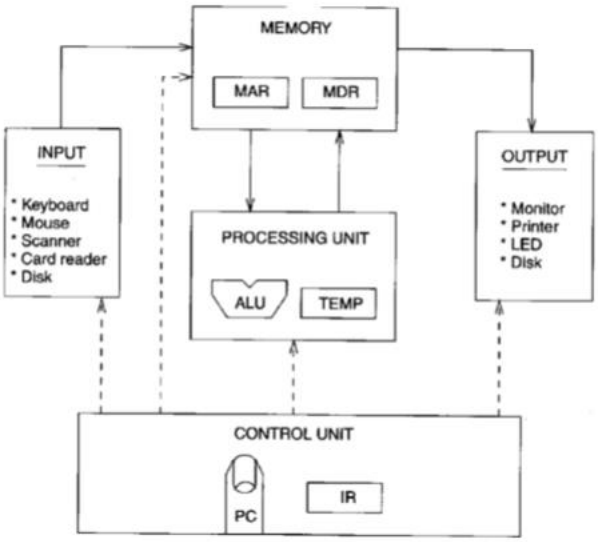
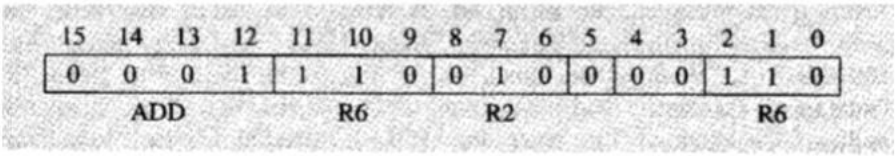
Von Neumann modellen er en abstrakt modell som forteller hvordan en prosessor fungerer i en datamaskin, modellen ser slik ut:



Modellen består av 6 komponenter:

- MAR, Memory Address Register
- MDR, Memory Data Register
- ALU, Arithmetic and Logic unit
- PC, Program Counter
- IR, Instruction Counter
- TEMP, Temporary storage

Det er disse komponentene som brukes under instruksjonssyklusen. Instruksjonssyklusen er hva prosessoren gjør når den får en oppgave, en slik oppgave kan for eksempel være å legge sammen to tall. Alt dette skjer også i form av bits (0 og 1).



Det første som skjer før selve Instruksjonssyklusen er at man gir prosessoren en instruksjon. I dette tilfelle ser vi at vi skal gjøre en «ADD» operasjon, dette leddet kalles for «Opcode», etter første leddet er resten av leddene under navnet «Operands» den forteller først hvilket tall som skal legges sammen, og til sist hvor den skal lagres. Så vi ser at instruksjonen forteller «Legge sammen R6 og R2 og lagre denne verdien i R6». Det neste som skjer nå er selve Instruksjonssyklusen.

Instruksjonssyklusen består av 6 steg:

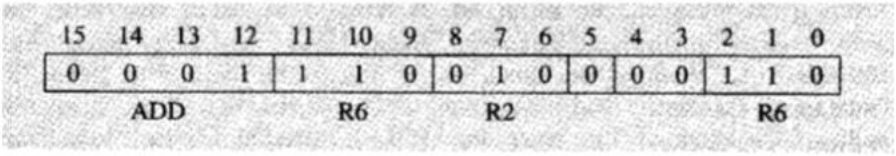
1. FETCH: hent den neste instruksjonen som skal gjøres
2. DECODE: tolk «opcode» og «operand(s)»
3. EVALUATE ADDRESS: evaluer minneadressen som er nødvendig
4. FETCH OPERANDS: hent «operands» fra minnet

5. EXECUTE: Behandlingsenheten gjør operasjonen (Behandlingsenheten består av «ALU» og «TEMP»)
6. STORE RESULT: lagre minne der det skulle lagres
   (7) start på nytt

For å fullføre disse 6 forskjellige stegene trenger vi å bruke de 6 forskjellige komponentene. Vi begynner steg 1:

- FETCH
  - «PC» fortell at vi skal last adressen for neste instruksjon i «MAR»
  - Les den nåværende instruksjonen fra «MDR» til «IR»
  - Øk «PC» til å pek på neste instruksjon

