Lab 20: HACK Assembler: Symbol and Instruction Lookup Table

1. Objective

In this lab, we will discuss the HACK assembler and design the lookup table for symbols and instructions. The implementation will be done using pseudo code.

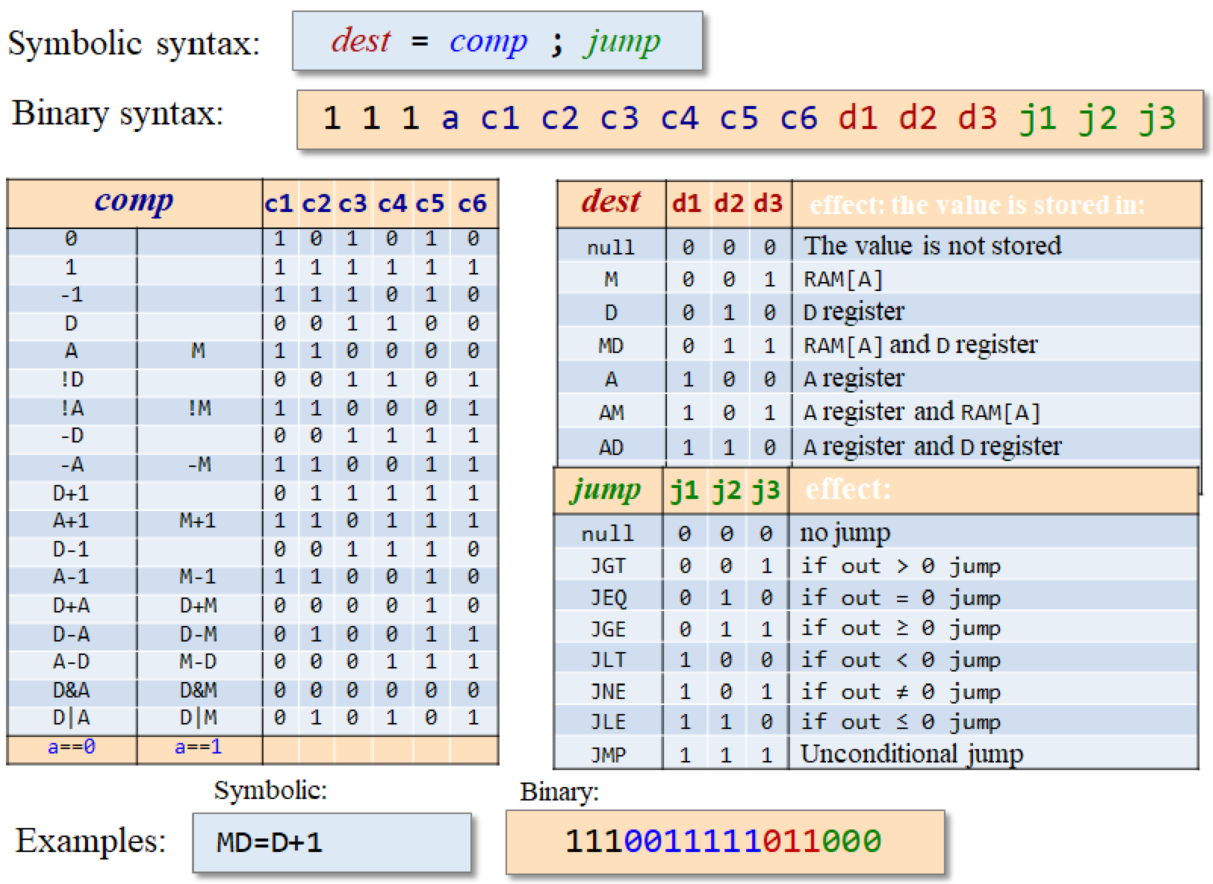
1.1 Assembler Overview

An assembler is a program that will take assembly code and translate it into the corresponding machine instructions (binary). The task for this lab and the next will be to design an assembler for HACK.

1.2 Look up tables

Look up tables are like dictionaries. You can give a dictionary a word and it will give you the definition of the word. In our case, we might ask it for a symbol and it would give us the address referred to by that symbol. Or we might look up a C instruction to get its equivalent binary.

