From Nand to Tetris in 12 Steps

The Elements of Computing Systems: Building a Modern Computer From First Principles
Noam Nisan and Shimon Schocken, MIT Press, 2005
The Computer Science Curriculum

Some Open Issues:
- Lack of integration
- Lack of comprehension

Our Solution:
- A new, integrative capstone course
Course Contents

- **Hardware**: Logic gates, Boolean arithmetic, multiplexors, flip-flops, registers, RAM units, counters, Hardware Description Language, chip simulation and testing.

- **Architecture**: ALU/CPU design and implementation, machine code, assembly language programming, addressing modes, memory-mapped input-output (I/O).

- **Data structures and algorithms**: Stacks, hashing, recursion, arithmetic algorithms, geometric algorithms, running time considerations.

- **Programming Languages**: Object-based design and programming, abstract data types, scoping rules, syntax and semantics, references, OS libraries.

- **Compilers**: Lexical analysis, top-down parsing, symbol tables, virtual stack-based machine, code generation, implementation of arrays and objects.

- **Software Engineering**: Modular design, the interface/implementation paradigm, API design and documentation, proactive test planning, quality assurance, programming at the large.
The Course Theme: Let’s Build a Computer

Course Goals

- **Explicit:** Let’s build a computer!
- **Implicit:** Understand ...
  - Key hardware & software abstractions
  - Key interfaces: compilation, VM, O/S
- **Appreciate:** Science history and method
- **Plus:** Have fun.

Course Methodology

- **Constructive:** do-it-yourself
- **Self-contained:** only requirement is programming
- **Guided:** all "plans" are given
- **Focused:** no optimization, no exceptions, no advanced features.
Enter the students data, ending with 'Q':

Name: DAN
Grade: 90

Name: PAUL
Grade: 80

Name: LISA
Grade: 100

Name: ANN
Grade: 90

Name: Q

The grades average is 90
The student with the highest grade is LISA
Sample Applications
Sample Applications
Sample Applications

```
PLEASE TYPE A NUMBER BETWEEN 1 - 9

YOU LOST :-(
```
Course map

- P = Instructional unit (chapter/lecture/project/week)
- Each unit is self-contained
- Each unit provides the building blocks with which we build the unit above it.

Hardware projects

Hardware projects:

- P1: Elementary logic gates
- P2: Combinational gates (ALU)
- P3: Sequential gates (memory)
- P4: Machine language
- P5: Computer architecture

Tools:

- HDL (Hard. Descr. Language)
- Test Description Language
- Hardware simulator.

Hardware platform:

- essential theory (Boolean algebra and gate logic)
- machine language
  - computer architecture
    - ALU / CPU
    - memory units
  - various combinational chips
  - various sequential chips

P1

P2

P3

P4

P5
Project 1: Elementary logic gates

**Given:** \( \text{Nand}(a,b) \), false

- \( \text{Not}(a) = \text{Nand}(a,a) \)
- \( \text{true} = \text{Not}(\text{false}) \)
- \( \text{And}(a,b) = \text{Not}(\text{Nand}(a,b)) \)
- \( \text{Or}(a,b) = \text{Not}(\text{And}(\text{Not}(a),\text{Not}(b))) \)
- \( \text{Mux}(s,a,b) = \text{Or}(\text{And}(s,a),\text{And}(\text{Not}(s),b)) \)
- Etc. - 12 gates altogether.

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>Nand(a,b)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
Building an **And** gate

CHIP And
{  IN  a, b;
    OUT out;
    // implementation missing
}

CHIP And
{  IN  a, b;
    OUT out;
    // implementation missing
}

**Contract:**
When running your .hdl on our .tst, your .out should be the same as our .cmp.

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>out</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

```
load And.hdl,
output-file And.out,
compare-to And.cmp,
output-list a b out;
set a 0, set b 0, eval, output;
set a 0, set b 1, eval, output;
set a 1, set b 0, eval, output;
set a 1, set b 1, eval, output;
```
Building an **And** gate