- DECODE



  - «ALU» finner ut av
    - Hva forteller «Opcode»
    - Hvilke deler er relevante i «Operands»
- EVALUATE ADDRESS
  - «ALU» skriver om adressen til «operands» slik at de blir forståelig, hvis det er nødvendig
- FETCH OPERANDS
  - «Opcode» adressen blir sendt til «MAR» og «Opcode» dataen blir sendt til «MDR»
  - Les data som ligger i «MDR» og send det til «TEMP»
  - «Operands» adressen blir sendt til «MAR» og «Operands» dataen blir sendt til «MDR»
  - Les data som ligger i «MDR» og send det til «TEMP»
- EXECUTE
  - Behandlingsenheten gjør operasjonen ved hjelp av «ALU» og «TEMP»
- STORE RESULTS
  - «Operands» lagrings-adressen blir sendt til «MAR»
  - Dataen fra operasjonen blir sendt til «MDR»

Oppgave 2

A

a. Funksjonen vil returnere: 3
b. Funksjonen vil returnere: 0

B

- Forskjellen mellom variablene «a» og «b» er at siden variabelen «a» er en pointer, består det mest sannsynligvis av et «array» med tekst, og variabelen «b» består bare av en enkel bokstav.

C

- Tegnet * betyr at disse variablene er en «pointer» som vil si at istedenfor å ha en egen verdi, holder de bare adressen til en verdi. Øverst i koden i argumentet til funksjonen ser vi «char *a», dette betyr funksjonen skal ha en «poniter» som peker til et array av tekststrenger.

  Øverst i funksjonen står det «*c = a». Dette betyr at den nye variabelen «c» skal peke på samme array som variabelen «a».

  I while-loopen står det «*c != 0», her de-refererer man «c» og denne while-loopen vil kjøre helt til den kommer til slutten av tekststreng arrayet.

  I if-statmentet står det «*c == b», her de-refererer man også «c» og denne vil være sann når c og b har samme verdi.

Oppgave 3a

```c
#include <stdio.h>


/* Oppgave 3a */
int lengste(char **A, int n)
{
    int tekststreng = 0;
    int lengste_tekststreng = 0;
    int posisjon = 0;

    //itererer gjennom hele arreyet
    for(int i = 0; i < n; i++)
    {
        //forteller størrelsen på et array
        //den funker slik at den skjekker hvor mye plass i minnet helle arrayet tar
        //så skjekker den hvor mye første element i arrayet tar i minnet
        //og deler disse to verdiene.
        //vi tar minus 1 fordi alle array ender med en '\0' og den vil vi ikke telle med
        //Dette kan man gjør siden man vet at et array bare har 1 type variabler, i vårt tilfelle er det char
        tekststreng = (sizeof(A[i])/sizeof(A[i][0])) - 1;

        //hvis det nyligste skjekket arreyet er større eller lik den forrige største
        if (tekststreng >= lengste_tekststreng)
        {
            //lagrer verdien slik at if-statementet funker igjennom hele iterasjonen
            lengste_tekststreng = tekststreng;
            //lagrer verdien til posisjon til den største arrayet
            posisjon = i;
        }
    }
    return posisjon;
}
```

```c
int main()
{
    /* Oppgave 3a */
    //vi har fra før av en dobbelpointer som inneholder et array med pointere, som peker til egne tekststrenger
    char *dobbelpointer[] = {/* pointere til tekststrenger */};
    int lengde = 10; //lengden på dobbelpointer

    int posisjon;
    posisjon = lengste(dobbelpointer, lengde);
```