2. Instructions Recap



Symbolic syntax: *dest* = *comp* ; *jump*

Binary syntax: 1 1 1 a c1 c2 c3 c4 c5 c6 d1 d2 d3 j1 j2 j3

comp		c1	c2	c3	c4	c5	c6
0		1	0	1	0	1	0
1		1	1	1	1	1	1
-1		1	1	1	0	1	0
D		0	0	1	1	0	0
A	M	1	1	0	0	0	0
!D		0	0	1	1	0	1
!A	!M	1	1	0	0	0	1
-D		0	0	1	1	1	1
-A	-M	1	1	0	0	1	1
D+1		0	1	1	1	1	1
A+1	M+1	1	1	0	1	1	1
D-1		0	0	1	1	1	0
A-1	M-1	1	1	0	0	1	0
D+A	D+M	0	0	0	0	1	0
D-A	D-M	0	1	0	0	1	1
A-D	M-D	0	0	0	1	1	1
D&A	D&M	0	0	0	0	0	0
D|A	D|M	0	1	0	1	0	1
a==0	a==1						

dest	d1	d2	d3	effect: the value is stored in:
null	0	0	0	The value is not stored
M	0	0	1	RAM[A]
D	0	1	0	D register
MD	0	1	1	RAM[A] and D register
A	1	0	0	A register
AM	1	0	1	A register and RAM[A]
AD	1	1	0	A register and D register

jump	j1	j2	j3	effect:
null	0	0	0	no jump
JGT	0	0	1	if out > 0 jump
JEQ	0	1	0	if out = 0 jump
JGE	0	1	1	if out ≥ 0 jump
JLT	1	0	0	if out < 0 jump
JNE	1	0	1	if out ≠ 0 jump
JLE	1	1	0	if out ≤ 0 jump
JMP	1	1	1	Unconditional jump

Examples: Symbolic: MD=D+1 Binary: 1110011111011000

2.1 A Instructions

In HACK assembly, an A instruction starts with an "at" symbol:
`@value`

2.2 C Instructions

C instructions are in the format:

`dest = comp ; jump`

`dest` and **`; jump`** are optional, but not both. One of those has to be present.

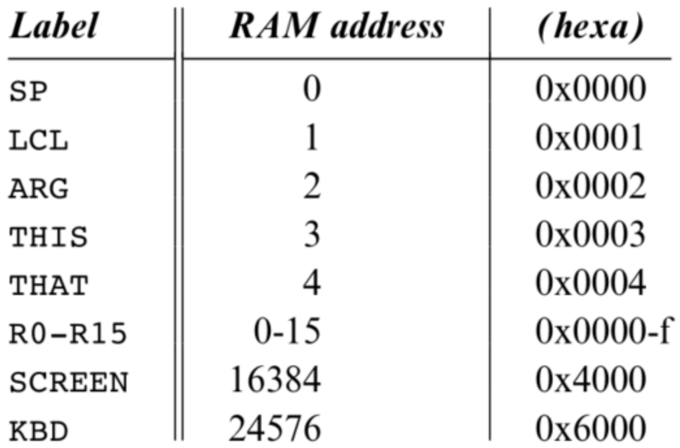
3. Symbols

Symbols are words that refer to some memory location. They are an alias for ROM addresses or data memory addresses.

Essentially, any time an A instruction is used which looks like `@wordAndNotANumber` , that should get replaced with its equivalent address, whether in the ROM or the data memory.

There are some predefined symbols in the HACK ISA. Aside from that, symbols come from labels and variables.

3.1 Predefined Symbols



Label	*RAM address*	*(hexa)*
SP	0	0x0000
LCL	1	0x0001
ARG	2	0x0002
THIS	3	0x0003
THAT	4	0x0004
R0-R15	0-15	0x0000-f
SCREEN	16384	0x4000
KBD	24576	0x6000

3.2 Labels

Labels are references to ROM addresses. They are used for control flow and branching.

Example:

```
1 (loop)
2    // do some things
3    @loop
4    0;JMP
```

Line 1 is a label. Labels are in the format **`(label)`**. They are enclosed in parentheses. Any time a label is referred to by an A instruction, such as in line 3, it is referring to the instruction directly after the label. In line 3, the symbol refers to the instructions at line 2.

3.3 Variables

Variables are another way that symbols are introduced into the program.

Example:

```
@sum
M = 0
@13
D = A
@sum
M = D
```

In the example above, sum is a variable (a reference to a memory location) which stores a value. The example initializes the value of sum to 0 and then changes it to 13. The computer will not know what “sum” is. Rather, it needs a memory location to look at. Variables are assigned starting at address 16 so if sum is the first variable, then it will map to memory address 16.

4. Symbol Look Up Table

A hash table or hash map will be used as the data structure for a look up table. Hash tables map a key to a value. In our case, it should map a string to an integer because the symbols are strings and the addresses are integers.

