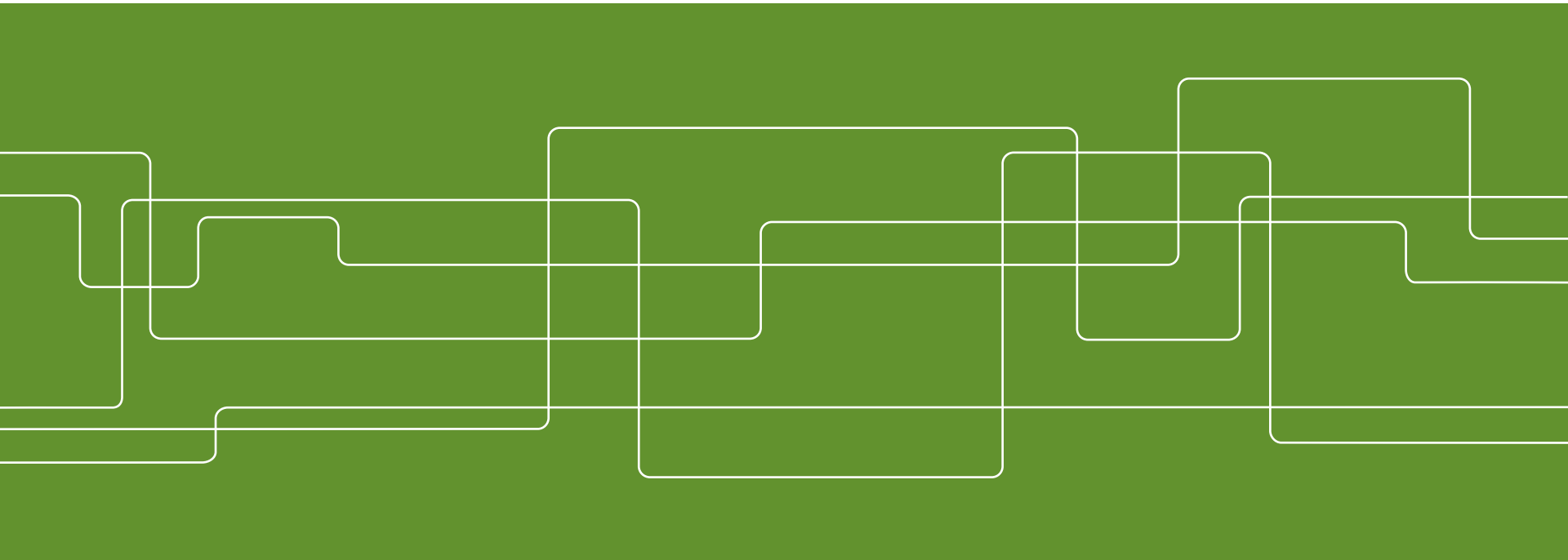

# EP1200 Introduction to Computing Systems Engineering

*Assembler*



# Outlook: from Hack to a “real” computer

Outlook

# Outlook: from Hack to a “real” computer

The Hack CPU and computer HW is as simple as possible

In reality design issues include

- Memory hierarchy – memory access needs time and energy
- Specific processors for specific tasks (graphics, floating point arithmetic)
- Pipelining
  - Several consecutive instructions are processed at the same time, in different stages (e.g., instruction decode and computation)
- Parallel processing
  - Instruction processed at several processors, or
  - Several instructions processed, if order does not matter
- Communication inside the computer - how processors, memory, I/O devices interact
  - Buses and switches – a small network in itself