Oppgave 3b

```c
/* Oppgave 3b */
int *finn(char **A, char *substreng)
{
    //det første man må gjør er å allokere minne til det nye arrayet med alle substrengene
    //den blir nå lagret på "heapen" og med å ta vare på adressen mister vi ikke veriden etter funksjonen har returnert arrayet
    //dette blir dobbelpointer siden vi vil ta vare på adressen til substrengene
    char *alleSubstrenger[] = malloc(sizeof(char));

    //lengden på A og Substreng, ville egentlig gitt den som et argument i funksjonen
    int lengdeA = 10;
    int lengdePaSubstreng = 20;

    int substrengCounter = 0;
    int lengdePaArray;

    //her starter man med å iterere gjennom pointerne inni arrayet A
    for(int i = 0; i < lengdeA; i++)
    {
        //finner størrelsen av Arrayet man skal skjekke om har en substreng
        lengdePaArray = (sizeof(A[i])/sizeof(A[i][0])) - 1;

        //itererer igjennom en tekststreng inni arrayet A
        for(int j = 0; j < lengdePaArray; j++)
        {
            //skjekker hvis en bokstav er lik en i substrengen
            if(A[i][j] == substreng[substrengCounter])
            {
                //hvis en bokstav er lik, så øker vi med 1 slik at substrengen går videre til neste bokstav
                substrengCounter++;
                //hvis en bokstav er lik, starter vi på nytt å iterere i tekststrengen inni arrayet A
                j=0;
```

```c
            //skjekker hvis en bokstav er lik en i substrengen
            if(A[i][j] == substreng[substrengCounter])
            {
                //hvis en bokstav er lik, så øker vi med 1 slik at substrengen går videre til neste bokstav
                substrengCounter++;
                //hvis en bokstav er lik, starter vi på nytt å iterere i tekststrengen inni arrayet A
                j=0;

                //skjekker hvis substrengCounter er lik lengden på substrengen
                if(substrengCounter == lengdePaSubstreng)
                {
                    //hvis substrengCounter og lengden på substrengen er like vet vi at tekststrengen inni arrayet A er en substreng
                    //vi vet tekststrengene er like fordi substrengCounter øker bare hvis de har matchende bokstaver,
                    //og den har økt helt opp til lengden av Substrengen, altså alle bokstavene matcher.
                    //vi legger den derfor inni arrayet som tar vare på alle substrengene
                    alleSubstrenger[i] = A[i];
                    //setter j som lengden av det nåværende arrayet slik at den kan gå videre til neste
                    j = lengdePaArray;
                }
            }
        }
    }
    return alleSubstrenger;
}
```

```c
    /* Oppgave 3b */
    //vi bruker samme dobbelpointer som i forrige oppgave, og antar at den er full av pointere med tekststrenger
    char substreng[] = "Dette er substrengen";

    //vi bruker denne til å ta vare på adressen til det nye arrayet med alle substrengene
    char *alleStrenger;
    alleStrenger = finn(dobbelpointer, substreng);

return 0;
}
```

Oppgave 4a og b

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

//nå går structen "stack" under navnet "stack_t"
typedef struct stack stack_t;

struct stack{
    //vi må ha en next slik at den kan peke på hva som er neste i stacken
    stack_t *next;
    //vi må ha en head slik at den første i stacken er definert som head
    stack_t *head;
    //verdien stakcen kan holde, altså en int verdi
    int verdi;
};

//vi lager en ny stack som ikke inneholder noe
stack_t *createstack(void)
{
    stack_t *NewStack = malloc(sizeof(stack_t)); //vi må allokere minne til stacken
        NewStack->head = NewStack; //dette er den første i stacken, derfor setter vi head er lik seg selv
        NewStack->next = NULL; //siden dette er den eneste i stacken må neste peke til NULL
        NewStack->verdi = 0; //den inneholder ingen verdi
}

//vi må kunne slette stacken fra minne hvis ønsket
void destroyStack(stack_t *stack)
{
    free(stack);
}
```

```c
//vi må kunne slette stacken fra minne hvis ønsket
void destroyStack(stack_t *stack)
{
    free(stack);
}

//legger til ett nytt heltatt i stacken
void PushStack(stack_t *stack, int verdi){
    stack_t *stackAdd = malloc(sizeof(stack_t)); //allokerer minne til den nye verdien
        stack->verdi = verdi; //setter verdien til stacken som man fikk som argument
        stack->next = stackAdd; //
        stack->head = stackAdd;
        stackAdd->next = NULL;
}

int main()
{
//vi lager stacken
int *stack;
stack = createstack();

//legger til verdi 5 i stacken
int leggtil = 5;
PushStack(stack, leggtil);

return 0;
}
```

Kilder

https://uit.instructure.com/courses/22505/files/folder/Forelesningsnotater/12_von_neumann?