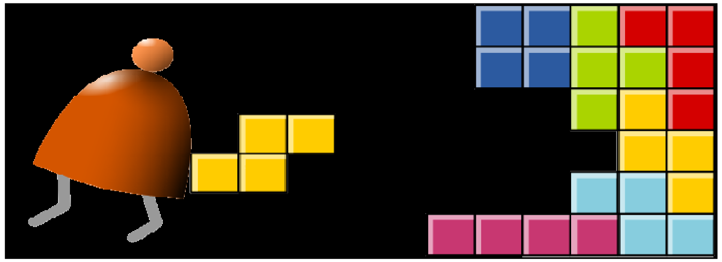
Boolean Logic



Building a Modern Computer From First Principles

www.nand2tetris.org

Boolean algebra

Some elementary Boolean functions:

- Not(x)
- And(x,y)
- Or(x,y)
- Nand(x,y)

x	Not(x)
0	1
1	0

x	y	And(x,y)
0	0	0
0	1	0
1	0	0
1	1	1

x	y	Or(x,y)
0	0	0
0	1	1
1	0	1
1	1	1

x	y	Nand(x,y)
0	0	1
0	1	1
1	0	1
1	1	0

Boolean functions:

x	y	z	$f(x,y,z)=(x+y)\bar{z}$
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	0

- A Boolean function can be expressed using a functional expression or a truth table expression
- Important observation:
  Every Boolean function can be expressed using And, Or, Not.

All Boolean functions of 2 variables

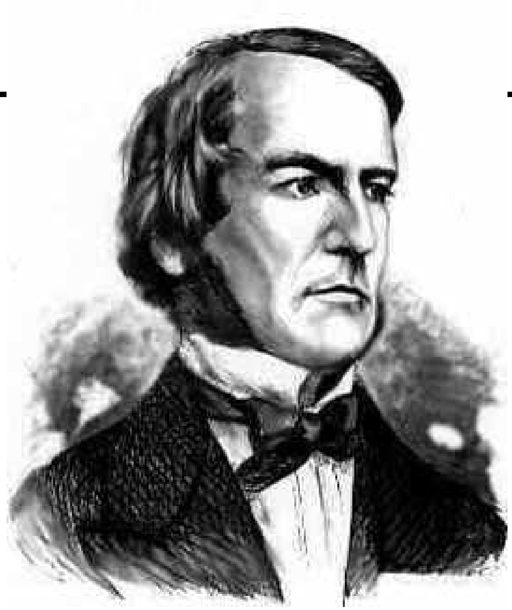
Function	x	0	0	1	1
	y	0	1	0	1
Constant 0	0	0	0	0	0
And	$x \cdot y$	0	0	0	1
x And Not y	$x \cdot \bar{y}$	0	0	1	0
x	x	0	0	1	1
Not x And y	$\bar{x} \cdot y$	0	1	0	0
y	y	0	1	0	1
Xor	$x \cdot \bar{y} + \bar{x} \cdot y$	0	1	1	0
Or	$x + y$	0	1	1	1
Nor	$\overline{x+y}$	1	0	0	0
Equivalence	$x \cdot y + \bar{x} \cdot \bar{y}$	1	0	0	1
Not y	$\bar{y}$	1	0	1	0
If y then x	$x + \bar{y}$	1	0	1	1
Not x	$\bar{x}$	1	1	0	0
If x then y	$\bar{x} + y$	1	1	0	1
Nand	$\overline{x \cdot y}$	1	1	1	0
Constant 1	1	1	1	1	1

Boolean algebra

<u>Given</u>: **`Nand(a,b), false`**

<u>We can build:</u>

- **`Not(a) = Nand(a,a)`**
- **`true = Not(false)`**
- **`And(a,b) = Not(Nand(a,b))`**
- **`Or(a,b) = Not(And(Not(a),Not(b)))`**
- **`Xor(a,b) = Or(And(a,Not(b)),And(Not(a),b)))`**
- Etc.



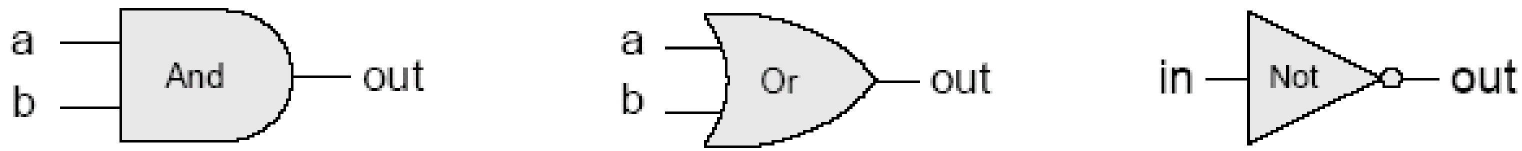
George Boole, 1815-1864

("*A Calculus of Logic*")

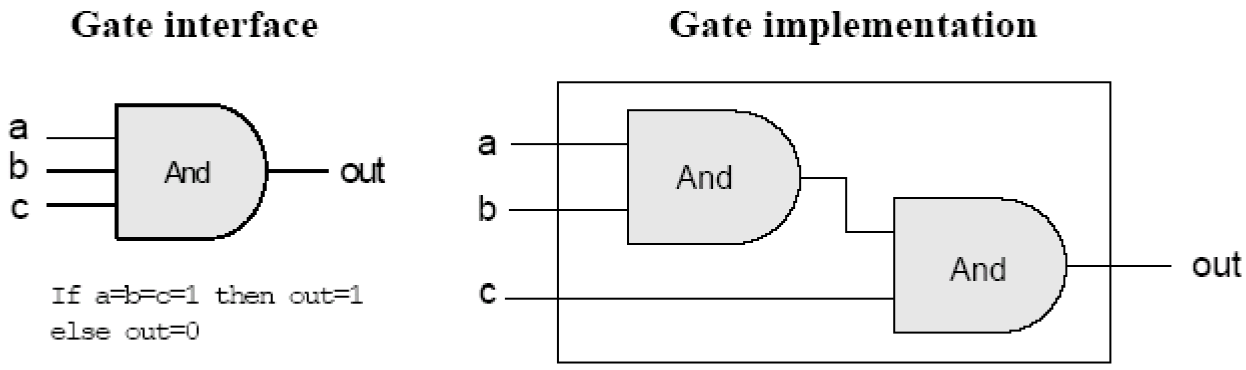
Gate logic

- Gate logic – a gate architecture designed to implement a Boolean function

- Elementary gates:

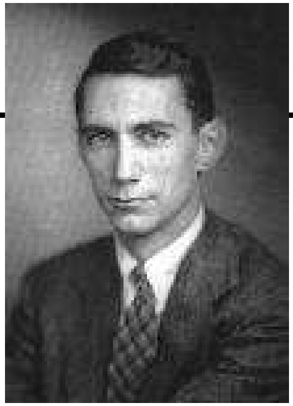


- Composite gates:



- Important distinction: Interface (*what*) VS implementation (*how*).

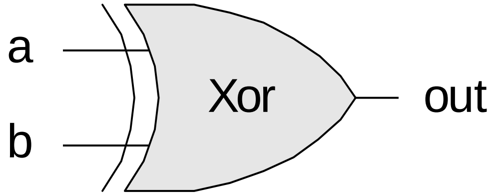
Gate logic



Claude Shannon, 1916-2001

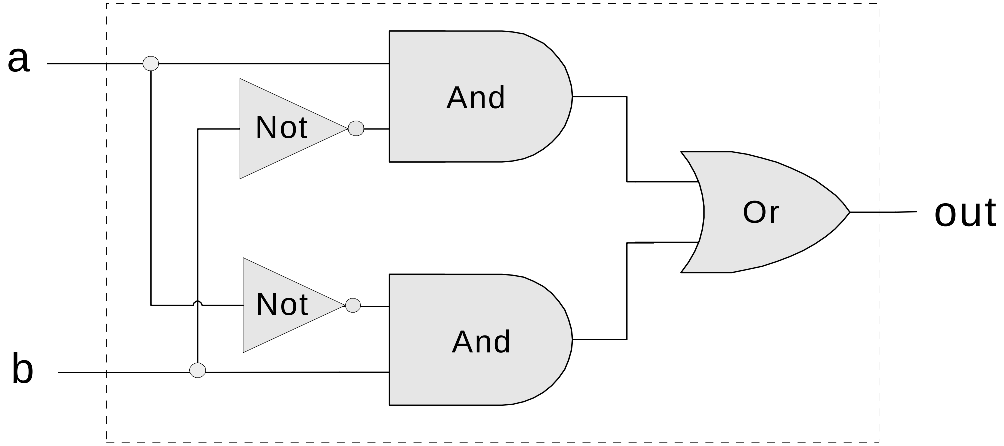
("Symbolic Analysis of Relay and Switching Circuits")

Interface



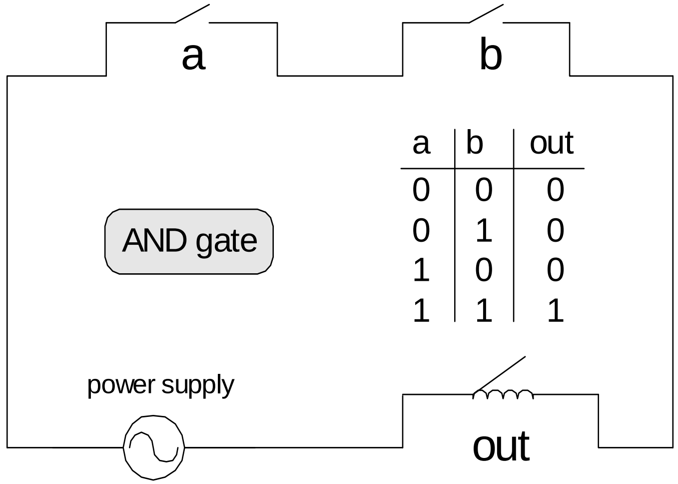
a	b	out
0	0	0
0	1	1
1	0	1
1	1	0

Implementation

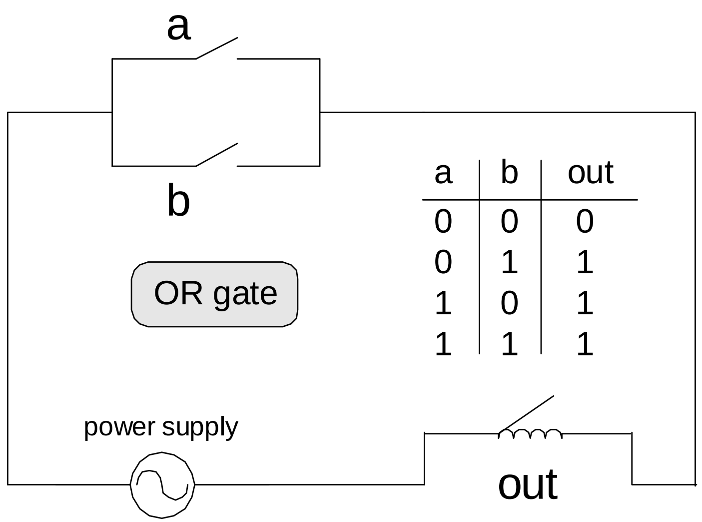


Xor(a,b) = Or(And(a,Not(b)),And(Not(a),b)))

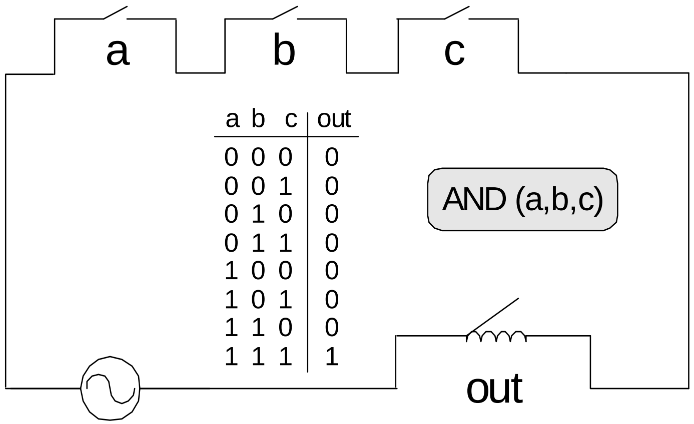
Circuit implementations



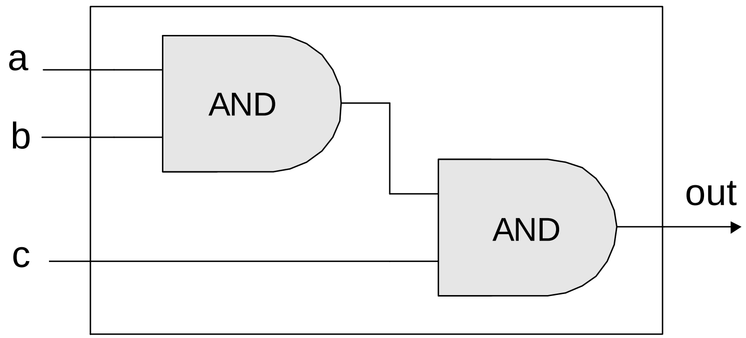
a	b	out
0	0	0
0	1	0
1	0	0
1	1	1



a	b	out
0	0	0
0	1	1
1	0	1
1	1	1



a	b	c	out
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1



- From a computer science perspective,
physical realizations of logic gates are irrelevant.

Project 1: elementary logic gates

Given: `Nand(a,b), false`

a	b	Nand(a,b)
0	0	1
0	1	1
1	0	1
1	1	0

Build:

- `Not(a) = ...`
- `true = ...`
- `And(a,b) = ...`
- `Or(a,b) = ...`
- `Mux(a,b,sel) = ...`
- Etc. - 12 gates altogether.

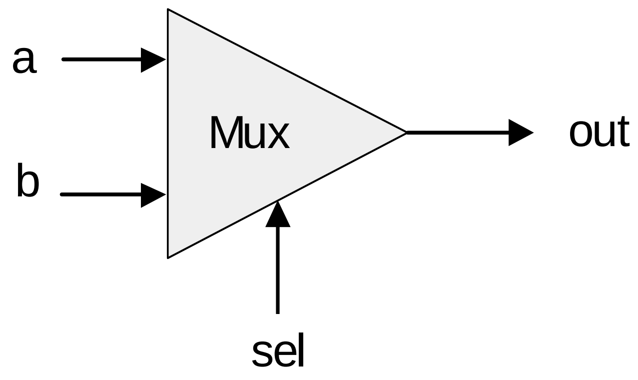
Q: Why these particular 12 gates?

A: Since …

- They are commonly used gates
- They provide all the basic building blocks needed to build our computer.

Multiplexer

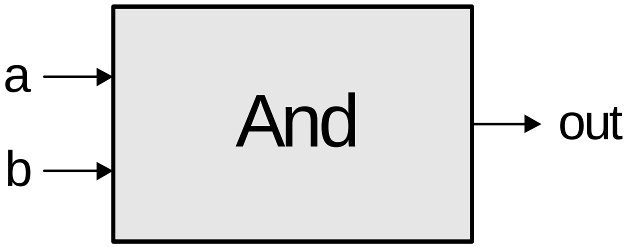
a	b	sel	out
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	1



sel	out
0	a
1	b

<u>Proposed Implementation:</u> based on Not, And, Or gates.

Example: Building an **And** gate



And.cmp

a	b	out
0	0	0
0	1	0
1	0	0
1	1	1

Contract:

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

And.hdl

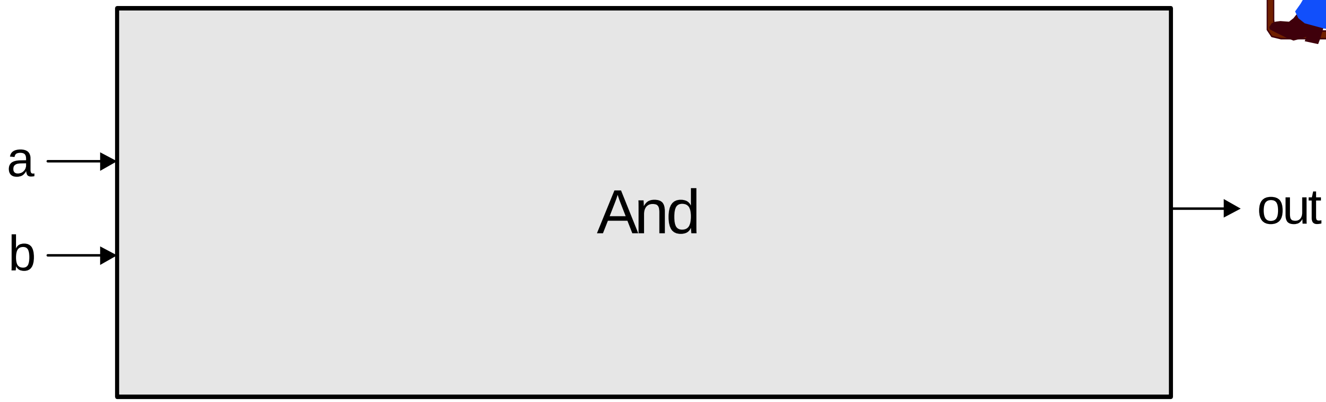
```
CHIP And
{   IN  a, b;
    OUT out;
    // implementation missing
}
```

