CS101 Computer Programming and Utilization

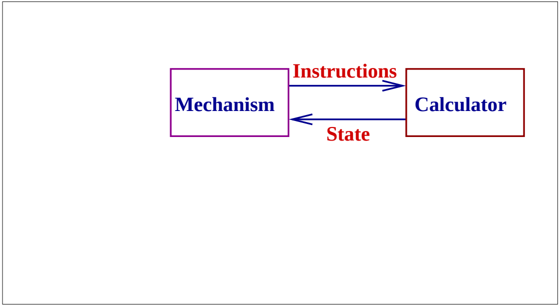
Milind Sohoni

May 10, 2006

1. The Basic Computer

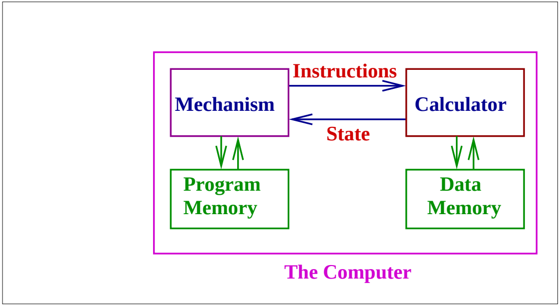
2. Programming Language
   - READIN and assignments
   - The IF-ENDIF instruction
   - The DO-WHILE instruction

In Summary



- We started off with the basic calculator and the BUM who executed our programs.
- Next we introduced more memory in the calculator so that programs became simpler.
- Finally, we replace the BUM by a cleverer mechanism:
  - who stored the program that we gave him.
  - could execute the TEST nos instruction and re-use the program code.
- Then we saw how to write some programs in for such a composite machine.

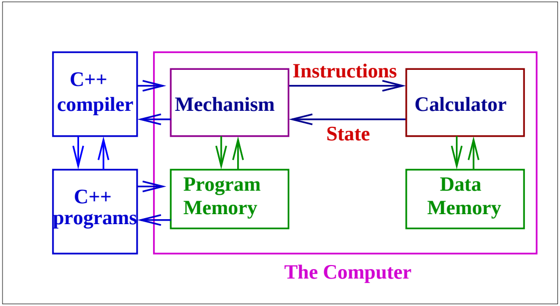
The Basic Computer



The basic computer is exactly this machine:

- It is an enhanced calculating machine with a richer instruction set for specific calculations.
- It has enhanced data memory (registers) which can stored 10^9 items.
- It has a mechanism which passes instructions to the calculator.
- It has a program memory, wherein the program to be executed is stored.

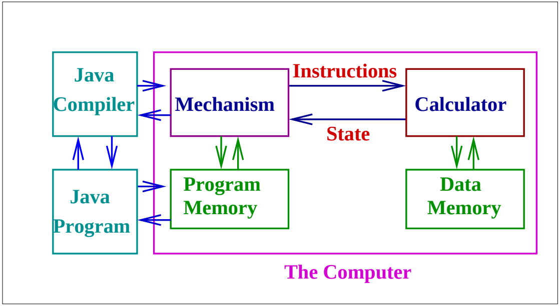
Programming Languages



Different programming languages such as C++, Java are front ends to the basic computer. These languages

- Allow the user to write programs in a more conceptual language.
- Translate this into the calculator language that we know.
- Store this translation into the progam memory.

Programming Languages



Different programming languages such as C++, Java are front ends to the basic computer. These languages

- Allow the user to write programs in a more conceptual language.
- Translate this into the calculator language that we know.
- Store this translation into the progam memory.

A Simple Programming Language

- A simple instruction:

```
M3=READIN 78
```

unfolds into

```
78  % put into display
STO 3 % put it into M1
```

- the instruction:

```
M3=READIN
```

prompts the user to input a number nos

```
STO 3 % put it into M1
```

This instructions puts user values into memory locations.

A Simple Programming Language

- A simple instruction:

  ```
  M3=READIN 78
  ```

  unfolds into

  ```
  78  % put into display
  STO 3 % put it into M1
  ```

- the instruction:

  ```
  M3=READIN
  ```

  prompts the user to input a number nos

  ```
  STO 3 % put it into M1
  ```

This instructions puts user values into memory locations.

- Another instruction: The ASSIGNMENT:

  ```
  M1= M1 + 5 * M3
  ```

  unfolds into

  ```
  RCL 1
  +
  5
  *
  RCL 3
  =
  STO 1
  ```

This instruction allows quick programming of arithmetic operations.

A Simple Programming Language

- A simple instruction:

```
M3=READIN 78
```

unfolds into

```
78  % put into display
STO 3 % put it into M1
```

- the instruction:

```
M3=READIN
```

prompts the user to input a number nos

```
STO 3 % put it into M1
```

This instructions puts user values into memory locations.

