

# Pseudocode Interpreter

# Overview

- Computer hardware executes programs that have been translated by a compiler or assembler to machine code.
- An interpreter is a program that *executes* a program rather than executing directly on the hardware.
- The program can be executed by:
  - interpreting the source code, time consuming; consider following the source code of a for-loop or recursion; advantage is complete information available at execution time.
  - lower level code, usually more efficient but less runtime information available.

# Overview

- Java compiler generates a byte code for execution on a hypothetical computer.
- Java Virtual Machine is a hypothetical computer implemented in software to execute Java program byte code
- JVM implements *fetch/execute* cycle in order to execute Java programs
- *fetch/execute* cycle fetches a machine instruction from memory and executes the instruction on data in memory
- We will implement a simple hypothetical computer to execute scientific applications written in machine language

# Scientific Pseudocode Design

- Pseudocode for scientific computing minimally requires the following:

<b>Functions</b>	<b>Example</b>
floating point arithmetic	$+$ , $-$ , $*$ , $/$ , $\sqrt{\quad}$
floating point relations	$=$ , $\neq$ , $<$ , $>$ , $\geq$ , $\leq$
indexing	$x[i] = y$ , $y = x[i]$
transfer of control	go to address
input-output	read $y$ , write $y$

# Hypothetical Computer Architecture

- **Word size**
  - signed 7 digits -9999999 to +9999999
- **Memory**
  - **Algorithm** 100 words – Address 00-99
  - **Data** 100 words – Address 00-99
- **CPU**
  - **Arithmetic**
  - **Floating point numbers**
  - **Array Indexing**

# Hypothetical Computer Architecture

Two separate address spaces

*Algorithm* – holds program algorithm machine code

*Data* – holds program data

Address

Algorithm

00	Add	x	y	z
01	Sub	y	y	y
:				
:				
98				
99				

Address

Data

00	x	3
01	y	4
:		
:		
98	z	5
99		

Program = Algorithm + Data

# Machine Instructions

- **Operands** - Define data or algorithm address.
- **Operations** – Define operation to perform on data.
  - $x = y + z$  has 3 operands, 2 operations.
  - One operand machine  $x = y + z$  is:
    - Load y
    - Add z
    - Store x
  - Three operand machine  $x = y + z$  is:
    - Add y, z, x

- **Format**

Operation	Operand 1	Operand 2	Destination
Add	y	z	x

# Machine Instruction Encoding

**Format**

Operation	Operand 1	Operand 2	Destination
-----------	-----------	-----------	-------------

**Operation Encoding**

<u>Address</u>	<u>Variable</u>
02	y
03	x
04	z

digit/sign	+	-
1	+	-
2	*	/
3	square	square root

**Example - Variables located at addresses (above left):**

Op	Operand 1	Operand 2	Destination	Comment
Add +1	y 02	z 04	x 03	$x = y + z$
Div -2	y 02	z 04	x 03	$x = y / z$
Square +3	y 02		x 03	$x = \text{square } y$



# Exercise 1

• **Format**

Operation	Operand 1	Operand 2	Destination
-----------	-----------	-----------	-------------

• **Operation Encoding**

<u>Address</u>	<u>Variable</u>
02	y
03	x
04	z

digit/sign	+	-
1	+	-
2	*	/
3	square	square root

Using the addresses above, give hypothetical machine instructions for:

Op	Operand 1	Operand 2	Destination	Comment
Add +1	y 02	z 04	x 03	$x = y + z$
				$z = y - x$
				$x = x * x$

# Comparisons alter execution control

- Flow of algorithm execution is normally sequential
- Flow can be altered to execute an instruction anywhere within the algorithm memory
- Conditional and unconditional branch

# Comparison Machine Instructions

digit/sign	+	-
1	+	-
2	*	/
3	square	square root
4	if = goto	if != goto
5	if >= goto	if < goto

## Example:

Operation	Operand 1	Operand 2	Destination	Comment
if < goto -5	x 03	y 02	01	if x < y goto 01

# Exercise 2

digit/sign	+	-
4	if = goto	if != goto
5	if >= goto	if < goto

Use variables at right to  
encode instructions  
below:

