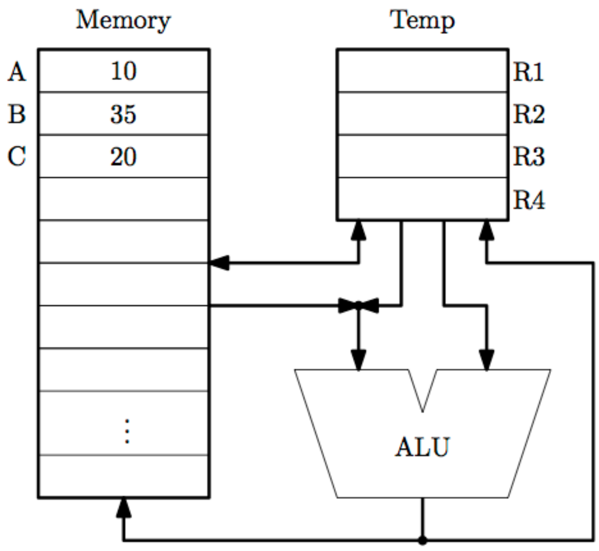

Figure 1.15: Register-Memory architecture

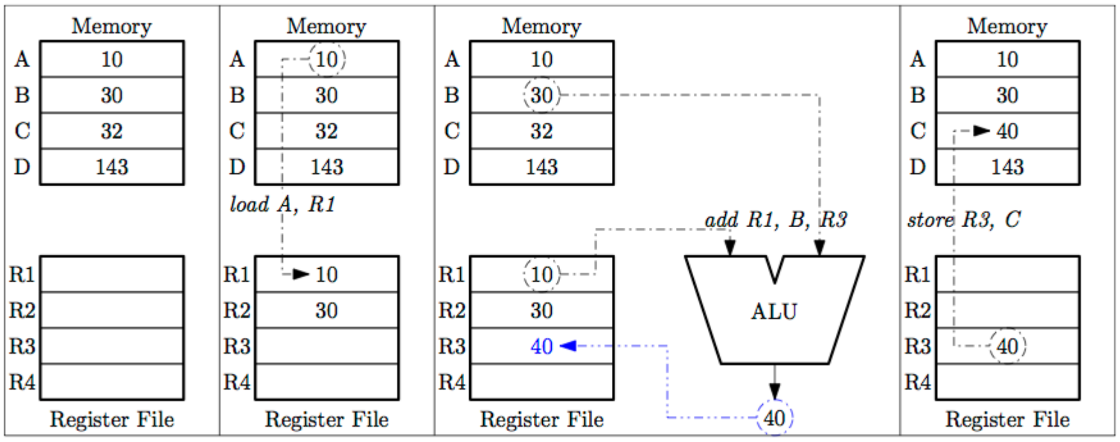

Figure 1.16: Execute $C = A + B$ in the Register-Memory architecture

c) *Memory-Memory architecture* allows multiple operands to be memory locations. It can not be found in any processor today.

Exercise 1.1 Explain how to calculate the following expression using the different architectures i.e. Stack, Accumulator, Register-Register, and Register-Memory.

```
A = (A+B)-(C+D)-E
```

Here, `A`, `B`, `C`, `D`, and `E` are memory locations.

References

[Gan09] D. Ganesan. *The Von Neumann Model.* `http://none.cs.umass.edu/~dganesan/courses/fall09/handouts/Chapter4.pdf`. [Online]. 2009.

[HP07] J. L. Hennessy and D. A. Patterson. *Computer Architecuture: A Quantitative Approach (4th edition)*. Morgan Kaufmann, 2007.

[Mut14] O. Mutlu. *18-447 Introduction to Computer Architecture – Spring 2014*. `http://www.ece.cmu.edu/~ece447/s14/doku.php?id=start`. [Online]. 2014.

[Wik14] Wikipedia Foundation Inc. *Insertion Sort.* `http://en.wikipedia.org/wiki/Insertion_sort`. [Online]. 2014.