- Another instruction: The ASSIGNMENT:

```
M1= M1 + 5 * M3
```

unfolds into

```
RCL 1
+
5
*
RCL 3
=
STO 1
```

This instruction allows quick programming of arithmetic operations.

In short, the new instructions saves us from writing long programs for conceptually easy steps.

The Quadratic Equation $x^2 + 3x + 2$ Revisited

```
M1=READIN 1 % A read in M1
M2=READIN 3 % B read in M2
M3=READIN 2 % C read in M3
```

This finishes the initialization. M6 and M7 contain the constants 2 and 4.

```
M4= M2*M2-4*M1*M3 % the discriminant
M4= M4 SQRT        % completed
```

This computes the discriminant.

```
M5= M2 MINUS             % M5=-B
M5= M5+M4 DIV 2 DIV M1 % root 1
```

Finally the root. Note that READIN statements are easy but ASSIGNMENT statements need some care.

Let us analyse the first two ASSIGNMENT statements:

```
M4= M2*M2-4*M1*M3       % the discriminant
M4= M4 SQRT              % completed
```

The first statement expands to:

```
RCL 2
*
RCL 2
-
4
*
RCL 1
*
RCL 3
=
STO 4
```

Given the current values of the registers, M4 contains $B^2 - 4AC$.

Let us analyse the first two ASSIGNMENT statements:

```
M4= M2*M2-4*M1*M3        % the discriminant
M4= M4 SQRT               % completed
```

The first statement expands to:

```
RCL 2
*
RCL 2
-
4
*
RCL 1
*
RCL 3
=
STO 4
```

Given the current values of the registers, M4 contains $B^2 - 4AC$.

The next assignment statement in peculiar:

```
M4= M4 SQRT
```

This translates to:

```
RCL 4
SQRT
=
STO 4
```

The current value of M4 is used to obtain the next value of M4 which is $\sqrt{B^2 - 4AC}$.

The IF-ENDIF instructions

The IF instructions is used as follows:

```
IF M4
```

unfolds into:

```
RCL 4
TEST  nos
```

The argument nos is captured by the ENDIF instructions as follows:

```
ENDIF
```

This records the line number of the next instruction.

The IF-ENDIF instructions

The IF instructions is used as follows:

```
IF M4
```

unfolds into:

```
RCL 4
TEST  nos
```

The argument nos is captured by the ENDIF instructions as follows:

```
ENDIF
```

This records the line number of the next instruction.

```
M1=READIN 1 % A read in M1
M2=READIN 3 % B read in M2
M3=READIN 2 % C read in M3

M4= M2*M2-4*M1*M3 % the discriminar

IF M4       %M4>0 then go to nos
STOP
ENDIF       %this is nos

M4= M4 SQRT         % completed
M5= M2 MINUS                 % M5=-B
M5= M5+M4 DIV 2 DIV M1 % root 1
```

In other words:

```
CODE BLOCK 1
IF M4
CODE BLOCK 2
ENDIF
CODE BLOCK 3
```

In other words:

```
CODE BLOCK 1
IF M4
CODE BLOCK 2
ENDIF
CODE BLOCK 3
```

causes the following two possibilities:

- if M4$>0 \Rightarrow$ CodeBlock1;CodeBlock3.
- if M4$<=0 \Rightarrow$ CodeBlock1;CodeBlock2;CodeBlock3.

Warning

The ENDIF of the IF must follow the IF.

In other words:

```
CODE BLOCK 1
IF M4
CODE BLOCK 2
ENDIF
CODE BLOCK 3
```

causes the following two possibilities:

- if M4$>0 \Rightarrow$ CodeBlock1;CodeBlock3.
- if M4$<=0 \Rightarrow$ CodeBlock1;CodeBlock2;CodeBlock3.

Warning

The ENDIF of the IF must follow the IF.

Assignment

- Write PL-code for computing the other root.
- Expand the last two ASSIGNMENT statements into CAL-code.
- Modify the quadratic programming code to take care of $a \neq 0$.
- Write PL-code for computing 2^n.

The DO-WHILE instruction

Here is another useful instruction:

```
DO
```

merely records the line number of the next instruction say nos as it scans the program.
The DO instruction must be coupled with the WHILE instruction:

```
WHILE M5
```

Let M10 be an unused register, The above instruction causes the following output:

```
M10=M5;
RCL 10
TEST  nos
```

summary...

The DO records the line number of the next instruction. Thus, the presence of a WHILE causes the execution to go to nos if M5$>$ 0. Otherwise the next statement is executed.