<u>Address</u>	<u>Variable</u>
02	y
03	x
04	z

Op	Operand 1	Operand 2	Destination	Comment
if < goto -5	y 02	z 04	08	if y < z goto 08
				if x = y goto 10
				if y >= z goto 12

# Moving

Operation	Operand 1	Operand 2	Destination	Comment
Move +0	y 02		x 03	x = y

# Indexing

<u>Addr</u>	<u>Variable</u>	
01	i	2
02	y	1
03	x	23
04		-4
05		<u>9</u>
06		-13
07	z	4

- Arrays are indexed starting at 0
- Indexing consists of two operations:
  - $z = x[i]$
  - $x[i] = z$
- For  $i=2$  the index  $x[i]$  accesses data address 05, calculated by:  
 $x \text{ address} + \text{value of } i = 03 + 2 = 05$

Operation	Operand 1	Operand 2	Destination	Comment
+6	x 03	i 01	z 07	$z = x[i]$
-6	z 07	x 03	i 01	$x[i] = z$

# Exercise 3

<u>Addr</u>	<u>Variable</u>	
01	i	2
02	y	1
03	x	23
04		-4
05		9
06		-13
07	z	4

- Give the machine code for:

Operation	Operand 1	Operand 2	Destination	Comment
+6				$x = x[ y ]$
-6				$y[ z ] = z$

# Loops

- Indexing is generally performed within sequential counting iteration (e.g. Java *for* statement)
- Requires an increment/decrement and comparison test of a variable
- Iteration continues until some terminating condition is met.
- The hypothetical machine iteration instruction does the following two step operation:
  - increment Operand 1
  - branches to Destination if Operand 1 < Operand 2

Operation	Operand 1	Operand 2	Destination	Comment
loop +7	i 01	n 06	05	i++ loop to 05 if i < n



# Exercise 4

- Iterate 5 times the instructions starting at algorithm address 10. State assumptions.
- Sum array f.

<u>Addr</u>	<u>Variable</u>	
01	i	0
02	n	5
03	f	23
04		-4
05		9
06		-13
07		4
08	Sum	0
09	x	0

Operation	Operand 1	Operand 2	Destination	Comment
loop +7	i 01	n 06	05	i++ loop to 05 if i < n
+6	x 03	i 01	z 07	z = x[i]
Add +1	y 02	z 04	x 03	x = y + z

# Solution

<u>Addr</u>	<u>Variable</u>	
01	i	0
02	n	5
03	f	23
04		-4
05		9
06		-13
07		4
08	Sum	0
09	x	0

<u>Addr</u>	<u>Instruction</u>	<u>Comment</u>
10	+6030109	x = f[i]
11	+1090808	Sum=Sum+x
12	+7010210	i++, loop to 10 if i<n

Operation	Operand 1	Operand 2	Destination	Comment
loop +7	i 01	n 06	05	i++ loop to 05 if i < n
+6	x 03	i 01	z 07	z = x[i]
Add +1	y 02	z 04	x 03	x = y + z

# Input/Output

- read x
- print x

Operation	Operand 1	Operand 2	Destination	Comment
+8	00	00	x 03	read x
-8	x 03	00	00	print x

# Stop

- To halt execution

Operation	Operand 1	Operand 2	Destination	Comment
+9	00	00	00	stop

# Exercise 4.5

1. Read in values for variable y and z.
2. Print the value of x[2].

<u>Addr</u>	<u>Variable</u>	
01	i	2
02	y	1
03	x	23
04		-4
05		9
06		-13
07	z	4

Operation	Operand 1	Operand 2	Destination	Comment
read +8	00	00	z 07	read z

# Exercise 4.5

## Solution

<u>Addr</u>	<u>Instruction</u>	<u>Comment</u>
10	+8000001	read y
11	+8000002	read z
12	+6030107	z = x[i]
13	-8070000	print z

<u>Addr</u>	<u>Variable</u>	
01	i	2
02	y	1
03	x	23
04		-4
05		9
06		-13
07	z	4

Operation	Operand 1	Operand 2	Destination	Comment
read +8	00	00	z 07	read z
+6	x 03	i 01	z 07	z = x[i]

# Complete Pseudocode Operations

digit/sign	+	-
0	move	
1	+	-
2	*	/
3	square	square root
4	if = goto	if != goto
5	if >= goto	if < goto
6	z=x[i]	x[i]=z
7	increment and test	
8	read	print
9	stop	