Interface: $\text{And}(a, b) = 1$ exactly when $a=b=1$

And.hdl

```hdl
CHIP And
{   IN  a, b;
    OUT out;
    // implementation missing
}
```
Building an **And** gate

**Implementation:** \( \text{And}(a,b) = \neg(\text{Nand}(a,b)) \)

```vhdl
CHIP And
{   IN a, b;
    OUT out;
    // implementation missing
}
```
Building an **And** gate

**Implementation:** $\text{And}(a,b) = \text{Not}(\text{Nand}(a,b))$

CHIP And
```
CHIP And
{
  IN  a, b;
  OUT out;
  // implementation missing
}
```
Building an **And** gate

**Implementation:** $\text{And}(a, b) = \text{Not}(\text{Nand}(a, b))$

**And.hdl**

```haskell
CHIP And
{
  IN  a, b;
  OUT out;
  Nand(a = a,
       b = b,
       out = x);
  Not(in = x, out = out)
}
```
Hardware simulator

HDL program

test script

gate diagram
Hardware simulator

HDL program

```
// Xor (exclusive or) gate
// if a<>b out=1 else out=0
CHIP Xor |
  ID a,b;
  OUT out;
  PARAS:
    Not (in=a, out=nota);
    Not (in=b, out=nob);
    And (a=b, not=1);    // a=b, output=1
    And (a=not,b=not, out=x);
    Or (a=x,b=y, out=y);
  }

load Xor,
output-file Xor.out,
constraint-file Xor.exp,
output-list a8'b3.1.3 b8'b3.1.3 out8'b3.1.3;

set a 0,
set b 0,
eval,
output;

set a 0,
set b 1,
eval,
output;

set a 1,
set b 0,
eval,
output;

set a 1,
set b 1,
 eval,
output;
```

Script restarted
Hardware simulator

HDL program

output file

Chip anatomy

CHIP Add16 {
  CHIP FullAdder {
    CHIP HalfAdder {
      CHIP Xor {
        CHIP And {
          CHIP Not {
            CHIP Nand {
              IN a, b;
              OUT out;
              PARTS:
              BUILTIN Nand; // Implemented by Nand.java
            }
          }
        }
      }
    }
  }
}

George Boole
1815-1864

Claude Shannon, 1916-2001
Chip anatomy

Simulator logic:
- Top-down expansion
- Nand is primitive (built-in)
- No chip.hdl? chip.java kicks in
- Instructors/architects can supply built-in versions of any chip.

Benefits:
- Behavioral simulation
- Chip GUI
- Order-free implementation
- Partial implementation is OK
- All HW projects are decoupled.
Hardware projects

Hardware projects:

- **P1**: Elementary logic gates
- **P2**: Combinational gates (ALU)
- **P3**: Sequential gates (memory)
- **P4**: Machine language
- **P5**: Computer architecture

Hardware platform:

- Machine language
- Computer architecture
- ALU / CPU
- Memory units
- Various combinational chips
- Various sequential chips
- Essential theory (Boolean algebra and gate logic)

Project 2: Combinational chips

**Half Adder**
- Inputs: a, b
- Outputs: sum, carry

**Full Adder**
- Inputs: a, b, c
- Outputs: sum, carry

**16-bit Adder**
- Inputs: a, b
- Outputs: sum, carry

**ALU**
- Inputs: x (16 bits), y (16 bits)
- Outputs: out (16 bits)

Output (x, y, control bits) =

- x + y, x - y, y - x,
- 0, 1, -1,
- x, y, -x, -y,
- x!, y!,
- x + 1, y + 1, x - 1, y - 1,
- x & y, x | y
ALU logic

<table>
<thead>
<tr>
<th>zx</th>
<th>nx</th>
<th>zy</th>
<th>ny</th>
<th>f</th>
<th>no</th>
<th>out=</th>
</tr>
</thead>
<tbody>
<tr>
<td>if zx then x=0</td>
<td>if nx then y=0</td>
<td>if ny then z=y</td>
<td>if f then out=x+y else out=x And y</td>
<td>if no then out=!out</td>
<td>f(X,Y)=</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>y</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>!x</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>!y</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>-x</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>-y</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>x+1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>y+1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>x-1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>y-1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>x+y</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>x-y</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>y-x</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>x&amp;y</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>x·y</td>
</tr>
</tbody>
</table>
A glimpse ahead:

<table>
<thead>
<tr>
<th>out (when a=0)</th>
<th>c1</th>
<th>c2</th>
<th>c3</th>
<th>c4</th>
<th>c5</th>
<th>c6</th>
<th>out (when a=1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>-1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>M</td>
</tr>
<tr>
<td>!D</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>!M</td>
</tr>
<tr>
<td>!A</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>!M</td>
</tr>
<tr>
<td>-D</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>-M</td>
</tr>
<tr>
<td>-A</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>-M</td>
</tr>
<tr>
<td>D+1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>A+1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>M+1</td>
</tr>
<tr>
<td>D-1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>A-1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>M-1</td>
</tr>
<tr>
<td>D+A</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>D+M</td>
</tr>
<tr>
<td>D-A</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>D-M</td>
</tr>
<tr>
<td>A-D</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>M-D</td>
</tr>
<tr>
<td>D&amp;A</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>D&amp;M</td>
</tr>
<tr>
<td>D</td>
<td>A</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>D</td>
</tr>
</tbody>
</table>
Hardware projects:

- **P1**: Elementary logic gates
- **P2**: Combinational gates (ALU)
- **P3**: Sequential chips (memory)
- **P4**: Machine language
- **P5**: Computer architecture
Project 3: Sequential chips

- DFF > Bit > Register > RAM8 > RAM64 > ... > RAM32K
Hardware projects:

- P1: Elementary logic gates
- P2: Combinational gates (ALU)
- P3: Sequential gates (memory)
- P4: Machine language
- P5: Computer architecture
Machine Language: **A-instruction**

```
@value // A register = value
```

**Symbolic:**

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

```
value (v = 0 or 1)
```

**Binary:**

```
0  V  V  V  V  V  V  V  V  V  V  V  V  V  V  V  V  V
```
Machine Language: **C-instruction**

\[
\text{dest} = \text{comp} ; \text{jump} \quad // \text{If dest is null, the } `=` \text{ is omitted}
\]

// If jump is null, the `;` is omitted

**comp** is one of:

\[0,1,-1,D,A,!D,!A,-D,-A,D+1,A+1,D-1,A-1,D+A,D-A,A-D,D\&A,D\mid A, M, \!M, \!M, M+1, M-1,D+M,D-M,M-D,D\&M,D\mid M\]

**dest** is one of:

\[
\text{Null, M, D, MD, A, AM, AD, AMD}
\]

**jump** is one of:

\[
\text{Null, JGT, JEQ, JGE, JLT, JNE, JLE, JMP}
\]
Machine Language: **C-instruction** (cont.)

**Symbolic:**  
\[ \text{dest}=\text{comp}; \text{jump} \quad \text{// Either the dest or jump fields may be empty.} \]  
\[ \text{// If dest is empty, the "=" is omitted;} \]  
\[ \text{// If jump is empty, the ";" is omitted.} \]