# Outlook: From Hack to a “real” computer

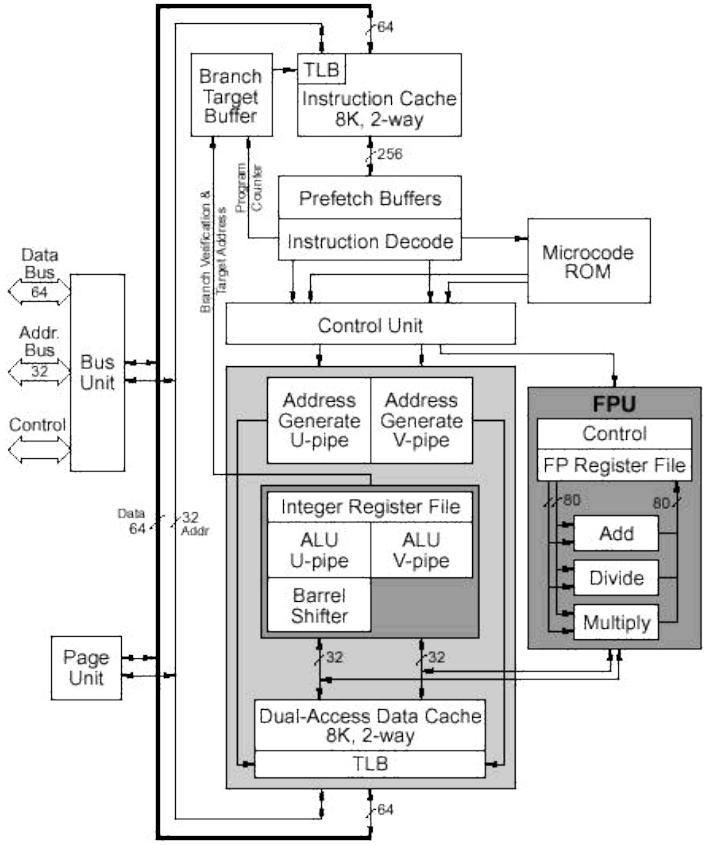
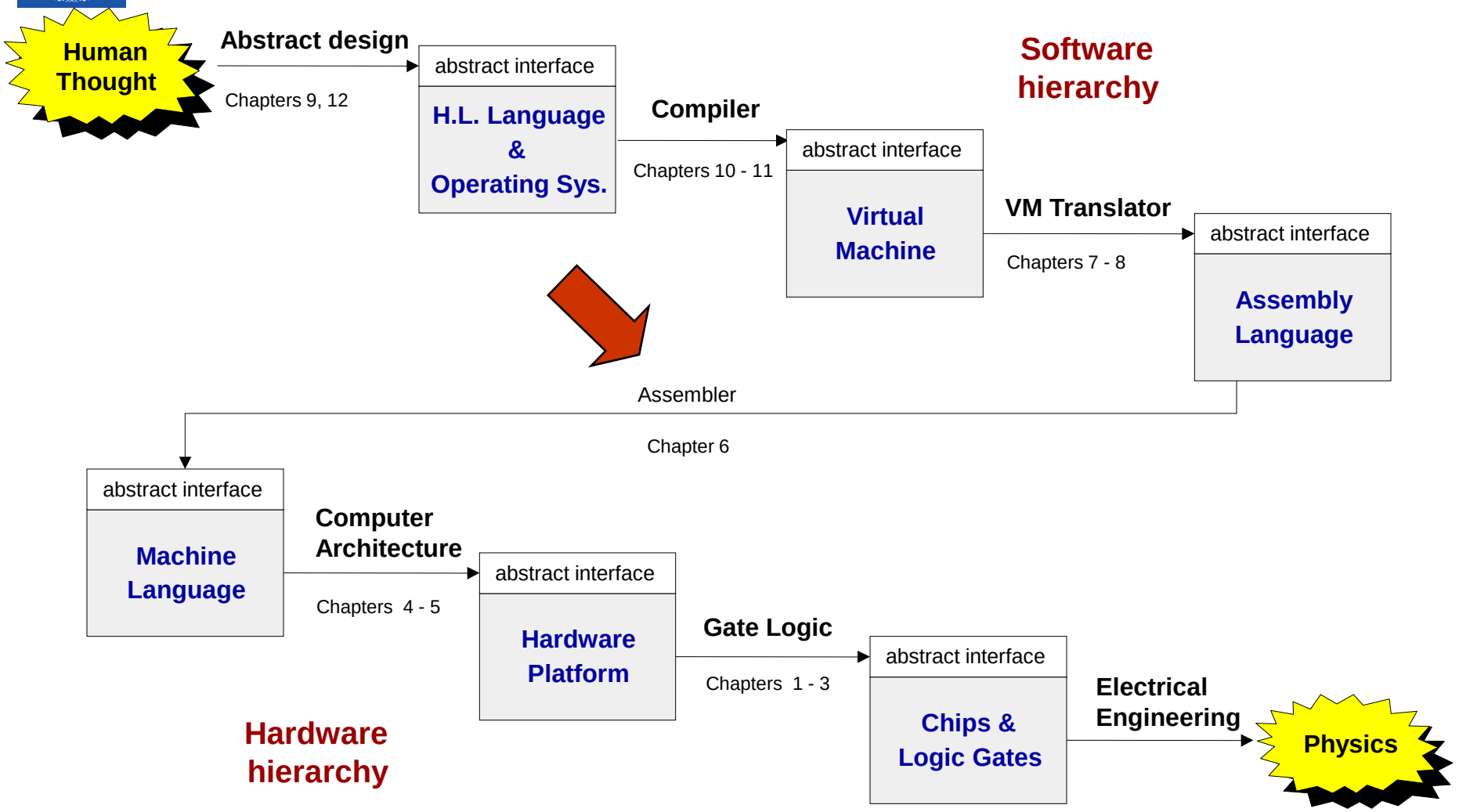



Figure 1. Pentium block diagram.

