Computer Organization

Symbolic HACK Instructions

HACK Assembly

- We have a base line set of instructions in the form of our A and C instructions, but these fall short in regards to human readability and convenience

- However, we can add abstraction to our instructions by utilizing the concept of symbolic notation
  - This can be acheived by adding additional functionality to our assembler

Assemblers

- **Assemblers** are programs that convert assembly instructions to binary machine code
- At minimum, assemblers should have the capability to translate our basic instructions to proper machine code
  - We can add levels of abstraction to increase readability and development

Symbolic Programming

- Symbolic programming is the idea of utilizing human-readable symbols/words in conjunction with a language’s syntax

- For the purpose of the HACK assembly language, we will be utilizing two forms of symbols
  - **variables** and **labels**

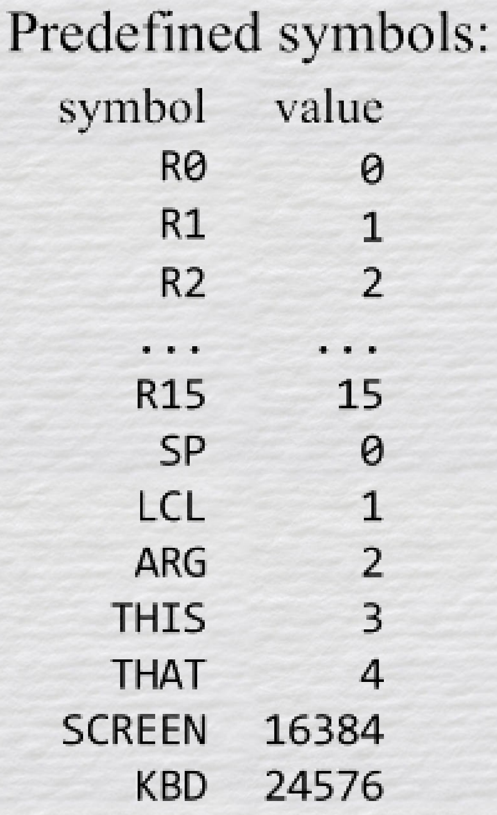
Variables

- Variables are used a means to create short-hand methods to access data stored in memory
    - The idea is that the assembler assigns a variable to a specific register in memory and uses that location when referencing the variable throughout the program

- For HACK, we have two forms of variables
    - **Predefined variables** & **User-Defined Variables**

Predefined Variables

- The HACK assembler allocates several locations in memory to predefined variables for ease-of-use during development
- Several of these will be explained in later chapters

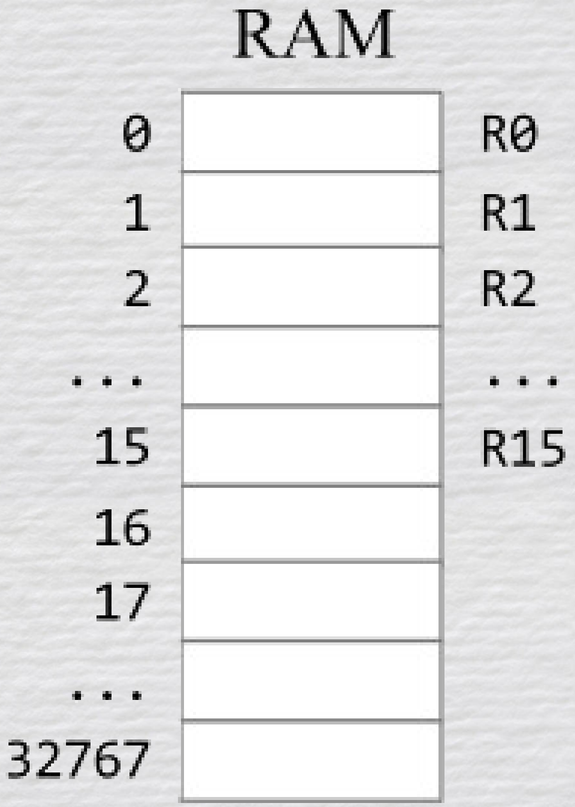


Predefined symbols:

symbol	value
R0	0
R1	1
R2	2
...	...
R15	15
SP	0
LCL	1
ARG	2
THIS	3
THAT	4
SCREEN	16384
KBD	24576

Predefined Variables - Example

```
// Sets R2 to R0 + R1 + 17
// D = R0
@R0
D=M
// D = D + R1
@R1
D=D+M
// D = D + 17
@17
D=D+A
// R2 = D
@R2
M=D
```