And.tst

```
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: And(a,b) = 1 exactly when a=b=1



And.hdl

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

Building an **And** gate

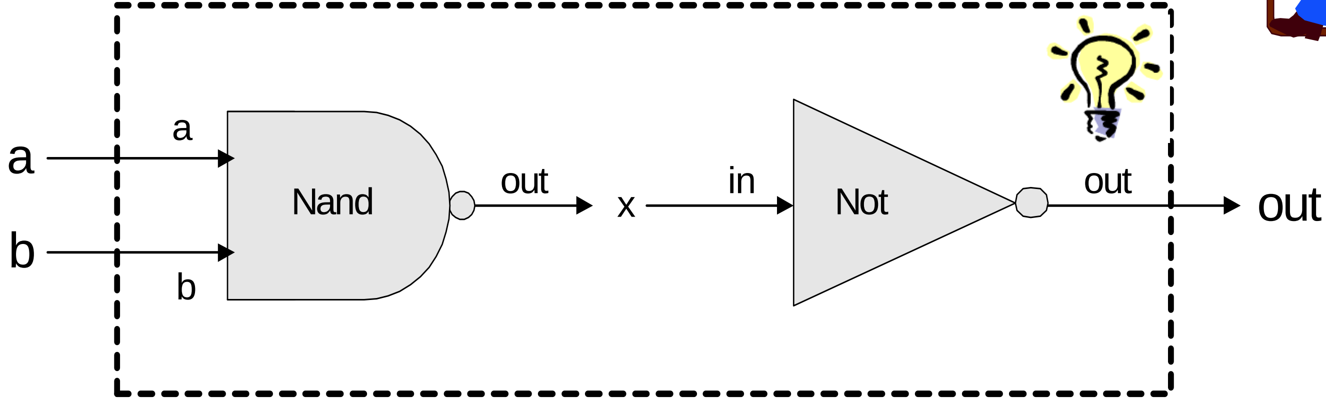
Implementation: And(a,b) = Not(Nand(a,b))

And.hdl

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

Building an **And** gate

Implementation: And(a,b) = Not(Nand(a,b))

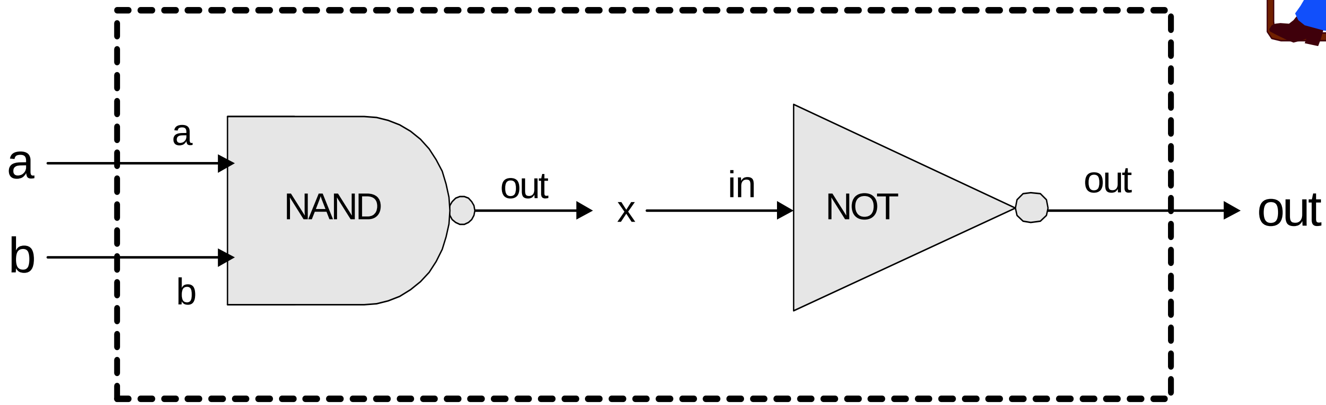


And.hdl

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

Building an **And** gate

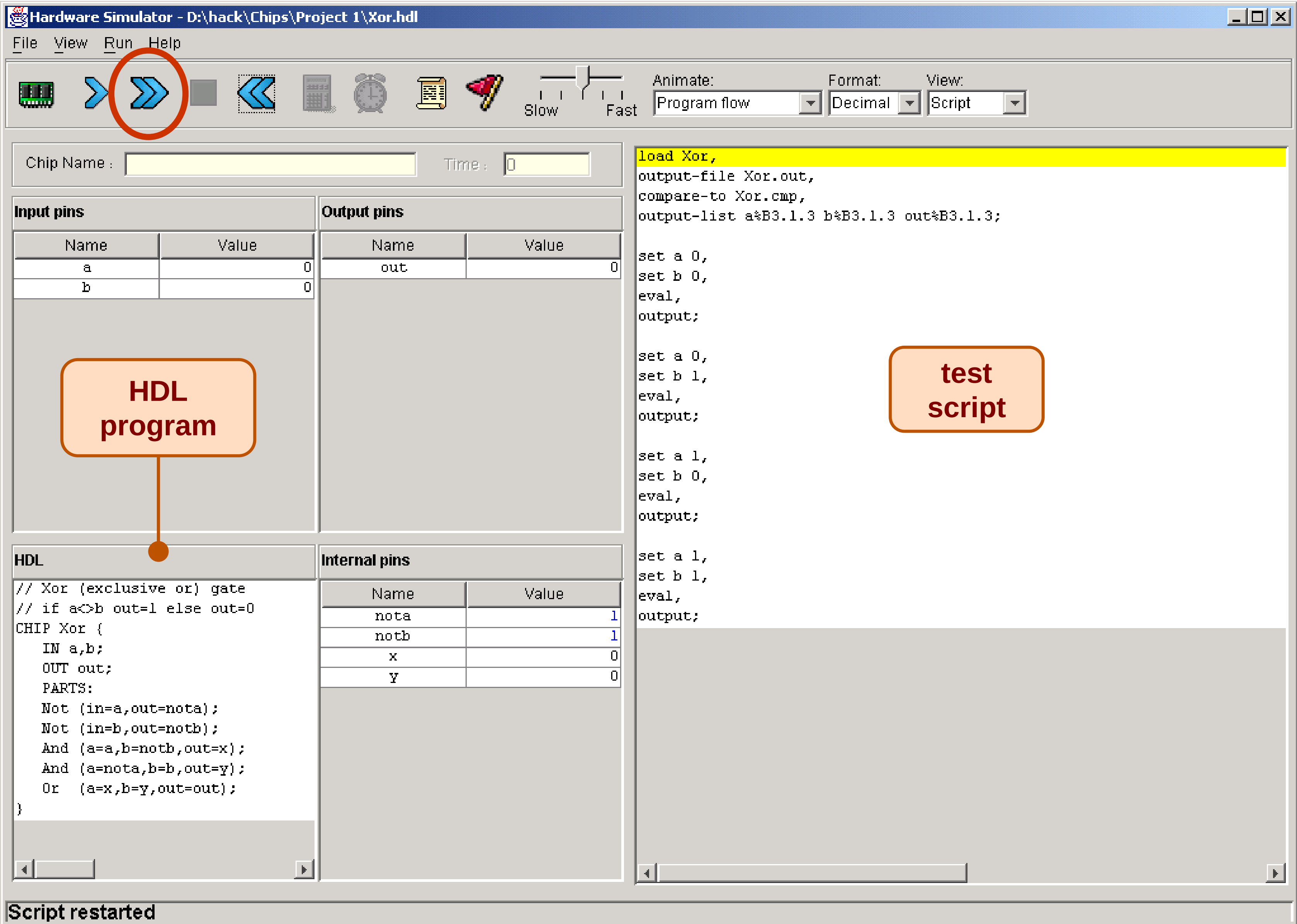
Implementation: And(a,b) = Not(Nand(a,b))



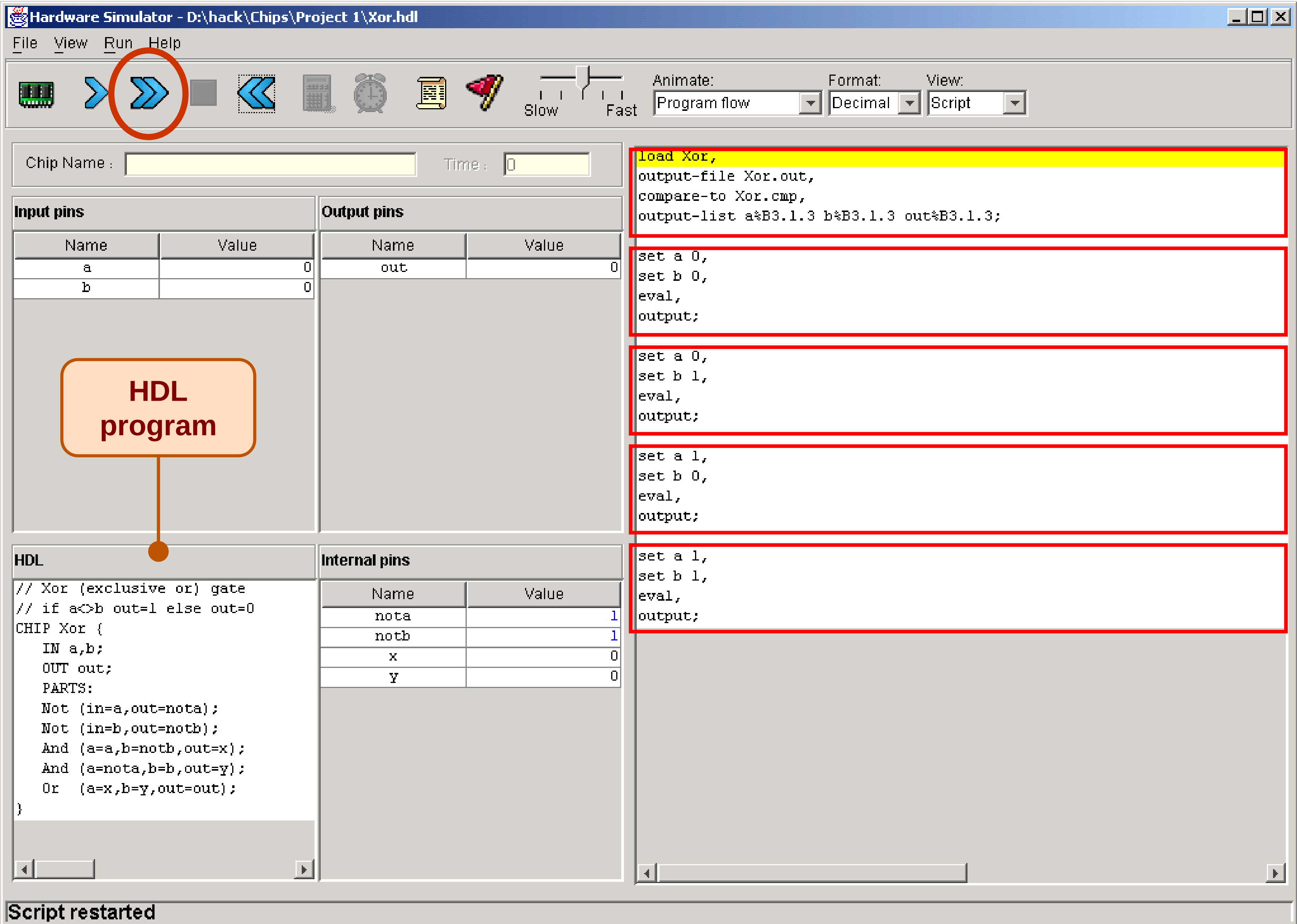
And.hdl

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

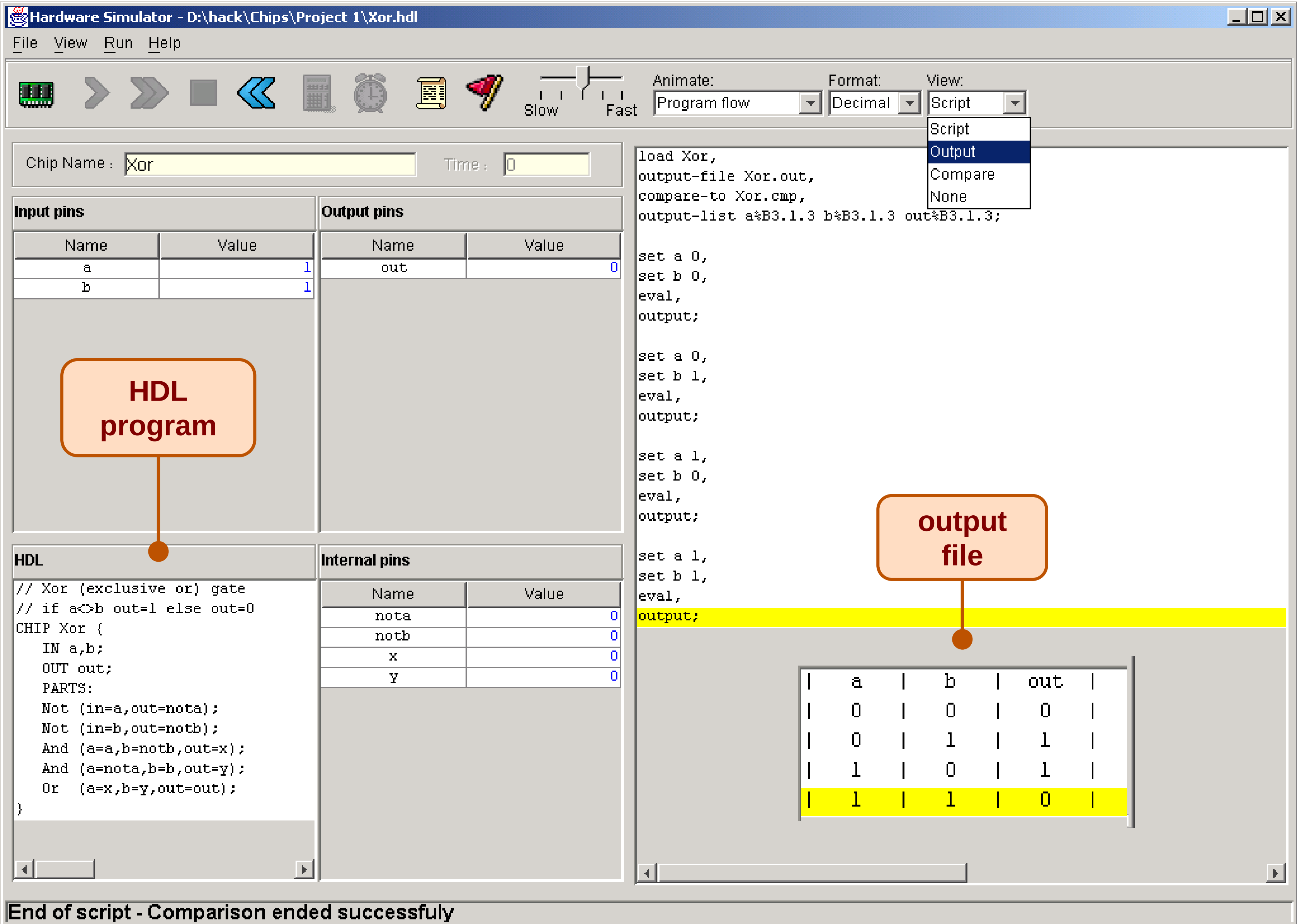
Hardware simulator (demonstrating Xor gate construction)



Hardware simulator



Hardware simulator



Project materials: www.nand2tetris.org

From NAND to Tetris

Building a Modern Computer From First Principles

- Home
- Projects
- Book
- Software
- Media
- Cool Stuff
- Terms
- Q&A
- About

Project 1: Logic Gates

Project 1 web site

Background

A typical computer architecture is based on a set of elementary logic gates like `And`, `Or`, etc., as well as their bit-wise versions `And16`, `Or16`, etc. (in a 16-bit machine). This project engages you in the construction of a typical set of elementary gates. These gates form the elementary building blocks from which more complex chips will be later constructed.

Objective

Build all the logic gates described in Chapter 1 (see list below), yielding a basic chip-set. The only building blocks that you can use in this project are primitive `Nand` gates and the composite gates that you will gradually build on top of them.

Chips

Chip (HDL)	Function	Test Script	Compare File
Nand	Nand gate (primitive)		
Not	Not gate	Not.tst	Not.cmp
And	And gate	And.tst	And.cmp
Or	Or gate	Or.tst	Or.cmp
Xor	Xor gate	Xor.tst	Xor.cmp
Mux	Mux gate	Mux.tst	Mux.cmp
DMux	DMux gate	DMux.tst	DMux.cmp
Not16	16-bit Not	Not16.tst	Not16.cmp

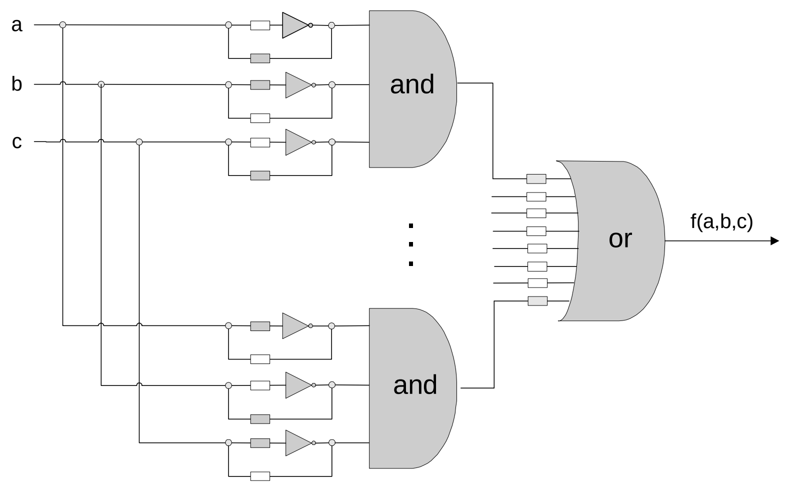
`And.hdl`,
`And.tst`,
`And.cmp` files

Project 1 tips

- Read the Introduction + Chapter 1 of the book
- Download the book's software suite
- Go through the hardware simulator tutorial
- Do Project 0 (optional)
- You're in business.

Perspective

- Each Boolean function has a canonical representation
- The canonical representation is expressed in terms of And, Not, Or
- And, Not, Or can be expressed in terms of Nand alone
- Ergo, every Boolean function can be realized by a standard PLD consisting of Nand gates only
- Mass production
- Universal building blocks, unique topology
- Gates, neurons, atoms, …



End notes: Canonical representation

Whodunit story: Each suspect may or may not have an alibi (*a*), a motivation to commit the crime (*m*), and a relationship to the weapon found in the scene of the crime (*w*). The police decides to focus attention only on suspects for whom the proposition Not(*a*) And (*m* Or *w*) is true.

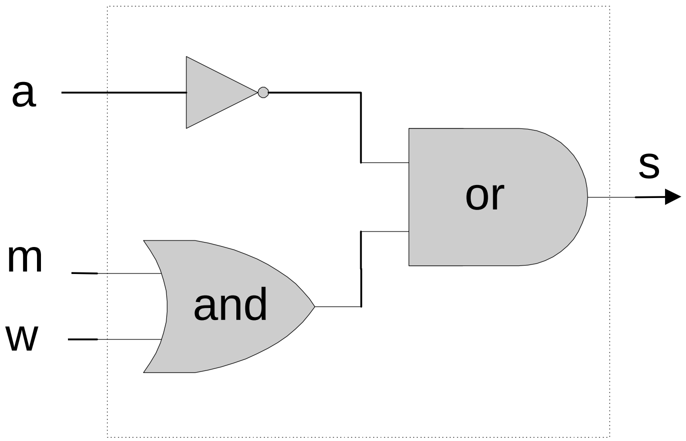
Truth table of the "suspect" function $s(a,m,w) = \bar{a} \cdot (m + w)$

a	m	w	*minterm*	suspect(a,m,w)= not(a) and (m or w)
0	0	0	$m_0 = \bar{a}\bar{m}\bar{w}$	0
0	0	1	$m_1 = \bar{a}\bar{m}w$	1
0	1	0	$m_2 = \bar{a}m\bar{w}$	1
0	1	1	$m_3 = \bar{a}mw$	1
1	0	0	$m_4 = a\bar{m}\bar{w}$	0
1	0	1	$m_5 = a\bar{m}w$	0
1	1	0	$m_6 = am\bar{w}$	0
1	1	1	$m_7 = amw$	0

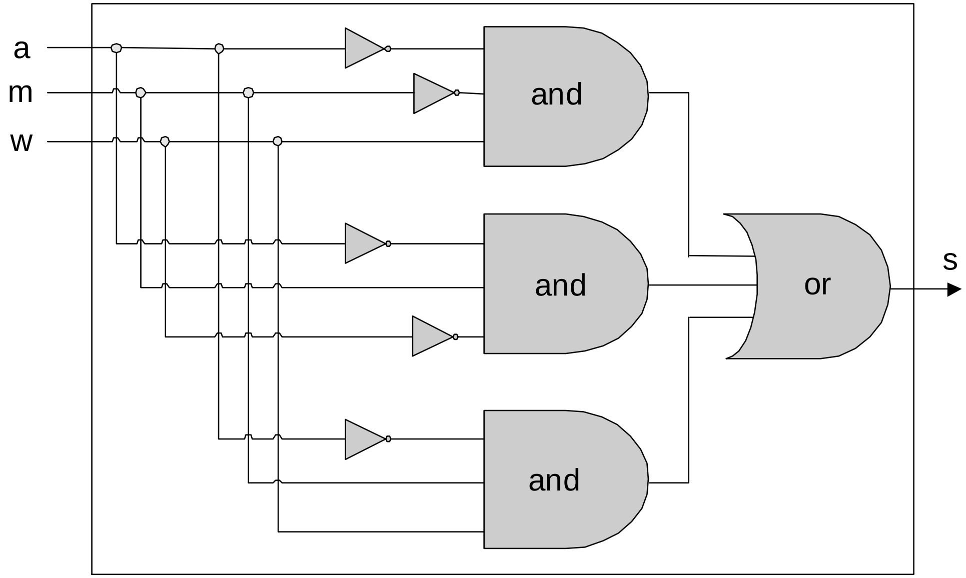
Canonical form: $s(a,m,w) = \bar{a}\,\bar{m}\,w + \bar{a}\,m\,\bar{w} + \bar{a}\,m\,w$

End notes: Canonical representation (cont.)

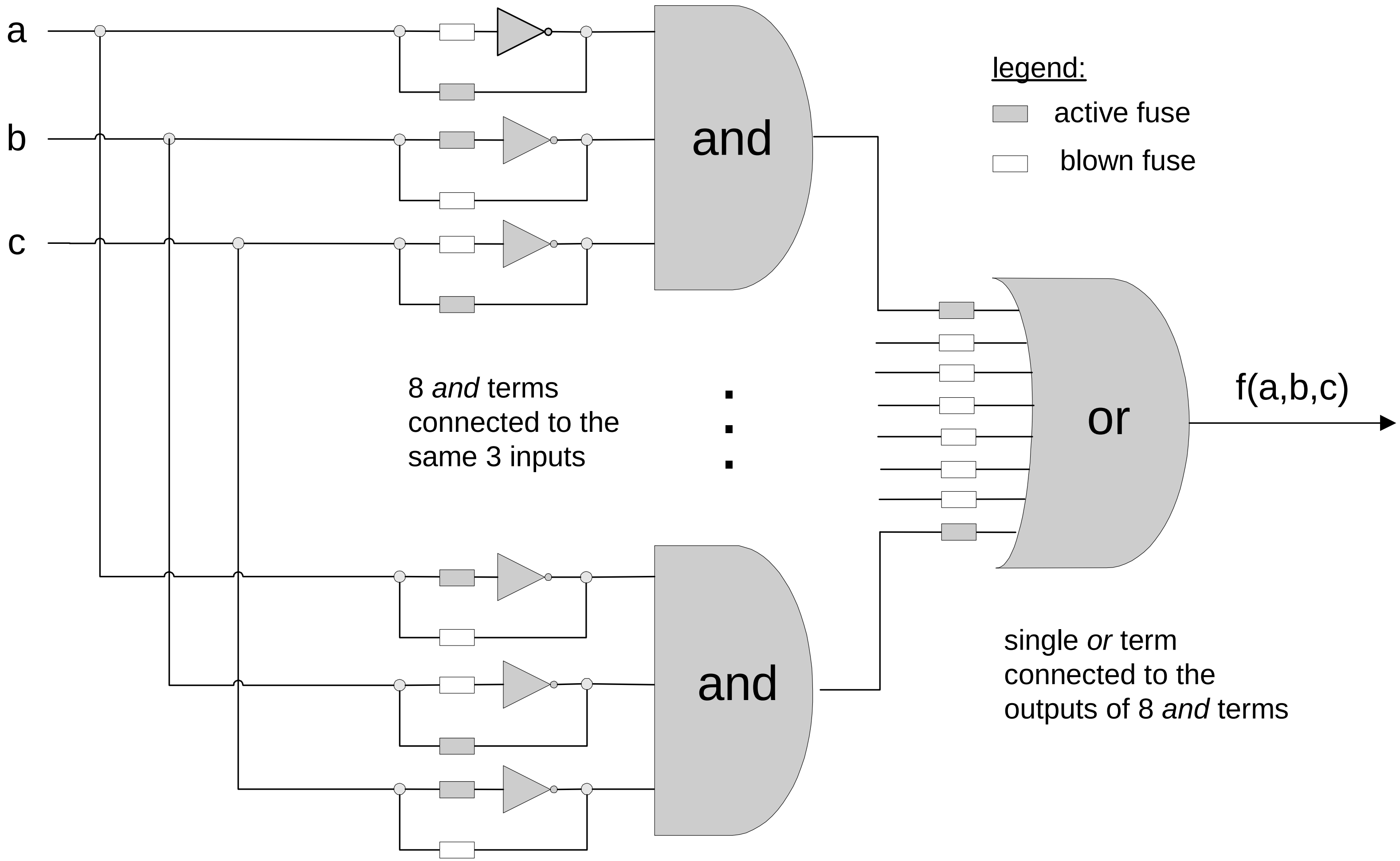
$$s(a,m,w) = \bar{a} \cdot (m + w)$$



$$s(a,m,w) = \bar{a}\,\bar{m}\,w + \bar{a}\,m\,\bar{w} + \bar{a}\,m\,w$$



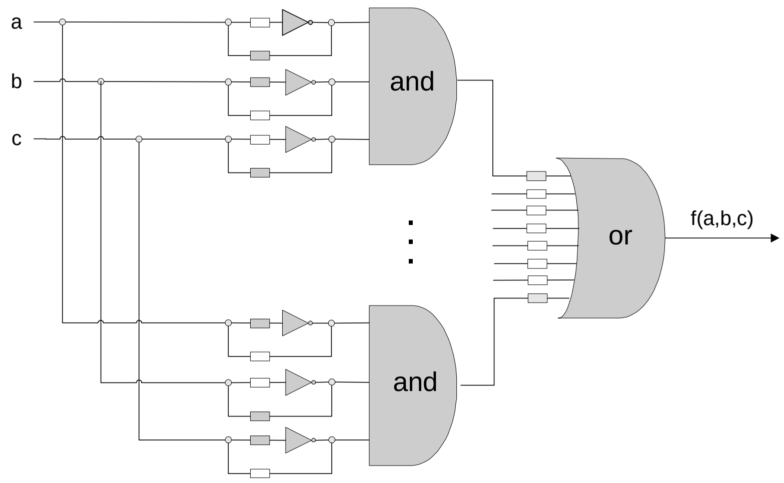
End notes: Programmable Logic Device for 3-way functions

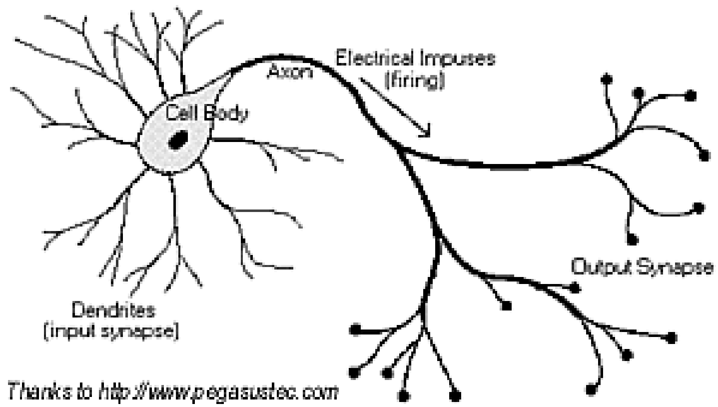


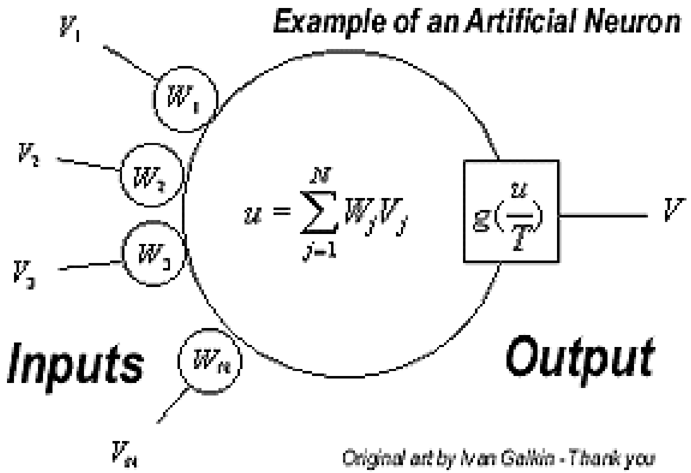
PLD implementation of $f(a,b,c) = a\,\bar{b}\,c + \bar{a}\,b\,\bar{c}$

(the on/off states of the fuses determine which gates participate in the computation)

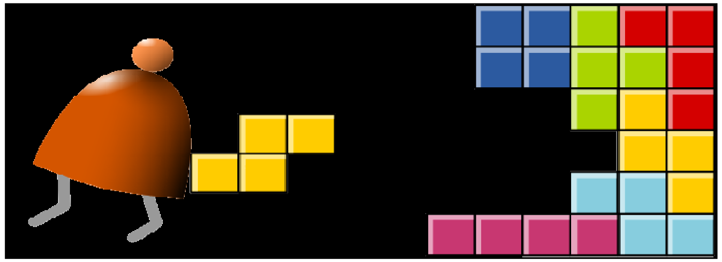
End notes: universal building blocks, unique topology







Boolean Arithmetic



Building a Modern Computer From First Principles

www.nand2tetris.org

Counting systems

quantity	decimal	binary	3-bit register
	0	0	000
*	1	1	001
**	2	10	010
***	3	11	011
****	4	100	100
*****	5	101	101
******	6	110	110
*******	7	111	111
********	8	1000	overflow
*********	9	1001	overflow
**********	10	1010	overflow

Rationale

$$(9038)_{ten} = 9\cdot 10^3 + 0\cdot 10^2 + 3\cdot 10^1 + 8\cdot 10^0 = 9038$$

$$(10011)_{two} = 1\cdot 2^4 + 0\cdot 2^3 + 0\cdot 2^2 + 1\cdot 2^1 + 1\cdot 2^0 = 19$$

$$(x_n x_{n-1} \ldots x_0)_b = \sum_{i=0}^{n} x_i \cdot b^i$$

Binary addition

Assuming a 4-bit system:

0	0	0	1		
	1	0	0	1	+
	0	1	0	1	
0	1	1	1	0	

no overflow

1	1	1	1		
	1	0	1	1	+
	0	1	1	1	
1	0	0	1	0	

overflow

- Algorithm: exactly the same as in decimal addition
- Overflow (MSB carry) has to be dealt with.

Representing negative numbers (4-bit system)

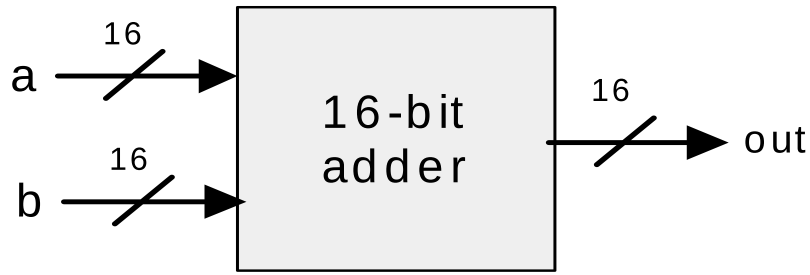
0	0000		
1	0001	1111	-1
2	0010	1110	-2
3	0011	1101	-3
4	0100	1100	-4
5	0101	1011	-5
6	0110	1010	-6
7	0111	1001	-7
		1000	-8

- The codes of all positive numbers begin with a "0"
- The codes of all negative numbers begin with a "1"
- To convert a number: leave all trailing 0's and first 1 intact, and flip all the remaining bits

<u>Example:</u> 2 - 5 = 2 + (-5) =

$$\begin{array}{r} 0\,0\,1\,0 \\ +\,1\,0\,1\,1 \\ \hline 1\,1\,0\,1 \end{array} = -3$$

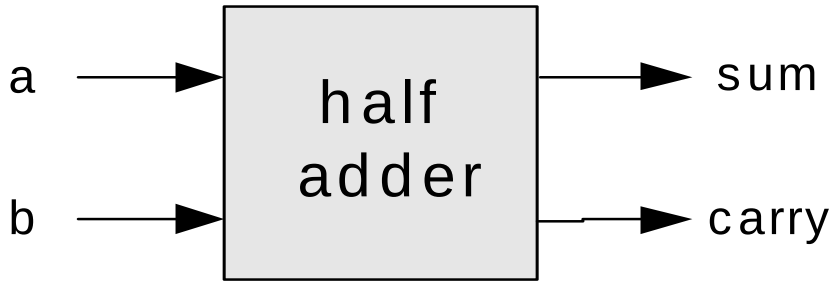
Building an Adder chip



- Adder: a chip designed to add two integers
- Proposed implementation:
  - Half adder: designed to add 2 bits
  - Full adder: designed to add 3 bits
  - Adder: designed to add two *n*-bit numbers.

Half adder (designed to add 2 bits)

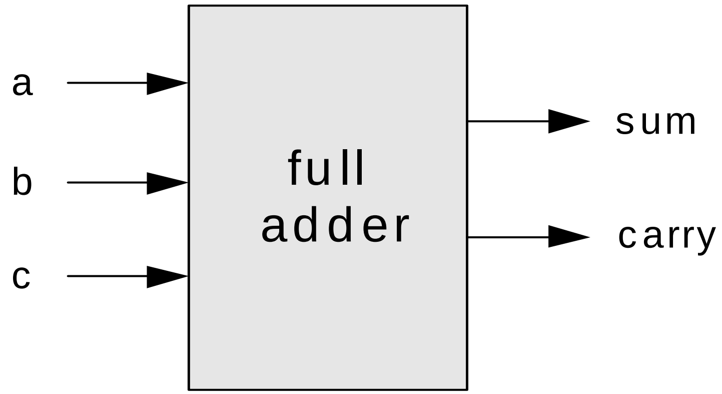
a	b	sum	carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1



<u>Implementation:</u> based on two gates that you've seen before.

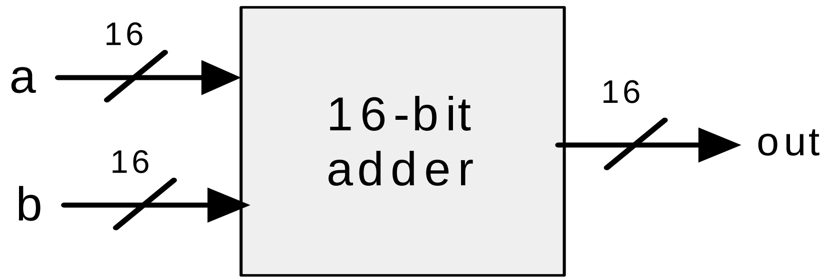
Full adder (designed to add 3 bits)

a	b	c	sum	carry
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1



Implementation: can be based on half-adder gates.

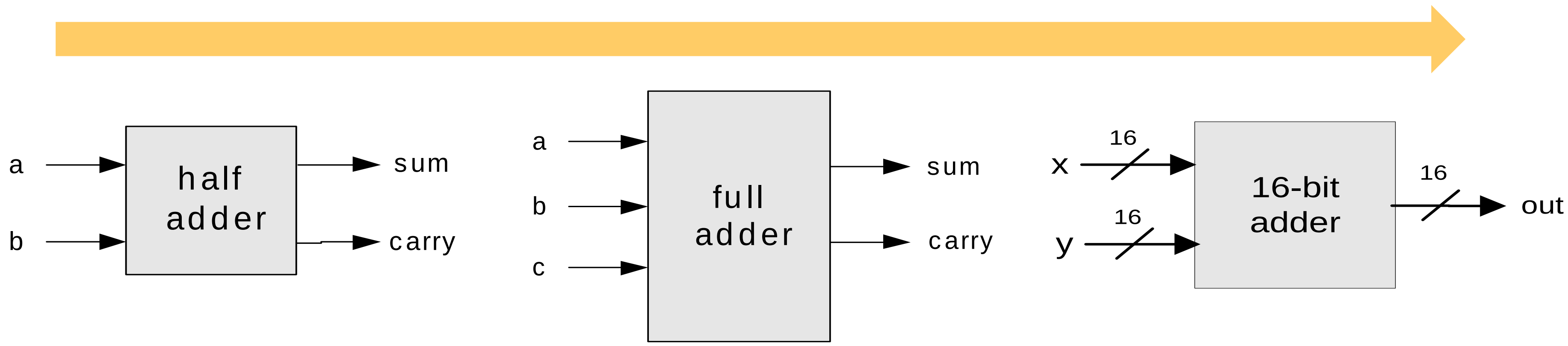
n-bit Adder (designed to add two 16-bit numbers)

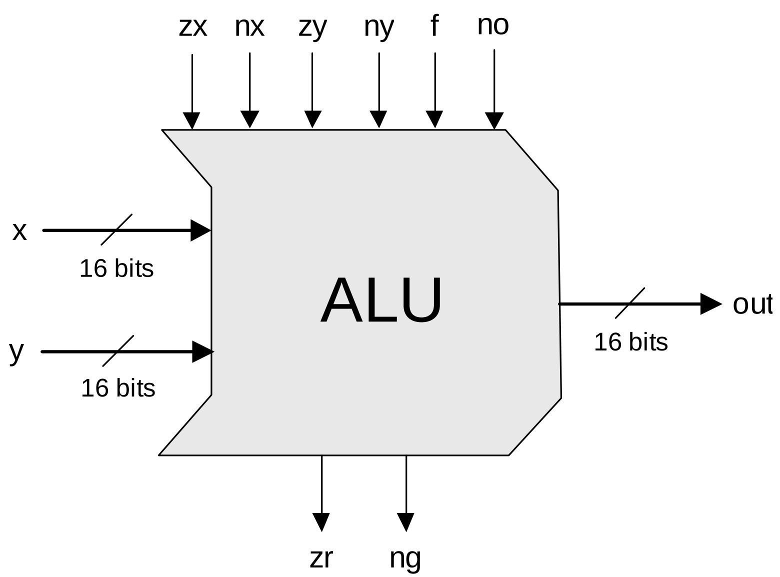


...	1	0	1	1	a	
						+
...	0	0	1	0	b	
...	1	1	0	1	out	

<u>Implementation:</u> array of full-adder gates.

The ALU (of the Hack platform)





out(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,

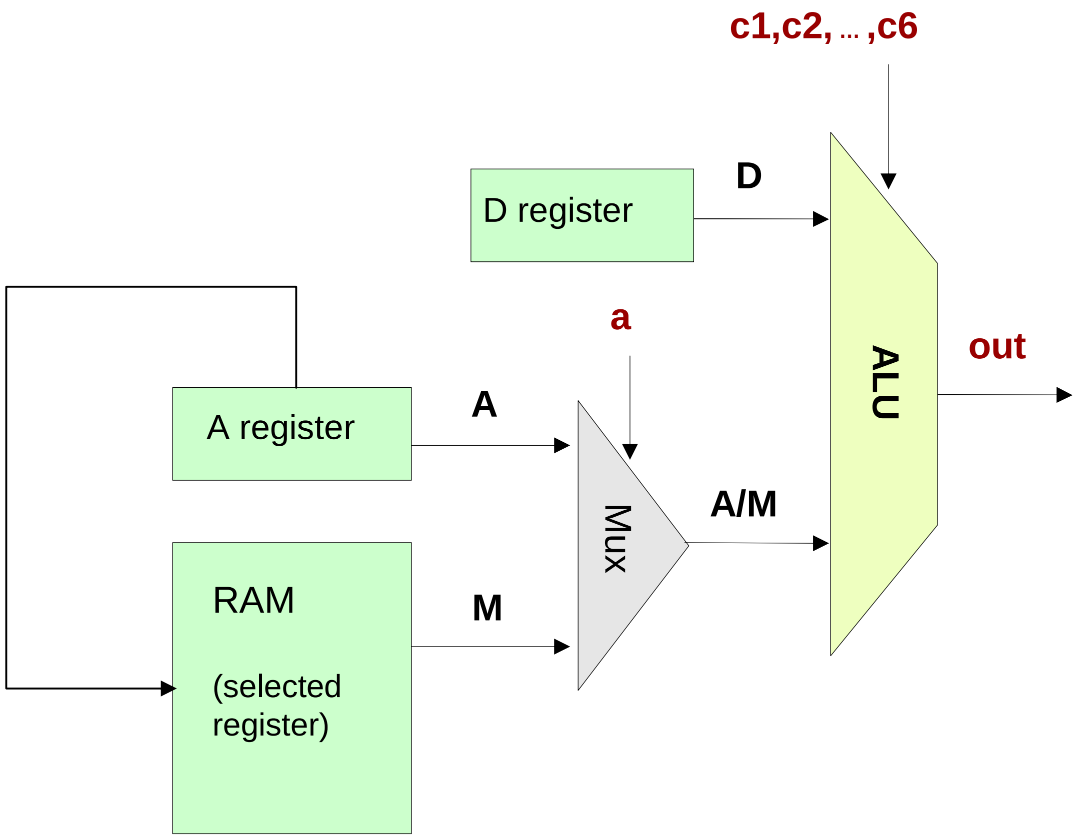
x&y, x|y

ALU logic (Hack platform)

These bits instruct how to pre-set the x input		These bits instruct how to pre-set the y input		This bit selects between + / And	This bit inst. how to post-set out	Resulting ALU output
zx	nx	zy	ny	f	no	out=
if zx then x=0	if nx then x=!x	if zy then y=0	if ny then y=!y	if f then out=x+y else out=x And y	if no then out=!out	f(x,y)=
1	0	1	0	1	0	0
1	1	1	1	1	1	1
1	1	1	0	1	0	-1
0	0	1	1	0	0	x
1						y
0						!x
1						!y
0						-x
1						-y
0	1	1	1	1	1	x+1
1	1	0	1	1	1	y+1
0	0	1	1	1	0	x-1
1	1	0	0	1	0	y-1
0	0	0	0	1	0	x+y
0	1	0	0	1	1	x-y
0	0	0	1	1	1	y-x
0	0	0	0	0	0	x&y
0	1	0	1	0	1	x|y

Implementation: build a logic gate architecture that "executes" the control bit "instructions": if zx==1 then set x to 0 (bit-wise), etc.

The ALU in the CPU context (a sneak preview of the Hack platform)

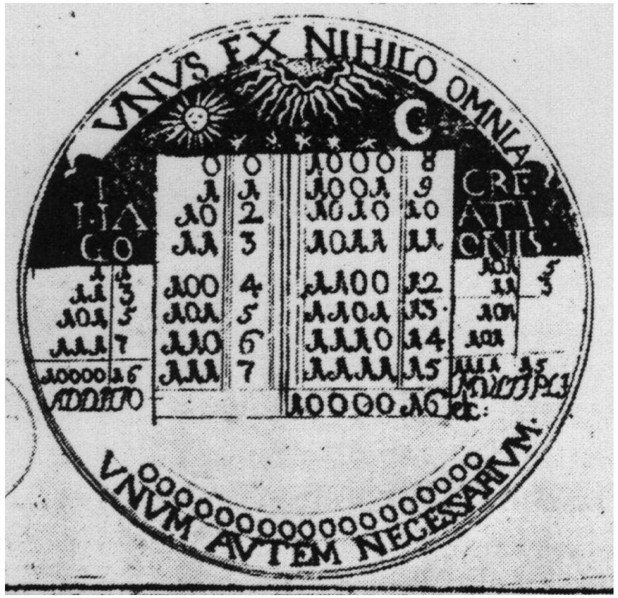


Perspective

- Combinational logic
- Our adder design is very basic: no parallelism
- It pays to optimize adders
- Our ALU is also very basic: no multiplication, no division
- Where is the seat of more advanced math operations?
  a typical hardware/software tradeoff.

Historical end-note: Leibnitz (1646-1716)

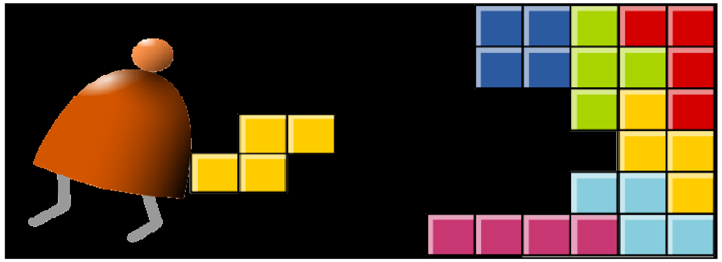
- "The binary system may be used in place of the decimal system; express all numbers by unity and by nothing"
- 1679: built a mechanical calculator (+, -, *, /)



Leibniz's medallion for the Duke of Brunswick

- CHALLENGE: "All who are occupied with the reading or writing of scientific literature have assuredly very often felt the want of a common scientific language, and regretted the great loss of time and trouble caused by the multiplicity of languages employed in scientific literature:
- SOLUTION: *"Characteristica Universalis"*: a universal, formal, and decidable language of reasoning
- The dream's end: Turing and Gödel in 1930's.

Sequential Logic



Building a Modern Computer From First Principles

www.nand2tetris.org

Sequential VS combinational logic

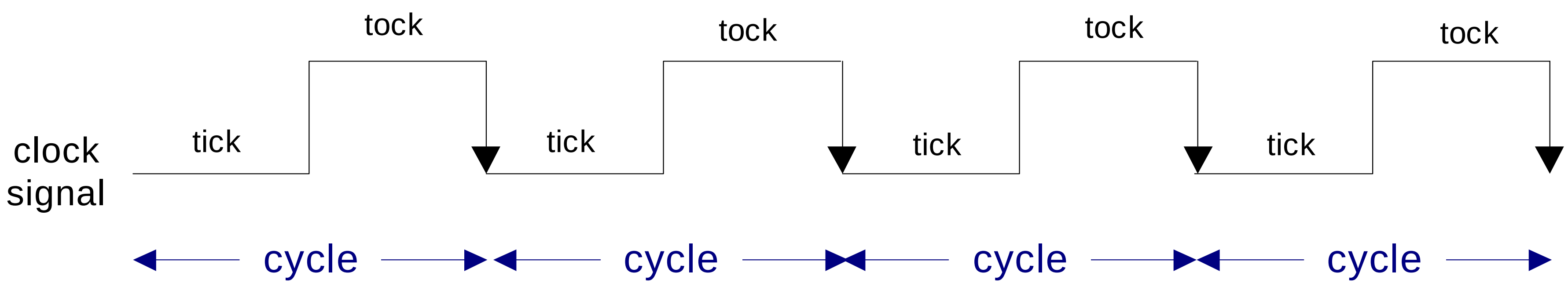
- Combinational devices: operate on data only; provide calculation services (e.g. Nand … ALU)
- Sequential devices: operate on data and a clock signal; as such, can be made to be *state*-aware and provide storage and synchronization services
- Sequential devices are sometimes called "clocked devices"
- The low-level behavior of clocked / sequential gates is tricky
- The good news:
  - All sequential chips can be based on one low-level sequential gate, called "data flip flop", or DFF
  - The clock-dependency details can be encapsulated at the low-level DFF level
  - Higher-level sequential chips can be built on top of DFF gates using combinational logic only.

Lecture plan

- Clock
- A hierarchy of memory chips:
  - Flip-flop gates
  - Binary cells
  - Registers
  - RAM
- Counters
- Perspective.

The Clock

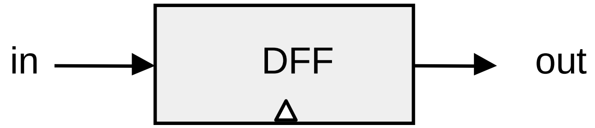
HW simulator demo



- In our jargon, a clock cycle = *tick*-phase (low), followed by a *tock*-phase (high)
- In real hardware, the clock is implemented by an oscillator
- In our hardware simulator, clock cycles can be simulated either
  - Manually, by the user, or
  - "Automatically," by a test script.

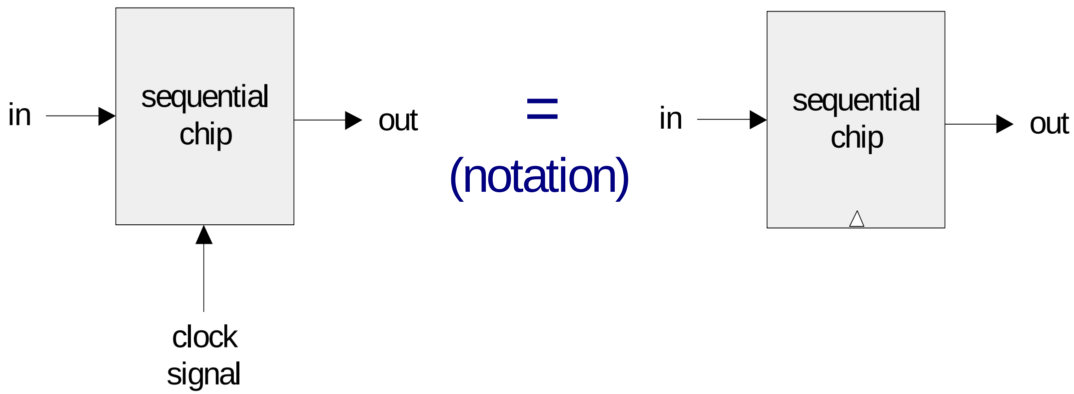
Flip-flop

HW simulator demo



out(t) = in(t-1)

- A fundamental state-keeping device
- For now, let us not worry about the DFF *implementation*
- Memory devices are made from numerous flip-flops, all regulated by the same master clock signal
- Notational convention:

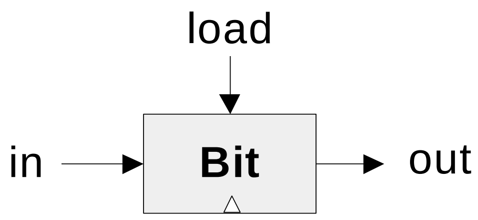


1-bit register (we call it "Bit")

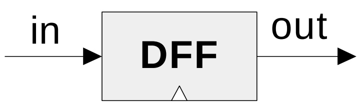
Objective: build a storage unit that can:

(a) Change its state to a given input

(b) Maintain its state over time (until changed)

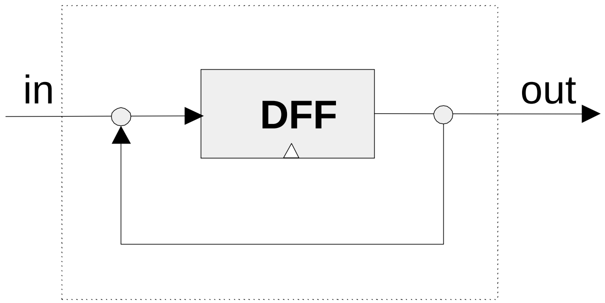


if load(t-1) then out(t)=in(t-1)
else out(t)=out(t-1)



out(t) = in(t-1)

Basic building block



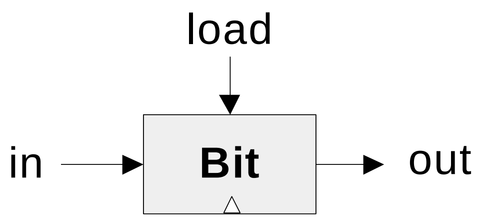
out(t) = out(t-1) ?
out(t) = in(t-1) ?

Won't work

Bit register (cont.)

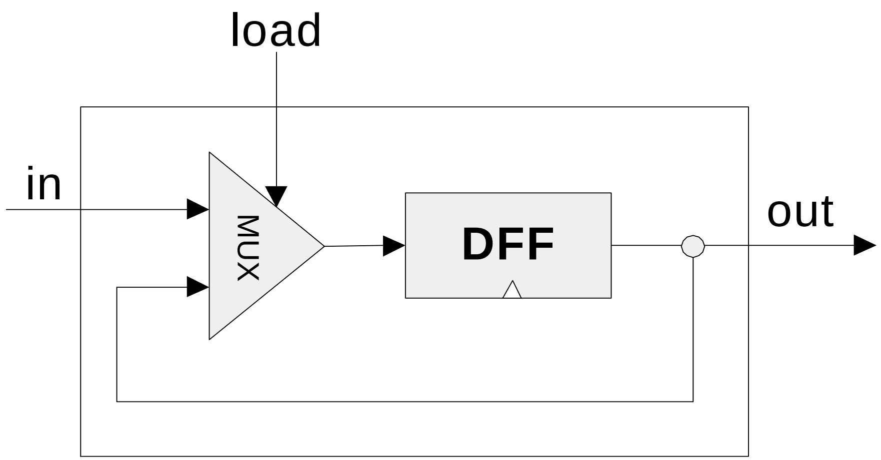
HW simulator demo

Interface



if load(t-1) then out(t)=in(t-1)
else out(t)=out(t-1)

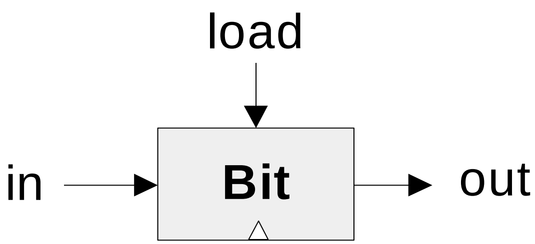
Implementation



- Load bit
- Read logic
- Write logic

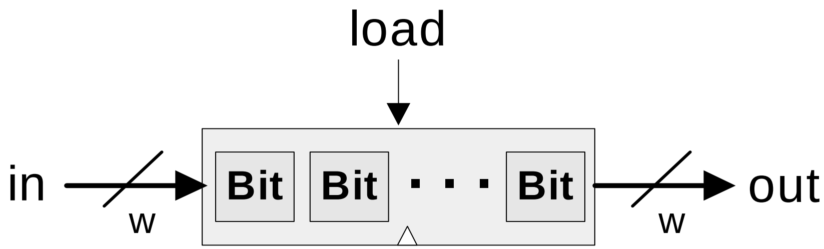
Multi-bit register

HW simulator demo



if load(t-1) then out(t)=in(t-1)
else out(t)=out(t-1)

1-bit register



if load(t-1) then out(t)=in(t-1)
else out(t)=out(t-1)

w-bit register

- Register's width: a trivial parameter
- Read logic
- Write logic

Aside: Hardware Simulation

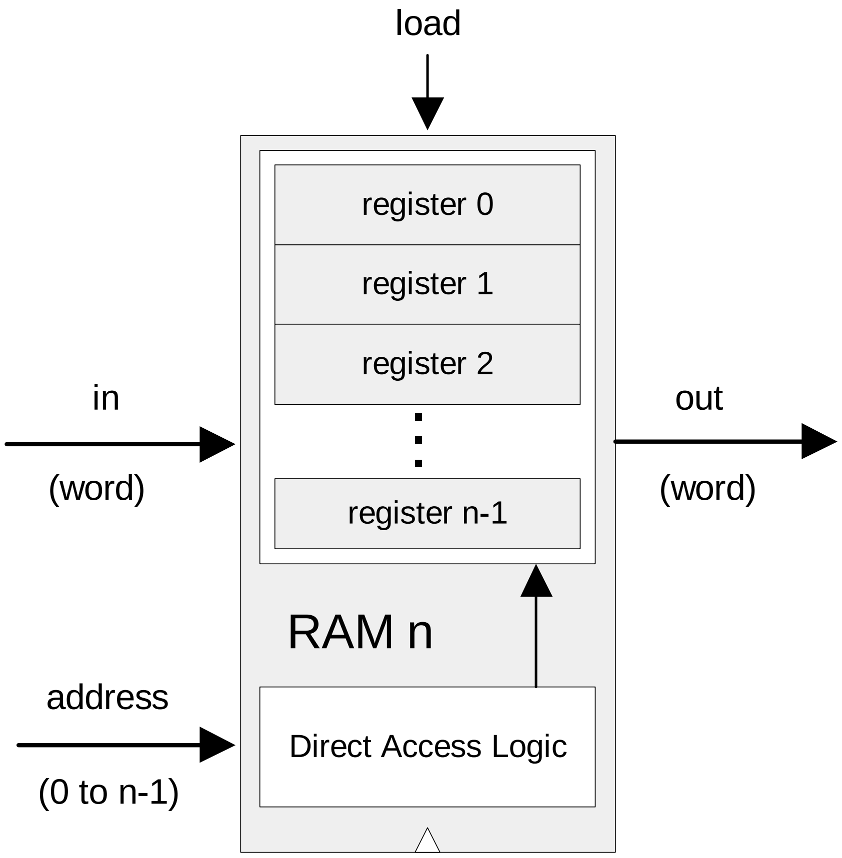
HW simulator demo

Relevant topics from the HW simulator tutorial:

- <u>Clocked chips:</u> When a clocked chip is loaded into the simulator, the clock icon is enabled, allowing clock control
- <u>Built-in chips:</u>
  - feature a standard HDL interface yet a Java implementation
  - Provide behavioral simulation services
  - May feature GUI effects (at the simulator level only).

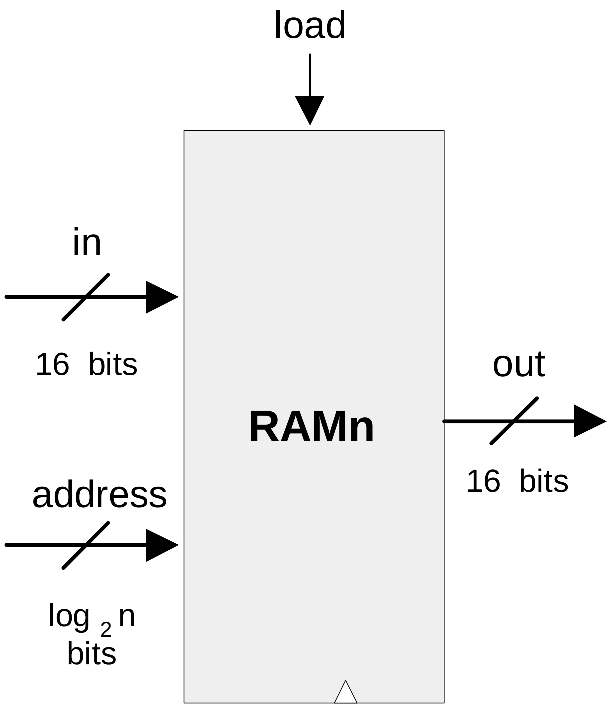
Random Access Memory (RAM)

HW simulator demo



- Read logic
- Write logic.

RAM interface

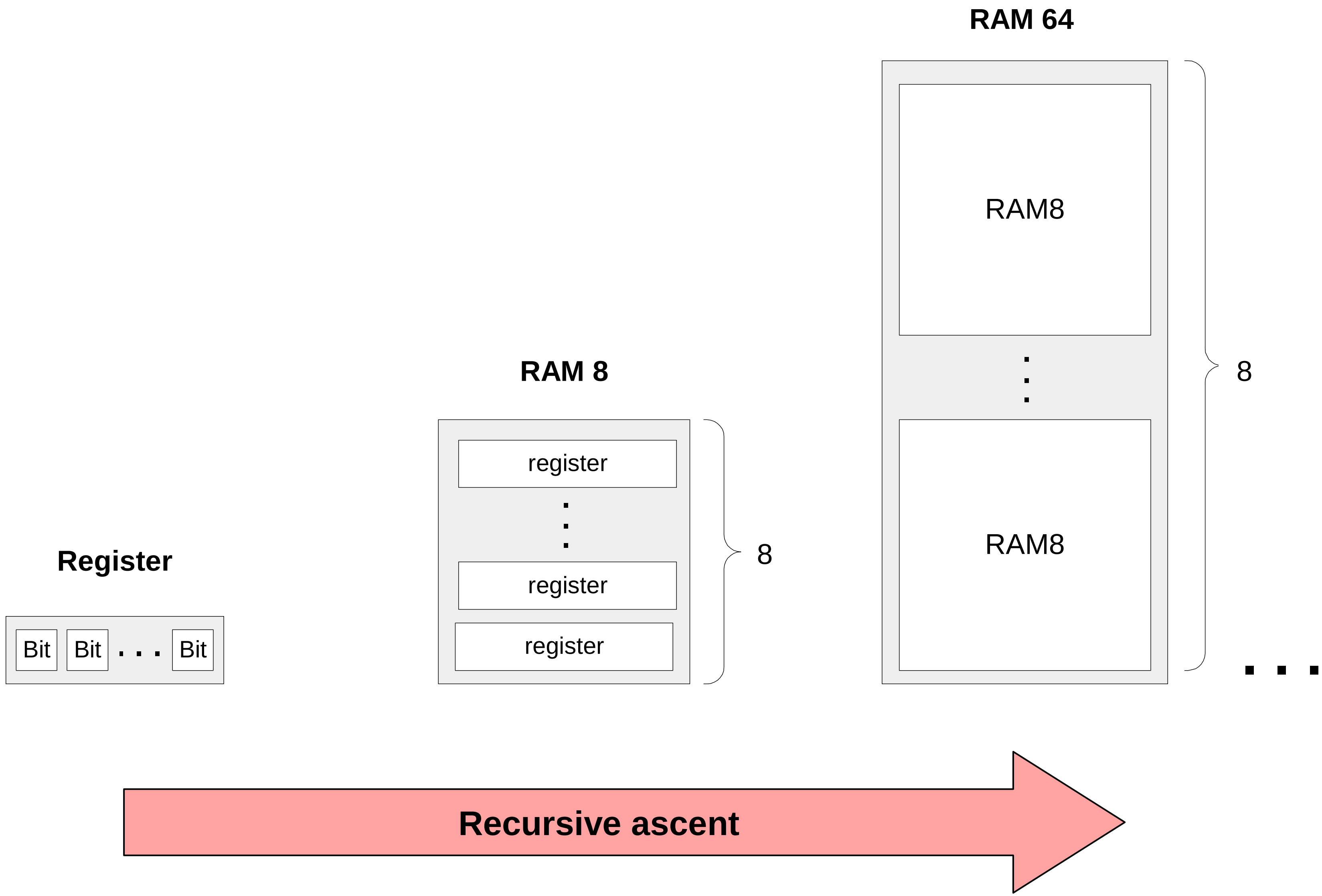



