Computer Architecture

Building a Modern Computer From First Principles

www.nand2tetris.org
Where we are at:

- **Human Thought**
  - Abstract design
    - Chapters 9, 12

- **Hardware hierarchy**
  - Machine Language
    - Abstract interface
    - Chapters 4 - 5
  - Computer Architecture
    - Abstract interface
    - Chapters 4 - 5
  - Hardware Platform
    - Abstract interface
    - Chapters 1 - 3
  - Chips & Logic Gates
    - Abstract interface
    - Chapters 7 - 8
  - Electrical Engineering
    - Abstract interface
    - Chapters 7 - 8
  - Physical Engineering
    - Abstract interface
    - Chapters 7 - 8

- **Software hierarchy**
  - Assembly Language
    - Abstract interface
    - Chapters 10 - 11
  - Virtual Machine
    - Abstract interface
    - Chapters 7 - 8
  - Compiler
    - Abstract interface
    - Chapters 10 - 11
  - Assembler
    - Abstract interface
    - Chapter 6
The Hack computer

A 16-bit machine consisting of the following elements:

- Computer
- Screen
- Keyboard
- The program is stored in a ROM.
The A-instruction

**symbolic**

@value

- value is a non-negative decimal number $\leq 2^{15} - 1$ or
- A symbol referring to such a constant

**binary**

0value

- value is a 15-bit binary number

Example

@21

0000 0000 0001 0101
The C-instruction

\[ \text{dest} = \text{comp} ; \text{jump} \]

symbolic

\[
\begin{array}{cccccccc}
111 & A & C_1 & C_2 & C_3 & C_4 & C_5 & C_6 & D_1 & D_2 & D_3 & J_1 & J_2 & J_3
\end{array}
\]

binary

opcode

not used

comp

dest

jump
The **C-instruction**

![C-instruction diagram](image)

<table>
<thead>
<tr>
<th>comp</th>
<th>c1</th>
<th>c2</th>
<th>c3</th>
<th>c4</th>
<th>c5</th>
<th>c6</th>
<th>(when a=1) comp</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>M</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>
</tr>
<tr>
<td>-1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</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>0</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>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>M+1</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>M-1</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>1</td>
</tr>
</tbody>
</table>
# The C-instruction

![C-instruction diagram](image)

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>D</td>
<td>M</td>
<td>d1</td>
<td>d2</td>
<td>d3</td>
<td>Mnemonic</td>
<td>Destination (where to store the computed value)</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td>-----</td>
<td>----</td>
<td>----</td>
<td>-----------</td>
<td>-----------------------------------------------</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>null</td>
<td>The value is not stored anywhere</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>M</td>
<td>Memory[A] (memory register addressed by A)</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>D</td>
<td>D register</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>MD</td>
<td>Memory[A] and D register</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>A</td>
<td>A register</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>AM</td>
<td>A register and Memory[A]</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>AD</td>
<td>A register and D register</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>AMD</td>
<td>A register, Memory[A], and D register</td>
</tr>
</tbody>
</table>
The **C**-instruction

<table>
<thead>
<tr>
<th>( j_1 ) ((out &lt; 0))</th>
<th>( j_2 ) ((out = 0))</th>
<th>( j_3 ) ((out &gt; 0))</th>
<th><strong>Mnemonic</strong></th>
<th><strong>Effect</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>null</td>
<td>No jump</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>JGT</td>
<td>If ( out &gt; 0 ) jump</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>JEQ</td>
<td>If ( out = 0 ) jump</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>JGE</td>
<td>If ( out \geq 0 ) jump</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>JLT</td>
<td>If ( out &lt; 0 ) jump</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>JNE</td>
<td>If ( out \neq 0 ) jump</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>JLE</td>
<td>If ( out \leq 0 ) jump</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>JMP</td>
<td>Jump</td>
</tr>
</tbody>
</table>
**Hack assembly/machine language**