A hash table can be created called `symbolTable`.

```
symbolTable = HashTable(string, integer)
```

4.1 Steps to creating the symbol table

Step 1. Populate the symbol table with predefined symbols.

Step 2. Populate symbol table with labels

Step 3. Populate symbol table with variables

The table can be populated with predefined symbols because we already know what they should translate to.

The reason that labels are handled before variables is because both labels and variables are referenced by `@symbol`. The only difference is that a label is also mentioned somewhere else in the form `(symbol).` There is no requirement that a label be defined before it is referenced, so the only way to differentiate between labels and variables is to first scan the entire file for tokens in the format **`(label)`** . Any A instructions that use a symbol but do not have a corresponding label can be considered a variables.

4.2 Add Predefined Symbols to Symbol Table

```
     symbolTable = HashTable(string, integer)

     symbolTable[“SP”] = 0
     symbolTable[“LCL”] = 1
     symbolTable[“ARG”] = 2
     symbolTable[“THIS”] = 3
     symbolTable[“THAT”] = 4

     for r in (0, 1, 2, ... , 15)
          symbolTable[“R” + string(r)] = r

     symbolTable[“SCREEN”] = 16384
     symbolTable[“KBD”] = 24576
```

4.2 Add Labels to Symbol Table

```
file = openFile(assemblyFile)
pc = 0

for line in file:
     if ( emptyLineOrComment(line) ):
          continue  // go to next line
     clean = removeCommentsAndWhitespace(line)

     if ( isLabel(clean) ):
          label = removeParenthesis(clean)

          if ( NOT symbolTable.contains(label) ):
               symbolTable[label] = pc

     if ( isInstruction(clean) ):
          pc = pc + 1
```

```
function isLabel(line):
     return true if line is a sequence of alphabet characters
     surrounded by parentheses. Return false otherwise.

function isInstruction(line):
     return true if the line is a valid A or C instruction
     format. Return false otherwise.
```

4.3 Add Variables to Symbol Table

```
file = openFile(assemblyFile)
nextAddress = 16    // variable assigned starting at addr 16

for line in file:
     clean = removeCommentsAndWhitespace(line)

     if ( isValidAInstruction(clean) ):
          AInstructionVal = clean.Strip("@")

          // the value is not a number and not in the symbol
          // table already. (this means it is not a label)
          if ( isNotNumber(AInstructionVal)
               AND NOT symbolTable.contains(AInstructionVal) ):

               symbolTable[AInstructionVal] = nextAddress
               nextAddress = nextAddress + 1
```

4.5 Using the Symbol Table

Now that the symbol table includes predefined symbols, labels, and variables, the assembler is able to translate any symbol.

For example, the instruction `@R5` can be translated like so:

```
     prefix = "0"
```

```
    addr = symbolTable[ instruction.Strip(“@”) ]
    bin = prefix + to15BitBinary(addr) // instruction in binary
```

5. Instruction Lookup Table

A Table has been constructed to look up symbols. Now, an table can also be constructed to look up C instructions, more specifically the COMP portion of a C instruction. The same can be done for DEST and JUMP to make things easier. Note -- the way it is constructed here, the compTable also contains the ‘a’ bit of a C instruction.

```
    compTable = HashTable(string, integer)
    destTable = HashTable(string, integer)
    jumpTable = HashTable(string, integer)

    compTable["0"] = "0101010";
    compTable["1"] = "0111111";
    compTable["-1"] = "0111010";
    ... // many entries omitted because it gets long
    compTable["A+1"] = "0110111";
    compTable["D-1"] = "0001110";
    compTable["A-1"] = "0110010";
    compTable["D+A"] = "0000010";
    ... // many entries omitted because it gets long
    compTable["D&M"] = "1000000";
    compTable["D|M"] = "1010101";

    destTable[“M”] = “001”
    destTable[“D] = “010”
    destTable[“MD”] = “011”
    ... // finish off destination

    jumpTable[“JGT”] = “001”
    jumpTable[“JEQ”] = “010”
    jumpTable[“JGE”] = “011”
    ... // finish off jump
```

5.1 Using the Instruction Lookup Tables

The binary of a C instruction can quickly be constructed now.

```
tokens = deconstructCInstruction(instruction)
prefix = "111"
dest  = "000"
jump = "000"
comp = compTable[ tokens.comp ]

if (tokens.dest != NULL):
     dest = destTable[ tokens.dest ]

if (token.jump != NULL):
     jump = jumpTable[ tokens.jump ]

binary = prefix + comp + dest + jump
```

Imagine that tokens is a class which stores each separate part of the C instruction.