```
Chip name:  RAMn   // n and k are listed below
Inputs:     in[16], address[k], load
Outputs:    out[16]
Function:   out(t)=RAM[address(t)](t)
            If load(t-1) then
               RAM[address(t-1)](t)=in(t-1)
Comment:    "=" is a 16-bit operation.
```

The specific RAM chips needed for the Hack platform are:

Chip name	n	K
RAM8	8	3
RAM64	64	6
RAM512	512	9
RAM4K	4096	12
RAM16K	16384	14

RAM anatomy


RAM 64
RAM8
RAM8
8
RAM 8
register
register
register
8
Register
Bit
Bit
Bit
. . .
Recursive ascent

Counter

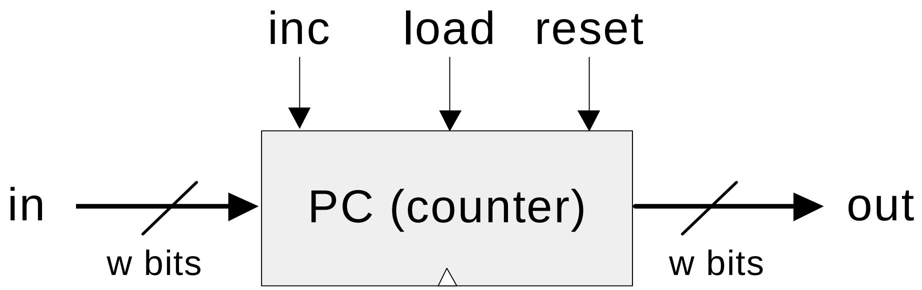
<u>Needed:</u> a storage device that can:

(a) set its state to some base value

(b) increment the state in every clock cycle

(c) maintain its state (stop incrementing) over clock cycles

(d) reset its state

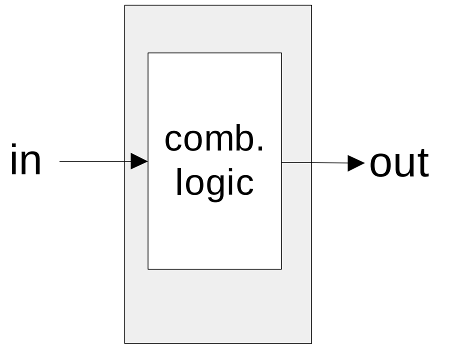



```
If reset(t-1) then out(t)=0
   else if load(t-1) then out(t)=in(t-1)
        else if inc(t-1) then out(t)=out(t-1)+1
             else out(t)=out(t-1)
```

- Typical function: *program counter*
- Implementation: register chip + some combinational logic.

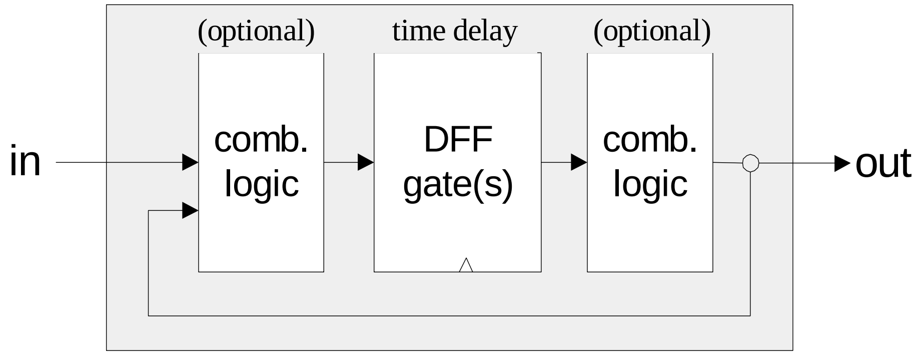
Recap: Sequential VS combinational logic

Combinational chip



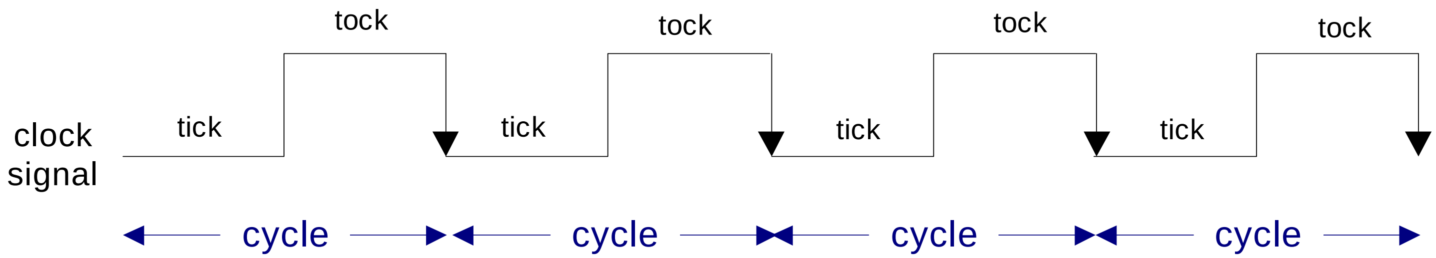
out = *some function of* (in)

Sequential chip

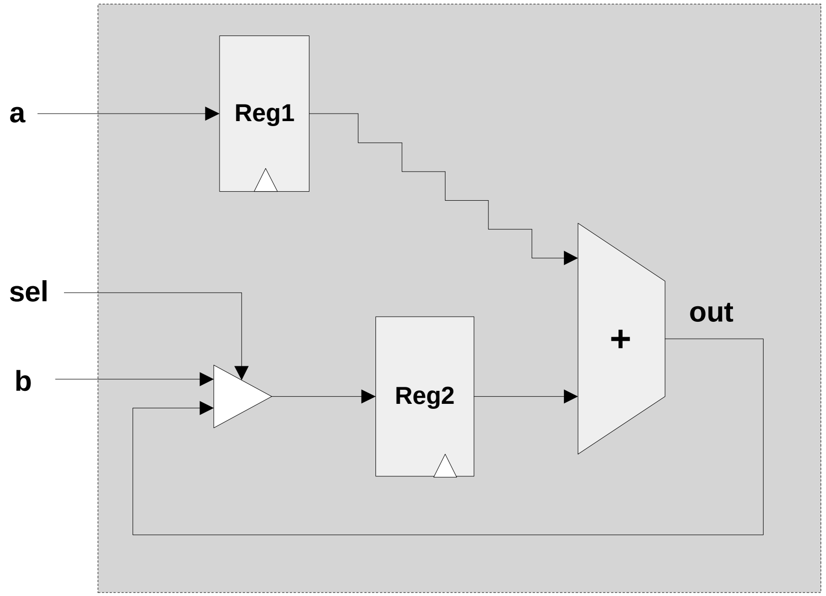


out(t) = *some function of* (in(t-1), out(t-1))

Time matters



- During a tick-tock cycle, the internal states of all the clocked chips are allowed to change, but their outputs are "latched"
- At the beginning of the next cycle, the outputs of all the clocked chips in the architecture commit to the new values.



<u>Implications:</u>

- Challenge: propagation delays
- Solution: clock synchronization
- Cycle length and processing speed.

Perspective

- All the memory units described in this lecture are standard

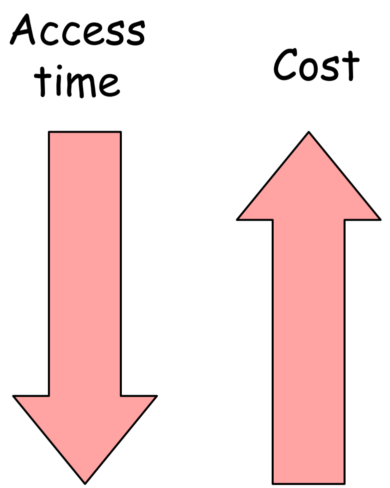
- Typical memory hierarchy

  - SRAM ("static"), typically used for the cache

  - DRAM ("dynamic"), typically used for main memory

  - Disk

  (Elaborate caching / paging algorithms)



- A Flip-flop can be built from Nand gates

- But ... real memory units are highly optimized, using a great variety of storage technologies.

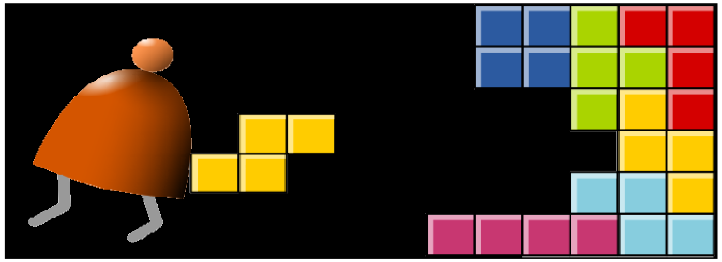
End notes: some poetry about the limits of logic ...

There exists a field where things are neither true nor false;
I'll meet you there. (Rumi)

In the place where we are always right
No flowers will bloom in springtime (Yehuda Amichai)

A mind all logic is like a knife all blade;
It makes the hand bleed that uses it.
(Rabindranath Tagor)

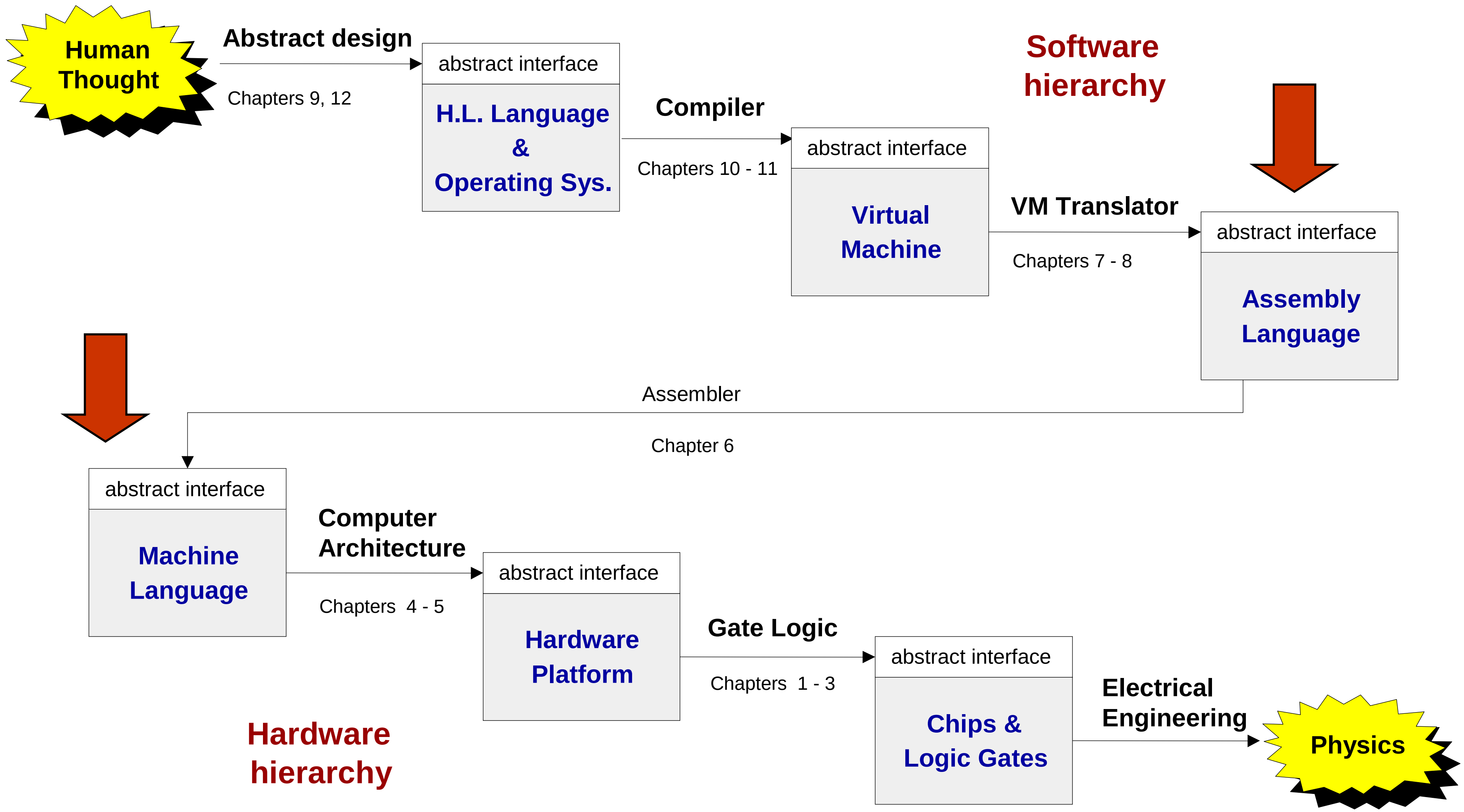
Machine Language



Building a Modern Computer From First Principles

www.nand2tetris.org

Where we are at:



Machine language

Abstraction – implementation duality:

- Machine language (= instruction set) can be viewed as a programmer-oriented abstraction of the hardware platform
- The hardware platform can be viewed as a physical means for realizing the machine language abstraction

Another duality:

- Binary version
- Symbolic version

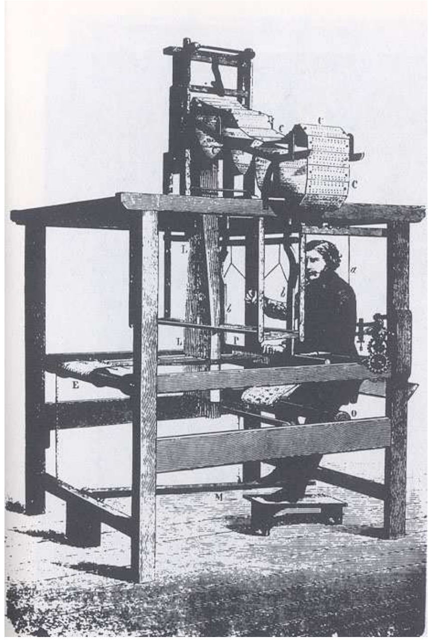
Loose definition:

- Machine language = an agreed-upon formalism for manipulating a *memory* using a *processor* and a set of *registers*
- Same spirit but different syntax across different hardware platforms.

Binary and symbolic notation

`1010 0001 0010 1011`

`ADD R1, R2, R3`



Jacquard loom

(1801)

Evolution:

- Physical coding
- Symbolic documentation
- Symbolic coding
- Translation and execution
- Requires a *translator*.

Augusta Ada King, Countess of Lovelace

(1815-1852)

Lecture plan

- Machine languages at a glance
- The Hack machine language:
  - Symbolic version
  - Binary version
- Perspective

(The assembler will be covered in lecture 6).

Typical machine language commands (a small sample)

```
// In what follows R1,R2,R3 are registers, PC is program counter,
// and addr is some value.