**Source code (example)**

```
// 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 code**

```
0000000000010000
1110111111010000
0000000000100000
1110101010001000
0000000001000000
1111110000010000
0000000000000000
1111010011010000
0000000000010010
111000110000001
0000000000000001
0000000000000000
1111110000010000
0000000000100000
1111111100000100
0000000000000010
1110101010000111
0000000000000000
1111110000010000
0000000000010010
111000110000001
0000000000000000
1111111100000100
0000000000000010
1110101010000111
```

**assemble**

Hack assembler or CPU emulator

**assembly code v.s. machine code**
The Hack computer

- A 16-bit Von Neumann platform
- The *instruction memory* and the *data memory* are physically separate
- Screen: 512 rows by 256 columns, black and white
- Keyboard: standard
- Designed to execute programs written in the Hack machine language
- Can be easily built from the chip-set that we built so far in the course

**Main parts of the Hack computer:**
- Instruction memory (ROM)
- Memory (RAM):  
  - Data memory  
  - Screen (memory map)  
  - Keyboard (memory map)
- CPU
- Computer (the logic that holds everything together).
Goal: select from one of n k-bit buses
- Implemented by layering k n-to-1 multiplexer
Hack ALU

\[ \text{out}(x, y, \text{control bits}) = \]
\[
\begin{align*}
&x+y, x-y, y-x, \\
&0, 1, -1, \\
&x, y, -x, -y, \\
&x!, y!, \\
&x+1, y+1, x-1, \\
&y-1, \\
&x\&y, x|y
\end{align*}
\]
## Hack ALU

<table>
<thead>
<tr>
<th>These bits instruct how to preset the x input</th>
<th>These bits instruct how to preset the y input</th>
<th>This bit selects between + / And</th>
<th>This bit instr. how to postset out</th>
<th>Resulting ALU output</th>
</tr>
</thead>
<tbody>
<tr>
<td>( zx )</td>
<td>( nx )</td>
<td>( zy )</td>
<td>( ny )</td>
<td>( f )</td>
</tr>
<tr>
<td>if ( zx ) then ( x=0 )</td>
<td>if ( nx ) then ( x=!x )</td>
<td>if ( zy ) then ( y=0 )</td>
<td>if ( ny ) then ( y=!y )</td>
<td>if ( f ) then out=x+y</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>else out=x&amp;y</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>if no then out=!out</td>
</tr>
<tr>
<td>( 1 )</td>
<td>( 1 )</td>
<td>( 1 )</td>
<td>( 0 )</td>
<td>( 1 )</td>
</tr>
<tr>
<td>( 0 )</td>
<td>( 1 )</td>
<td>( 1 )</td>
<td>( 0 )</td>
<td>( 0 )</td>
</tr>
<tr>
<td>( 1 )</td>
<td>( 0 )</td>
<td>( 1 )</td>
<td>( 0 )</td>
<td>( 0 )</td>
</tr>
<tr>
<td>( 1 )</td>
<td>( 1 )</td>
<td>( 0 )</td>
<td>( 0 )</td>
<td>( 1 )</td>
</tr>
<tr>
<td>( 0 )</td>
<td>( 0 )</td>
<td>( 1 )</td>
<td>( 1 )</td>
<td>( 1 )</td>
</tr>
<tr>
<td>( 1 )</td>
<td>( 1 )</td>
<td>( 0 )</td>
<td>( 1 )</td>
<td>( 1 )</td>
</tr>
<tr>
<td>( 0 )</td>
<td>( 0 )</td>
<td>( 1 )</td>
<td>( 1 )</td>
<td>( 1 )</td>
</tr>
<tr>
<td>( 1 )</td>
<td>( 1 )</td>
<td>( 0 )</td>
<td>( 1 )</td>
<td>( 1 )</td>
</tr>
<tr>
<td>( 0 )</td>
<td>( 0 )</td>
<td>( 1 )</td>
<td>( 1 )</td>
<td>( 1 )</td>
</tr>
<tr>
<td>( 1 )</td>
<td>( 1 )</td>
<td>( 0 )</td>
<td>( 1 )</td>
<td>( 1 )</td>
</tr>
<tr>
<td>( 0 )</td>
<td>( 0 )</td>
<td>( 1 )</td>
<td>( 1 )</td>
<td>( 1 )</td>
</tr>
<tr>
<td>( 1 )</td>
<td>( 1 )</td>
<td>( 0 )</td>
<td>( 1 )</td>
<td>( 1 )</td>
</tr>
<tr>
<td>( 0 )</td>
<td>( 0 )</td>
<td>( 1 )</td>
<td>( 1 )</td>
<td>( 1 )</td>
</tr>
<tr>
<td>( 1 )</td>
<td>( 1 )</td>
<td>( 0 )</td>
<td>( 1 )</td>
<td>( 1 )</td>
</tr>
<tr>
<td>( 0 )</td>
<td>( 0 )</td>
<td>( 1 )</td>
<td>( 1 )</td>
<td>( 1 )</td>
</tr>
<tr>
<td>( 0 )</td>
<td>( 0 )</td>
<td>( 1 )</td>
<td>( 1 )</td>
<td>( 1 )</td>
</tr>
<tr>
<td>( 1 )</td>
<td>( 1 )</td>
<td>( 0 )</td>
<td>( 1 )</td>
<td>( 1 )</td>
</tr>
<tr>
<td>( 0 )</td>
<td>( 0 )</td>
<td>( 1 )</td>
<td>( 1 )</td>
<td>( 1 )</td>
</tr>
<tr>
<td>( 0 )</td>
<td>( 0 )</td>
<td>( 1 )</td>
<td>( 1 )</td>
<td>( 1 )</td>
</tr>
<tr>
<td>( 1 )</td>
<td>( 1 )</td>
<td>( 0 )</td>
<td>( 1 )</td>
<td>( 1 )</td>
</tr>
</tbody>
</table>

\( f(x, y) = \)
Registers