Contributed by Rajesh Kothandapani
http://www.laynetworks.com

# Where we are at:




# Why care about assemblers?

Because …

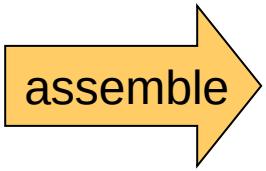
- Assemblers are the first step of the software hierarchy ladder
- An assembler is a translator of a simple language – needs simple programming tools
- Writing an assembler = practice for writing compilers

# Assembler example

**For now, ignore all details!**

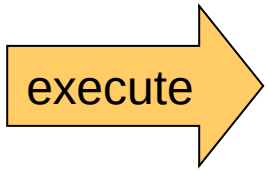
Source code (example)

```
// Computes 1+...+RAM[0]
// And stored the sum in RAM[1]
    @i
    M=1
    @sum
    M=0
(LOOP)
    @i
    D=M
    @R0
    D=D-M
    @WRITE
    D;JGT
    ...    // Etc.
```




Target code

```
0000000000010000
1110111111001000
0000000000010001
1110101010001000
0000000000010000
1111110000010000
0000000000000000
1111010011010000
0000000000010010
1110001100000001
0000000000010000
1111110000010000
0000000000010001
  ...
```



# Assembler example

**For now, ignore all details!**

Source code (example)

```
// Computes 1+...+RAM[0]
// And stored the sum in RAM[1]
    @i
    M=1
    @sum
    M=0
(LOOP)
    @i
    D=M
    @R0
    D=D-M
    @WRITE
    D;JGT
    ...    // Etc.
```

assemble

Target code

```
0000000000010000
1110111111001000
0000000000010001
1110101010001000
0000000000010000
1111110000010000
0000000000000000
1111010011010000
0000000000010010
1110001100000001
0000000000010000
1111110000010000
0000000000010001
  ...
```

execute

The program translation challenge

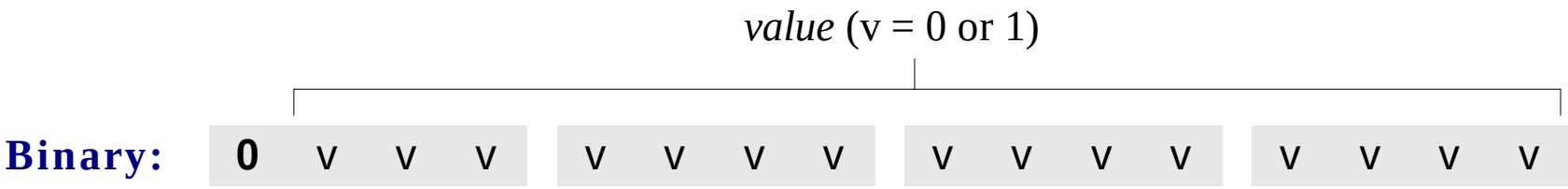
- Extract the program's **semantics** from the source program, using the **syntax** rules of the source language
- Re-express the program's **semantics** in the target language, using the **syntax** rules of the target language

Assembler = simple translator

- Translates each assembly instruction into one binary machine instruction
- Handles symbols (e.g. i, sum, LOOP, …) – maintains a Symbol table <symbol, address>

# Translating / assembling A-instructions

**Symbolic:** @*value* // Where *value* is either a non-negative decimal number
// or a symbol referring to such number.

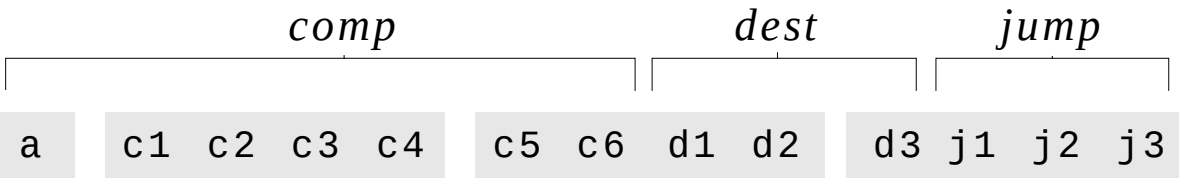



Translation to binary:

- If *value* is a non-negative decimal number, simple
- If *value* is a symbol (label or variable) get address from the symbol table

# Translating / assembling C-instructions

**Symbolic:** *dest*=*comp*;*jump*  // Either the *dest* or *jump* fields may be empty.
// If *dest* is empty, the "=" is ommitted;
// If *jump* is empty, the ";" is omitted.