ADD R1,R2,R3       // R1 ← R2 + R3

ADDI R1,R2,addr    // R1 ← R2 + addr

AND R1,R1,R2       // R1 ← R1 and R2 (bit-wise)

JMP addr           // PC ← addr

JEQ R1,R2,addr     // IF R1 == R2 THEN PC ← addr ELSE PC++

LOAD R1, addr      // R1 ← RAM[addr]

STORE R1, addr     // RAM[addr] ← R1

NOP                // Do nothing

// Etc. – some 50-300 command variants
```

The Hack computer

A 16-bit machine consisting of the following elements:

Data memory: **RAM** – an addressable sequence of registers

Instruction memory: **ROM** – an addressable sequence of registers

Registers: **D**, **A**, **M**, where **M** stands for **RAM[A]**

Processing: **ALU**, capable of computing various functions

Program counter: **PC**, holding an address

Control: The **ROM** is loaded with a sequence of 16-bit instructions, one per memory location, beginning at address 0. Fetch-execute cycle: later

Instruction set: Two instructions: *A*-instruction, *C*-instruction.

The A-instruction

```
@value        // A ← value
```

Where *value* is either a number or a symbol referring to some number.

Used for:

- Entering a constant value
  (**A** = **value**)

Coding example:

```
@17      // A = 17
D = A    // D = 17
```

- Selecting a **RAM** location
  (**register** = **RAM[A]**)

```
@17      // A = 17
D = M    // D = RAM[17]
```

- Selecting a **ROM** location
  (**PC** = **A**)

Later

```
@17      // A = 17
JMP      // fetch the instruction
         // stored in ROM[17]
```

The C-instruction (first approximation)

dest = *x* + *y*

dest = *x* - *y*

dest = *x*

dest = 0

dest = 1

dest = -1

x = {A, D, M}

y = {A, D, M , 1}

dest = {A, D, M, MD, A, AM, AD, AMD, null}

Exercise: Implement the following tasks using Hack commands:

- Set **D** to **A-1**
- Set both **A** and **D** to **A + 1**
- Set **D** to **19**
- Set both **A** and **D** to **A + D**
- Set **RAM[5034]** to **D - 1**
- Set **RAM[53]** to **171**
- Add 1 to **RAM[7]**,
  and store the result in **D**.

The C-instruction (first approximation)

```
dest = x + y
dest = x - y
dest = x
dest = 0
dest = 1
dest = -1
```

x = {A, D, M}

y = {A, D, M , 1}

dest = {A, D, M, MD, A, AM, AD, AMD, null}

Symbol table:

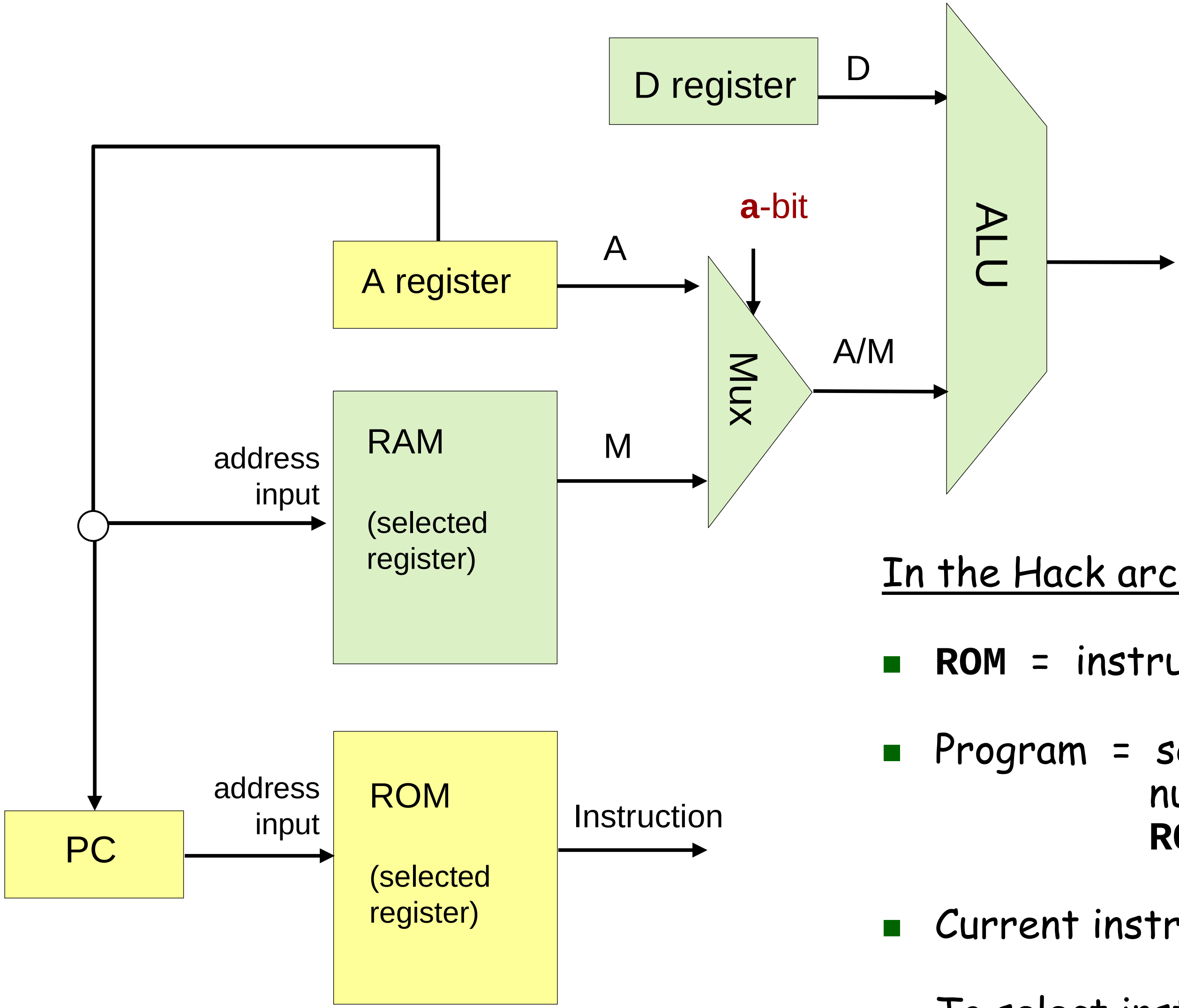
j	3012
sum	4500
q	3812
arr	20561

(All symbols and values are arbitrary examples)

Exercise: Implement the following tasks using Hack commands:

- `sum = 0`
- `j = j + 1`
- `q = sum + 12 – j`
- `arr[3] = -1`
- `arr[j] = 0`
- `arr[j] = 17`
- `etc.`

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**

Coding examples (practice)

Exercise: Implement the following tasks using Hack commands:

- goto 50
- if D==0 goto 112
- if D<9 goto 507
- if RAM[12] > 0 goto 50
- if sum>0 goto END
- if x[i]<=0 goto NEXT.

Hack commands:

A-command: @value // set A to value

C-command: dest = comp ; jump // dest = and ;jump
// are optional

Where:

comp = 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|A , M , !M , -M ,M+1, M-1 , D+M , D-M ,
M-D , D&M , D|M

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

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

In the command dest = comp; jump, the jump materialzes if (comp jump 0) is true. For example, in D=D+1,JLT, we jump if D+1 < 0.

Hack convention:

- True is represented by -1
- False is represented by 0

Symbol table:

sum	2200
x	4000
i	6151
END	50
NEXT	120

(All symbols and values in are arbitrary examples)

IF logic – Hack style

High level:

```
if condition {
   code block 1}
else {
   code block 2}
code block 3
```

Hack:

```
   D ← not condition
   @IF_TRUE
   D;JEQ
   code block 2
   @END
   0;JMP
(IF_TRUE)
   code block 1
(END)
    code block 3
```

Hack convention:

- True is represented by -1
- False is represented by 0

WHILE logic – Hack style

High level:

```
while condition {
    code block 1
}
Code block 2
```

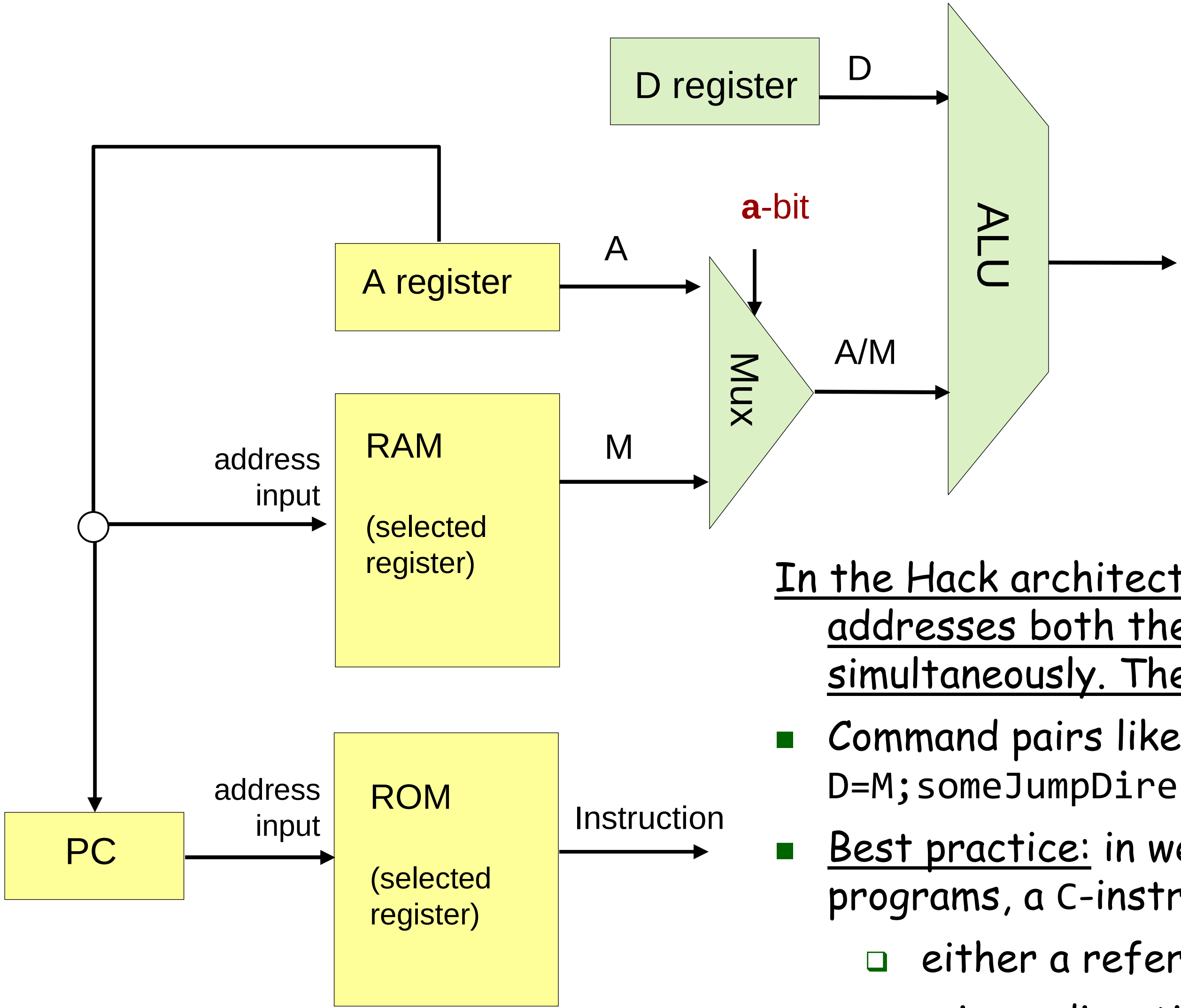
Hack:

```
(LOOP)
     D ← not condition)
     @END
     D;JEQ
     code block 1
     @LOOP
     0;JMP
(END)
     code block 2
```

Hack convention:

- True is represented by -1
- False is represented by 0

Side note (focus on the yellow chip parts only)



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.

Complete program example

C language code:

```
// Adds 1+...+100.
 into i = 1;
 into sum = 0;
 while (i <= 100){
    sum += i;
    i++;
 }
```

Hack assembly code:

```
// Adds 1+...+100.
      @i      // i refers to some RAM location
      M=1     // i=1
      @sum    // sum refers to some RAM location
      M=0     // sum=0
(LOOP)
      @i
      D=M      // D = i
      @100
      D=D-A    // D = i - 100
      @END
      D;JGT    // If (i-100) > 0 goto END
      @i
      D=M      // D = i
      @sum
      M=D+M    // sum += i
      @i
      M=M+1    // i++
      @LOOP
      0;JMP    // Got LOOP
 (END)
      @END
      0;JMP    // Infinite loop
```

Hack assembly convention:

- Variables: lower-case
- Labels: upper-case
- Commands: upper-case

Demo
CPU emulator

Symbols in Hack assembly programs

<u>Symbols created by Hack programmers and code generators:</u>

- Label symbols: Used to label destinations of goto commands. Declared by the pseudo command **(XXX)**. This directive defines the symbol **XXX** to refer to the instruction memory location holding the next command in the program (within the program, **XXX** is called "label")
- Variable symbols: Any user-defined symbol **xxx** appearing in an assembly program that is not defined elsewhere using the **(xxx)** directive is treated as a variable, and is "automatically" assigned a unique RAM address, starting at RAM address 16

  By convention, Hack programmers use lower-case and upper-case letters for variable names and labels, respectively.

<u>Predefined symbols:</u>

- I/O pointers: The symbols **SCREEN** and **KBD** are "automatically" predefined to refer to RAM addresses 16384 and 24576, respectively (base addresses of the Hack platform's *screen* and *keyboard* memory maps)
- Virtual registers: covered in future lectures.
- VM control registers: covered in future lectures.

<u>Q</u>: Who does all the "automatic" assignments of symbols to RAM addresses?

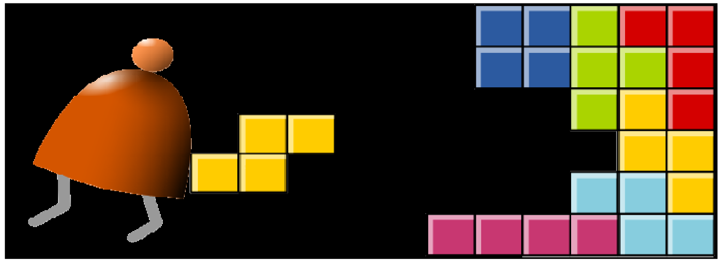
<u>A</u>: The *assembler*, which is the program that translates symbolic Hack programs into binary Hack program. As part of the translation process, the symbols are resolved to RAM addresses. (more about this in future lectures)

```
// Typical symbolic
// Hack code, meaning
// not important
  @R0
   D=M
   @INFINITE_LOOP
   D;JLE
   @counter
   M=D
   @SCREEN
   D=A
   @addr
   M=D
(LOOP)
   @addr
   A=M
   M=-1
   @addr
   D=M
   @32
   D=D+A
   @addr
   M=D
   @counter
   MD=M-1
   @LOOP
   D;JGT
(INFINITE_LOOP)
   @INFINITE_LOOP
   0;JMP
```

Perspective

- Hack is a simple machine language
- User friendly syntax: **`D=D+A`** instead of **`ADD D,D,A`**
- Hack is a "½-address machine": any operation that needs to operate on the RAM must be specified using two commands: an **A**-command to address the RAM, and a subsequent **C**-command to operate on it
- A Macro-language can be easily developed
- A Hack assembler is needed and will be discusses and developed later in the course.

Computer Architecture

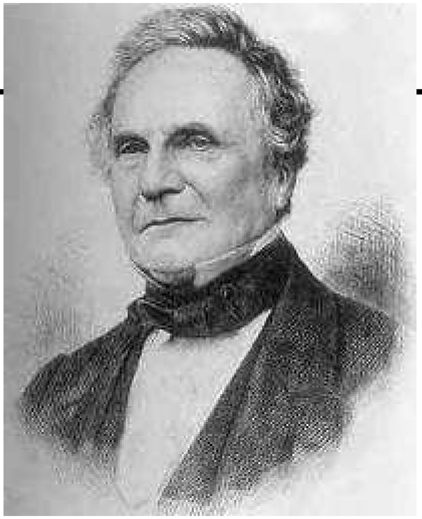


Building a Modern Computer From First Principles

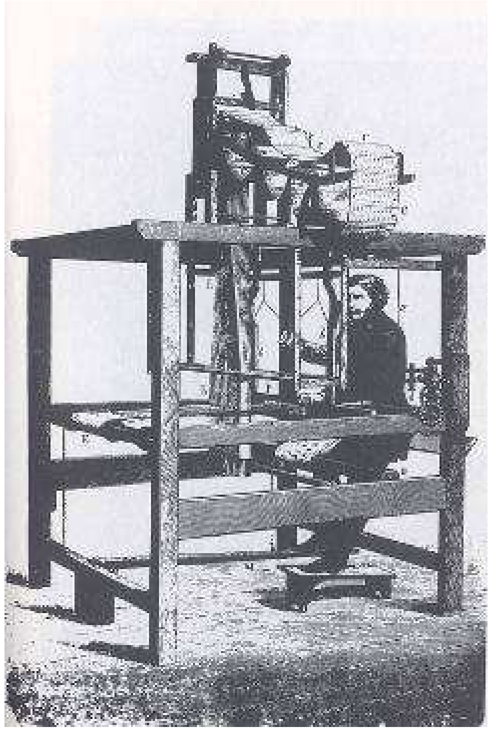
www.nand2tetris.org

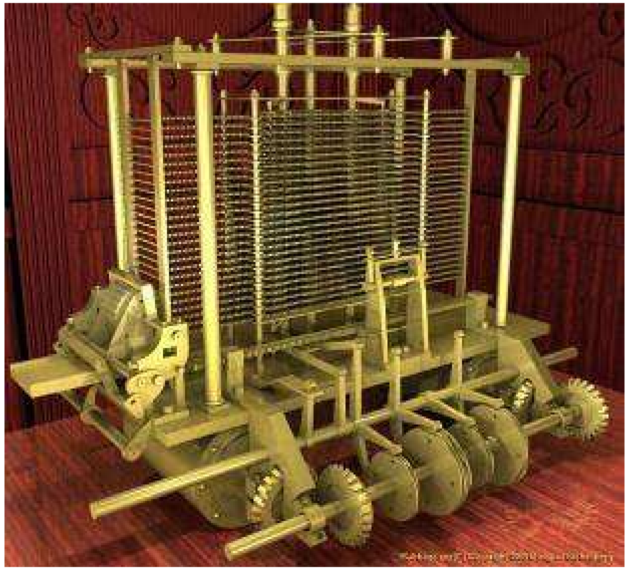
Babbage's Analytical Engine (1835)

"We may say most aptly that the Analytical Engine weaves algebraic patterns just as the Jacquard-loom weaves flowers and leaves"
(Ada Lovelace)



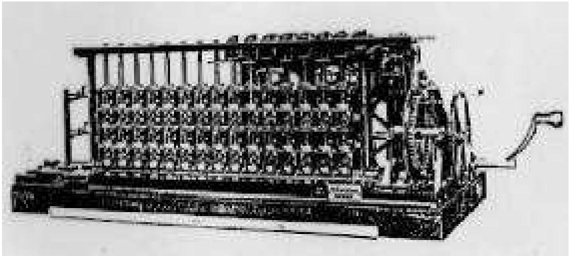
Charles Babbage (1791-1871)



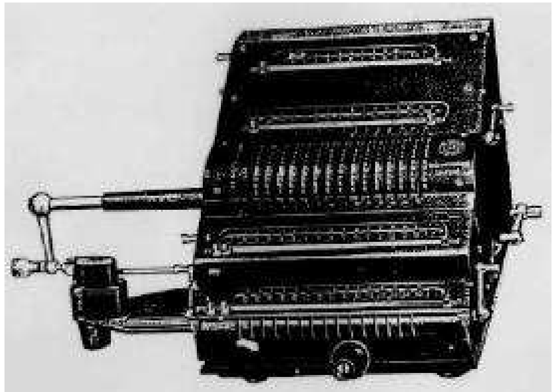


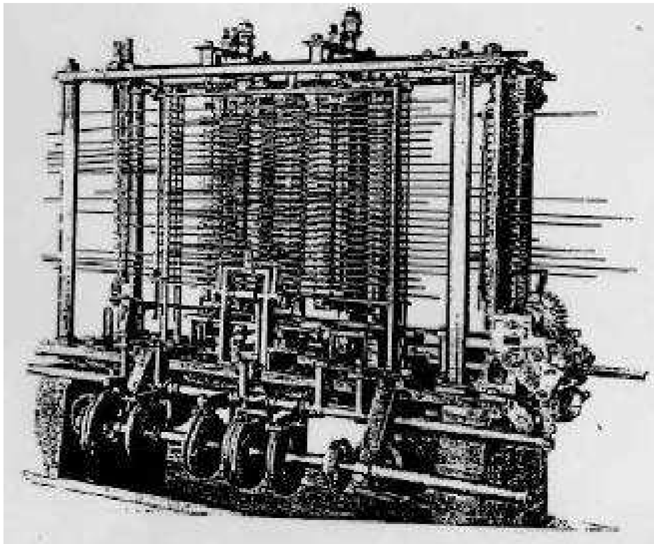
Some early computers and computer scientists

Blaise Pascal
1623-1662

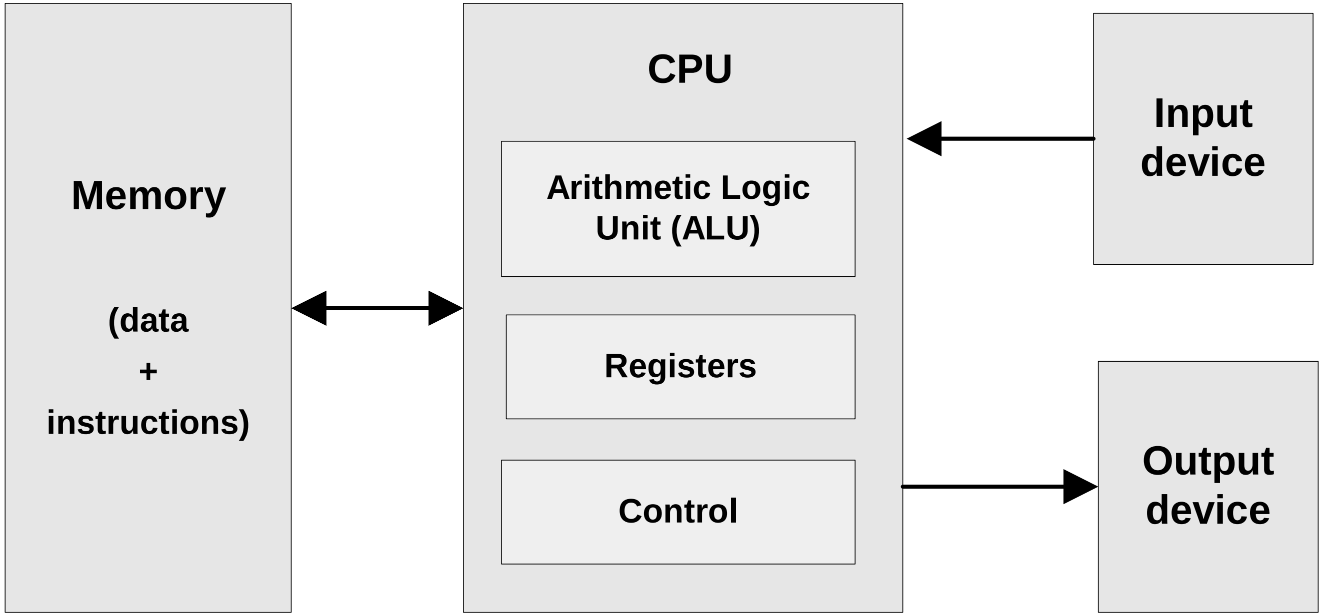


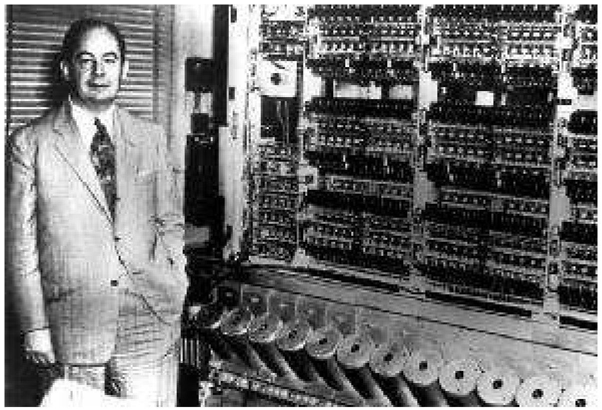
Gottfried Leibniz
1646-1716





Von Neumann machine (circa 1940)



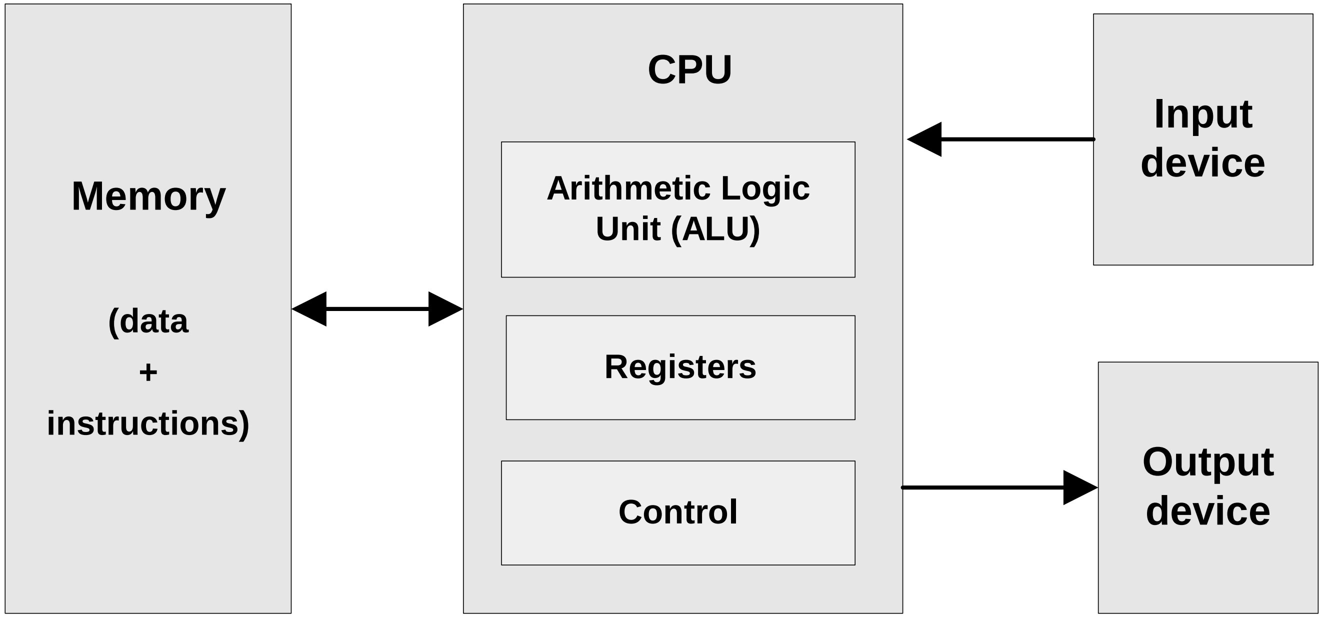


Stored program concept!

John Von Neumann (and others) ... made it possible

Andy Grove (and others) ... made it small and fast.

Processing logic: fetch-execute cycle



Executing the *current instruction* involves one or more of the following micro-tasks:

- Have the ALU compute some function out = f (register values)
- Write the ALU output to selected registers
- As a side-effect of this computation, figure out which instruction to fetch and execute next.

The Hack chip-set and hardware platform

Elementary logic gates

- `Nand`
- `Not`
- `And`
- `Or`
- `Xor`
- `Mux`
- `Dmux`
- `Not16`
- `And16`
- `Or16`
- `Mux16`
- `Or8Way`
- `Mux4Way16`
- `Mux8Way16`
- `DMux4Way`
- `DMux8Way`

done

Combinational chips

- `HalfAdder`
- `FullAdder`
- `Add16`
- `Inc16`
- `ALU`

done

Sequential chips

- `DFF`
- `Bit`
- `Register`
- `RAM8`
- `RAM64`
- `RAM512`
- `RAM4K`
- `RAM16K`
- `PC`

done

Computer Architecture

- `Memory`
- `CPU`
- `Computer`

this lecture

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

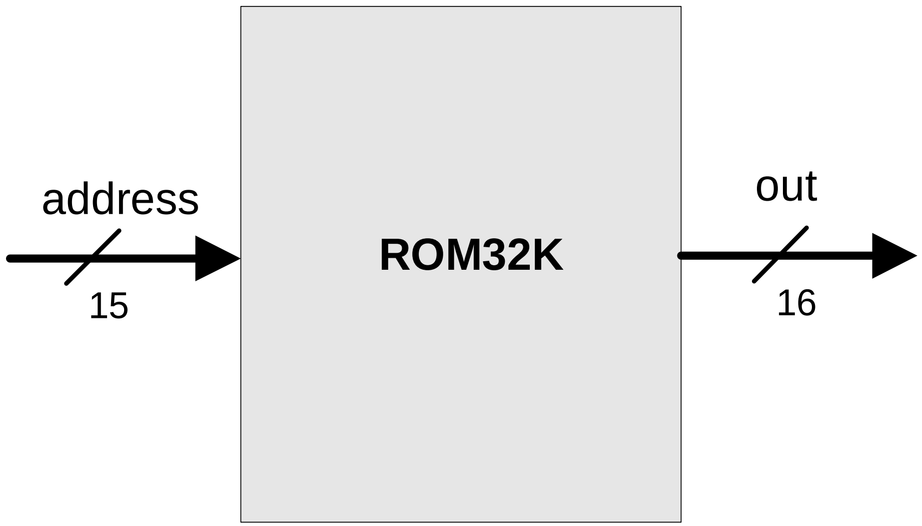
<u>Main parts of the Hack computer:</u>

- Instruction memory (ROM)
- Memory (RAM):
  - Data memory
  - Screen (memory map)
  - Keyboard (memory map)
- CPU
- Computer (the logic that holds everything together).

Lecture / construction plan

- Instruction memory
- Memory:
  - Data memory
  - Screen
  - Keyboard
- CPU
- Computer

Instruction memory



<u>Function:</u>

- The ROM is pre-loaded with a program written in the Hack machine language
- The ROM chip always emits a 16-bit number:

```
out = ROM32K[address]
```

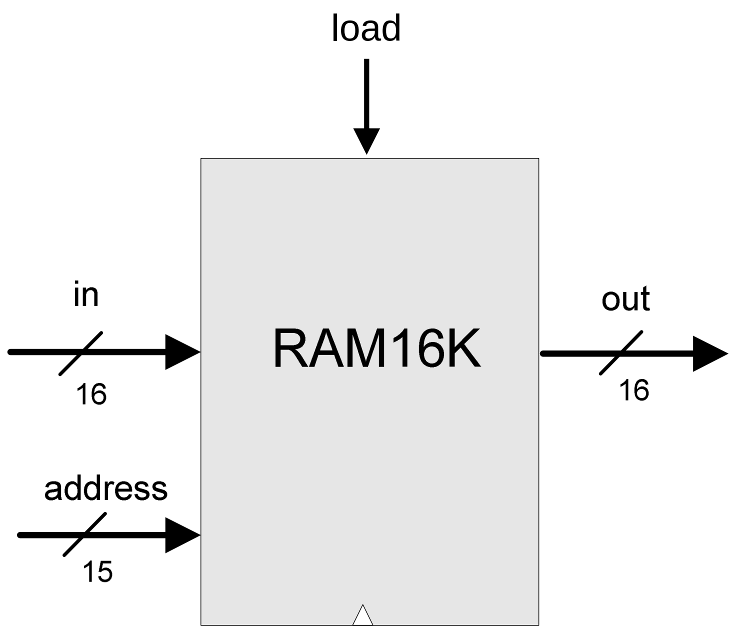
- This number is interpreted as the *current instruction.*

Data memory

<u>Low-level (hardware) read/write logic:</u>

To read `RAM[k]`: set `address` to k,
probe `out`

To write `RAM[k]=x`: set `address` to k,
set `in` to x,
set `load` to 1,
run the clock



<u>High-level (OS) read/write logic:</u>

To read `RAM[k]`: use the OS command `out = peek(k)`

To write `RAM[k]=x`: use the OS command `poke(k,x)`

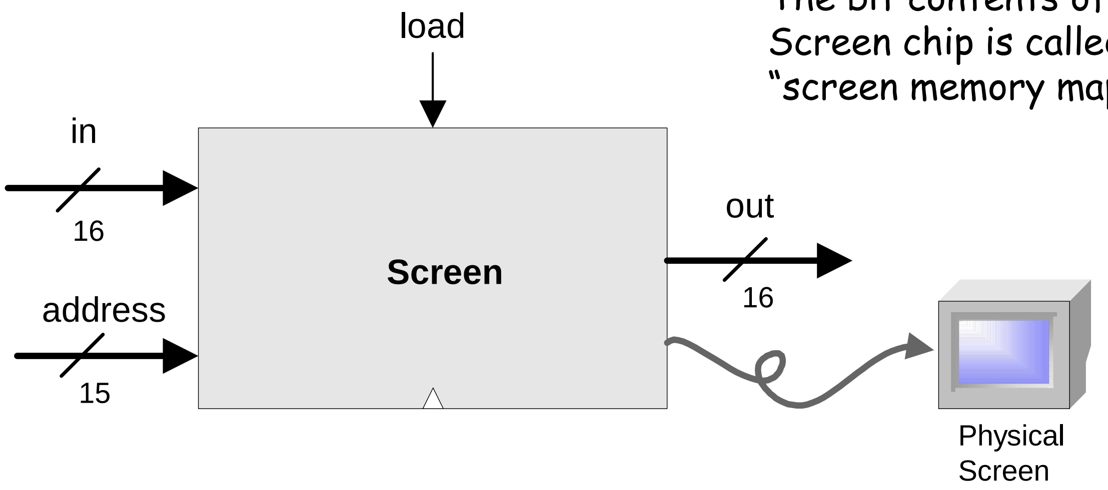
`peek` and `poke` are OS commands whose implementation should effect the same behavior as the low-level commands

More about `peek` and `poke` this later in the course, when we'll write the OS.

Lecture / construction plan

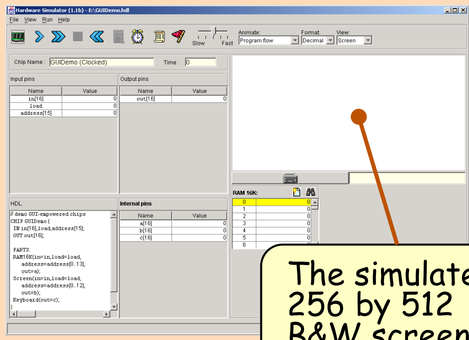
- Instruction memory
- Memory:
  - Data memory
  - Screen
  - Keyboard
- CPU
- Computer

Screen



The bit contents of the Screen chip is called the "screen memory map"

Simulated screen:



The simulated 256 by 512 B&W screen

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.

The Screen chip has a basic RAM chip functionality:

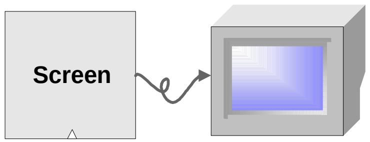
- read logic: `out = Screen[address]`
- write logic: `if load then Screen[address] = in`

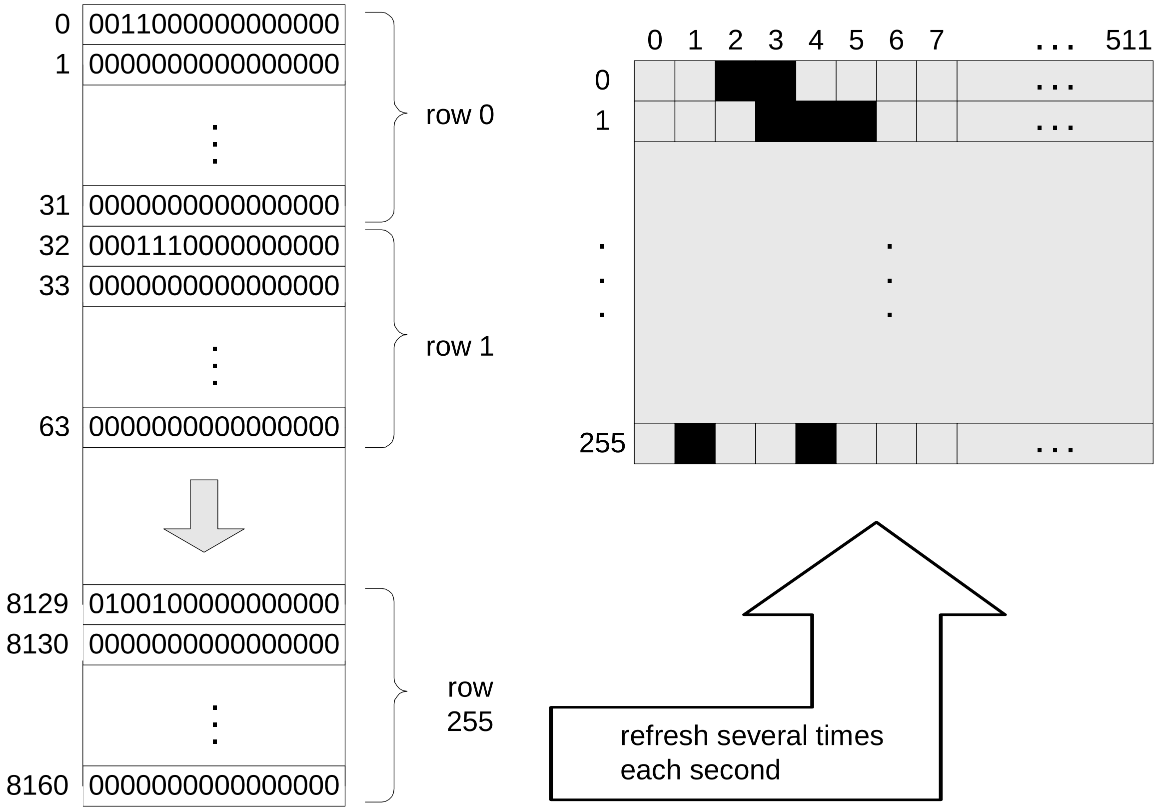
Side effect:

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

Screen memory map

In the Hack platform, the screen is implemented as an 8K 16-bit RAM chip.

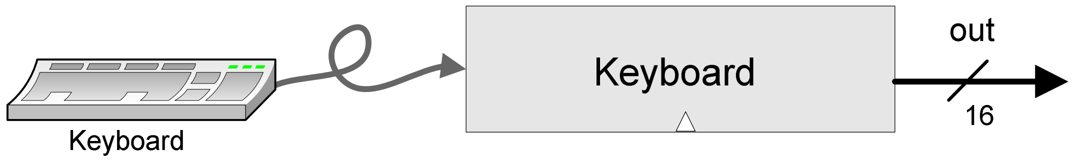




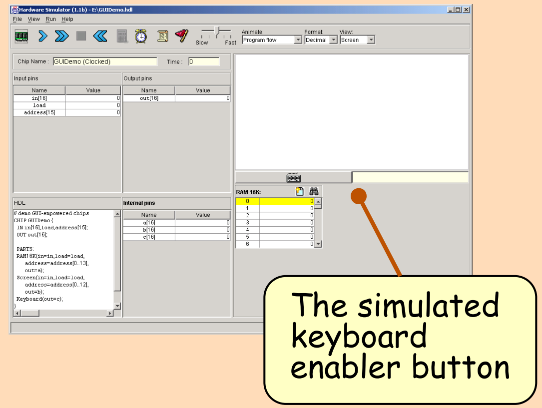
<u>How to set the `(row,col)` pixel of the screen to black or to white:</u>

- Low-level (machine language): Set the `col%16` bit of the word found at `Screen[row*32+col/16]` to `1` or to `0` (`col/16` is integer division)
- High-level: Use the OS command `drawPixel(row,col)` (effects the same operation, discussed later in the course, when we'll write the OS).

Keyboard



Simulated keyboard:



The keyboard is implemented as a built-in **`Keyboard.hdl`** chip. When this java chip is loaded into the simulator, it connects to the regular keyboard and pipes the scan-code of the currently pressed key to the keyboard memory map.

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:

Key pressed	Keyboard output	Key pressed	Keyboard output
newline	128	end	135
backspace	129	page up	136
left arrow	130	page down	137
up arrow	131	insert	138
right arrow	132	delete	139
down arrow	133	esc	140
home	134	f1-f12	141-152

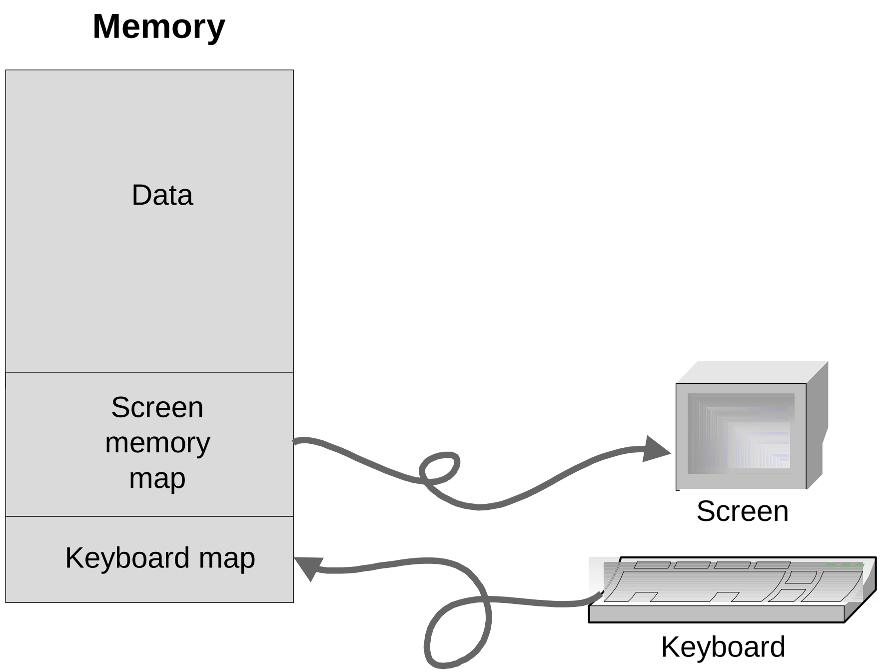
How to read the keyboard:

- 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).

Lecture / construction plan

- Instruction memory
- Memory:
  - Data memory
  - Screen
  - Keyboard
- CPU
- Computer

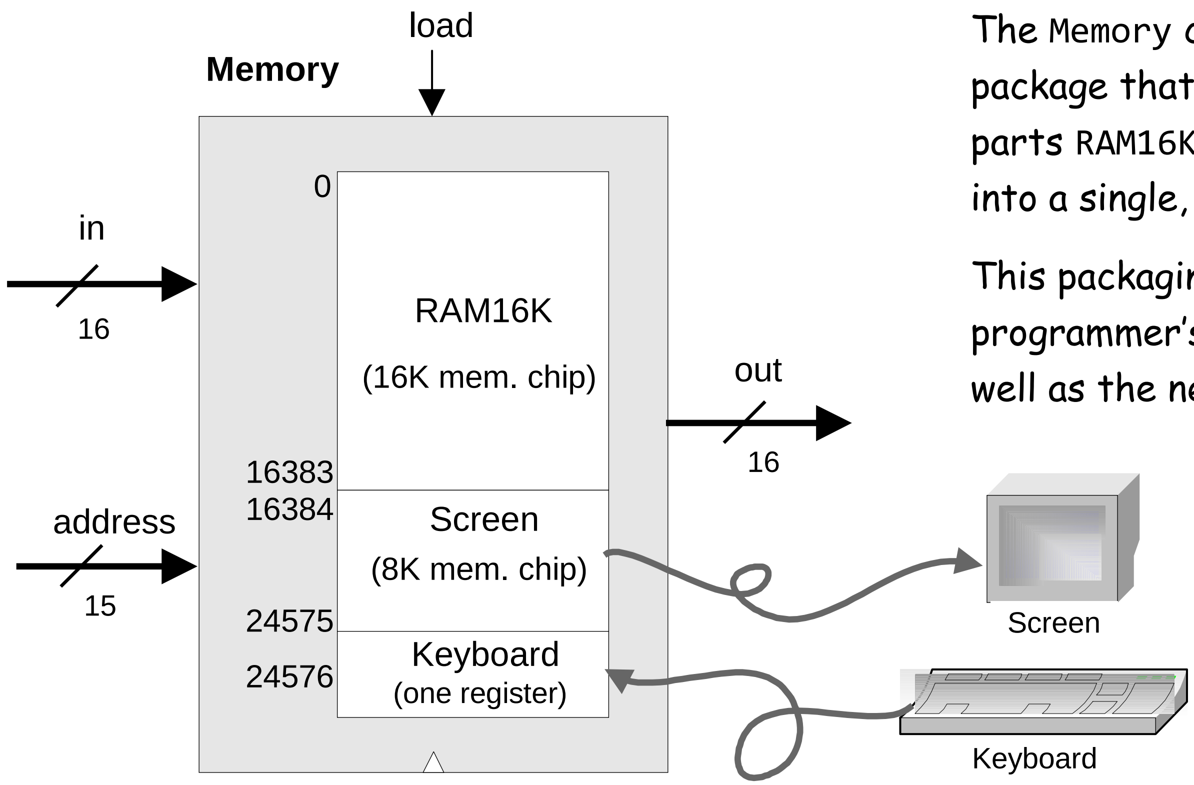
Memory: conceptual / programmer's view



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.

<u>Access logic:</u>

- 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.

Lecture / construction plan

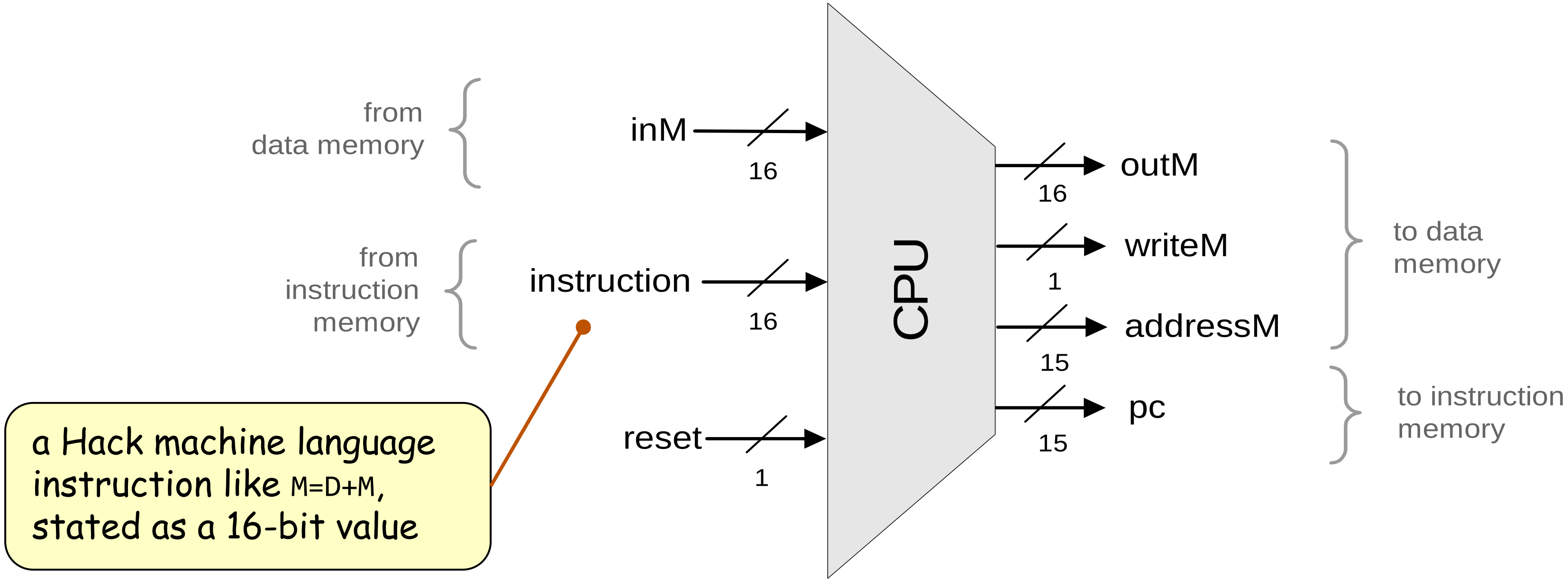
- Instruction memory
- Memory:
  - Data memory
  - Screen
  - Keyboard
- CPU
- Computer

A pledge to patience ...

"At times … the fragments that I lay out for your inspection may seem not to fit well together, as if they were stray pieces from separate puzzles. In such cases, I would counsel patience. There are moments when a large enough fragment can become a low wall, a second fragment another wall to be raised at a right angle to the first. A few struts and beams later, and we may made ourselves a rough foundation … But it can consume the better part of a chapter to build such a foundation; and as we do so the fragment that we are examining may seem unconnected to the larger whole. Only when we step back can we see that we have been assembling something that can stand in the wind."

From: Sailing the Wind Dark Sea (Thomas Cahill)

CPU



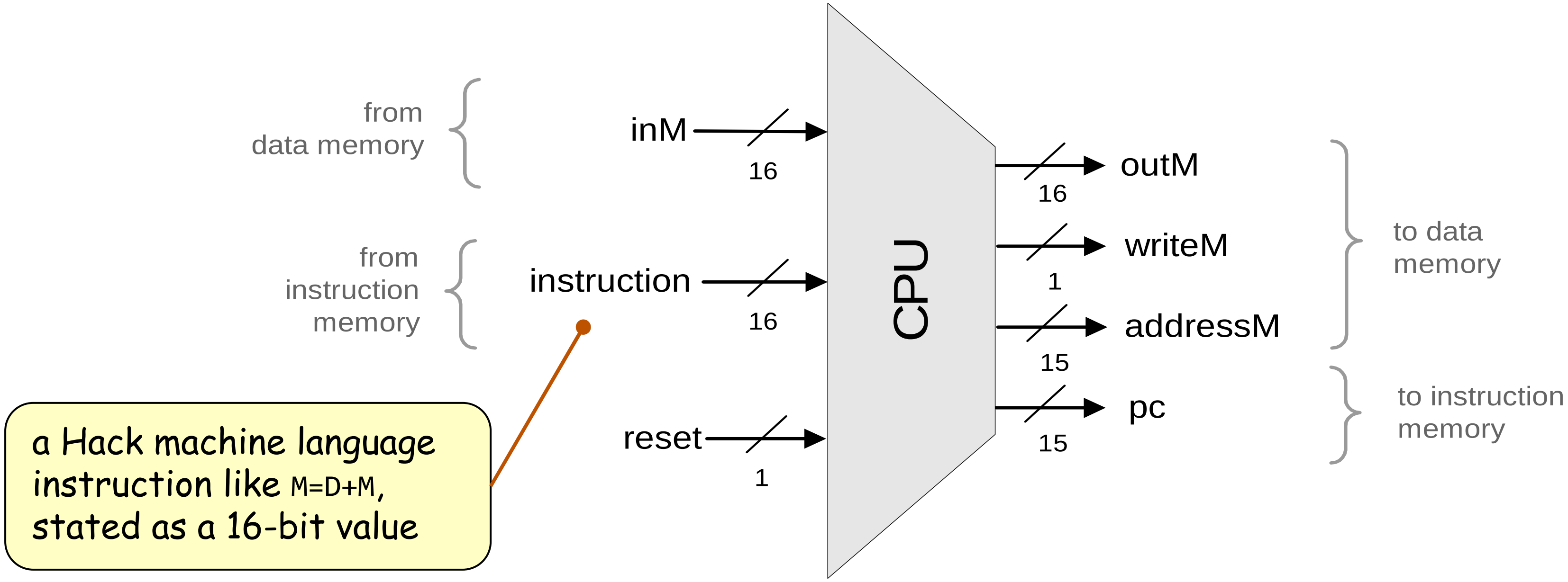
<u>CPU internal components</u> (invisible in this chip diagram): `ALU` and 3 registers: `A`, `D`, `PC`

<u>CPU execute logic:</u>

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
- 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.

CPU



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

<u>CPU fetch logic:</u>

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

<u>If reset==0:</u> the CPU uses this information (the jump bits and the ALU control bits) as follows:
If there should be a jump, the PC is set to the value of A; else, PC is set to PC+1

<u>If reset==1:</u> the PC is set to 0. (restarting the computer)

The **C**-instruction revisited

dest = comp; jump

	comp		***dest***	***jump***
binary:	**1** 1 1 a	c1 c2 c3 c4	c5 c6 d1 d2	d3 j1 j2 j3

(when a=0) ***comp***	c1	c2	c3	c4	c5	c6	(when a=1) ***comp***
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	M
!D	0	0	1	1	0	1	
!A	1	1	0	0	0	1	!M
-D	0	0	1	1	1	1	
-A	1	1	0	0	1	1	-M
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	AM	A register and Memory[A]
1	1	0	AD	A register and D register
1	1	1	AMD	A register, Memory[A], and D register

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 $out \geq 0$ jump
1	0	0	JLT	If $out < 0$ jump
1	0	1	JNE	If $out \neq 0$ jump
1	1	0	JLE	If $out \leq 0$ jump
1	1	1	JMP	Jump

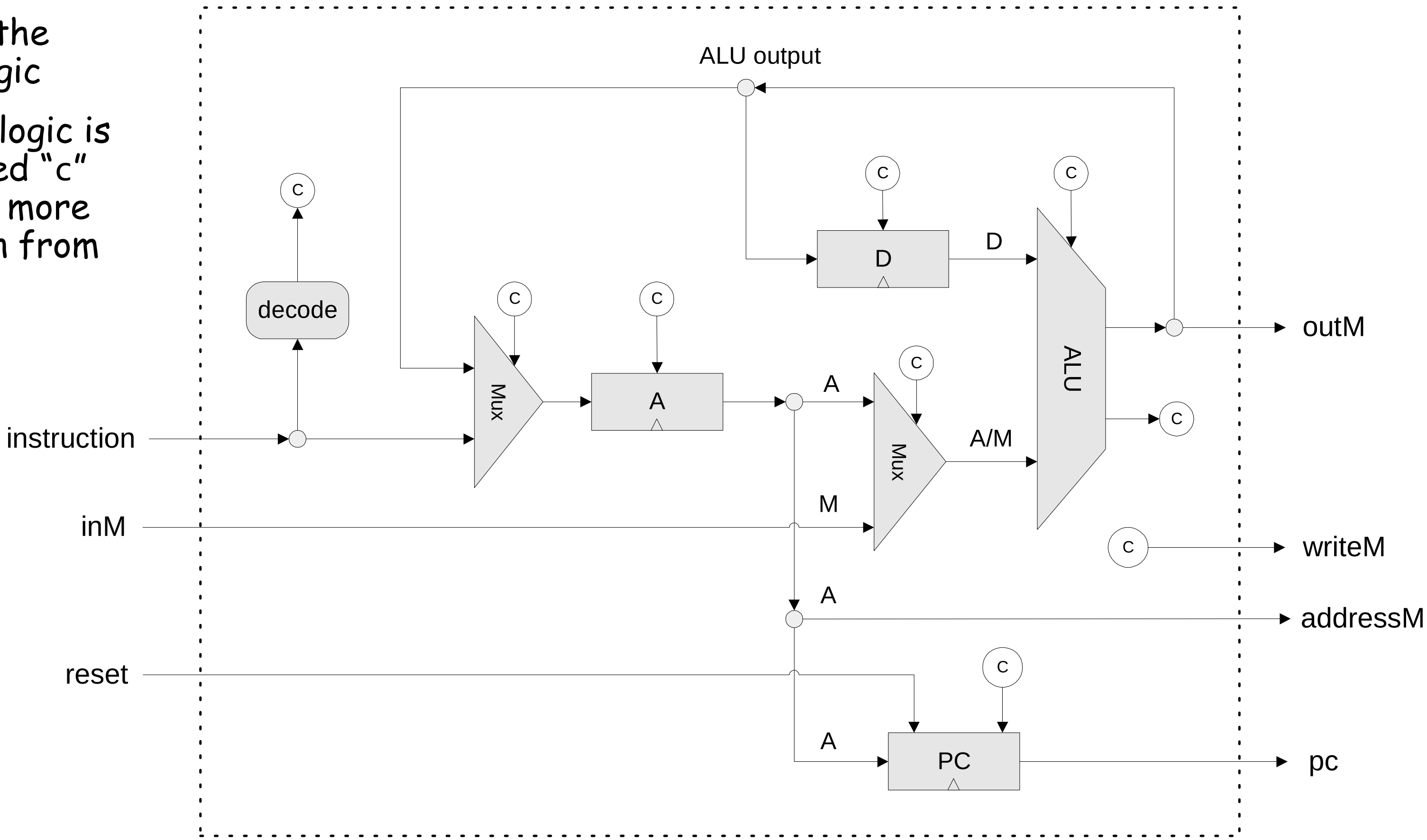
CPU implementation

dest = comp; jump

	comp		dest	jump
binary:	1 1 1 a	c1 c2 c3 c4	c5 c6 d1 d2	d3 j1 j2 j3

Chip diagram:

- Includes most of the CPU's execution logic
- The CPU's control logic is hinted: each circled "c" represents one or more control bits, taken from the instruction
- The "decode" bar does not represent a chip, but rather indicates that the instruction bits are decoded somehow.



Cycle:

- Execute
- Fetch

Execute logic:

- Decode
- Execute

Fetch logic:

If there should be a jump, set PC to A
else set PC to PC+1

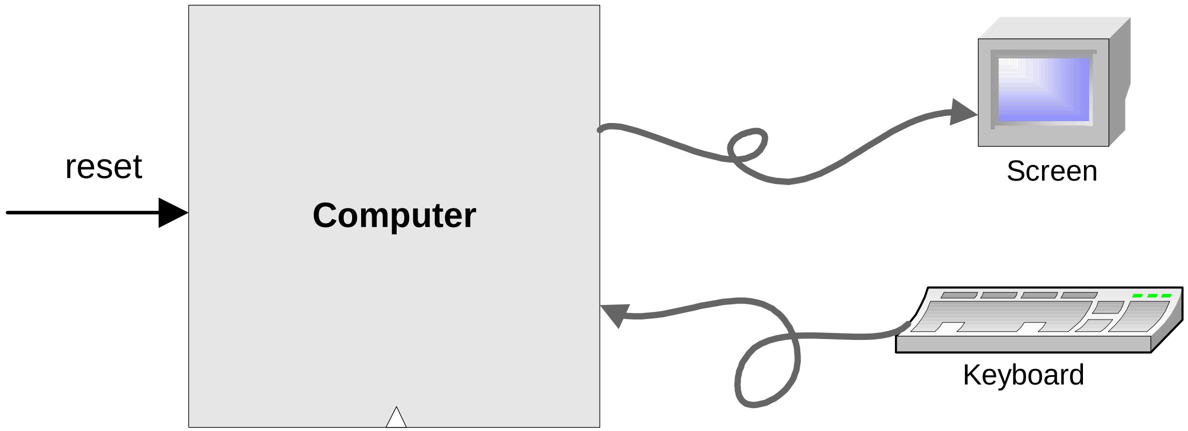
Resetting the computer:

Set reset to 1,
then set it to 0.

Lecture / construction plan

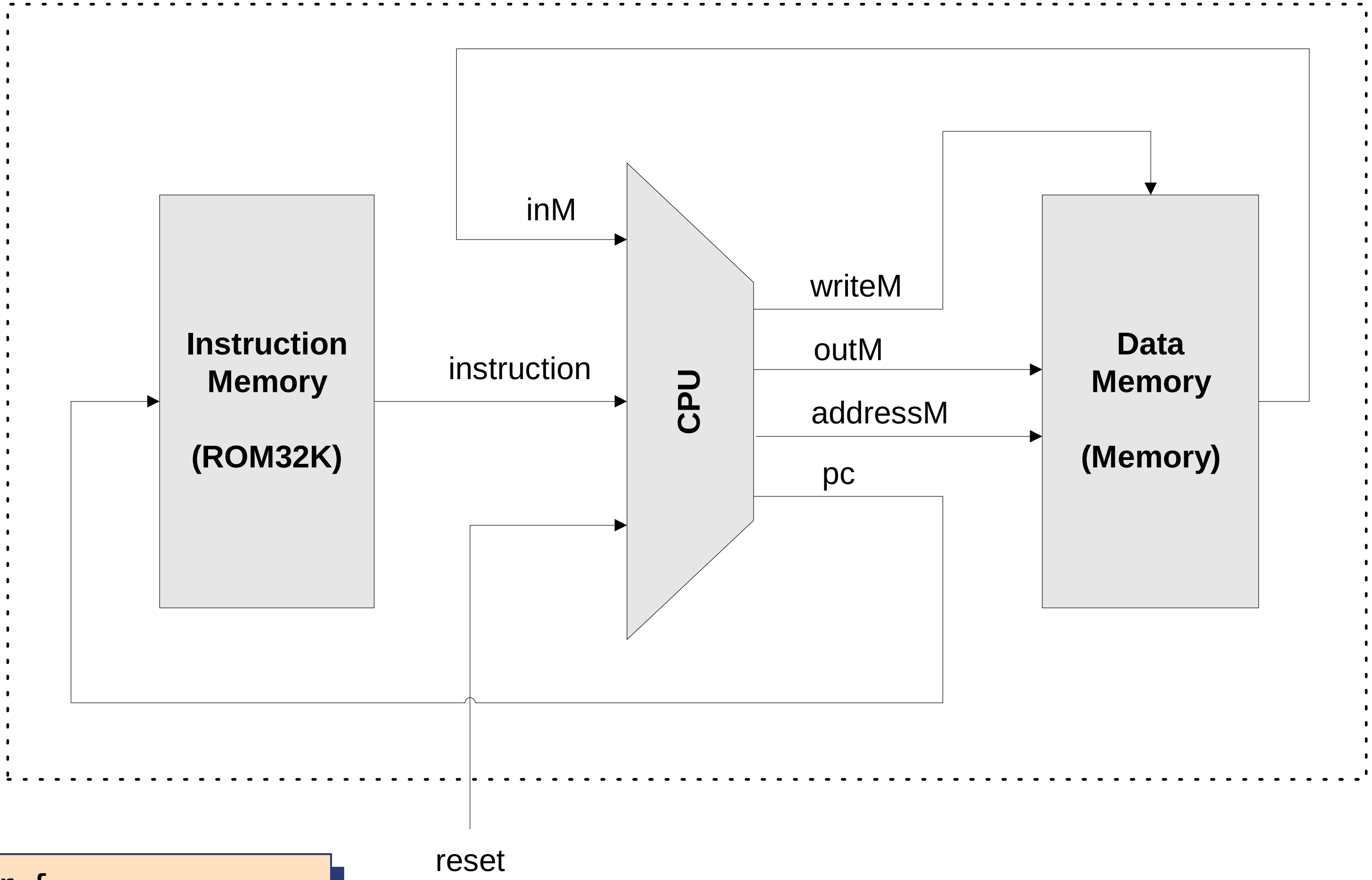
- Instruction memory
- Memory:
  - Data memory
  - Screen
  - Keyboard
- CPU
- Computer

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

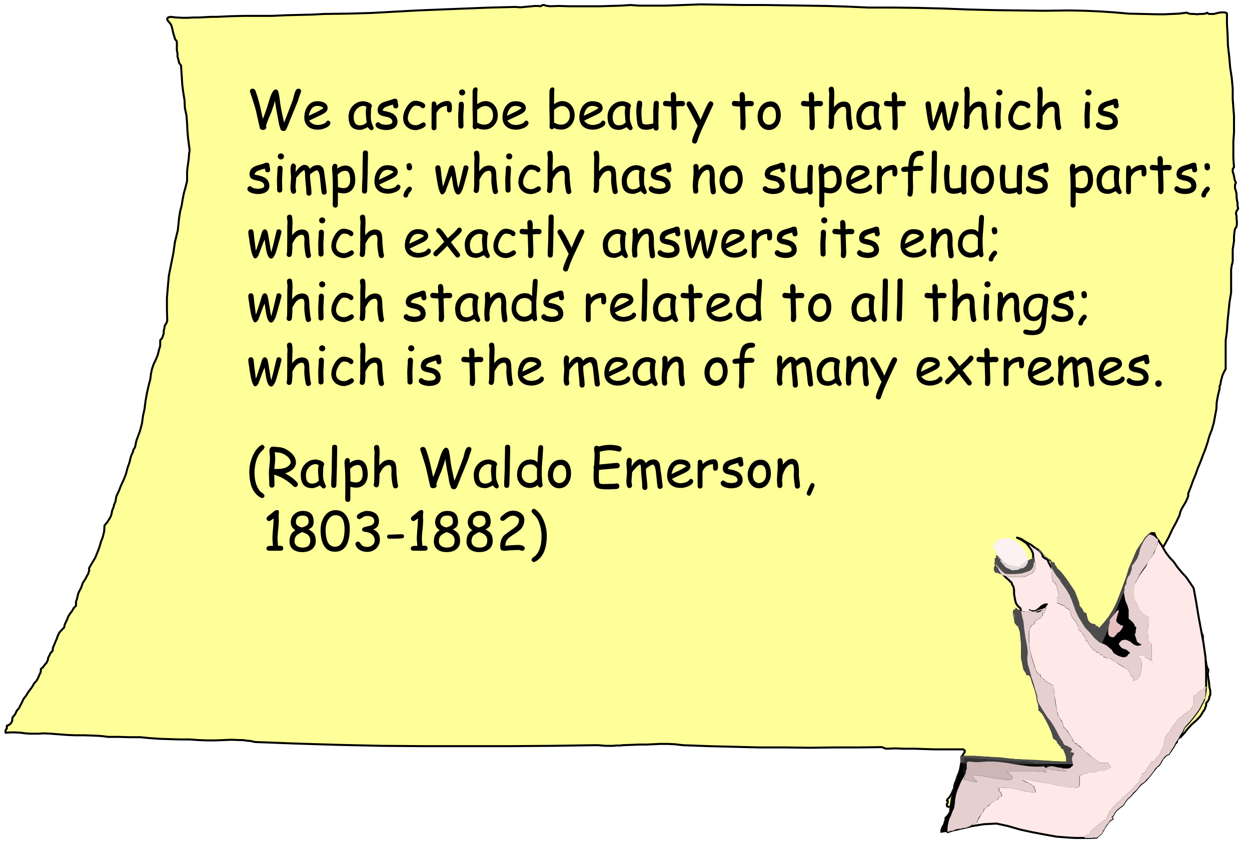