k-bit register.
- Stores k bits.
- Register contents always available on output.
- If write enable is asserted, k input bits get copied into register.

**Ex:** Program Counter, 16 TOY registers, 256 TOY memory locations.

16-bit Register Interface

16-bit Register Implementation
Function:

- The ROM is pre-loaded with a program written in the Hack machine language.
- The ROM chip always emits a 16-bit number:
  \[ \text{out} = \text{ROM32K}[\text{address}] \]
- This number is interpreted as the current instruction.
RAM (data memory)

- We will discuss the details for Hack's data memory later.
Clock

- **Fundamental abstraction:** regular on-off pulse.
  - **on:** fetch phase
  - **off:** execute phase
- **External analog device.**
- **Synchronizes operations of different circuit elements.**
- **Requirement:** clock cycle longer than max switching time.
Design a processor

- How to build a processor (Hack, this time)
  - Develop instruction set architecture (ISA)
    - 16-bit words, two types of machine instructions
  - Determine major components
    - ALU, registers, program counter, memory
  - Determine datapath requirements
    - Flow of bits
  - Analyze how to implement each instruction
    - Determine settings of control signals
Hack programming reference card

**Hack commands:**

**A-command:**  \( @\text{value} \)  // \( A<-\text{value}; \ M=\text{RAM}[A] \)

**C-command:**  \( \text{dest} = \text{comp} \; ; \text{jump} \)  // \( \text{dest} = \text{and} \; ; \text{jump} \)

// are optional

Where:

\( \text{comp} = \)


**dest =** M, D, A, MD, AM, AD, AMD, or null

**jump =** JGT, JEQ, JGE, JLT, JNE, JLE, JMP, or null

In the command \( \text{dest} = \text{comp}; \text{jump} \), the jump materializes \( (PC<-A) \) if \( (\text{comp} \; \text{jump} \; 0) \) is true. For example, in \( D=D+1,JLT \), we jump if \( D+1 < 0 \).
Fetch and execute

- In Toy, we have two phases: fetch and execution.

- We use two cycles since fetch and execute phases each access memory and alter program counter.
  - fetch [set memory address from pc]
  - fetch and clock [write instruction to IR]
  - execute [set ALU inputs from registers]
  - execute and clock [write result of ALU to registers]
Both fetch and execute would access memory. To avoid conflict, we add a MUX. Similar for PC.
In addition, we need a register IR to store the instruction.
In Hack, we avoid it by using two separate memory chips, one for data and the other for instruction.
Design a processor

- How to build a processor (Hack, this time)
  - Develop instruction set architecture (ISA)
    - 16-bit words, two types of machine instructions

- Determine major components
  - ALU, registers, program counter, memory

- Determine datapath requirements
  - Flow of bits

- Analyze how to implement each instruction
  - Determine settings of control signals
Program counter

- Program counter emits the address of the next instruction.
  - To start/restart the program execution: PC=0
  - No jump: PC++
  - Unconditional jump: PC=A
  - Conditional jump: if (cond.) PC=A else PC++

Note that the design is slightly different from your project #3.
Program counter

if (reset) PC=0
else if (W) PC=Din
else PC++
Program counter

\[
\begin{align*}
&\text{if (reset) } PC=0 \\
&\text{else if (W) } PC=\text{Din} \\
&\text{else } PC++
\end{align*}
\]
Hack architecture (component)
Design a processor

How to build a processor (Hack, this time)

- Develop instruction set architecture (ISA)
  - 16-bit words, two types of machine instructions

- Determine major components
  - ALU, registers, program counter, memory

- Determine datapath requirements
  - Flow of bits

- Analyze how to implement each instruction
  - Determine settings of control signals
 Hack architecture (data path)

@value // A<-value; M=RAM[A]
[ADM] = x op y; jump // x=D; y=A or M; if jump then PC<-A
Hack architecture (data path)