**Binary:**  
![Binary representation of C-instruction](image)

<table>
<thead>
<tr>
<th>((\text{when } a=0))</th>
<th>(c_1)</th>
<th>(c_2)</th>
<th>(c_3)</th>
<th>(c_4)</th>
<th>(c_5)</th>
<th>(c_6)</th>
<th>((\text{when } a=1))</th>
<th>(c_1)</th>
<th>(c_2)</th>
<th>(c_3)</th>
<th>(c_4)</th>
<th>(c_5)</th>
<th>(c_6)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(D)</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>(M)</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>null</td>
<td>The value is not stored anywhere</td>
<td></td>
</tr>
<tr>
<td>(1)</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>(M)</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>(M)</td>
<td>Memory[A] (memory register addressed by A)</td>
<td></td>
</tr>
<tr>
<td>(\neg 1)</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>(M)</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>(D)</td>
<td>D register</td>
<td></td>
</tr>
<tr>
<td>(D)</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>(M)</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>(MD)</td>
<td>Memory[A] and D register</td>
<td></td>
</tr>
<tr>
<td>(A)</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>(M)</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>(A)</td>
<td>A register</td>
<td></td>
</tr>
<tr>
<td>(!D)</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>(M)</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>(AM)</td>
<td>A register and Memory[A]</td>
<td></td>
</tr>
<tr>
<td>(!A)</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>(M)</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>(AD)</td>
<td>A register and D register</td>
<td></td>
</tr>
<tr>
<td>(\neg D)</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>(M)</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>(AMD)</td>
<td>A register, Memory[A], and D register</td>
<td></td>
</tr>
<tr>
<td>(D+1)</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>(M+1)</td>
<td>(j_1)</td>
<td>(j_2)</td>
<td>(j_3)</td>
<td>(null)</td>
<td>No jump</td>
<td></td>
</tr>
<tr>
<td>(A+1)</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>(M+1)</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>(JGT)</td>
<td>If out &gt; 0 jump</td>
<td></td>
</tr>
<tr>
<td>(D-1)</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>(M-1)</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>(JEQ)</td>
<td>If out = 0 jump</td>
<td></td>
</tr>
<tr>
<td>(A-1)</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>(M-1)</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>(JGE)</td>
<td>If out ≥ 0 jump</td>
<td></td>
</tr>
<tr>
<td>(D+X)</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>(D+X)</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>(JLT)</td>
<td>If out &lt; 0 jump</td>
<td></td>
</tr>
<tr>
<td>(A-D)</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>(AD)</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>(JNE)</td>
<td>If out ≠ 0 jump</td>
<td></td>
</tr>
<tr>
<td>(D&amp;A)</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>(D&amp;A)</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>(JLE)</td>
<td>If out ≤ 0 jump</td>
<td></td>
</tr>
<tr>
<td>(D</td>
<td>A)</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>(D</td>
<td>A)</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>(JMP)</td>
</tr>
</tbody>
</table>
Hardware projects:

- P1: Elementary logic gates
- P2: Combinational gates (ALU)
- P3: Sequential gates (memory)
- P4: Machine language
- P5: Computer architecture
Project 5: CPU

[Diagram of a CPU with labels for instruction, inM, reset, etc.]
Project 5: Computer

Many other computer models can be built.

J. Von Neumann
(1903-1957)
Input / Output Devices

Data Memory

- RAM (16K)
- screen memory map (8K)
- keyboard memory map

Addressing:
- in (16)
- address (15)
- load

Output:
- out (16)
- screen
- Keyboard
Recap: the Hack chip-set and hardware platform

<table>
<thead>
<tr>
<th>Elementary logic gates</th>
<th>Combinational chips</th>
<th>Sequential chips</th>
<th>Computer Architecture</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>(Project 1):</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Nand (primitive)</td>
<td>HalfAdder</td>
<td>DFF (primitive)</td>
<td>Memory</td>
</tr>
<tr>
<td>Not</td>
<td>FullAdder</td>
<td>Bit</td>
<td>CPU</td>
</tr>
<tr>
<td>And</td>
<td>Add16</td>
<td>Register</td>
<td>Computer</td>
</tr>
<tr>
<td>Or</td>
<td>Inc16</td>
<td>RAM8</td>
<td></td>
</tr>
<tr>
<td>Xor</td>
<td>ALU</td>
<td>RAM64</td>
<td></td>
</tr>
<tr>
<td>Mux</td>
<td>DMux</td>
<td>RAM512</td>
<td></td>
</tr>
<tr>
<td>Dmux</td>
<td>Not16</td>
<td>RAM4K</td>
<td></td>
</tr>
<tr>
<td>And16</td>
<td>Or16</td>
<td>RAM16K</td>
<td></td>
</tr>
<tr>
<td>Or16</td>
<td>Mux16</td>
<td>PC</td>
<td></td>
</tr>
<tr>
<td>Mux16</td>
<td>Or8Way</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Or8Way</td>
<td>Mux4Way16</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mux4Way16</td>
<td>Mux8Way16</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mux8Way16</td>
<td>DMux4Way</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DMux4Way</td>
<td>DMux8Way</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Software projects