# Translating / assembling C-instructions

- Translate the a,c,d,j bits:
  - Use the definitions in the tables

| (when a=0) *comp* | c1 | c2 | c3 | c4 | c5 | c6 | (when a=1) |
|---|---|---|---|---|---|---|---|
| 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 | 1 | 1 | 0 | 0 | 0 | 0 | |
| !D | 0 | 0 | 1 | 1 | 0 | 1 | |
| !A | 1 | 1 | 0 | 0 | 0 | 1 | |
| -D | 0 | 0 | 1 | 1 | 1 | 1 | |
| -A | 1 | 1 | 0 | 0 | 1 | 1 | |
| D+1 | 0 | 1 | 1 | 1 | 1 | 1 | |
| A+1 | 1 | 1 | 0 | 1 | 1 | 1 | M+1 |
| D-1 | 0 | 0 | 1 | 1 | 1 | 0 | |
| A-1 | 1 | 1 | 0 | 0 | 1 | 0 | M-1 |
| D+A | 0 | 0 | 0 | 0 | 1 | 0 | D+M |
| D-A | 0 | 1 | 0 | 0 | 1 | 1 | D-M |
| A-D | 0 | 0 | 0 | 1 | 1 | 1 | M-D |
| D&A | 0 | 0 | 0 | 0 | 0 | 0 | D&M |
| D\|A | 0 | 1 | 0 | 1 | 0 | 1 | D\|M |

| d1 | d2 | d3 | *Mnemonic* | *Destination (where to store the computed value)* |
|---|---|---|---|---|
| 0 | 0 | 0 | null | The value is not stored anywhere |
| 0 | 0 | 1 | M | Memory[A] (memory register addressed by A) |
| 0 | 1 | 0 | D | D register |
| 0 | 1 | 1 | MD | Memory[A] and D register |
| 1 | 0 | 0 | A | A register |
| 1 | 0 | 1 | | |
| 1 | 1 | 0 | | |
| 1 | 1 | 1 | | |

| j1 (*out* < 0) | j2 (*out* = 0) | j3 (*out* > 0) | Mnemonic | Effect |
|---|---|---|---|---|
| 0 | 0 | 0 | null | No jump |
| 0 | 0 | 1 | JGT | If *out* > 0 jump |
| 0 | 1 | 0 | JEQ | If *out* = 0 jump |
| 0 | 1 | 1 | JGE | If *out* ≥ 0 jump |
| 1 | 0 | 0 | JLT | If *out* < 0 jump |
| 1 | 0 | 1 | JNE | If *out* ≠ 0 jump |
| 1 | 1 | 0 | JLE | If *out* ≤ 0 jump |
| 1 | 1 | 1 | JMP | Jump |

# The overall assembler logic

For each (real) command

- Parsing
  - break the command into its underlying fields (mnemonics)
- Code generation
  - A-instruction: replace the symbolic reference (if any) with the corresponding memory address, using the symbol table
  - C-instruction: for each field in the instruction, generate the corresponding binary code
- Assemble the translated binary codes into a complete 16-bit machine instruction
- Write the 16-bit instruction to the output file

- Note that comment lines and label declarations (pseudo commands) generate no code

```
    @R0
    D=M
    @END
    D;JLE
    @counter
    M=D
    @SCREEN
    D=A
    @x
    M=D
(LOOP)
    @x
    A=M
    M=-1
    @x
    D=M
    @32
    D=D+A
    @x
    M=D
    @counter
    MD=M-1
    @LOOP
    D;JGT
(END)
    @END
    0;JMP
```


# Handling symbols (aka *symbol resolution*)

Assembly programs can have many different symbols

- Labels that mark destinations of goto commands (ROM)
  - `LOOP, END`
- Pre-defined variables that mark special memory locations (RAM)
  - `R0, SCREEN`
- User defined variables (RAM)
  - `counter, x`
- Symbols are maintained with the help of a symbol table.