# Design Issues

- Orthogonality – We have 2 independent axes, m rows by n columns
- Symmetry – code inverse for inverse operation (+2 inverse of -2)
- Learning 10+1 facts (0-9 + inverse) know 10\*2 results
  - $m+n < m*n$
- Exceptions increase learning cost and reduce results of symmetry
  - $m+n+e < m*n-e$  must be true for symmetry to be advantageous.

digit/sign	+	-
0	move	
1	+	-
2	*	/
3	square	square root
4	if = goto	if != goto
5	if >= goto	if < goto
6	z=x[i]	x[i]=z
7	increment and test	
8	read	print
9	stop	



# Programs in memory

- Program = Algorithm + Data
- Stored in memory

## Algorithm

Addr	Instruction	Comment
00	+1020403	$x = y + z$
01	-2020403	$x = y / z$
02	+3020003	$x = \text{square } y$
03	-5030201	if $x < y$ goto 01
04	+6030104	$z = x[i]$
05	-6040301	$x[i] = z$

## Data

Addr	Data in memory
00	+0000000
01 i	+0000002
02 y	+0000012
03 x	+0000045
04 z	+0000007
05	+0000032
06 n	+0000075
07	+0000065
08	+0000051

# Program Structure

1. **Variables** - Data memory image
2. **+9999999** as a separator
3. **Instructions** - Algorithm memory image
4. **+9999999** as separator
5. **Program data** to be read

# Example Program

```
+0000000 // (00) Constant 0 Initial data
+0000021 // (01) x
+0000054 // (02) y
-0000019 // (03) z
+0000005 // (04) Constant 5
+9999999
+1010203 // (00) z = x + y
-8030000 // (01) Print z
+8000001 // (02) Read x
+8000002 // (03) Read y
-2010401 // (04) x = x / 5
+9000000 // (05) Stop
+9999999 // Data input
+0000012 // x by READ
-0000123 // y execution
```

# Exercise 5

Give the complete hypothetical machine program to read a positive number, compute and print its factorial based on the algorithm:

```
int i=1;
int n;
int nfact=1;
read n;
do {
    i++;
    nfact = nfact * i;
} while (i < n);
print nfact;
```

digit/sign	+	-
0	move	
1	+	-
2	*	/
3	square	square root
4	if = goto	if != goto
5	if >= goto	if < goto
6	z=x[i]	x[i]=z
7	increment and test	
8	read	print
9	stop	

# Exercise 5 Solution

```
int i=1;
int n;
int nfact=1;
read n;
do {
    i++;
    nfact = nfact * i;
} while (i < n);
print nfact;
```

## Algorithm

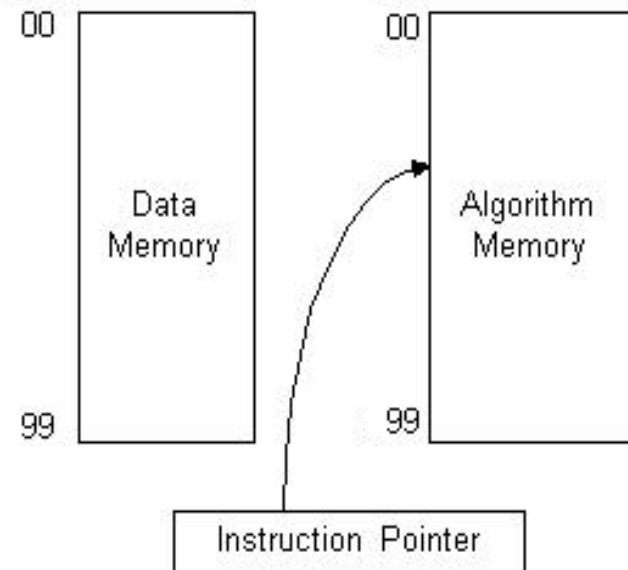
Addr	Instruction	Comment
00	+8000002	read n
01	+2010303	nfact=nfact*i
02	+7010201	while i < n
03	-8030000	print nfact
04		
05		