```
@\text{value} \quad // \ A<-value; \ M=\text{RAM}[A]
[\text{ADM}] = x \text{ op } y; \ \text{jump} \quad // x=D; \ y=A \ or \ M; \ if \ jump \ then \ \text{PC}<-A
```

Design a processor

- How to build a processor (Hack, this time)
  - Develop instruction set architecture (ISA)
    - 16-bit words, two types of machine instructions
  - Determine major components
    - ALU, registers, program counter, memory
  - Determine datapath requirements
    - Flow of bits

- Analyze how to implement each instruction
  - Determine settings of control signals
Hack architecture (data path)
Hack architecture (control)

A total of 13 control signals
Hack architecture (control)
Hack architecture (control)
Hack architecture (control)
Hack architecture (control)
Hack architecture (control)
Hack architecture (control)

- **Inputs:** instruction, zr, ng
  - instruction

- **Outputs:**
  - OP_ALU
  - MUX_A
  - MUX_ALU
  - W_A
  - W_D
  - writeM
  - W_PC

```
<table>
<thead>
<tr>
<th>I15</th>
<th>I14..0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>value</td>
</tr>
</tbody>
</table>
```

```
111A C1 C2 C3 C4 C5 C6 D1 D2 D3 J1 J2 J3
```
Hack architecture (trace @10 / D=M+1;JGE )
Hack architecture (CPU interface)
**Hack CPU**

**CPU internal components** (invisible in this chip diagram): ALU and 3 registers: A, D, PC

**CPU execute logic:**

The CPU executes the instruction according to the Hack language specification:

- The D and A values, if they appear in the instruction, are read from (or written to) the respective CPU-resident registers.
- If the instruction is @x, then x is stored in the A-register; and the emitted addressM is updated.
- The M value, if there is one in the instruction’s RHS, is read from inM.
- If the instruction’s LHS includes M, then the ALU output is placed in outM, the value of the CPU-resident A register is placed in addressM, and writeM is asserted.

A Hack machine language instruction like M=D+M, stated as a 16-bit value

Hack CPU

CPU internal components (invisible in this chip diagram): ALU and 3 registers: A, D, PC

CPU fetch logic:
Recall that:
1. the instruction may include a jump directive (expressed as non-zero jump bits)
2. the ALU emits two control bits, indicating if the ALU output is zero or less than zero

If \( \text{reset} = 0 \): the CPU uses this information (the jump bits and the ALU control bits) as follows:
   - If there should be a jump, the \( \text{PC} \) is set to the value of \( A \); else, \( \text{PC} \) is set to \( \text{PC} + 1 \)
   - The updated \( \text{PC} \) value is emitted by \( \text{pc} \).

If \( \text{reset} = 1 \): the \( \text{PC} \) is set to 0. \( \text{pc} \) emits 0. (restarting the computer)
Control (focus on the yellow chips only)

In the Hack architecture:

- ROM = instruction memory
- Program = sequence of 16-bit numbers, starting at ROM[0]
- Current instruction = ROM[PC]
- To select instruction n from the ROM, we set A to n, using the instruction @n
In the Hack architecture, the A register addresses both the RAM and the ROM simultaneously. Therefore:

- Command pairs like `@addr` followed by `D=M;someJumpDirective` make no sense.
- Best practice: in well-written Hack programs, a C-instruction should contain
  - either a reference to `M`, or
