Pseudo-Code
Part 2

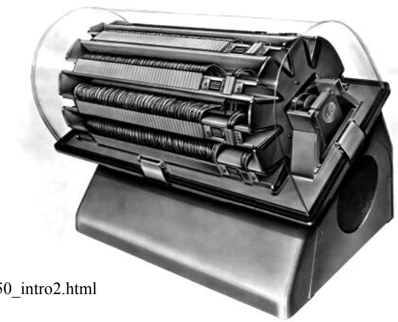
CS4100

February 2, 2011

Based on slides by Istvan Jonyer

Hardware Assumptions

- The IBM 650 will serve as the hardware
  - 1 word: 10 decimal digits + 1 sign
  - 2000 byte memory
    - 1000 for data
    - 1000 for program



http://www-03.ibm.com/ibm/history/exhibits/650/650_intro2.html

Principles of Programming

- Impossible error principle
  - Making errors impossible to commit is preferable to detecting them after their commission.
  - E.g.: Cannot modify the program accidentally, since memory modifying operations are for “data memory” only

Language Design

- 1 word can be enough to specify a 3-operand instruction
  - Operation: sign + 1 digit
    - Supports 20 operations
  - 3 3-digit operands
    - Each accessing memory locations in data area
  - Orthogonal design:
    - Operations should be more intuitive than machine code
    - Use the *sign* to get more orthogonality

Principles of Programming

- Orthogonality principle
  - Independent functions should be controlled by independent mechanisms.

Specifics

- Instruction format:
  - op src1 src2 dst
  - E.g.: x+y→z : +1 010 150 200
    - "Add values at location 010 and 150, and save it to location 200"
  - Orthogonal design: subtract should be '-1'

Arithmetic Operations

	+	-
1	+	-
2	*	/
3	x^2	square root

Comparisons

- Comparisons alter control flow
  - `if` *x* `<` *y* `then go to` *z*
  - First 2 operands are data locations, *dst* is address of next instruction

Extended Instruction Table

	+	-
1	+	-
2	*	/
3	x^2	square root
4	=	≠
5	≥	<

What else do we need?

- Moving
  - Could do “add 0” to an address, but that could be inefficient
  - Dedicate an operation to moving
  - Second operand is not used
  - “+0 src 000 dst”

Indexing

- Need
  - Base address
  - Index
- Base and index take up 2 operands; what can we do with 3rd?
  - Save value of indexed element for other operations
- Index operations:
  - Get: $x_i \rightarrow z$: `+6 xxx iii zzz`
  - Put: $x \rightarrow y_i$: `-6 xxx yyy iii`

Looping

- Looping through the elements of an array is frequently used
- What’s needed?
  - Iterator variable (array index *i*)
  - Upper bound (*n*)
  - Address of beginning of loop (*d*)
  - “`+7 iii nnn ddd`”

Principles of Programming

- The abstraction principle
  - Avoid requiring something to be stated more than once; factor out the recurring pattern.

Input/Output

- Program needs to read data from input and write data to output
  - Needs only a memory location to read from or write to
  - Read: "`+8 000 000 dst`"
  - Print: "`-8 000 000 src`"

Complete Instruction Set

	+	-
0	Move	
1	+	-
2	*	/
3	x^2	square root
4	=	≠
5	≥	<
6	GetArray	PutArray
7	Incr. & test	
8	Read	Print
9	Stop	

Program Structure

Initial data values

+9999999999

Program instructions

+9999999999

Input data

Implementing the Interpreter

- How to implement the interpreter for our pseudo-coded program?
  - Model interpreter behavior after manual execution
  - Cheat: Implement using a high-level language ☺
  - We have to simulate the hardware in software

Data Structures

- What data structures are needed to simulate the IBM 650?
  - Data memory
  - Program memory
  - Instruction pointer

Structure of the Interpreter

1. Read the next instruction
2. Decode the instruction
3. Execute the operation
4. Continue from step 1

- Where do we update the instruction pointer (IP)?
  - Step 4? No: we may need to jump, which would be overwritten
  - Increment in step 1; overwrite if needed

Decoding Instructions

- Extract part of instruction
  - `dst = ` *`instruction`* ` mod 1000`
- Select operation
  - Big switch-statement (case-statement)
- Arithmetic operations
  - Straight-forward
- Control-flow
  - IP may also need to be altered

Labeling

- What if we need to insert an instruction?
  - All addresses would have to be shifted, and the code updated
- Solution:
  - Use labels for loops, instead of absolute memory addresses
  - Define label:
    - `-7 0LL 000 000`
    - Only 100 numeric labels are possible (00-99)
  - Modify control flow instructions to jump to labels

Interpreting Labels

- How do we handle labels in the interpreter?
  - Look through all instructions from beginning of program?
    - Yes, but that is slow. This is how some interpreters work. (BASIC, for instance)
  - Create label table with absolute addresses for labels and bind addresses
    - Much faster. Compilers do it this way.

Principles of Programming

- Labeling principle
  - Do not require users to know absolute numbers or addresses. Instead associate labels with number or addresses.

Data Labels?

- If we can jump to a label, we could use labels for variables as well
- Construct symbol table
- This idea is easily extended to instructions as well to form a symbolic pseudo-code

Data Declaration

- We could extend the language to include symbols not only for program instructions but for data declarations as well
- In initial data values:

```
+0 sss nnn 000
±dddddddddd
```

  – Declare *n* values of *d* referenced by symbol *s*
  – Symbolic notation:

```
VAR sss nnn
±dddddddddd
```

  – n=1 : simple variable
  – n>1 : array

Debugging?

- Debugging always has to be done…
- Can facilitate debugging by printing instructions executed in order
- Interpreter can include *trace* flag

```
if trace is enabled
  print IP, instruction
```


Complete Symbolic Language

	+	-
0	move MOVE	
1	+ ADD	- SUB
2	* MULT	/ DIV
3	X^2 SQR	square root SQRT
4	= EQ	≠ NE
5	≥ GE	< LT
6	GetArray GETA	PutArray PUTA
7	Incr. & test LOOP	Label LABL
8	input READ	output PRNT
9	end STOP	Trace TRAC

Complete Symbolic Language

- Additional symbols
  – LABL nn
    - Declare label *n*
  – VAR sss nnn
    - Declare variable *s*[*n*]
  – END
    - Delimiter between variables, program and input
    - Defined as -9999999999
  – TRAC
    - Enable/disable tracing
    - Tracing is turned off by default. Encountering this operation toggles tracing.

Sample Program

```
VAR ZRO 1
+0000000000
VAR I 1
+0000000000
VAR SUM 1
+0000000000
...
END
READ N
LABL 20
READ TMP
GE   TMP ZRO  40
SUB  ZRO TMP TMP
LABL  40
PUTA TMP DTA   I
LOOP   I   N  20
...
STOP
END
+0000000005
+0000000020
...
```


Principles of Programming

- Security principle
  - No program that violates the definition of the language, or its own intended structure, should escape detection.