```
CHIP Computer {
    IN reset;
    PARTS:
    // implementation missing
}
```

Implementation:

Simple, the chip-parts do all the hard work.

The spirit of things



Lecture plan

- ✓ Instruction memory
- ✓ Memory:
    - Data memory
    - Screen
    - Keyboard
- ✓ CPU
- ✓ Computer

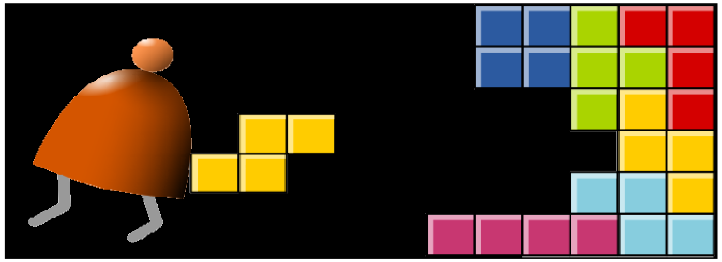
Perspective: from here to a “real” computer

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

Perspective: some issues we haven't discussed (among many)

- CISC / RISC (hardware / software trade-off)
- Hardware diversity: desktop, laptop, hand-held, game machines, …
- General-purpose vs. embedded computers
- Silicon compilers
- And more ...

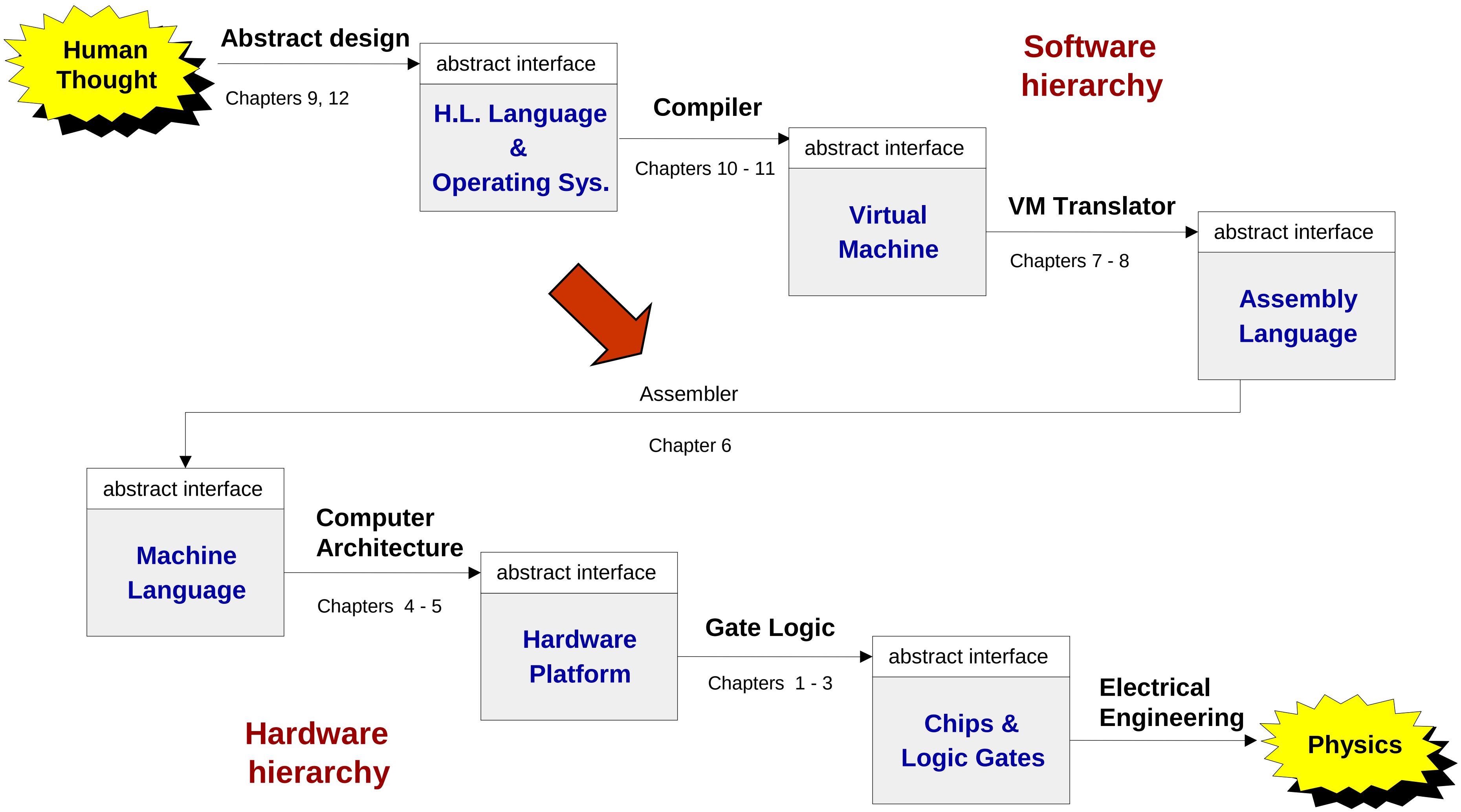
Assembler



Building a Modern Computer From First Principles

www.nand2tetris.org

Where we are at:



Why care about assemblers?

Because ...

- Assemblers employ nifty programming tricks
- Assemblers are the first rung up the software hierarchy ladder
- An assembler is a translator of a simple language
- Writing an assembler = low-impact practice for writing compilers.

Assembly example

For now, ignore all details!

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
    ...    // Etc.
```

assemble

Target code

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

execute

<u>The program translation challenge</u>

- 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

<u>Assembler = simple translator</u>

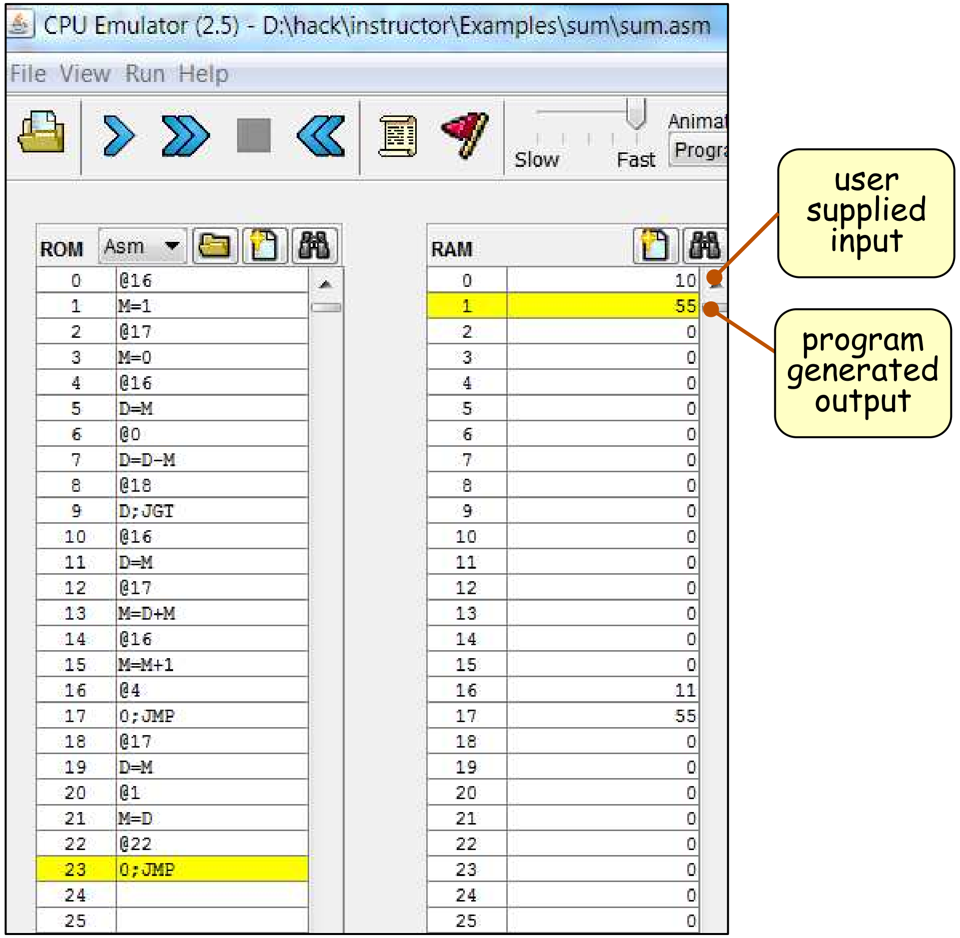
- Translates each assembly command into one or more binary machine instructions
- Handles symbols (e.g. `i`, `sum`, `LOOP`, …).

Revisiting Hack low-level programming: an example

Assembly program (sum.asm)

```
// Computes 1+...+RAM[0]
// And stores 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
    @0
    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
    @1
    M=D  // RAM[1] = the sum
(END)
    @END
    0;JMP
```

CPU emulator screen shot after running this program



The CPU emulator allows loading and executing symbolic Hack code. It resolves all the symbolic symbols to memory locations, and executes the code.

The assembler's view of an assembly program

Assembly program

```
// Computes 1+...+RAM[0]
// And stores 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
    @0
    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
    @1
    M=D  // RAM[1] = the sum
(END)
    @END
    0;JMP
```

Assembly program =
a stream of text lines, each being one of the following:

- A-instruction
- C-instruction
- Symbol declaration: (SYMBOL)
- Comment or white space:
  // comment

The challenge:

Translate the program into a sequence of 16-bit instructions that can be executed by the target hardware platform.

Translating / assembling A-instructions

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

Translation to binary:

- If *value* is a non-negative decimal number, simple
- If *value* is a symbol, later.

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.

Binary:

				comp						*dest*			*jump*		
1	1	1	a	c1	c2	c3	c4	c5	c6	d1	d2	d3	j1	j2	j3

(when a=0) *comp*	c1	c2	c3	c4	c5	c6	(when a=1) *comp*
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			
A	1	1	0	0			
!D	0	0	1	1			
!A	1	1	0	0	0	1	!M
-D	0	0	1	1	1	1	
-A	1	1	0	0	1	1	-M
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
				Memory[A] and D register
				A register
				A register and Memory[A]
1	1	0	AD	A register and D register
1	1	1	AMD	A register, Memory[A], and D register

Translation to binary: simple!

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 $out \geq 0$ jump
1	0	0	JLT	If $out < 0$ jump
1	0	1	JNE	If $out \neq 0$ jump
1	1	0	JLE	If $out \leq 0$ jump
1	1	1	JMP	Jump

The overall assembly logic

Assembly program

```
// Computes 1+...+RAM[0]
// And stores 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
    @0
    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
    @1
    M=D  // RAM[1] = the sum
(END)
    @END
    0;JMP
```

<u>For each (real) command</u>

- Parse the command,
  i.e. break it into its underlying fields
- A-instruction: replace the symbolic reference (if any) with the corresponding memory address, which is a number

  (how to do it, later)
- 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.

Handling symbols (aka *symbol resolution*)

Assembly programs typically have many symbols:

- Labels that mark destinations of goto commands
- Labels that mark special memory locations
- Variables

These symbols fall into two categories:

- User–defined symbols (created by programmers)
- Pre-defined symbols (used by the Hack platform).

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: user-defined symbols

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
```

Label symbols: Used to label destinations of goto commands. Declared by the pseudo-command (XXX). This directive defines the symbol XXX to refer to the instruction memory location holding the next command in the program

Variable symbols: Any user-defined symbol xxx appearing in an assembly program that is not defined elsewhere using the (xxx) directive is treated as a variable, and is automatically assigned a unique RAM address, starting at RAM address 16

(why start at 16? Later.)

By convention, Hack programmers use lower-case and upper-case to represent variable and label names, respectively

Q: Who does all the "automatic" assignments of symbols to RAM addresses?

A: As part of the program translation process, the assembler resolves all the symbols into RAM addresses.

Handling symbols: pre-defined symbols

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
```

Virtual registers:
The symbols R0,…, R15 are automatically predefined to refer to RAM addresses 0,…,15

I/O pointers: The symbols SCREEN and KBD are automatically predefined to refer to RAM addresses 16384 and 24576, respectively (base addresses of the *screen* and *keyboard* memory maps)

VM control pointers: the symbols SP, LCL, ARG, THIS, and THAT (that don't appear in the code example on the right) are automatically predefined to refer to RAM addresses 0 to 4, respectively

(The VM control pointers, which overlap R0,…, R4 will come to play in the virtual machine implementation, covered in the next lecture)

Q: Who does all the "automatic" assignments of symbols to RAM addresses?

A: As part of the program translation process, the assembler resolves all the symbols into RAM addresses.

Handling symbols: symbol table

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
```

Symbol table

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

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

Handling symbols: constructing the symbol table

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
```

Symbol table

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

Initialization: create an empty symbol table and populate it with all the pre-defined symbols

First pass: go through the entire source code, and add all the user-defined label symbols to the symbol table (without generating any code)

Second pass: go again through the source code, and use the symbol table to translate all the commands. In the process, handle all the user-defined variable symbols.

The assembly process (detailed)

- Initialization: create the symbol table and initialize it with the pre-defined symbols
- First pass: march through the source code without generating any code.
  For each label declaration (LABEL) that appears in the source code,
  add the pair <LABEL , *n* > to the symbol table
- Second pass: march again through the source code, and process 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, look it up in the symbol table and proceed as follows:
    - If the symbol is found, replace it with its numeric value and complete the command's translation
    - If the symbol is not found, then it must represent a new variable:
      add the pair <xxx , *n* > to the symbol table, where *n* is the next available RAM address, and complete the command's translation.

      (Platform design decision: the allocated RAM addresses are running, starting at address 16).

The result ...

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
```

assemble

Target 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
```

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

Proposed assembler implementation

An assembler program can be written in any high-level language.

We propose a language-independent design, as follows.

<u>Software modules:</u>

- **`Parser:`** Unpacks each command into its underlying fields
- **`Code:`** Translates each field into its corresponding binary value, and assembles the resulting values
- **`SymbolTable:`** Manages the symbol table
- **`Main:`** Initializes I/O files and drives the show.

<u>Proposed implementation stages</u>

- Stage I: Build a basic assembler for programs with no symbols
- Stage II: Extend the basic assembler with symbol handling capabilities.

Parser (a software module in the assembler program)

Parser: Encapsulates access to the input code. Reads an assembly language command, parses it, and provides convenient access to the command's components (fields and symbols). In addition, removes all white space and comments.

Routine	**Arguments**	**Returns**	**Function**
Constructor / initializer	Input file / stream	--	Opens the input file/stream and gets ready to parse it.
`hasMoreCommands`	--	Boolean	Are there more commands in the input?
`advance`	--	--	Reads the next command from the input and makes it the current command. Should be called only if `hasMoreCommands()` is true. Initially there is no current command.
`commandType`	--	`A_COMMAND`, `C_COMMAND`, `L_COMMAND`	Returns the type of the current command: • `A_COMMAND` for `@Xxx` where `Xxx` is either a symbol or a decimal number • `C_COMMAND` for `dest=comp;jump` • `L_COMMAND` (actually, pseudo-command) for `(Xxx)` where `Xxx` is a symbol.

Parser (a software module in the assembler program) / continued

`symbol`	--	string	Returns the symbol or decimal `Xxx` of the current command `@Xxx` or `(Xxx)`. Should be called only when `commandType()` is `A_COMMAND` or `L_COMMAND`.
`dest`	--	string	Returns the dest mnemonic in the current *C*-command (8 possibilities). Should be called only when `commandType()` is `C_COMMAND`.
`comp`	--	string	Returns the comp mnemonic in the current *C*-command (28 possibilities). Should be called only when `commandType()` is `C_COMMAND`.
`jump`	--	string	Returns the jump mnemonic in the current *C*-command (8 possibilities). Should be called only when `commandType()` is `C_COMMAND`.

Code (a software module in the assembler program)

Code: Translates Hack assembly language mnemonics into binary codes.			
Routine	**Arguments**	**Returns**	**Function**
`dest`	mnemonic (string)	3 bits	Returns the binary code of the `dest` mnemonic.
`comp`	mnemonic (string)	7 bits	Returns the binary code of the `comp` mnemonic.
`jump`	mnemonic (string)	3 bits	Returns the binary code of the `jump` mnemonic.

SymbolTable (a software module in the assembler program)

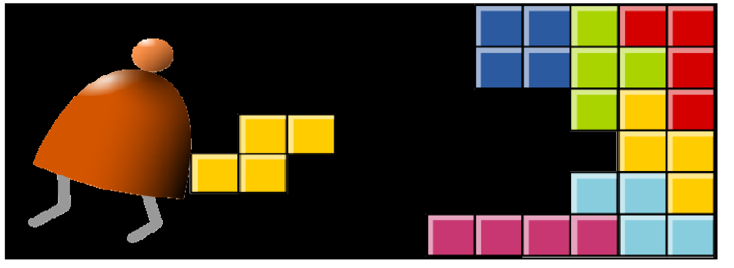
SymbolTable: A symbol table that keeps a correspondence between symbolic labels and numeric addresses.			
Routine	**Arguments**	**Returns**	**Function**
`Constructor`	--	--	Creates a new empty symbol table.
`addEntry`	`symbol` (string), `address` (int)	--	Adds the pair `(symbol,address)` to the table.
`contains`	`symbol` (string)	Boolean	Does the symbol table contain the given `symbol`?
`GetAddress`	`symbol` (string)	int	Returns the address associated with the `symbol`.

Perspective

- Simple machine language, simple assembler
- Most assemblers are not stand-alone, but rather encapsulated in a translator of a higher order
- C programmers that understand the code generated by a C compiler can improve their code considerably
- C programming (e.g. for real-time systems) may involve re-writing critical segments in assembly, for optimization
- Writing an assembler is an excellent practice for writing more challenging translators, e.g. a VM Translator and a compiler, as we will do in the next lectures.

Virtual Machine

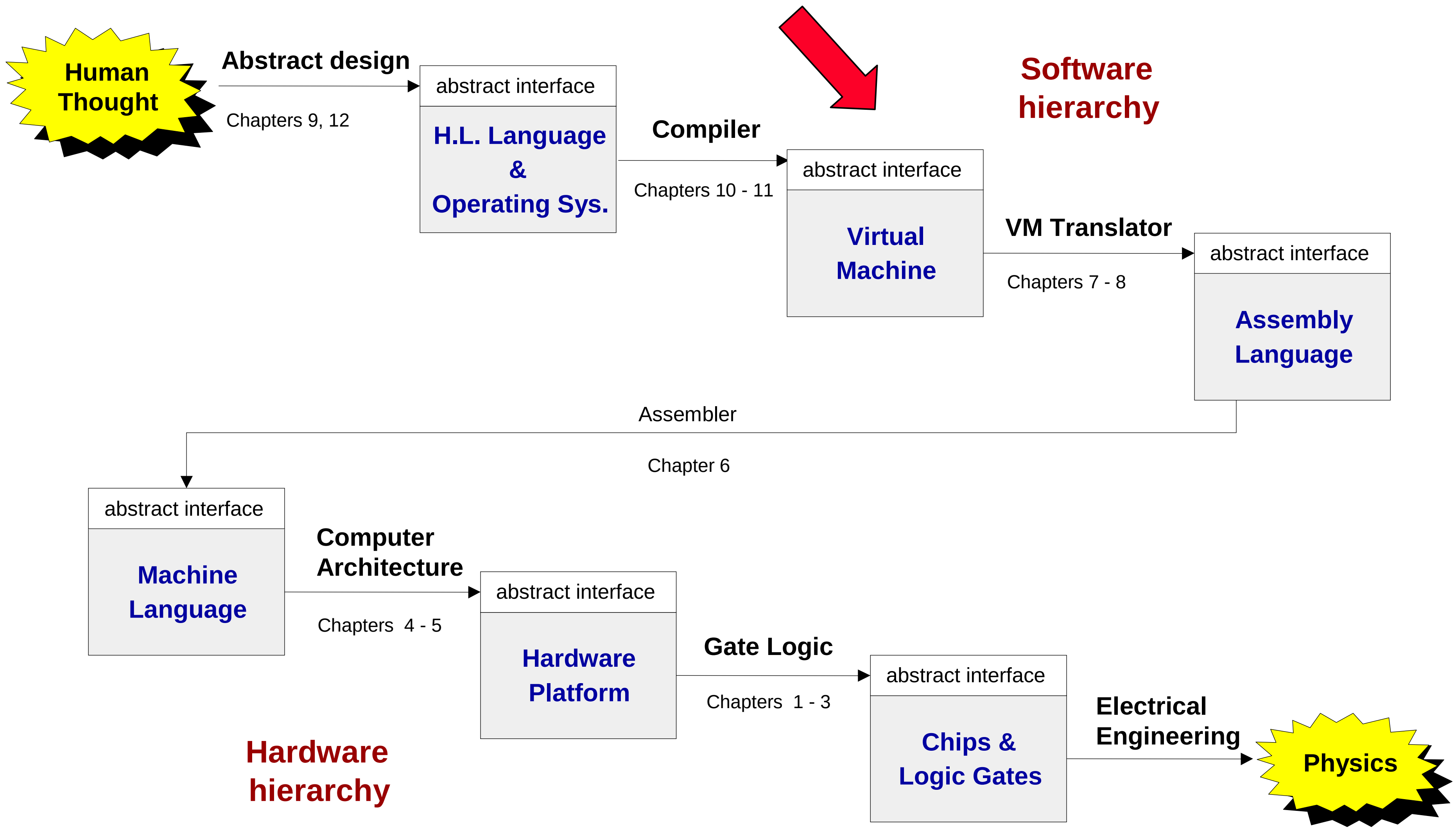
Part I: Stack Arithmetic



Building a Modern Computer From First Principles

www.nand2tetris.org

Where we are at:



Motivation

Jack code (example)

```
class Main {
  static int x;

  function void main() {
    // Inputs and multiplies two numbers
    var int a, b, x;
    let a = Keyboard.readInt("Enter a number");
    let b = Keyboard.readInt("Enter a number");
    let x = mult(a,b);
    return;
  }
}

  // Multiplies two numbers.
  function int mult(int x, int y) {
    var int result, j;
    let result = 0; let j = y;
    while ~(j = 0) {
      let result = result + x;
      let j = j - 1;
    }
    return result;
  }
}
```

<u>Our ultimate goal:</u>

Translate high-level programs into executable code.

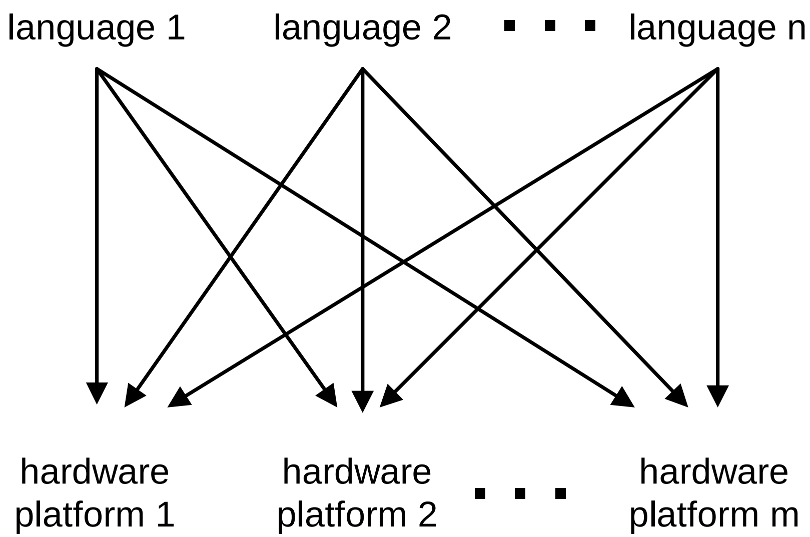
Compiler

Hack code

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

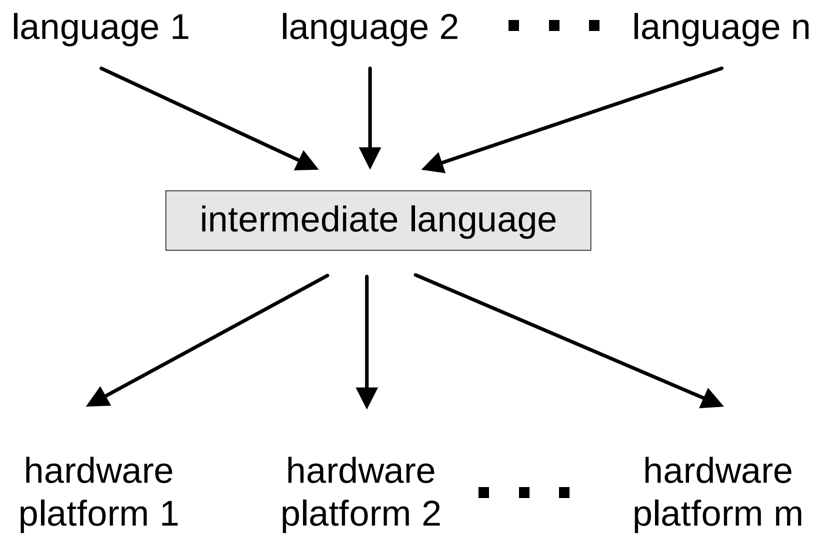
Compilation models

direct compilation:



requires $n \cdot m$ translators

2-tier compilation:

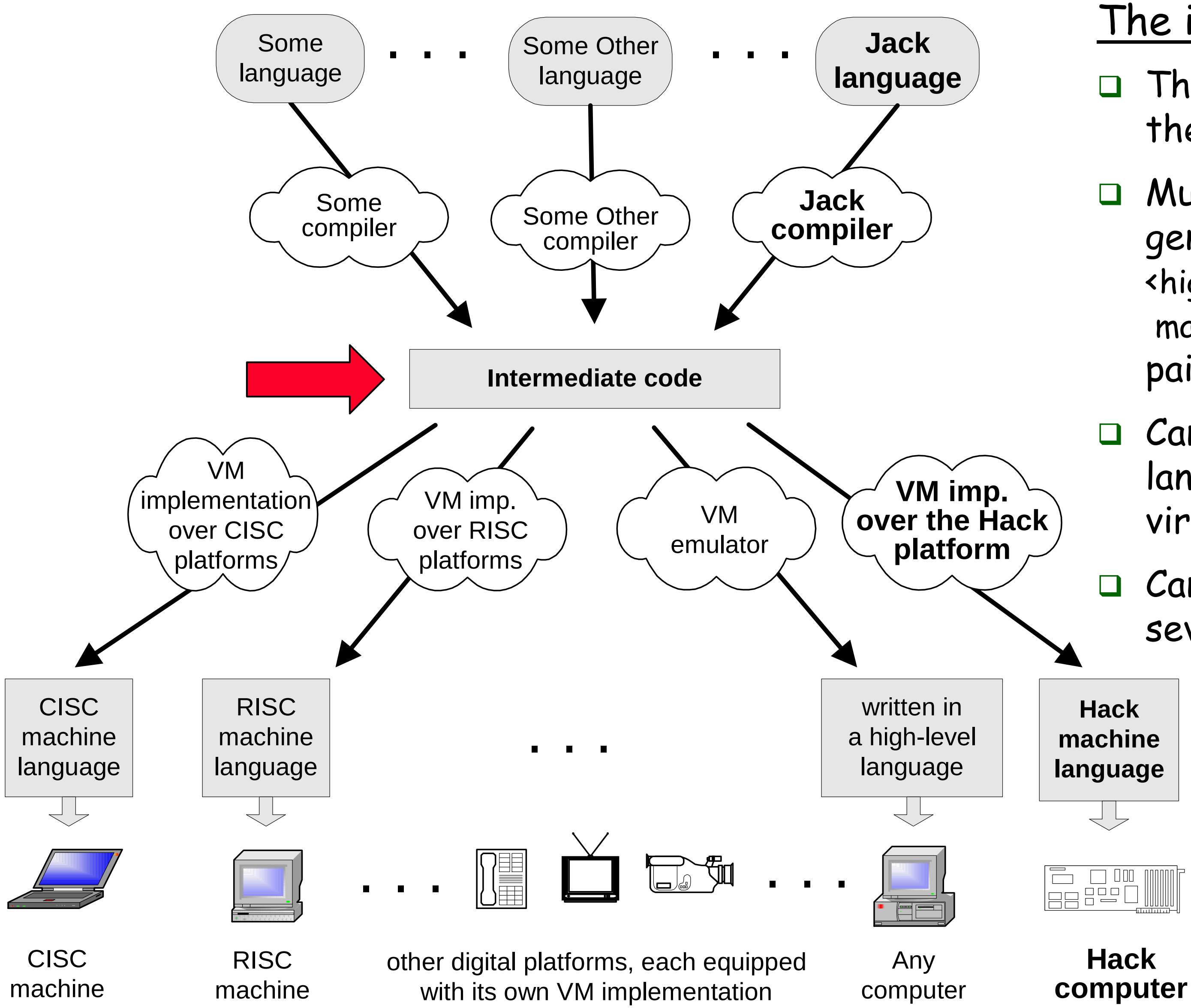


requires $n + m$ translators

Two-tier compilation:

- First compilation stage: depends only on the details of the source language
- Second compilation stage: depends only on the details of the target language.

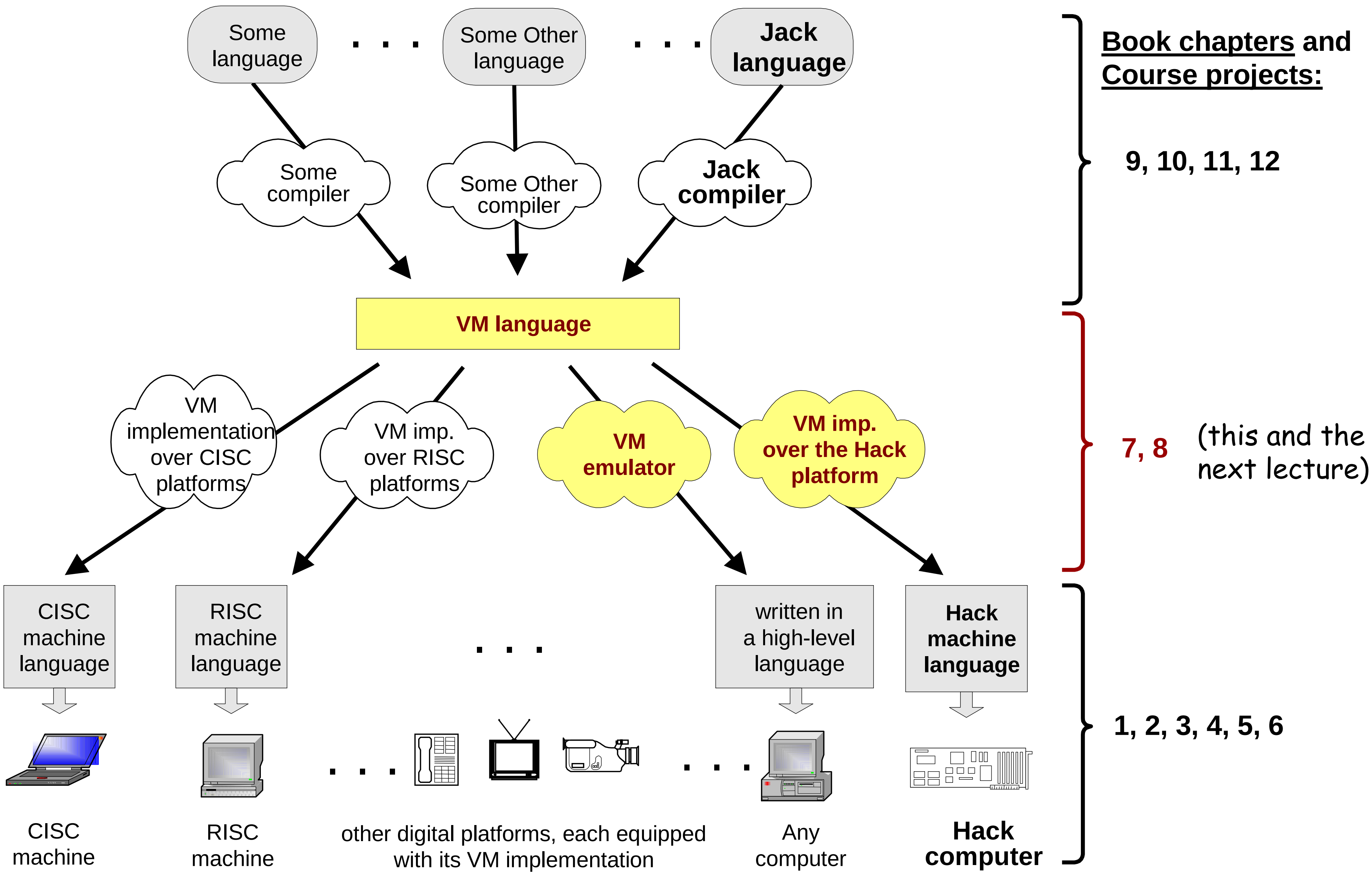
The big picture



The intermediate code:

- The interface between the 2 compilation stages
- Must be sufficiently general to support many <high-level language, machine-language> pairs
- Can be modeled as the language of an abstract virtual machine (VM)
- Can be implemented in several different ways.

Focus of this lecture (yellow):



The VM model and language

Perspective:

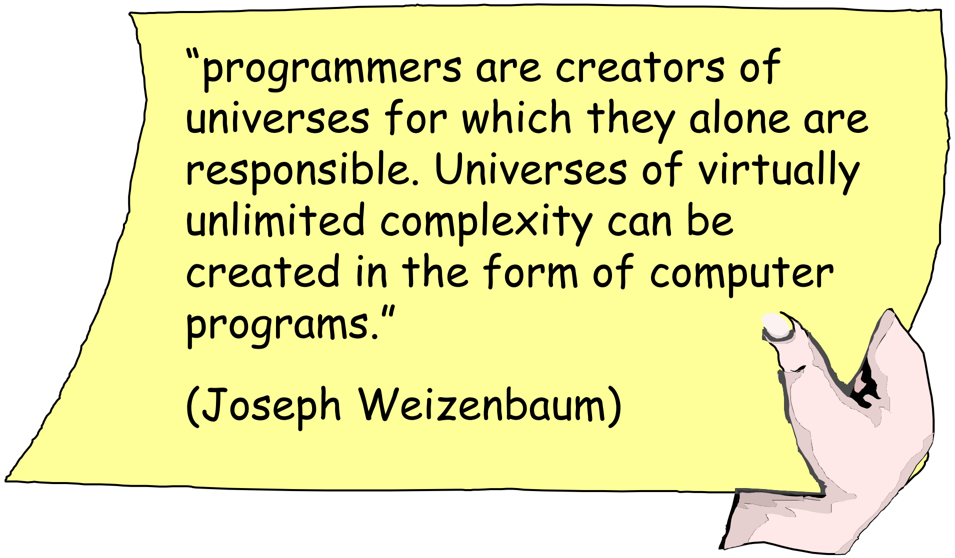
From here till the end of the next lecture we describe the VM model used in the Hack-Jack platform

Other VM models (like Java's JVM/JRE and .NET's IL/CLR) are similar in spirit but differ in scope and details.