  - a jump directive, but not both.
The Hack computer (put together)

A 16-bit machine consisting of the following elements:

Both memory chips are 16-bit wide and have 15-bit address space.
The RAM used in Hack is different from a normal RAM. It also plays the role for I/O.

Programmers usually use high-level library for I/O, such as printf, drawline.

But, at low-level, we usually need to manipulate bits directly for I/O.
Displays

- CRT displays
  - resolution
  - refresh rate
keyboard
Using the memory:

- To record or recall values (e.g. variables, objects, arrays), use the first 16K words of the memory.
- To write to the screen (or read the screen), use the next 8K words of the memory.
- To read which key is currently pressed, use the next word of the memory.
Memory: physical implementation

The Memory chip is essentially a package that integrates the three chip-parts RAM16K, Screen, and Keyboard into a single, contiguous address space.

This packaging effects the programmer’s view of the memory, as well as the necessary I/O side-effects.

Access logic:

- Access to any address from 0 to 16,383 results in accessing the RAM16K chip-part
- Access to any address from 16,384 to 24,575 results in accessing the Screen chip-part
- Access to address 24,576 results in accessing the keyboard chip-part
- Access to any other address is invalid.
Data memory

**Low-level (hardware) read/write logic:**

To read $\text{RAM}[k]$: set address to $k$, probe out

To write $\text{RAM}[k]=x$: set address to $k$, set in to $x$, set load to 1, run the clock

**High-level (OS) read/write logic:**

To read $\text{RAM}[k]$: use the OS command $\text{out} = \text{peek}(k)$

To write $\text{RAM}[k]=x$: use the OS command $\text{poke}(k, x)$

peek and poke are OS commands whose implementation should effect the same behavior as the low-level commands

More about peek and poke this later in the course, when we’ll write the OS.
In the Hack platform, the screen is implemented as an 8K 16-bit RAM chip with a side effect of refreshing.

The Screen chip has a basic RAM chip functionality:

- **read logic:** \( \text{out} = \text{Screen}[\text{address}] \)
- **write logic:** if load then \( \text{Screen}[\text{address}] = \text{in} \)

**Side effect:**

Continuously refreshes a 256 by 512 black-and-white screen device

When loaded into the hardware simulator, the built-in Screen.hdl chip opens up a screen window; the simulator then refreshes this window from the screen memory map several times each second.
To set pixel \((row, col)\) black
1. \(word = \text{Screen}[32 \times row + \frac{col}{16}]\)  
   \(\text{(RAM}[16384 + 32 \times row + \frac{col}{16}]\))
2. Set the \((\text{col}\%16)\)-th bit of word to 1
3. Commit word to the RAM
   \(\text{(High-level: use the OS command drawPixel(row, col))}\)
keyboard

- A 16-bit register is used to keep the key stroke.

When a key is pressed on the keyboard, the key’s scan code appears in the keyboard memory map.
keyboard

- A 16-bit register is used to keep the key stroke.

When a key is pressed on the keyboard, the key's scan code appears in the keyboard memory map.
Keyboard

Keyboard chip: a single 16-bit register

Input: scan-code (16-bit value) of the currently pressed key, or 0 if no key is pressed