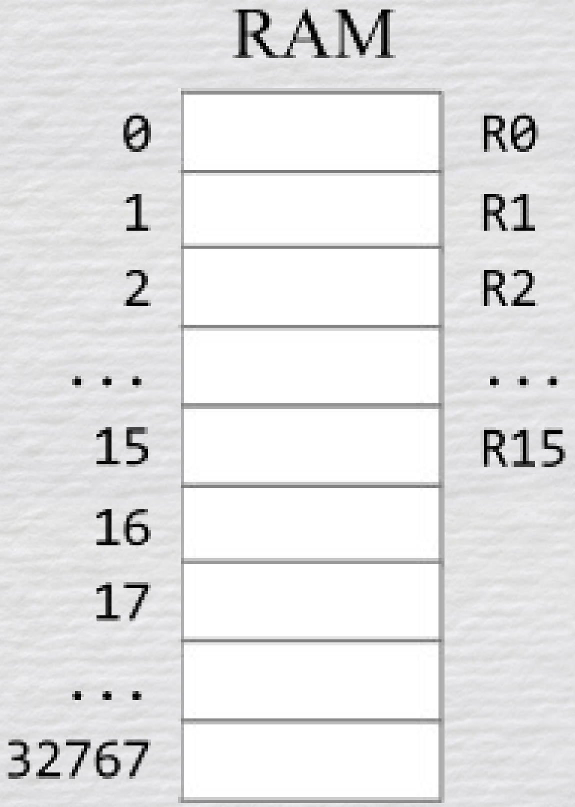
User-Defined Variables

- User-Defined Variables are what allow us a great deal of flexibility when developing low-level code

- Instead of directly accessing memory via numerical addresses, we instead create variables and the assembler will allocate said variables to their own memory address

User-Defined Variables – cont.

- User-Defined Variables are allocated to memory beginning at **RAM[16]**
  - The reason being that the first 16 addresses (RAM[0] through RAM[15]) are designated to Predefined Variables



User-Defined Variables – Syntax

<u>Syntax:</u>

```
@const
```

where *const* is a constant

```
@sym
```

where *sym* is a symbol bound to a constant

<u>Example:</u>

```
@19    // A ← 19
```

```
@x
```

For example, if x is bound to 21, this instruction will set A to 21

User-Defined Variables – Examples

Typical instructions:

@*constant*	A ← *constant*
@*symbol*	A ← the constant which is bound to *symbol*

```
D=0
M=1
D=-1
M=0
...
```

```
D=M
A=M
M=D
D=A
...
```

```
D=D+A
D=A+1
D=D+M
M=M-1
...
```

```
// sum = 0
?
```

```
// x = 512
?
```

```
// n = n - 1
?
```

```
// sum = sum + x
?
```

User-Defined Variables – Answer

Typical instructions:

@*constant*	A ← *constant*
@*symbol*	A ← the constant which is bound to *symbol*

```
D=0
M=1
D=-1
M=0
...
```

```
D=M
A=M
M=D
D=A
...
```

```
D=D+A
D=A+1
D=D+M
M=M-1
...
```

```
// sum = 0
@sum
M=0
```

```
// x = 512
@512
D=A
@x
M=D
```

```
// n = n - 1
@n
M=M-1
```

```
// sum = sum + x
@sum
D=M
@x
D=D+M
@sum
M=D
```

Labels

- Previously, we used branching instructions by directly telling the assembler what instruction we wanted to jump to

- This is cumbersome for several reasons
  - Almost impossible to decipher in complex code
  - Not flexible if code needs further revisions

- We solve this through the use of **labels**

Labels

Hack assembly

```
   ...
(LOOP)
   // if (i = 0) goto CONT
   @i
   D=M
   @CONT
   D;JEQ
   do this
   ...
   // goto LOOP
   @LOOP
   0;JMP
(CONT)
   do that
   ...
```

<u>Hack assembly syntax:</u>

- A label *sym* is declared using (*sym*)
- Any label *sym* declared somewhere in the program can appear in a @*sym* instruction
- The assembler resolves the labels to actual addresses.

Labels

Examples (similar to what we did before):

```
// goto 48
@48
0;JMP
```

```
// if (D > 0) goto 21
@21
D;JGT
```

```
// if (RAM[100] < 0) goto 35
@100
D=M
@35
D;JLT
```

Same examples, using *labels*

```
// goto LOOP
@LOOP
0;JMP
```

```
// if (D > 0) goto CONT
@CONT
D;JGT
```

```
// if (x < 0) goto NEG
@x
D=M
@NEG
D;JLT
```

Hack convention:

Variables: lower-case symbols

Labels: upper-case symbols