Here is the log example again:

```
M1=READIN 178 % the value of n
M2=0  % this stores log
M3=1  % this stores 2^log
M4=M1-M3

DO  *      nos=5

M2=M2+1     % add 1
M3=M3*10     % multiply by 10
M4=M1-M3
                M10=M4
WHILE M4  *     RCL 10
                TEST nos
STOP
```

Here is the log example again:

```
M1=READIN 178 % the value of n
M2=0  % this stores log
M3=1  % this stores 2^log
M4=M1-M3

DO  *      nos=5

M2=M2+1     % add 1
M3=M3*10     % multiply by 10
M4=M1-M3
                   M10=M4
WHILE M4  *        RCL 10
                   TEST nos
STOP
```

Let us see what happens:

- The first time the DO instruction is encountered, the line number is noted of the next instruction, which is 5.
- Next:

	M1	M2	M3	M4
do 1	178	0	1	177
while 1	178	1	10	168
do 2	178	1	10	168
while 1	178	2	100	78
do 3	178	2	100	78
while 1	178	3	1000	-822
STOP				

DO-WHILE abstracted

The following code

```
CODE BLOCK 1

DO

CODE BLOCK 2

WHILE M4

CODE BLOCK 3
```

causes the following execution:

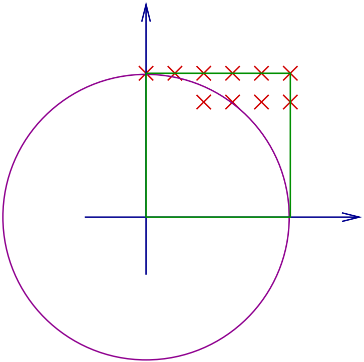
```
CB1
 CB2  first time (always)
 CB2  M4 >0
 CB2  M4 >0
 CB2   M4 non-positive
CB3
```

Caution

The WHILE must always come after the DO.

Compute $\pi/4$

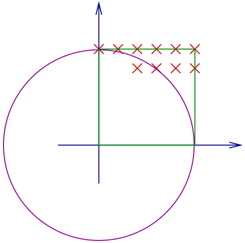
```
M1=READIN 100
M2=1 DIV M1 % the delta
M3=0;   % count
M11=1
 do
 M10=1
  do
  M4=M10*M10+M11*M11-1
    IF M4
    M3=M3+1
    ENDIF
  M10=M10-M2
  while M10
 M11=M11-M2
 while M11
M3=M3 DIV M1 DIV M1
```



Compute $\pi/4$

```
M1=READIN 100
M2=1 DIV M1 % the delta
M3=0;   % count
M11=1
 do
 M10=1
  do
  M4=M10*M10+M11*M11-1
    IF M4
    M3=M3+1
    ENDIF
  M10=M10-M2
  while M10
 M11=M11-M2
 while M11

M3=M3 DIV M1 DIV M1
```

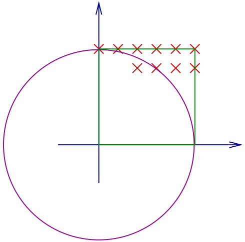


- M11 changes only in the green loop. Thus it is constant in the blue loop and the IF-ENDIF.
- For this fixed value of M11, M10 is initialized to 1. In the blue loop, this value goes from M10=1, 0.99,... upto M10=0.01. Thus the IF-ENDIF is executed exactly 100 times for each value of M11.

Compute $\pi/4$

```
M1=READIN 100
M2=1 DIV M1 % the delta
M3=0;   % count
M11=1
 do
 M10=1
  do
  M4=M10*M10+M11*M11-1
    IF M4
    M3=M3+1
    ENDIF
  M10=M10-M2
  while M10
 M11=M11-M2
 while M11

M3=M3 DIV M1 DIV M1
```



- At M10=0.0, the blue loop stops and a new value of M11 is computed.
- Thus there are 100×100 iterations of the IF-ENDIF which counts the number of points in the circle. Finally, the approximation to $\pi/4$ is computed.

Nesting

Putting one DO-WHILE inside another is called Nesting. The language is responsible for correctly identifying each WHILE with the corresponding DO. This is done in the same way as brackets are matched.

Let { stand for DO and } for WHILE. Then the following sequence:

Stands for

```
DO
DO
WHILE
WHILE
DO
WHILE
```

$\{ \{ \} \} \{ \}$

Nesting

Putting one DO-WHILE inside another is called Nesting. The language is responsible for correctly identifying each WHILE with the corresponding DO. This is done in the same way as brackets are matched.

Let { stand for DO and } for WHILE. Then the following sequence:

	Stands for
DO DO WHILE WHILE DO WHILE	{ { } } { }

Problem

- Given the following sequence of valid brackets, tell which open-brackets match with which closed bracket.

1	2	3	4	5	6	7	8	9	10	11	12
{	{	{	}	}	{	}	}	{	{	}	}

- Given a sequence a open and close brackets, how will you detect if it is a valid sequence?