Software hierarchy

- Operating system (P12)
- Prog. language (P9)
- Compiler (P10, 11)
- Stack machine
- Virtual machine (P7, 8)
- Assembly
- Assembler (P6)
- Machine language
Project 6: Assembler

Sum.asm

// Computes sum=1+2+ ... +100.
@i    // i=1
M=1
@sum  // sum=0
M=0
(LOOP)
@i    // if i-100>0 goto END
D=M
@100
D=D-A
@END
D;jgt
@i    // sum+=i
D=M
@sum
M=D+M
@i    // i++
M=M+1
@LOOP // goto LOOP
0;jmp
(END)
@END
0;JMP

Sum.bin

0000000000010000
1110111111001000
0000000000010001
1110101010001000
0000000000100000
1111000000010000
0000000000100000
1111110000010000
0000000000100000
1110011011010000
0000000000010010
1110000000000001
0000000000100000
1111110000010000
0000000000010001
1111000010001000
0000000000010000
1111110011101000
0000000000010010
1110101010000000
1110101010000111

Ada Lovelace
(1815-1852)
Assembler in action

<table>
<thead>
<tr>
<th>Source</th>
<th>Destination</th>
</tr>
</thead>
<tbody>
<tr>
<td>// compute sum=A+..+100</td>
<td>0000000000000001</td>
</tr>
<tr>
<td>@1</td>
<td>1110111111111101000</td>
</tr>
<tr>
<td>B=1 # source=1</td>
<td>000000000000000011001</td>
</tr>
<tr>
<td>@sum # allocated at JAS[16]</td>
<td>1110101010000000</td>
</tr>
<tr>
<td>END</td>
<td>000000000000000001001</td>
</tr>
<tr>
<td>(LOOP)</td>
<td>1110110111011000</td>
</tr>
<tr>
<td>@1</td>
<td>00000000000000000000000000000000000000000001</td>
</tr>
<tr>
<td>P=P</td>
<td>1110110111001000</td>
</tr>
<tr>
<td>@100</td>
<td>000000000000000011001</td>
</tr>
<tr>
<td>D=A # if count=100 ...</td>
<td>1110101010000000</td>
</tr>
<tr>
<td>END</td>
<td>000000000000000001001</td>
</tr>
<tr>
<td>D,GET # goto end</td>
<td>1110110111011000</td>
</tr>
<tr>
<td>@1</td>
<td>000000000000000011001</td>
</tr>
<tr>
<td>P=P</td>
<td>1110110111001000</td>
</tr>
<tr>
<td>@sum</td>
<td>00000000000000000000000000000000000000000001</td>
</tr>
<tr>
<td>B=B+# sum=sum+count</td>
<td>1111111111111100100</td>
</tr>
<tr>
<td>@1</td>
<td>000000000000000011001</td>
</tr>
<tr>
<td>LOOP</td>
<td>1110110111001000</td>
</tr>
<tr>
<td>0,REP</td>
<td>000000000000000011001</td>
</tr>
<tr>
<td>(END)</td>
<td>11101010100000111</td>
</tr>
<tr>
<td>0,REP</td>
<td>000000000000000011001</td>
</tr>
<tr>
<td>0,REP</td>
<td>11101010100000111</td>
</tr>
</tbody>
</table>

Project 6: Build an assembler

File compilation succeeded
CPU Emulator

instruction memory

data memory

keyboard enabler

256 by 512 pixels simulated screen

program counter

address register

ALU

Data register
Software projects

Software hierarchy

- Operating system (P12)
- Program language (P9)
- Compiler (P10, 11)
- Stack machine
- Virtual machine (P7, 8)
- Assembly
- Assembler (P6)
- Machine language
The Big Picture