## Data

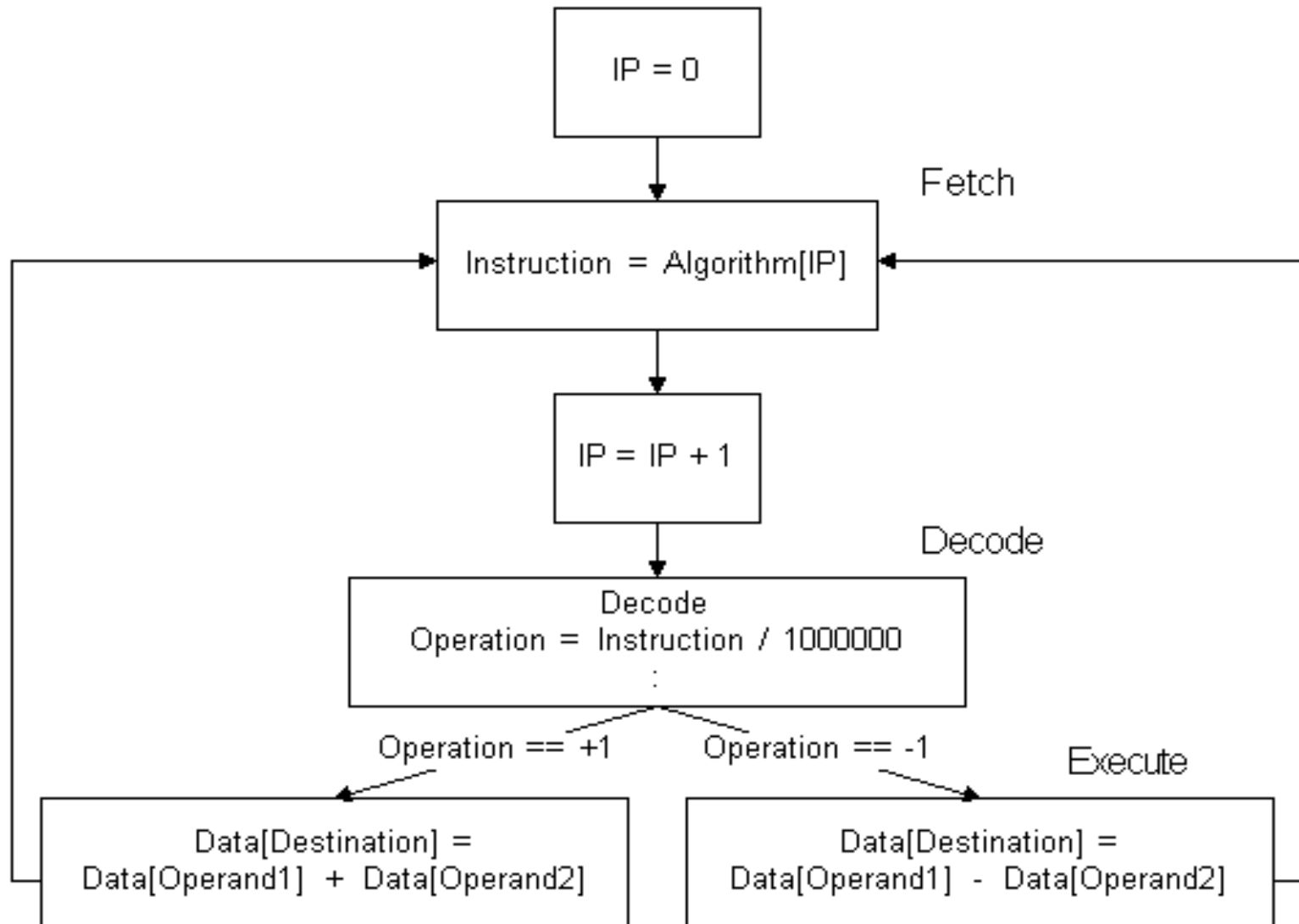
Addr	Data in memory
00	+0000000
01 i	+0000001
02 n	+0000000
03 nfact	+0000001
04 z	+0000007

# Interpreter Implementation

- Hypothetical machine hardware does not exist
- Program execution performed by software called an *interpreter*.
- Interpreter executes hypothetical machine programs by simulating hypothetical machine architecture.
- The minimal architecture requires:
  - *data* memory
  - *algorithm* memory
  - register for pointing to the address of the *next* algorithm instruction to fetch from algorithm memory.