Several different ways to think about the notion of a virtual machine:

- **Abstract software engineering view:**
  the VM is an interesting abstraction that makes sense in its own right
- **Practical software engineering view:**
  the VM code layer enables "managed code" (e.g. enhanced security)
- **Pragmatic compiler writing view:**
  a VM architecture makes writing a compiler much easier
  (as we'll see later in the course)
- **Opportunistic empire builder view:**
  a VM architecture allows writing high-level code once and have it run on many target platforms with little or no modification.

Yet another view (poetic)



Our VM model + language are an example of one such universe.

Lecture plan

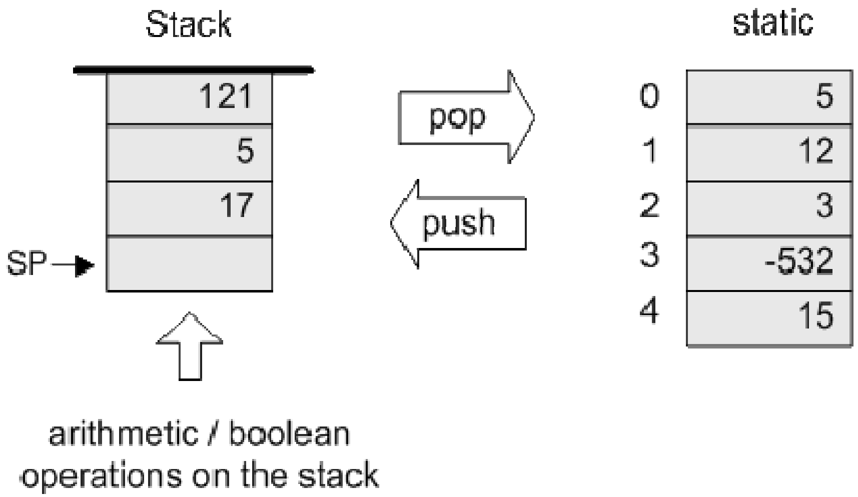
Goal: Specify and implement a VM model and language:

Arithmetic / Boolean commands	Program flow commands
`add`	`label` (declaration)
`sub`	`goto` (label)
`neg`	`if-goto` (label)
`eq`	Next lecture
`gt`	
`lt`	**Function calling commands**
`and`	`function` (declaration)
`or`	`call` (a function)
`not`	`return` (from a function)
Memory access commands	
`pop x` (pop into x, which is a variable)	
`push y` (y being a variable or a constant)	
This lecture	

Our game plan: (a) describe the VM abstraction (above)
(b) propose how to implement it over the Hack platform.

Our VM model is *stack-oriented*

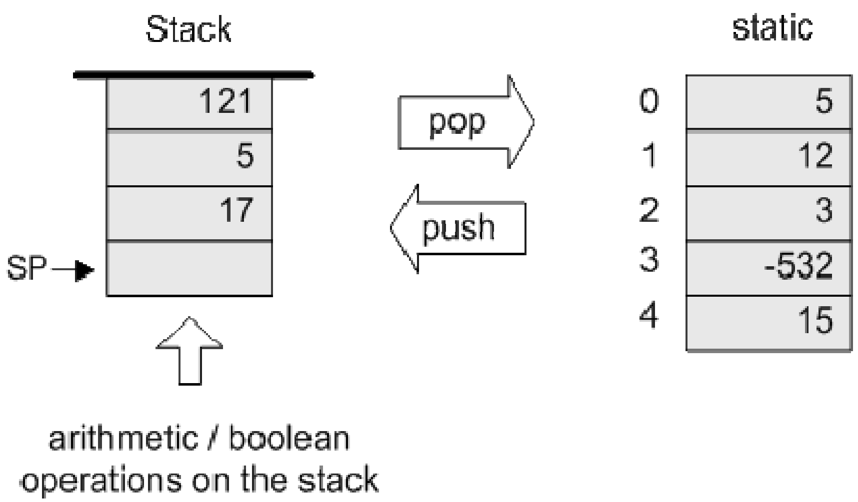
- All operations are done on a stack
- Data is saved in several separate *memory segments*
- All the memory segments behave the same
- One of the memory segments m is called `static`, and we will use it (as an arbitrary example) in the following examples:



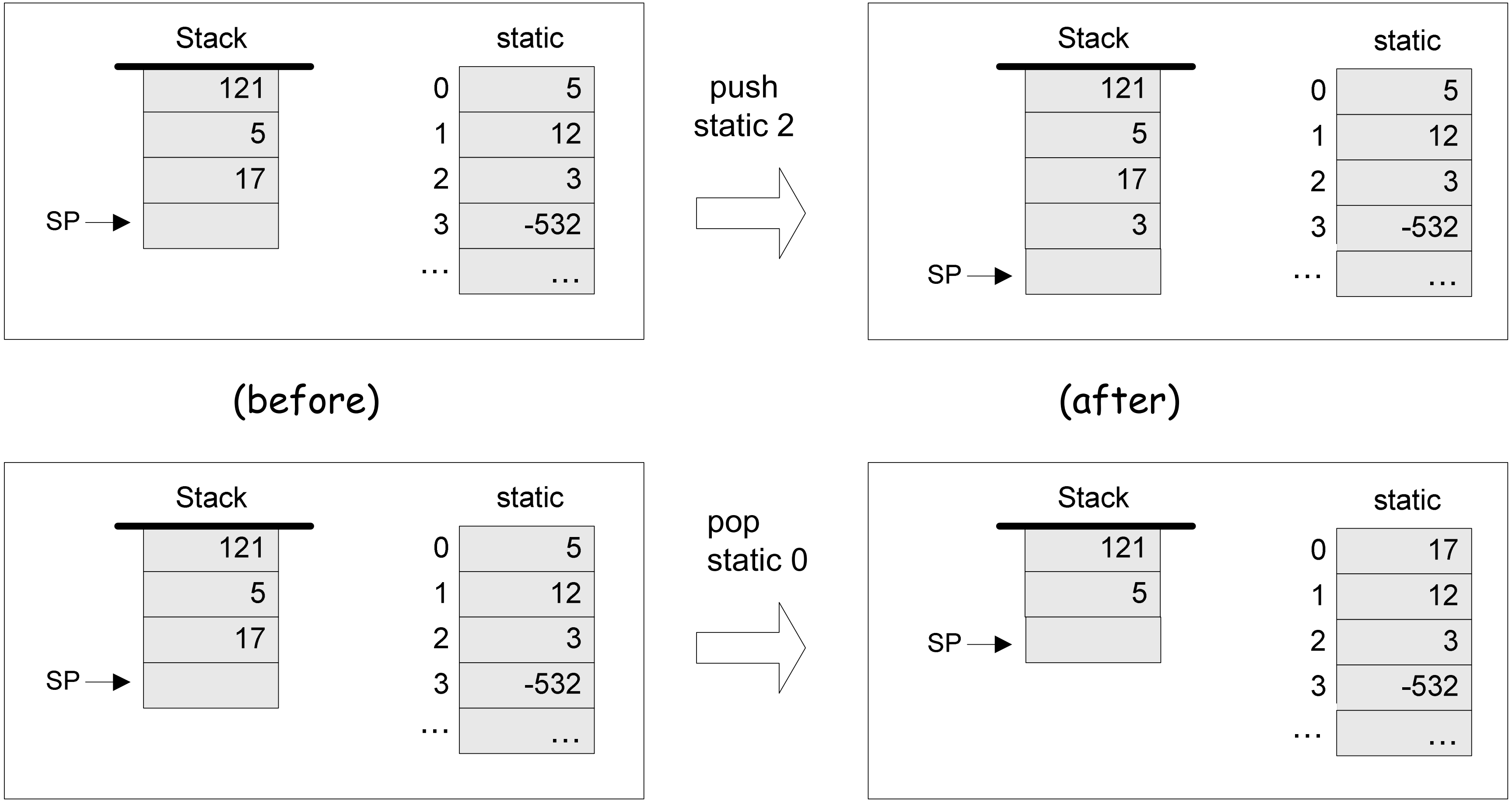
Data types

Our VM model features a single 16-bit data type that can be used as:

- an integer value (16-bit 2's complement: `-32768`, ... , `32767`)
- a Boolean value (`0` and `-1`, standing for `true` and `false`)
- a pointer (memory address)



Memory access operations



<u>The stack:</u>

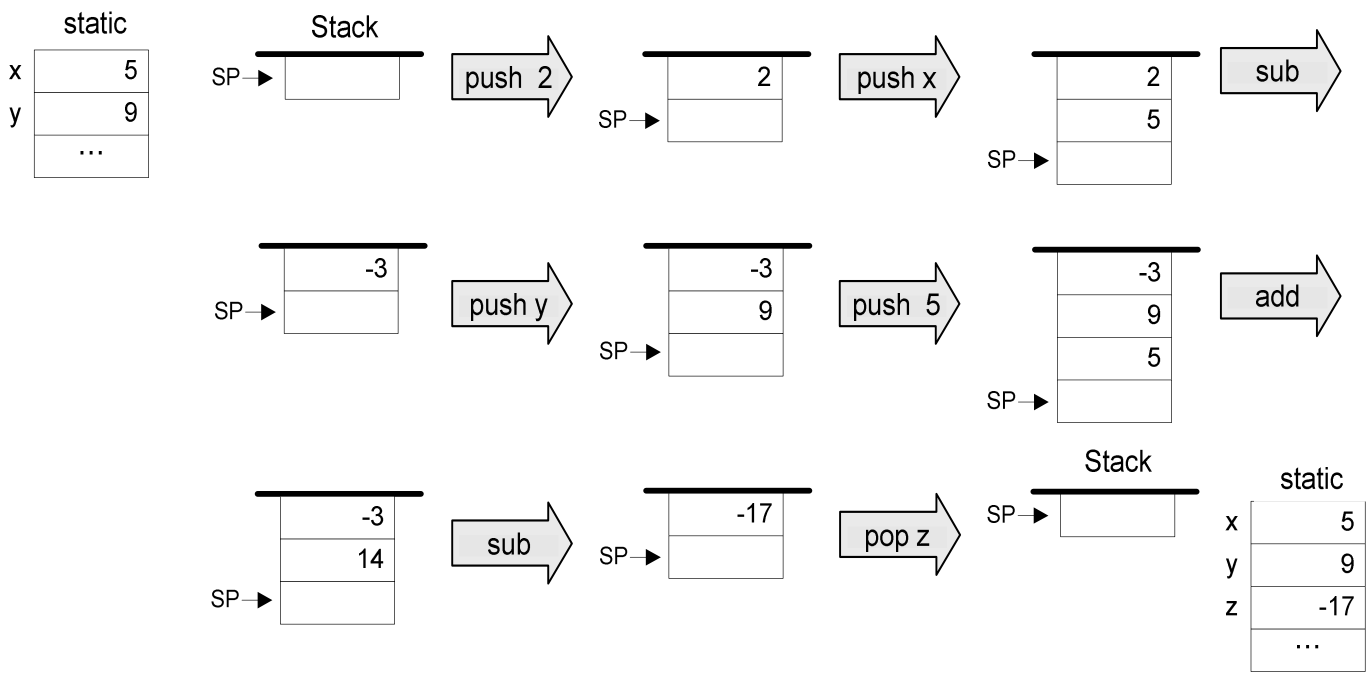
- A classical LIFO data structure
- Elegant and powerful
- Several hardware / software implementation options.

Evaluation of arithmetic expressions

VM code (example)

```
// z=(2-x)-(y+5)
push 2
push x
sub
push y
push 5
add
sub
pop z
```

(suppose that
x refers to static 0,
y refers to static 1, and
z refers to static 2)

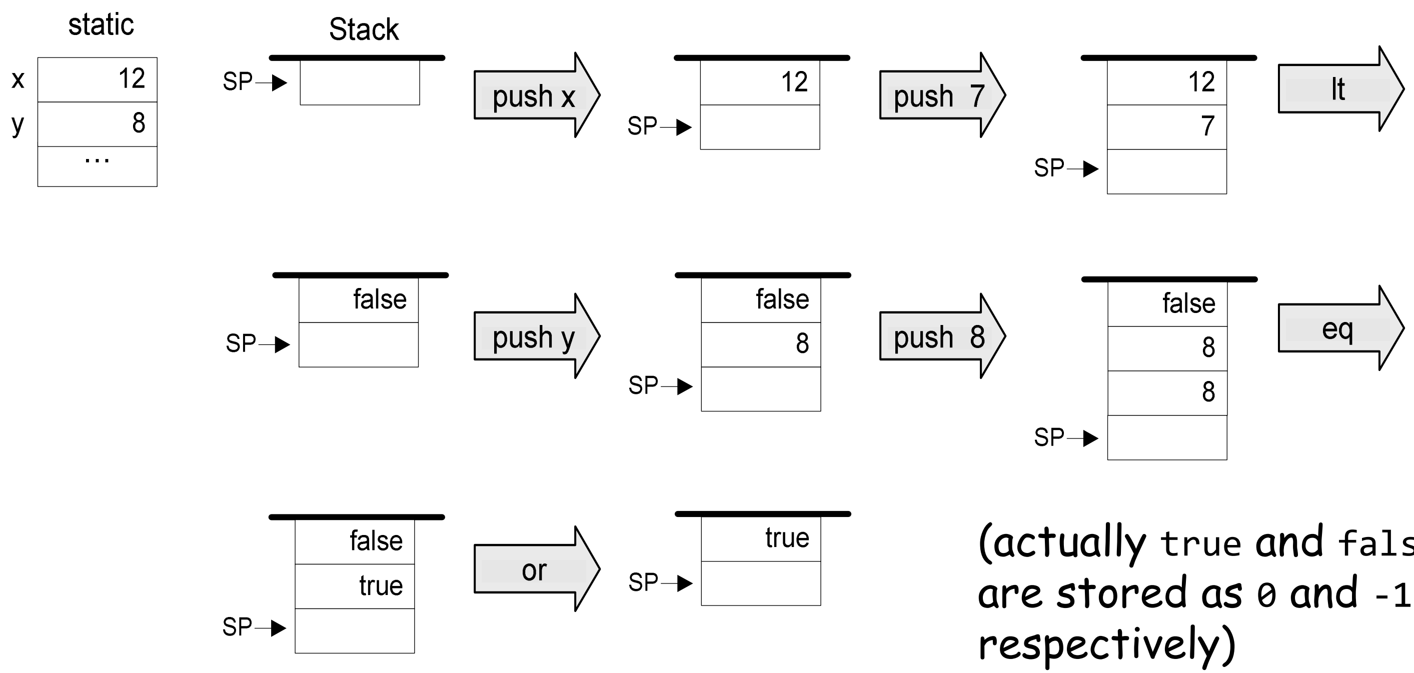


Evaluation of Boolean expressions

VM code (example)

```
// (x<7) or (y=8)
push x
push 7
lt
push y
push 8
eq
or
```

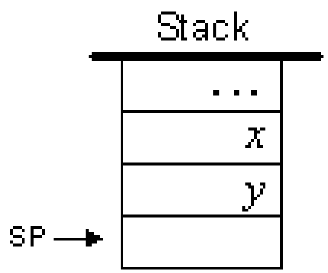
(suppose that
x refers to static 0, and
y refers to static 1)



(actually true and false are stored as 0 and -1, respectively)

Arithmetic and Boolean commands in the VM language (wrap-up)

Command	Return value (after popping the operand/s)	Comment	
`add`	$x+y$	Integer addition	(2's complement)
`sub`	$x-y$	Integer subtraction	(2's complement)
`neg`	$-y$	Arithmetic negation	(2's complement)
`eq`	true if $x=y$ and false otherwise	Equality	
`gt`	true if $x>y$ and false otherwise	Greater than	
`lt`	true if $x<y$ and false otherwise	Less than	
`and`	x And y	Bit-wise	
`or`	x Or y	Bit-wise	
`not`	Not y	Bit-wise	



The VM's Memory segments

A VM program is designed to provide an interim abstraction of a program written in some high-level language

Modern OO high-level languages normally feature the following variable kinds:

Class level:

- Static variables (class-level variables)
- Private variables (aka "object variables" / "fields" / "properties")

Method level:

- Local variables
- Argument variables

When translated into the VM language,

The static, private, local and argument variables are mapped by the compiler on the four memory segments `static`, `this`, `local`, `argument`

In addition, there are four additional memory segments, whose role will be presented later: `that`, `constant`, `pointer`, `temp`.

Memory segments and memory access commands

The VM abstraction includes 8 separate memory segments named:

`static`, `this`, `local`, `argument`, `that`, `constant`, `pointer`, `temp`

As far as VM programming commands go, all memory segments look and behave the same

To access a particular segment entry, use the following generic syntax:

Memory access VM commands:

- pop *memorySegment index*
- push *memorySegment index*

Where *memorySegment* is `static`, `this`, `local`, `argument`, `that`, `constant`, `pointer`, or `temp`

And *index* is a non-negative integer

Notes:

(In all our code examples thus far, *memorySegment* was `static`)

The different roles of the eight memory segments will become relevant when we'll talk about the compiler

At the VM abstraction level, all memory segments are treated the same way.

VM programming

VM programs are normally written by *compilers*, not by humans

However, compilers are written by humans ...

In order to write or optimize a compiler, it helps to first understand the spirit of the compiler's target language – the VM language

So, we'll now see an example of a VM program

The example includes three new VM commands:

- `function` *functionSymbol* // function declaration
- `label` *labelSymbol* // label declaration
- `if-goto` *labelSymbol* // `pop x`
  // if `x=true`, jump to execute the command after *labelSymbol*
  // else proceed to execute the next command in the program

For example, to effect `if (x > n) goto loop`, we can use the following VM commands:

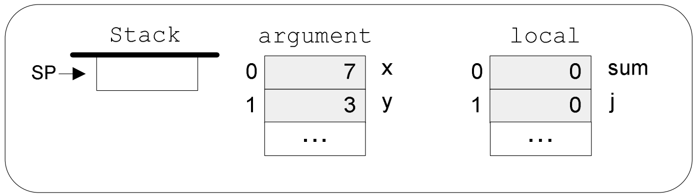
```
push x
push n
gt
if-goto loop            // Note that x, n, and the truth value were removed from the stack.
```

VM programming (example)

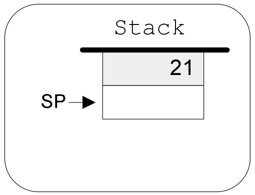
High-level code

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

Just after mult(7,3) is entered:



Just after mult(7,3) returns:



VM code (first approx.)

```
function mult(x,y)
   push 0
   pop result
   push y
   pop j
label loop
   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