Output: same

Special keys:

<table>
<thead>
<tr>
<th>Key pressed</th>
<th>Keyboard output</th>
<th>Key pressed</th>
<th>Keyboard output</th>
</tr>
</thead>
<tbody>
<tr>
<td>newline</td>
<td>128</td>
<td>end</td>
<td>135</td>
</tr>
<tr>
<td>backspace</td>
<td>129</td>
<td>page up</td>
<td>136</td>
</tr>
<tr>
<td>left arrow</td>
<td>130</td>
<td>page down</td>
<td>137</td>
</tr>
<tr>
<td>up arrow</td>
<td>131</td>
<td>insert</td>
<td>138</td>
</tr>
<tr>
<td>right arrow</td>
<td>132</td>
<td>delete</td>
<td>139</td>
</tr>
<tr>
<td>down arrow</td>
<td>133</td>
<td>esc</td>
<td>140</td>
</tr>
<tr>
<td>home</td>
<td>134</td>
<td>f1-f12</td>
<td>141-152</td>
</tr>
</tbody>
</table>

How to read the keyboard:

- Low-level (hardware): probe the contents of the Keyboard chip
- High-level: use the OS command keyPressed()
  (effects the same operation, discussed later in the course, when we’ll write the OS).
## Some scan codes

<table>
<thead>
<tr>
<th>Key</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>48</td>
</tr>
<tr>
<td>1</td>
<td>49</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>9</td>
<td>57</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Key</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>(space)</td>
<td>32</td>
</tr>
<tr>
<td>!</td>
<td>33</td>
</tr>
<tr>
<td>“</td>
<td>34</td>
</tr>
<tr>
<td>#</td>
<td>35</td>
</tr>
<tr>
<td>$</td>
<td>36</td>
</tr>
<tr>
<td>%</td>
<td>37</td>
</tr>
<tr>
<td>&amp;</td>
<td>38</td>
</tr>
<tr>
<td>‘</td>
<td>39</td>
</tr>
<tr>
<td>(</td>
<td>40</td>
</tr>
<tr>
<td>)</td>
<td>41</td>
</tr>
<tr>
<td>*</td>
<td>42</td>
</tr>
<tr>
<td>+</td>
<td>43</td>
</tr>
<tr>
<td>,</td>
<td>44</td>
</tr>
<tr>
<td>-</td>
<td>45</td>
</tr>
<tr>
<td>.</td>
<td>46</td>
</tr>
<tr>
<td>/</td>
<td>47</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Key</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>new line</td>
<td>128</td>
</tr>
<tr>
<td>back space</td>
<td>129</td>
</tr>
<tr>
<td>left arrow</td>
<td>130</td>
</tr>
<tr>
<td>up arrow</td>
<td>131</td>
</tr>
<tr>
<td>right arrow</td>
<td>132</td>
</tr>
<tr>
<td>down arrow</td>
<td>133</td>
</tr>
<tr>
<td>home</td>
<td>134</td>
</tr>
<tr>
<td>end</td>
<td>135</td>
</tr>
<tr>
<td>Page up</td>
<td>136</td>
</tr>
<tr>
<td>Page down</td>
<td>137</td>
</tr>
<tr>
<td>insert</td>
<td>138</td>
</tr>
<tr>
<td>delete</td>
<td>139</td>
</tr>
<tr>
<td>esc</td>
<td>140</td>
</tr>
<tr>
<td>f1</td>
<td>141</td>
</tr>
<tr>
<td>f12</td>
<td>152</td>
</tr>
</tbody>
</table>

When no key is pressed, the resulting code is 0.
Keyboard memory map

- To check which key is currently pressed:
  - Probe the content of the Keyboard chip
  - In the Hack computer, probe the content of RAM[24576]
  - If the register contains 0, no key is pressed.
The Hack computer (put together)

A 16-bit machine consisting of the following elements:

Both memory chips are 16-bit wide and have 15-bit address space.
Assembly programming with I/O

Hack language convention:
- SCREEN: base address of the screen memory map, 16,384.
- KBD: address of the keyboard memory map, 24,576.
Example: draw a rectangle

- Draw a filled rectangle at the upper left corner of the screen, 16 pixels wide and RAM[0] pixels long. (demo)
Example: draw a rectangle (pseudo code)