Projects 1-6

Proj. 9: building an app.
Proj. 12: building the OS

Projects 7-8

Projects 10-11

VM language

VM implementation over CISC platforms

CISC machine language

Any computer

CISC machine

VM imp. over the Hack platform

VM emulator

Hack machine language

Hack computer

Some language

Some compiler

VM language

Some Other language

Some Other compiler

Jack language

Jack compiler

Other digital platforms, each equipped with its VM implementation

written in a high-level language

Some language

Some Other language

Jack language

Some Other compiler

VM implementation over RISC platforms

CISC machine language

RISC machine language

Projects 7-8

Some language

Some Other language

Jack language

Some Other compiler

VM implementation over RISC platforms

CISC machine language

RISC machine language

Projects 10-11

Some language

Some Other language

Jack language

Some Other compiler

VM implementation over CISC platforms

CISC machine language

RISC machine language

Projects 1-6

Some language

Some Other language

Jack language

Some Other compiler

VM implementation over RISC platforms

CISC machine language

RISC machine language

Projects 7-8
The VM language

**Arithmetic commands**
- add
- sub
- neg
- eq
- gt
- lt
- and
- or
- not

**Memory access commands**
- pop segment i
- push segment i

**Program flow commands**
- label symbol
- goto symbol
- if-goto symbol

**Function calling commands**
- function functionName nLocals
- call functionName nArgs
- return
## The VM abstraction: a Stack Machine

### High-level code

```c
function mult(x,y) {
    int result, j;
    result=0;
    j=y;
    while ~(j=0) {
        result=result+x;
        j=j-1;
    }
    return result;
}
```

### Stack machine code

```assembly
function mult(x,y) {
    push 0
    pop result
    push y
    pop j
    label loop
    push j
    push 0 eqif-goto end
    push result
    push x add pop result
    push j
    push 0
    eq
    if-goto end
    push result
    push x add
    pop result
    push j
    push 1 sub
    pop j
goto loop
label end
push result
return
}
```

### VM code

```assembly
function mult 2
    push constant 0
    pop local 0
    push argument 1
    pop local 1
    label loop
    push local 1
    push constant 0 eq
    if-goto end
    push local 0
    push argument 0 add
    pop local 0
    push local 1
    push constant 1 sub
    pop local 1
goto loop
label end
push local 0
return
```
Virtual Machine Emulator

- VM code
- Working stack
- Call stack
- Static segments
- Local segments
- Argument segments
- Global stack
- Stack
- RAM
- Default test script
- Virtual memory segments
- Host RAM (the RAM is not part of the VM)
Projects 7,8: Implement the VM over the Hack platform

Mult.vm

```vm
function mult 2  // 2 local variables
    push constant 0  // result=0
    pop local 0
    push argument 1  // j=y
    pop local 1
label loop
    push constant 0  // if j==0 goto end
    push local 1
    eq
    if-goto end
    push local 0    // result=result+x
    push argument 0
    add
    pop local 0
    push local 1    // j=j-1
    push constant 1
    sub
    pop local 1
    goto loop
label end
    push local 0    // return result
    return
```

Mult.asm

```asm
... 
A=M-1  
M=0  
@5  
D=A  
@LCL  
A=M-D  
D=M  
@R6  
M=D  
@SP  
AM=M-1  
D=M  
@ARG  
A=M  
M=D  
D=A  
@SP  
M=D+1  
@LCL  
D=M  
... 
```

VM Translator
Software hierarchy

- Operating system
- Program language
- Compiler
- Stack machine
- Virtual machine
- Assembly
- Assembler
- Machine language
class Math {

/** Returns n! */
function int factorial(int n)
    if (n = 0) {
        return 1;
    }
    else {
        return n * Math.factorial(n - 1);
    }
}

/** Returns e=\sum(1/n!) where n goes from 0 to infinity */
function Fraction e (int n){
    var int i;
    let i = 0;
    let e = Fraction.new(0,1);  // start with e=0
    // approximate up to n
    while (i < n) {
        let e = e.plus(Fraction.new(1, Math.factorial(i)));
        let i = i + 1;
    }
    return e;
}