# Fetch/Execute Cycle as Finite State Machine



# Interpreter Data Models Hardware

1. **int IP** - Instruction Pointer
2. **int Algorithm[100]** - Algorithm memory
3. **float Data[100]** - Data memory
4. **int Instruction** - Instruction fetched
5. **int Operation** - Operation field of instruction
6. **int Operand1** - Operand 1 field of instruction
7. **int Operand2** - Operand 2 field of instruction
8. **int Destination** - Destination field of instruction



1.IP = 0

2.Fetch

Instruction = Algorithm[IP]

IP = IP + 1

3.Decode

Operation = Instruction / 1000000

Decode Operand1, Operand2, and Destination addresses.

4.Execute each instruction case

if (Operation == +1)

Data[Destination] = Data[Operand1] + Data[Operand2]

if (Operation == +2)

Data[Destination] = Data[Operand1] \* Data[Operand2]

5.Go to 2.

<b>Execute:</b>	+1010203
	+2010303

### Algorithm

Addr	Instruction	Comment
00	+1010203	x=y+z
01	+2010303	x=y*x
02		
03		
04		

### Data

Addr	Data in memory
00	+0000000
01 y	+0000006
02 z	+0000005
03 x	+0000000
04	

**Execute:** +1010203

1. **IP = 0** - Initializes instruction pointer to next instruction in *algorithm memory*.
2. **Fetch** - Reads *next* instruction to execute from algorithm memory and increments instruction pointer.

**Instruction = Algorithm[0]** (*Instruction = +1010203*)

**IP = IP + 1** (*IP = 1*)

3. **Decode** - Decode instruction fields into Operation code, Operand1, Operand2, and Destination addresses.

**Operation = Instruction / 1000000** (*Operation = +1*)

Decode Operand1, Operand2, and Destination similarly.  
(*Operand1 = 01, Operand2 = 02,  
Destination = 03*)

4. **Execute** - Execute hypothetical machine instruction by determining *operation* to be performed then performing operation on *data* addresses.

**if (Operation == +1)**

**Data[Destination] = Data[Operand1] + Data[Operand2]**

(*Data[03] = Data[01]+Data[02] = 13+54 = 67*)

5. **Go to 2** - Continue the fetch and execute cycle.

# Exercise 6

1. Give the interpreter statement to decode Operand1.

```
Operation = Instruction / 1000000
```

```
Instruction = +1010203
```

2. Give the interpreter statement to execute the -5 operation.

```
if (Operation == +1)
```

```
    Data[Destination] =
```

```
        Data[Operand1] + Data[Operand2]
```

3. Give the interpreter statement to execute the +6 operation
4. Give the interpreter statements to turn tracing ON/OFF and display IP, Operation, Operand1, Operand2, and Destination prior to the instruction execution. Use operation -9 to toggle tracing ON/OFF.

# Loader

- Program = Algorithm + Data
- Program must be loaded from input into memory for execution
- Given the program structure, steps are:
  1. Load Data until  
+99999999
  2. Load Algorithm until  
+99999999

```
+0000000 // Data
+0000021
+0000054
-0000019
+0000005
+9999999 // Algorithm
+1010203
-8030000
+8000001
+8000002
-2010401
+9000000
+9999999 // Read
+0000012
-0000123
```

# Loader

- Load data into memory
  1. address = 0
  2. Read data value
  3. while (data value != +9999999.0)
  4.     Data[address] = data value
  5.     address++
  6.     Read data value
- Load algorithm into memory
  1. address = 0
  2. Read instruction
  3. while (instruction != +9999999)
  4.     Algorithm[address] = instruction
  5.     address++
  6.     Read instruction

```
+0000000 //Data
+0000021
+0000054
-0000019
+0000005
+9999999 //Algorithm
+1010203
-8030000
+8000001
+8000002
-2010401
+9000000
+9999999 //Read
+0000012
-0000123
```

# Interpreter Conclusion

- Advantages:
  - Hardware portability
  - Software development before hardware
  - Potential of extensive program runtime checking
  - Others?
- Disadvantages
  - Slower execution compared to native code
  - Others?