```plaintext
// for (i=0; i<n; i++)
// draw 16 black pixels at the beginning of row i

addr = SCREEN
n = RAM[0]
i = 0

LOOP:
    if (i>n) goto END
    RAM[addr] = -1 // 1111 1111 1111 1111
    addr = addr+32 // advances to the next row
    i++;
    goto LOOP

END:
    goto END
```
Example: draw a rectangle (assembly)

```
addr = SCREEN
n = RAM[0]
i = 0

LOOP:
  if (i>n) goto END
  RAM[addr] = -1
  addr = addr+32
  i++;
  goto LOOP

END:
  goto END
```

@SCREEN
D=A
@addr
M=D // addr = SCREEN

@0
D=M
@n
M=D // n = RAM[0]

@i
M=0 // i=0
Example: draw a rectangle (assembly)

.addr = SCREEN
n = RAM[0]
i = 0

LOOP:
   if (i>n) goto END
   RAM[addr] = -1
   addr = addr+32
   i++;
   goto LOOP

END:
   goto END

(LOOP)
@i
D=M
@n
D=D-M
@END
D; JGT

@addr
A=M
M=-1
Example: draw a rectangle (assembly)

 addr = SCREEN
 n = RAM[0]
i = 0

 LOOP:
  if (i>n) goto END
  RAM[addr] = -1
  addr = addr+32
  i++;
goto LOOP
END:
  goto END
Example: draw a rectangle (assembly)

addr = SCREEN
n = RAM[0]
i = 0

LOOP:
    if (i>n) goto END
    RAM[addr] = -1
    addr = addr+32
    i++;
    goto LOOP

END:
    goto END

@32
D=A
@addr
M=D+M
    // addr = addr+32

@i
M=M+1
    // i++

@LOOP
0; JMP
    // goto LOOP

(END)

@END
0; JMP
Example: draw a rectangle (assembly)

```
@32
D=A
@addr
M=D+M    // addr = addr+32

@i
M=M+1    // i++

@LOOP
0; JMP   // goto LOOP

(END)

@END
0; JMP

addr = SCREEN
n = RAM[0]
i = 0

LOOP:
    if (i>n) goto END
    RAM[addr] = -1
    addr = addr+32
    i++;
    goto LOOP

END:
    goto END
```
Example: draw a rectangle (assembly)

```
@32
D=A
addr
M=D+M  // addr = addr+32
i
M=M+1  // i++
LOOP
  @i
  M=M+1  // i++
  @LOOP
  0; JMP  // goto LOOP
(END)
  @END
  0; JMP

addr = SCREEN
n = RAM[0]
i = 0

LOOP:
  if (i>n) goto END
  RAM[addr] = -1
  addr = addr+32
  i++;
  goto LOOP

END:
  goto END
```
Project #5: Computer-on-a-chip interface

Chip Name: Computer  // Topmost chip in the Hack platform
Input:          reset
Function: When reset is 0, the program stored in the computer's ROM executes. When reset is 1, the execution of the program restarts. Thus, to start a program's execution, reset must be pushed “up” (1) and “down” (0).

From this point onward the user is at the mercy of the software. In particular, depending on the program's code, the screen may show some output and the user may be able to interact with the computer via the keyboard.
Computer-on-a-chip implementation

CHIP Computer {
    IN reset;
    PARTS:
        // implementation missing
}

Implementation:
- You need to implement Memory and CPU first.
- Simple, the chip-parts do all the hard work.
Perspective: from here to a “real” computer

- Caching
- More I/O units
- Special-purpose processors (I/O, graphics, communications, ...)
- Multi-core / parallelism
- Efficiency
- Energy consumption considerations
- And more ...