} // end Math
Project 9: Write a Jack program

Demo Pong
Software projects

Software hierarchy

- Operating system
- Compiler
- Stack machine
- Virtual machine
- Assembly
- Assembler
- Machine language
- Programming language

Compiler project I: Syntax Analysis

Prog.jack

(5+y)*2 - sqrt(x*4)

Prog.xml

```xml
<expression>
  <term>
    <symbol> ( </symbol>
    <expression>
      <term>
        <integerConstant> 5 </integerConstant>
      </term>
      <symbol> + </symbol>
      <term>
        <identifier> y </identifier>
      </term>
    </expression>
    <symbol> ) </symbol>
  </term>
  <symbol> * </symbol>
  <expression>
    <term>
      <integerConstant> 2 </integerConstant>
    </term>
    <symbol> * </symbol>
    <expression>
      <term>
        <integerConstant> 4 </integerConstant>
      </term>
      <symbol> + </symbol>
      <term>
        <identifier> x </identifier>
      </term>
    </expression>
  </expression>
  <symbol> - </symbol>
</expression>
```

Jack Grammar

Syntax Analyzer
Compiler project II: Code Generation

Prog.jack

\[(5+y) \times 2 - \sqrt{x \times 4}\]

Syntax analyzer

Project 9

Prog.vm

push 5
push y
add
push 2
call mult
push x
call mult
call sqrt
sub

Project 10

Code generator
Software projects

Software hierarchy

- Operating system
- Program language
- Compiler
- Stack machine
- Virtual machine
- Assembly
- Assembler
- Machine language
/** Computes the average of a sequence of integers. */
class Main {
    function void main() {
        var Array a;
        var int length;
        var int i, sum;

        let length = Keyboard.readInt("How many numbers? ");
        let a = Array.new(length); // Constructs the array
        let i = 0;

        while (i < length) {
            let a[i] = Keyboard.readInt("Enter the next number: ");
            let sum = sum + a[i];
            let i = i + 1;
        }

        do Output.printString("The average is: ");
        do Output.printInt(sum / length);
        do Output.println();
        return;
    }
}
**OS Libraries**

- **Math**: Provides basic mathematical operations;
- **String**: Implements the `String` type and string-related operations;
- **Array**: Implements the `Array` type and array-related operations;
- **Output**: Handles text output to the screen;
- **Screen**: Handles graphic output to the screen;
- **Keyboard**: Handles user input from the keyboard;
- **Memory**: Handles memory operations;
- **Sys**: Provides some execution-related services.
class Math {
class String {
Class Array {
class Output {
Class Screen {
class Memory {
Class Keyboard {
Class Sys {
    function void halt():
    function void error(int errorCode)
    function void wait(int duration)
}
Recap

250 lines of Jack code
1000 lines of VM code
4,000 lines of Assembly
30,000 lines of Hack code
500,000 bits
2,000,000 logic gates
1 God

application (e.g. Pong)
15 obj-oriented bat / ball operations

high-level language / OS
250 lines of Jack code

virtual machine
1000 lines of VM code

assembly
4,000 lines of Assembly

machine language
30,000 lines of Hack code

architecture
500,000 bits

chip-set
2,000,000 logic gates

logic
1 God
God gave us 0 and Nand

Everything else was done by humans.
Why the approach works

Take home

- Initial goal: Acquire enough hands-on knowledge for building a computer system
- Final lesson: This includes some of the most beautiful topics in applied CS

Occam razor

- No optimization
- No advanced features
- No exceptions

Highly-Managed

- Detailed API’s are given
- Hundreds of test files and programs
- Modular projects, unit-testing.
Abstraction–Implementation Paradigm

Human Thought

Abstract design

P9, P12

H.L. Language & Operating Sys.

Compiler

P10, P11

Virtual Machine

VM Translator

P7, P8

Assembly Language

Assembler

P6

Computer Architecture

Machine Language

abstract interface

P4, P5

Hardware Platform

Gate Logic

P1, P2, P3

Chips & Logic Gates

abstract interface

Electrical Engineering

Physics

Hardware platform

Software hierarchy

www.idc.ac.il/tecs