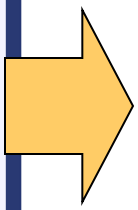
Typical symbolic Hack assembly code:

```
     @R0
     D=M
     @END
     D;JLE
     @counter
     M=D
     @SCREEN
     D=A
     @x
     M=D
(LOOP)
     @x
     A=M
     M=-1
     @x
     D=M
     @32
     D=D+A
     @x
     M=D
     @counter
     MD=M-1
     @LOOP
     D;JGT
(END)
     @END
     0;JMP
```

# Handling symbols: symbol table

Source code (example)

```
// Computes 1+...+RAM[0]
// And stored the sum in RAM[1]
0     @i
1     M=1    // i = 1
2     @sum
3     M=0    // sum = 0
  (LOOP)
4     @i     // if i>RAM[0] goto WRITE
5     D=M
6     @R0
7     D=D-M
8     @WRITE
9     D;JGT
10    @i     // sum += i
11    D=M
12    @sum
13    M=D+M
14    @i     // i++
15    M=M+1
16    @LOOP  // goto LOOP
17    0;JMP
   (WRITE)
18    @sum
19    D=M
20    @R1
21    M=D   // RAM[1] = the sum
   (END)
22    @END
23    0;JMP
```



Symbol table

| | |
|---|---|
| R0 | 0 |
| R1 | 1 |
| R2 | 2 |
| ... | ... |
| R15 | 15 |
| SCREEN | 16384 |
| KBD | 24576 |
| SP | 0 |
| LCL | 1 |
| ARG | 2 |
| THIS | 3 |
| THAT | 4 |
| LOOP | 4 |
| WRITE | 18 |
| END | 22 |
| i | 16 |
| sum | 17 |

Predefined RAM locations, filled in before the assembler process

Labels

Variables

This symbol table is generated by the assembler, and used to translate the symbolic code into binary code.

# The assembly process (detailed)

Initialization:

- Create the symbol table, include pre-defined symbols

First pass:

- Go through the source code without generating any code.
- For each (LABEL) add the pair <LABEL, *n* > to the symbol table (what is n?)

Second pass: march again through the source code, and translate each line:

- If the line is a C-instruction, simple
- If the line is @xxx where xxx is a number, simple
- If the line is @xxx and xxx is a symbol, use the symbol table
  - Add <symbol,n> pair, or get n, if entry already added.

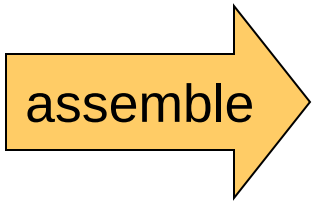
Typical symbolic Hack assembly code:

```
    @R0
    D=M
    @END
    D;JLE
    @counter
    M=D
    @SCREEN
    D=A
    @x
    M=D
(LOOP)
    @x
    A=M
    M=-1
    @x
    D=M
    @32
    D=D+A
    @x
    M=D
    @counter
    MD=M-1
    @LOOP
    D;JGT
(END)
    @END
    0;JMP
```

# The result ...

Source Assembly code

```
// Computes 1+...+RAM[0]
// And stored the sum in RAM[1]
    @i
    M=1   // i = 1
    @sum
    M=0   // sum = 0
(LOOP)
    @i    // if i>RAM[0] goto WRITE
    D=M
    @R0
    D=D-M
    @WRITE
    D;JGT
    @i    // sum += i
    D=M
    @sum
    M=D+M
    @i    // i++
    M=M+1
    @LOOP // goto LOOP
    0;JMP
(WRITE)
    @sum
    D=M
    @R1
    M=D  // RAM[1] = the sum
(END)
    @END
    0;JMP
```




Target Machine Language code

```
0000000000010000
1110111111001000
0000000000010001
1110101010001000
0000000000010000
1111110000010000
0000000000000000
1111010011010000
0000000000010010
1110001100000001
0000000000010000
1111110000010000
0000000000010001
1111000010001000
0000000000010000
1111110111001000
0000000000000100
1110101010000111
0000000000010001
1111110000010000
0000000000000001
1110001100001000
0000000000010110
1110101010000111
```


# Assembler implementation (Chapter 6 project)

- Read Chapter 6 Assembler
- Set up a high level programming environment (suggested: Python)

- Project: Complete the provided Assembler implementation
  - Understand the provided incomplete code
  - Add: Parsing and code generation for *dest* in C-instructions
  - Add: Symbol table handling: address resolution for LABELs

- Hand in Project 5 by April 7, 8:00.