```
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
```

VM programming: multiple functions

Compilation:

- A Jack application is a set of 1 or more class files (just like `.java` files).
- When we apply the Jack compiler to these files, the compiler creates a set of 1 or more `.vm` files (just like `.class` files). Each method in the Jack app is translated into a VM function written in the VM language
- Thus, a VM file consists of one or more VM functions.

Execution:

- At any given point of time, only one VM function is executing (the "current function"), while 0 or more functions are waiting for it to terminate (the functions up the "calling hierarchy")
- For example, a `main` function starts running; at some point we may reach the command `call factorial`, at which point the `factorial` function starts running; then we may reach the command `call mult`, at which point the `mult` function starts running, while both `main` and `factorial` are waiting for it to terminate

`The stack:` a global data structure, used to save and restore the resources (memory segments) of all the VM functions up the calling hierarchy (e.g. `main` and `factorial`). The tip of this stack if the working stack of the current function (e.g. `mult`).

Lecture plan

Goal: Specify and implement a VM model and language:

Arithmetic / Boolean commands	Program flow commands
`add`	`label` (declaration)
`sub`	`goto` (label)
`neg`	`if-goto` (label)
`eq`	Next lecture
`gt` This lecture	**Function calling commands**
`lt`	`function` (declaration)
`and`	`call` (a function)
`or`	`return` (from a function)
`not`	
Memory access commands	
`pop x` (pop into x, which is a variable)	
`push y` (y being a variable or a constant)	

Method: (a) specify the abstraction (stack, memory segments, commands)

→ (b) propose how to implement the abstraction over the Hack platform.

Implementation

VM implementation options:

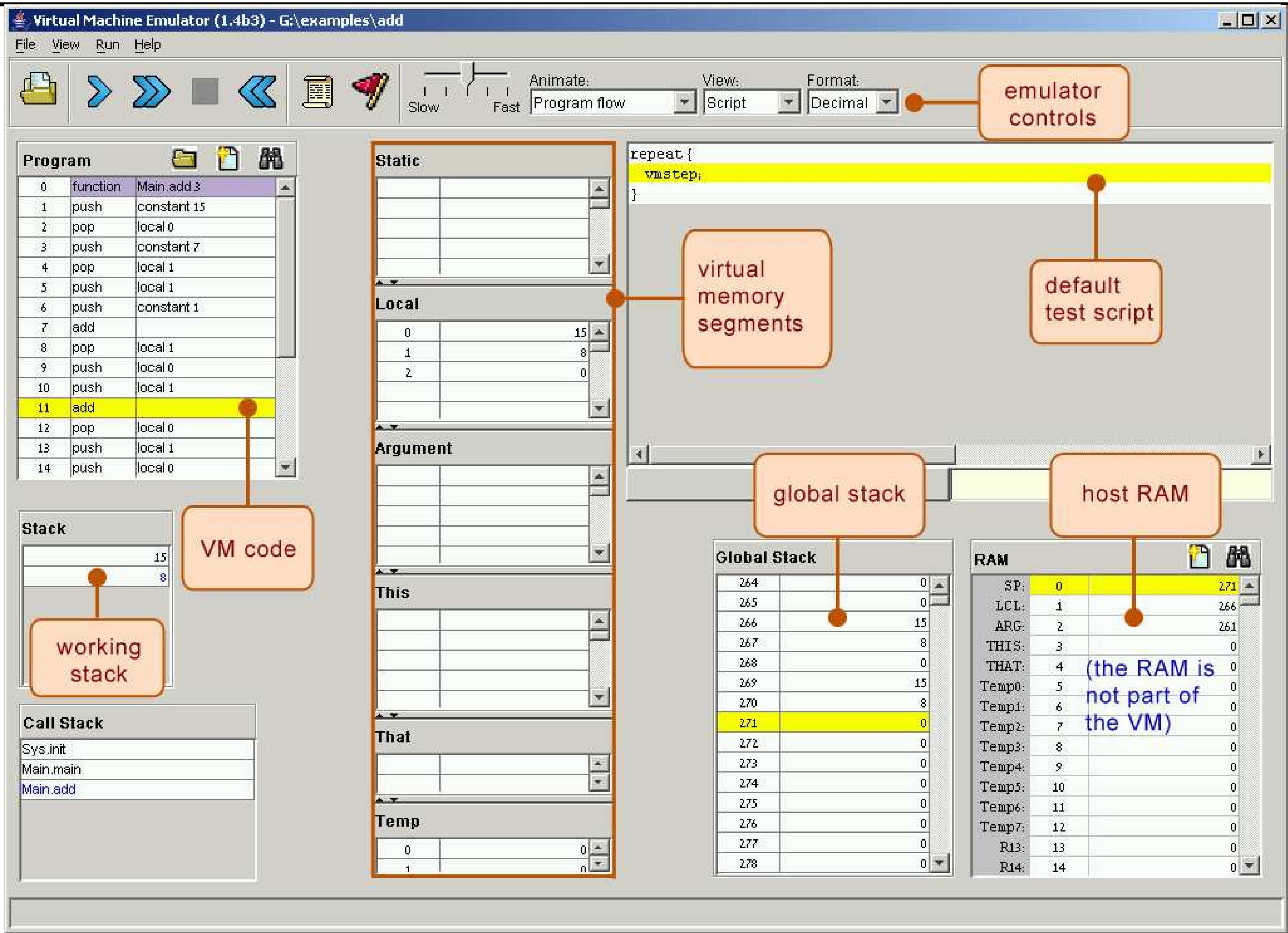
- Software-based (e.g. emulate the VM model using Java)
- Translator-based (e. g. translate VM programs into the Hack machine language)
- Hardware-based (realize the VM model using dedicated memory and registers)

Two well-known translator-based implementations:

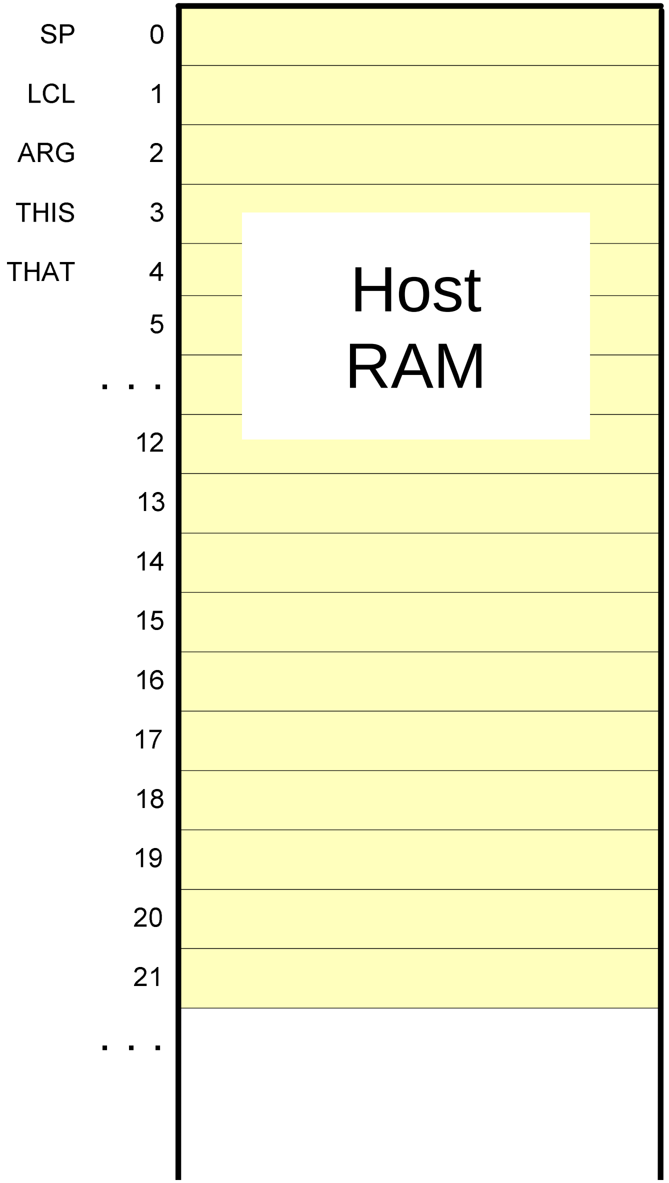
JVM: Javac translates Java programs into bytecode;
The JVM translates the bytecode into
the machine language of the host computer

CLR: C# compiler translates C# programs into IL code;
The CLR translated the IL code into
the machine language of the host computer.

Software implementation: Our VM emulator (part of the course software suite)



VM implementation on the Hack platform



The stack: a global data structure, used to save and restore the resources of all the VM functions up the calling hierarchy.

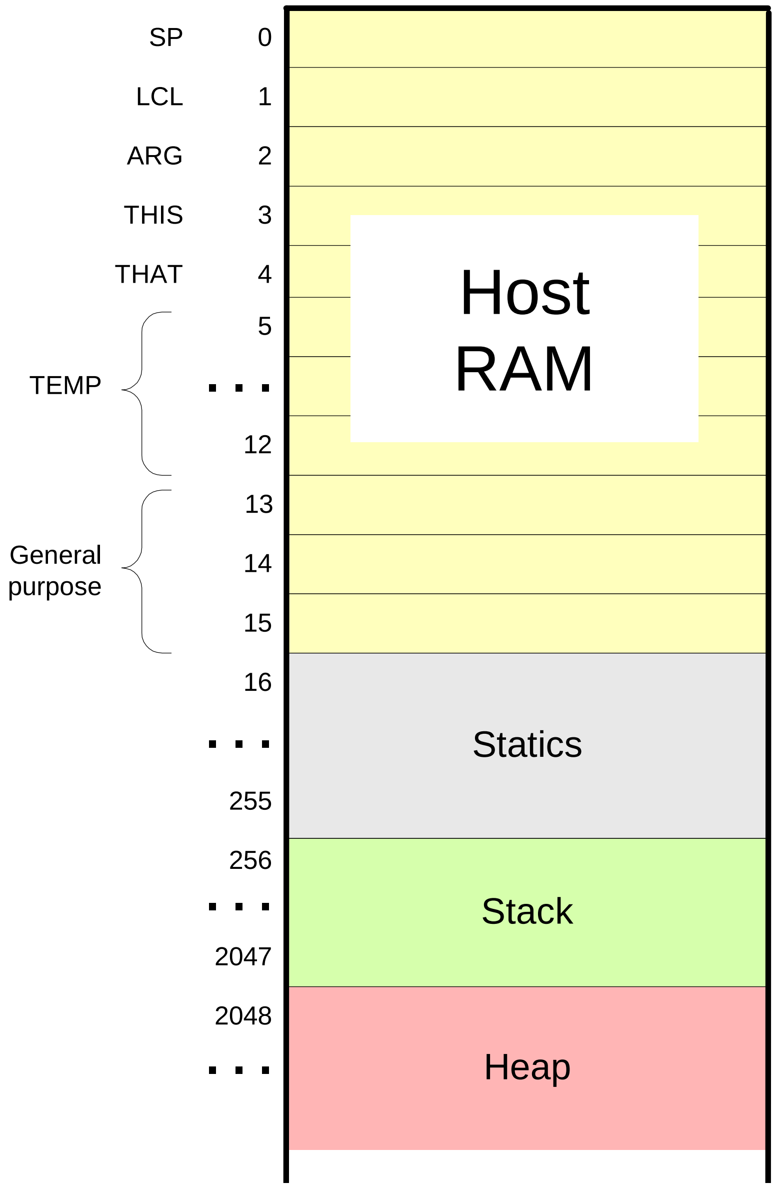
The tip of this stack if the working stack of the current function

`static, constant, temp, pointer`:
Global memory segments, all functions see the same four segments

`local,argument,this,that`:
these segments are local at the function level; each function sees its own, private copy of each one of these four segments

The challenge:
represent all these logical constructs on the same single physical address space -- the host RAM.

VM implementation on the Hack platform



Basic idea: the mapping of the stack and the global segments on the RAM is easy (fixed); the mapping of the function-level segments is dynamic, using pointers

The stack: mapped on `RAM[256 ... 2047]`;
The stack pointer is kept in RAM address `SP`

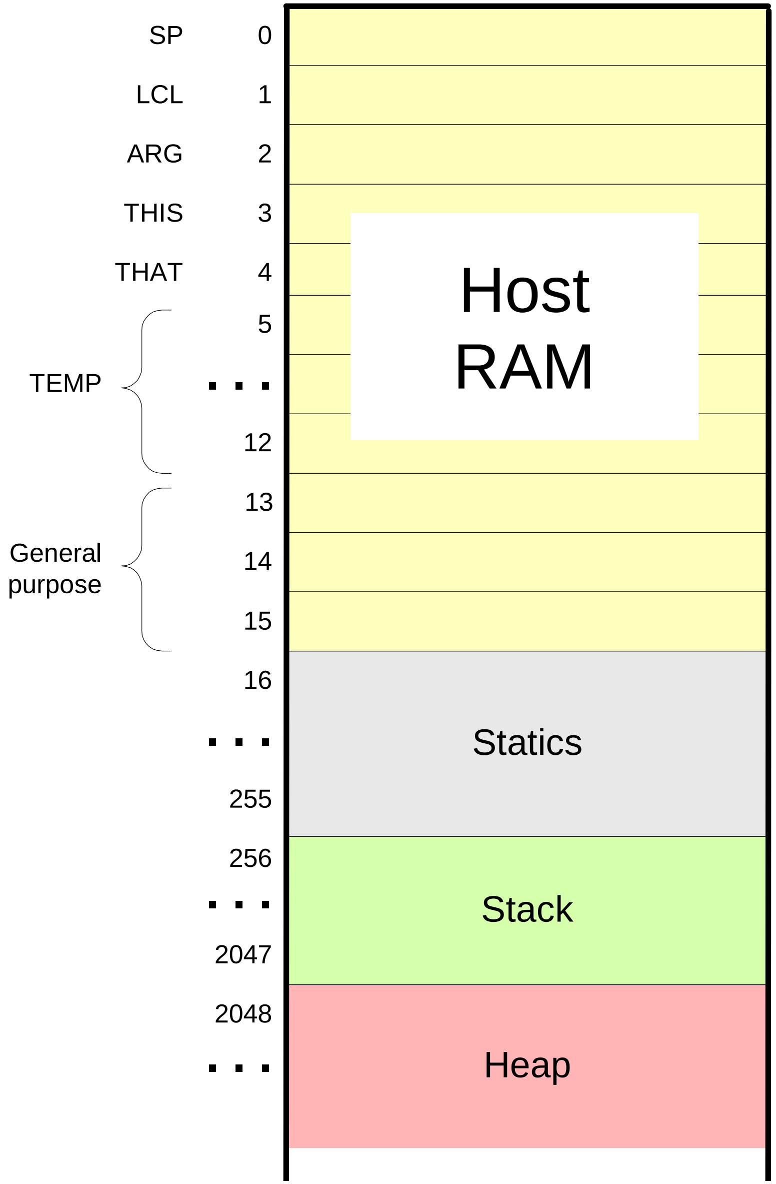
static: mapped on `RAM[16 ... 255]`;
each segment reference `static` *i* appearing in a VM file named `f` is compiled to the assembly language symbol `f.i` (recall that the assembler further maps such symbols to the RAM, from address 16 onward)

local,argument,this,that: these method-level segments are mapped somewhere from address 2048 onward, in an area called "heap". The base addresses of these segments are kept in RAM addresses `LCL`, `ARG`, `THIS`, and `THAT`. Access to the *i*-th entry of any of these segments is implemented by accessing `RAM[segmentBase +` *i*`]`

constant: a truly a virtual segment:
access to `constant` *i* is implemented by supplying the constant *i*.

pointer: discussed later.

VM implementation on the Hack platform



Practice exercises

Now that we know how the memory segments are mapped on the host `RAM`, we can write Hack commands that realize the various VM commands. for example, let us write the Hack code that implements the following VM commands:

- `push constant 1`
- `pop static 7` (suppose it appears in a VM file named `f`)
- `push constant 5`
- `add`
- `pop local 2`
- `eq`

Tips:

1. The implementation of any one of these VM commands requires several Hack assembly commands involving pointer arithmetic (using commands like `A=M`)
2. If you run out of registers (you have only two ...), you may use `R13`, `R14`, and `R15`.

Proposed VM translator implementation: Parser module

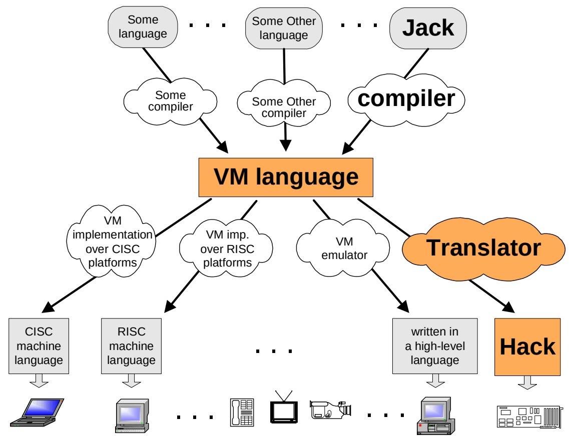
Parser: Handles the parsing of a single `.vm` file, and encapsulates access to the input code. It reads VM commands, parses them, and provides convenient access to their components. In addition, it removes all white space and comments.

Routine	**Arguments**	**Returns**	**Function**
Constructor	Input file / stream	--	Opens the input file/stream and gets ready to parse it.
`hasMoreCommands`	--	boolean	Are there more commands in the input?
`advance`	--	--	Reads the next command from the input and makes it the current command. Should be called only if `hasMoreCommands` is true. Initially there is no current command.
`commandType`	--	`C_ARITHMETIC, C_PUSH, C_POP, C_LABEL, C_GOTO, C_IF, C_FUNCTION, C_RETURN, C_CALL`	Returns the type of the current VM command. `C_ARITHMETIC` is returned for all the arithmetic commands.
`arg1`	--	string	Returns the first arg. of the current command. In the case of `C_ARITHMETIC`, the command itself (`add`, `sub`, etc.) is returned. Should not be called if the current command is `C_RETURN`.
`arg2`	--	int	Returns the second argument of the current command. Should be called only if the current command is `C_PUSH`, `C_POP`, `C_FUNCTION`, or `C_CALL`.

Proposed VM translator implementation: CodeWriter module

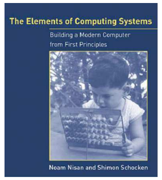
CodeWriter: Translates VM commands into Hack assembly code.			
Routine	**Arguments**	**Returns**	**Function**
Constructor	Output file / stream	--	Opens the output file/stream and gets ready to write into it.
`setFileName`	fileName (string)	--	Informs the code writer that the translation of a new VM file is started.
`writeArithmetic`	command (string)	--	Writes the assembly code that is the translation of the given arithmetic command.
`WritePushPop`	command (`C_PUSH` or `C_POP`), segment (string), index (int)	--	Writes the assembly code that is the translation of the given `command`, where `command` is either `C_PUSH` or `C_POP`.
`Close`	--	--	Closes the output file.
Comment: More routines will be added to this module in the next lecture / chapter 8.			

Perspective



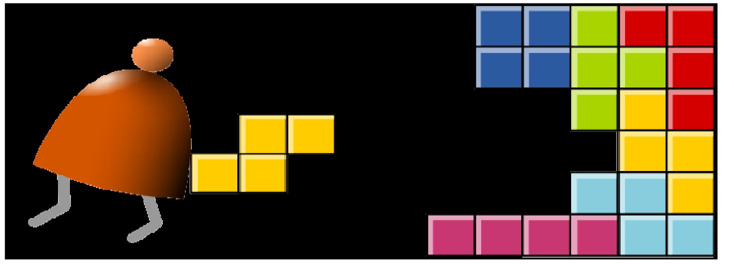
- In this lecture we began the process of building a compiler
- Modern compiler architecture:
  - Front-end (translates from a high-level language to a VM language)
  - Back-end (translates from the VM language to the machine language of some target hardware platform)
- Brief history of virtual machines:
  - 1970's: p-Code
  - 1990's: Java's JVM
  - 2000's: Microsoft .NET
- A full blown VM implementation typically also includes a common software library (can be viewed as a mini, portable OS).
- We will build such a mini OS later in the course.

The big picture

Java	Microsoft .net		
❑ JVM	❑ CLR	❑ VM	❑ 7, 8
❑ Java	❑ C#	❑ Jack	❑ 9
❑ Java compiler	❑ C# compiler	❑ Jack compiler	❑ 10, 11
❑ JRE	❑ .NET base class library	❑ Mini OS	❑ 12
			(Book chapters and Course projects)

Virtual Machine

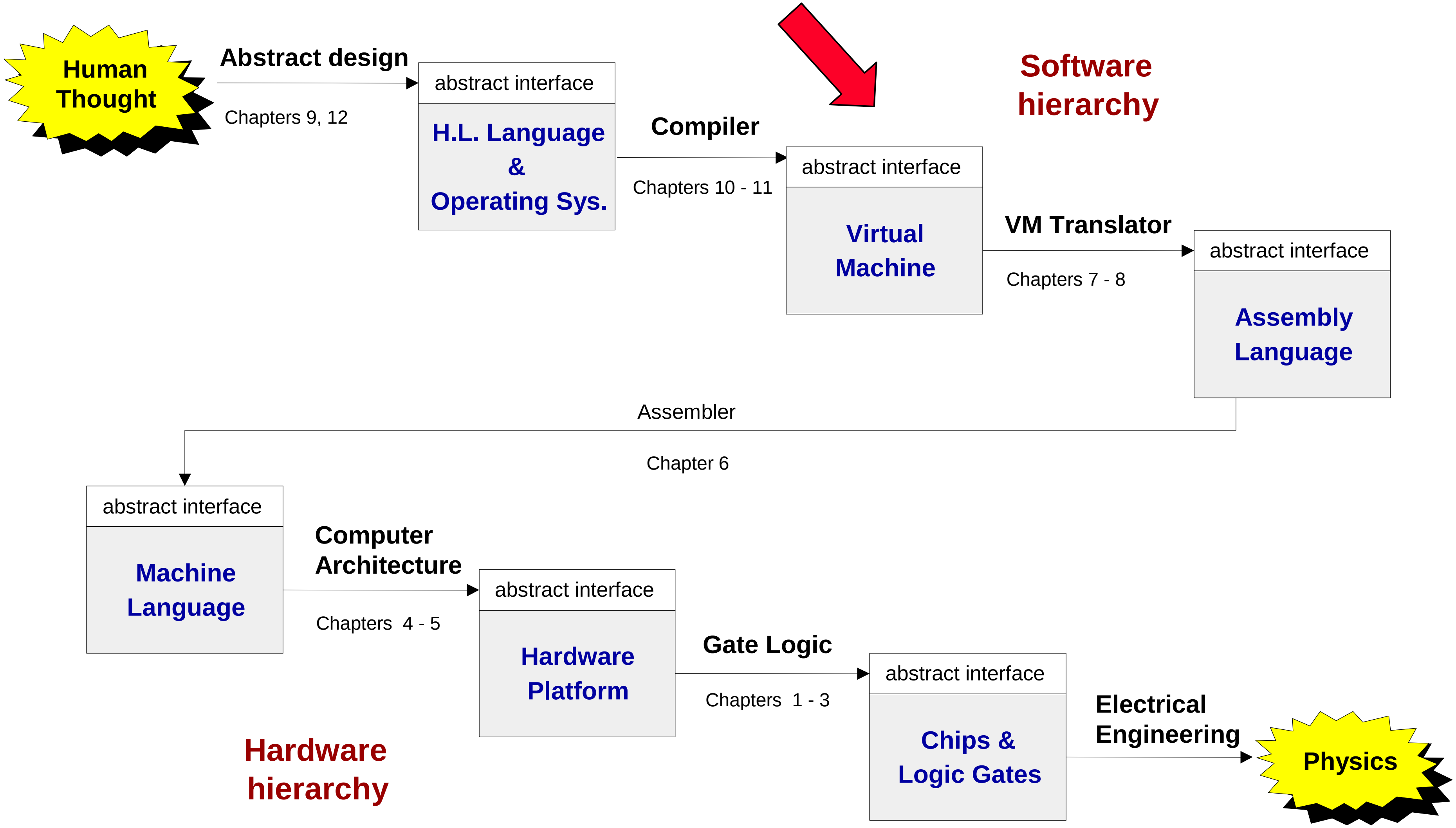
Part II: Program Control



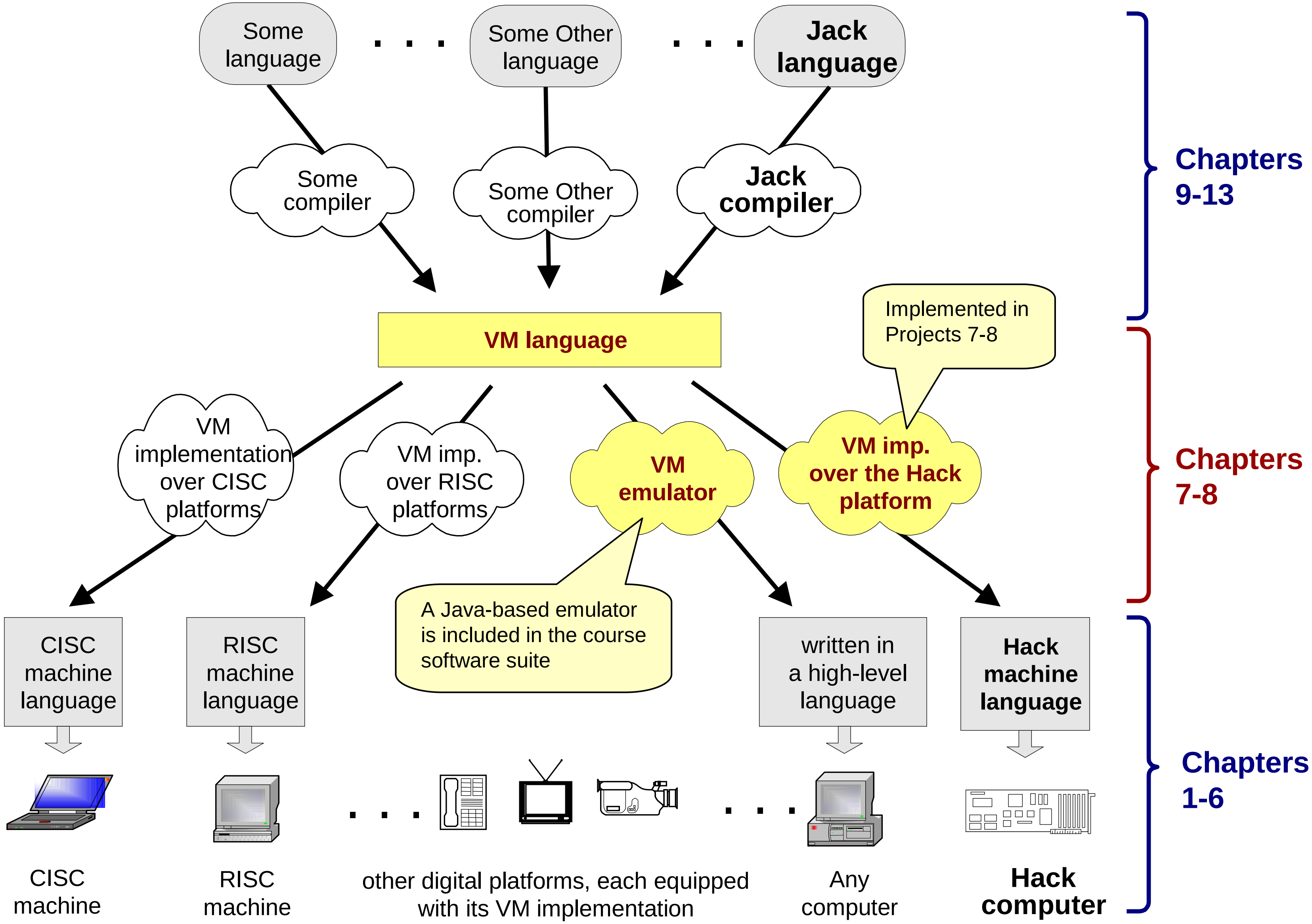
Building a Modern Computer From First Principles

www.nand2tetris.org

Where we are at:



The big picture



The VM langauge

Goal: Complete the specification and implementation of the VM model and language

Arithmetic / Boolean commands	Program flow commands
`add`	`label` (declaration)
`sub`	`goto` (label)
`neg`	`if-goto` (label)
`eq`	
`gt`	
`lt`	**Function calling commands**
`and`	`function` (declaration)
`or`	`call` (a function)
`not`	`return` (from a function)
Memory access commands	
`pop x` (pop into x, which is a variable)	
`push y` (y being a variable or a constant)	

previous lecture

this lecture

Method: (a) specify the abstraction (model's constructs and commands)
(b) propose how to implement it over the Hack platform.

The compilation challenge

Source code (high-level language)

```
class Main {
  static int x;

  function void main() {
    // Inputs and multiplies two numbers
    var int a, b, c;
    let a = Keyboard.readInt(“Enter a number”);
    let b = Keyboard.readInt(“Enter a number”);
    let c = Keyboard.readInt(“Enter a number”);
    let x = solve(a,b,c);
    return;
  }
}

  // Solves a quadearic equation (sort of)
  function int solve(int a, int b, int c) {
     var int x;
     if (~(a = 0))
        x=(-b+sqrt(b*b–4*a*c))/(2*a);
     else
        x=-c/b;
     return x;
  }
}
```

Our ultimate goal:

Translate high-level programs into executable code.

Compiler

Target code

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

The compilation challenge / two-tier setting

Jack source code

```
if (~(a = 0))
   x = (-b+sqrt(b*b–4*a*c))/(2*a)
else
   x = -c/b
```

Compiler

VM (pseudo) code

```
   push a
   push 0
   eq
   if-goto elseLabel
   push b
   neg
   push b
   push b
   call mult
   push 4
   push a
   call mult
   push c
   call mult
   call sqrt
   add
   push 2
   push a
   call mult
   div
   pop x
   goto contLable
elseLabel:
   push c
   neg
   push b
   call div
   pop x
contLable:
```

VM translator

Machine code

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

- We'll develop the compiler later in the course
- We now turn to describe how to complete the implementation of the VM language
- That is -- how to translate each VM command into assembly commands that perform the desired semantics.

The compilation challenge

Typical compiler's source code input:

```
// Computes x = (-b + sqrt(b^2 -4*a*c)) / 2*a
if (~(a = 0))
   x = (-b + sqrt(b * b – 4 * a * c)) / (2 * a)
else
   x = - c / b
```

program flow logic (branching)

(this lecture)

boolean expressions

(previous lecture)

function call and return logic

(this lecture)

arithmetic expressions

(previous lecture)

<u>How to translate such high-level code into machine language?</u>

- In a two-tier compilation model, the overall translation challenge is broken between a *front-end* compilation stage and a subsequent *back-end* translation stage
- In our Hack-Jack platform, all the above sub-tasks (handling arithmetic / boolean expressions and program flow / function calling commands) are done by the back-end, i.e. by the VM translator.

Lecture plan

Arithmetic / Boolean commands

- `add`
- `sub`
- `neg`
- `eq`
- `gt`
- `lt`
- `and`
- `or`
- `not`

previous lecture

Memory access commands

- `pop x` (pop into x, which is a variable)
- `push y` (y being a variable or a constant)

Program flow commands

- `label` (declaration)
- `goto` (label)
- `if-goto` (label)

Function calling commands

- `function` (declaration)
- `call` (a function)
- `return` (from a function)

Program flow commands in the VM language

VM code example:

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

In the VM language, the program flow abstraction is delivered using three commands:

```
label c       // label declaration

goto c        // unconditional jump to the
              // VM command following the label c

if-goto c     // pops the topmost stack element;
              // if it's not zero, jumps to the
              // VM command following the label c
```

<u>How to translate these three abstractions into assembly?</u>

- Simple: label declarations and goto directives can be effected directly by assembly commands
- More to the point: given any one of these three VM commands, the VM Translator must emit one or more assembly commands that effects the same semantics on the Hack platfrom
- How to do it? see project 8.

Lecture plan

<u>Arithmetic / Boolean commands</u>

add
sub
neg
eq
gt
lt
and
or
not

previous lecture

<u>Memory access commands</u>

`pop x` (pop into x, which is a variable)
`push y` (y being a variable or a constant)

<u>Program flow commands</u>

`label`	(declaration)
`goto`	(label)
`if-goto`	(label)

<u>Function calling commands</u>

`function`	(declaration)
`call`	(a function)
`return`	(from a function)

Subroutines

```
// Compute x = (-b + sqrt(b^2 -4*a*c)) / 2*a
if (~(a = 0))
   x = (-b + sqrt(b * b – 4 * a * c)) / (2 * a)
else
   x = - c / b
```

Subroutines = a major programming artifact

- Basic idea: the given language can be extended at will by user-defined commands (aka *subroutines* / *functions* / *methods* ...)
- Important: the language's primitive commands and the user-defined commands have the same look-and-feel
- This transparent extensibility is the most important abstraction delivered by high-level programming languages
- The challenge: implement this abstraction, i.e. allow the program control to flow effortlessly between one subroutine to the other

"A well-designed system consists of a collection of black box modules, each executing its effect like magic"
(Steven Pinker, *How The Mind Works*)

Subroutines in the VM language

Calling code (example)

```
...
// computes (7 + 2) * 3 - 5
push constant 7
push constant 2
add
push constant 3
call mult
push constant 5
sub
...
```

VM subroutine call-and-return commands

Called code, aka "callee" (example)

```
function mult 1
  push constant 0
  pop local 0 // result (local 0) = 0
label loop
  push argument 0
  push constant 0
  eq
  if-goto end // if arg0 == 0, jump to end
  push argument 0
  push 1
  sub
  pop argument 0  // arg0--
  push argument 1
  push local 0
  add
  pop local 0  // result += arg1
  goto loop
label end
  push local 0  // push result
  return
```

The invocation of the VM's primitive commands and subroutines follow exactly the same rules:

- The caller pushes the necessary argument(s) and calls the command / function for its effect
- The called command / function is responsible for removing the argument(s) from the stack, and for popping onto the stack the result of its execution.

Function commands in the VM language

```
function g nVars   // here starts a function called g,
                   // which has nVars local variables

call g nArgs       // invoke function g for its effect;
                   // nArgs arguments have already been pushed onto the stack

return             // terminate execution and return control to the caller
```

Q: Why this particular syntax?

A: Because it simplifies the VM implementation (later).

Function call-and-return conventions

Calling function

```
function demo 3
  ...
  push constant 7
  push constant 2
  add
  push constant 3
  call mult
  ...
```

called function aka "callee" (example)

```
function mult 1
  push constant 0
  pop local 0 // result (local 0) = 0
label loop
  ...         // rest of code ommitted
label end
  push local 0  // push result
  return
```

Although not obvious in this example, every VM function has a private set of 5 memory segments (`local`, `argument`, `this`, `that`, `pointer`)

These resources exist as long as the function is running.

Call-and-return programming convention

- The caller must push the necessary argument(s), call the callee, and wait for it to return
- Before the callee terminates (returns), it must push a return value
- At the point of return, the callee's resources are recycled, the caller's state is re-instated, execution continues from the command just after the `call`
- **Caller's net effect:** the arguments were replaced by the return value (just like with primitive commands)

Behind the scene

- Recycling and re-instating subroutine resources and states is a major headache
- Some agent (either the VM or the compiler) should manage it behind the scene "like magic"
- In our implementation, the magic is VM / stack-based, and is considered a great CS gem.

The function-call-and-return protocol

```
function g nVars
call g nArgs
return
```

The caller's view:

- Before calling a function *g*, I must push onto the stack as many arguments as needed by *g*
- Next, I invoke the function using the command `call g nArgs`
- After *g* returns:
  - The arguments that I pushed before the call have disappeared from the stack, and a return value (that always exists) appears at the top of the stack
  - All my memory segments (`local`, `argument`, `this`, `that`, `pointer`) are the same as before the call.

Blue = VM function writer's responsibility

Black = black box magic, delivered by the VM implementation

Thus, the VM implementation writer must worry about the "black operations" only.

The callee's (*g* 's) view:

- When I start executing, my `argument` segment has been initialized with actual argument values passed by the caller
- My `local` variables segment has been allocated and initialized to zero
- The `static` segment that I see has been set to the `static` segment of the VM file to which I belong, and the working stack that I see is empty
- Before exiting, I must push a value onto the stack and then use the command `return`.

The function-call-and-return protocol: the VM implementation view

```
function g nVars
call g nArgs
return
```

When function *f* calls function *g*, the VM implementation must:

- Save the return address within *f* 's code:
  the address of the command just after the `call`
- Save the virtual segments of *f*
- Allocate, and initialize to 0, as many local variables as needed by *g*
- Set the `local` and `argument` segment pointers of *g*
- Transfer control to *g*.

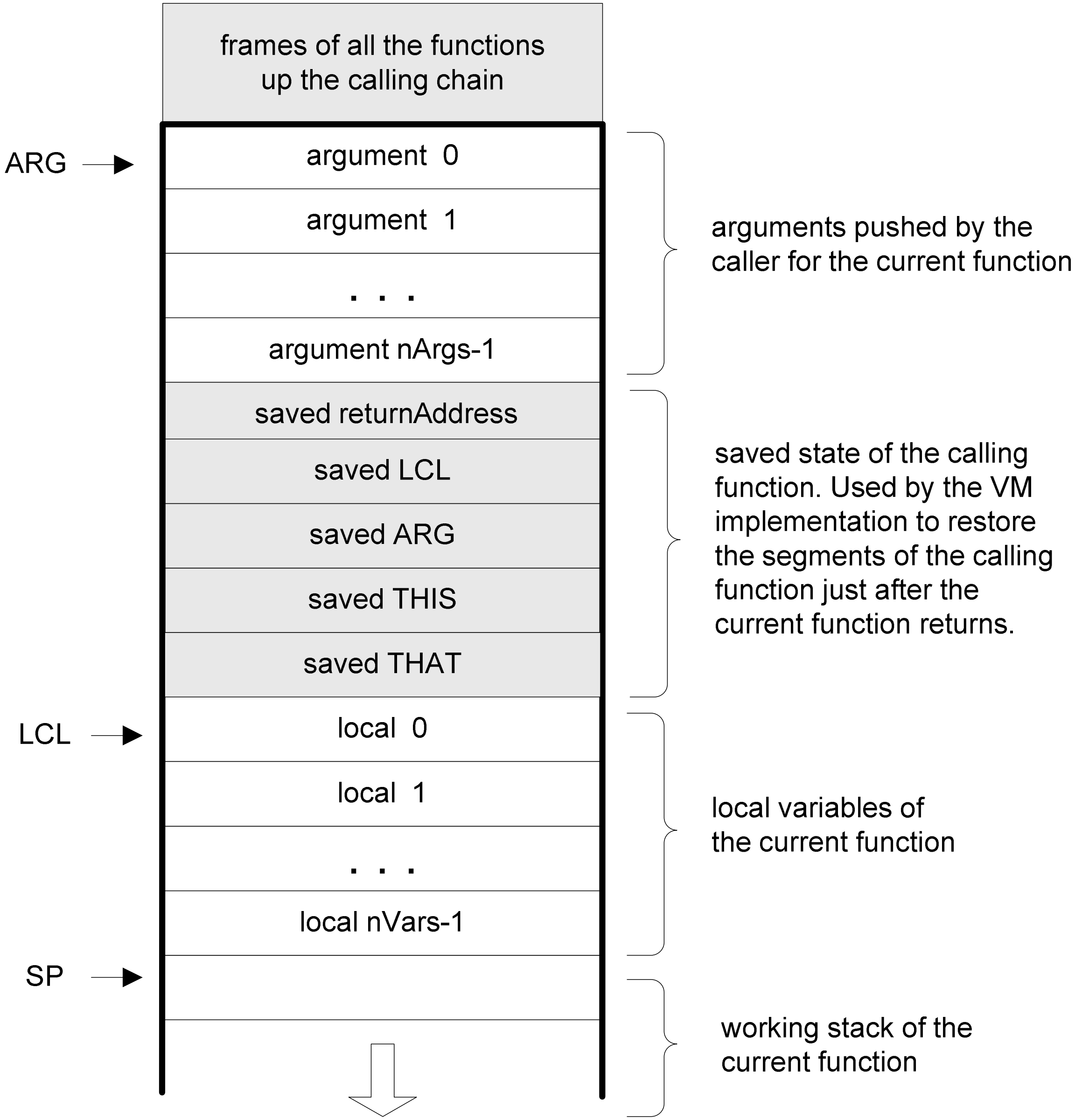
When *g* terminates and control should return to *f*, the VM implementation must:

- Clear *g* 's arguments and other junk from the stack
- Restore the virtual segments of *f*
- Transfer control back to *f*
  (jump to the saved return address).

Q: How should we make all this work “like magic”?

A: We’ll use the stack cleverly.

The implementation of the VM's stack on the host Hack RAM



<u>Global stack:</u>
the entire RAM area dedicated to hold the stack

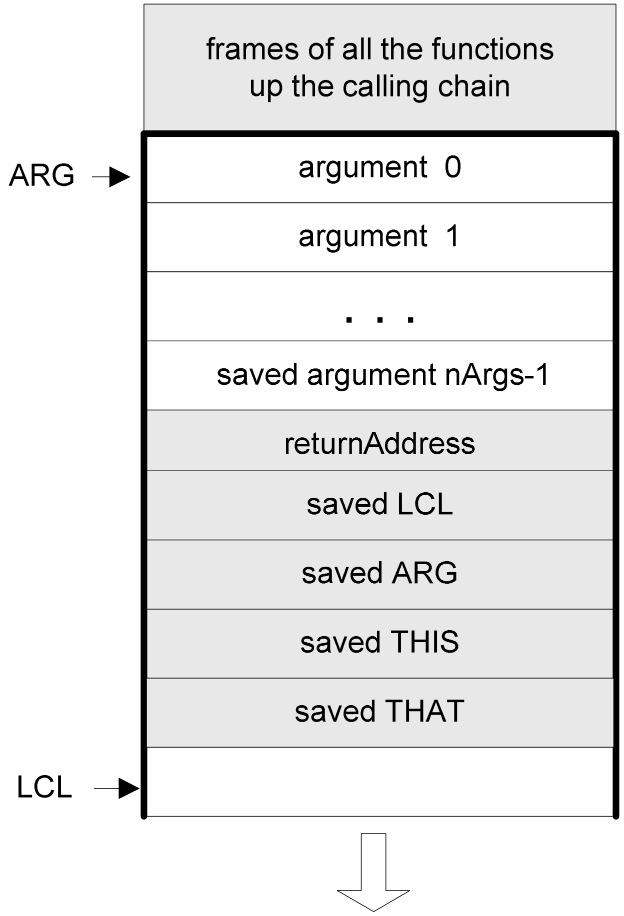
<u>Working stack:</u>
from SP onwards: the stack that the current function sees

- At any point of time, only one function (the *current function*) is executing; other functions may be waiting up the calling chain
- Shaded areas: irrelevant to the current function
- The current function sees only the working stack, as well as its virtual memory segments
- The rest of the stack holds the frozen states of all the functions up the calling hierarchy.

Implementing the `call g nArgs` command

`call g nArgs`

```
  // In the course of implementing the code of f
  // (the caller), we arrive to the command call g nArgs.
  // we assume that nArgs arguments have been pushed
  // onto the stack. What do we do next?
  // We generate a symbol, let's call it returnAddress;
  // Next, we effect the following logic:
  push returnAddress // saves the return address
  push LCL           // saves the LCL of f
  push ARG           // saves the ARG of f
  push THIS          // saves the THIS of f
  push THAT          // saves the THAT of f
  ARG = SP-nArgs-5   // repositions SP for g
  LCL = SP           // repositions LCL for g
  goto g             // transfers control to g
returnAddress:       // the generated symbol
```



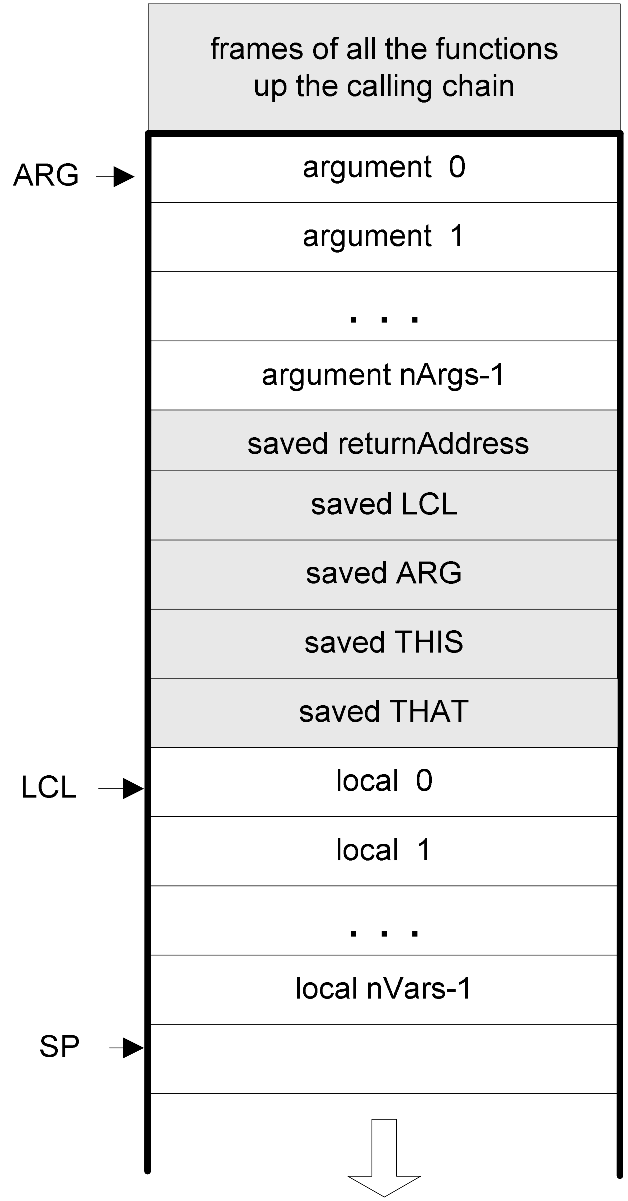
None of this code is executed yet ... At this point we are just *generating code* (or simulating the VM code on some platform)

<u>Implementation:</u> If the VM is implemented as a program that translates VM code into assembly code, the translator must emit the above logic in assembly.

Implementing the `function g nVars` command

`function g nVars`

```
// to implement the command function g nVars,
// we effect the following logic:

g:
  repeat nVars times:
  push 0
```

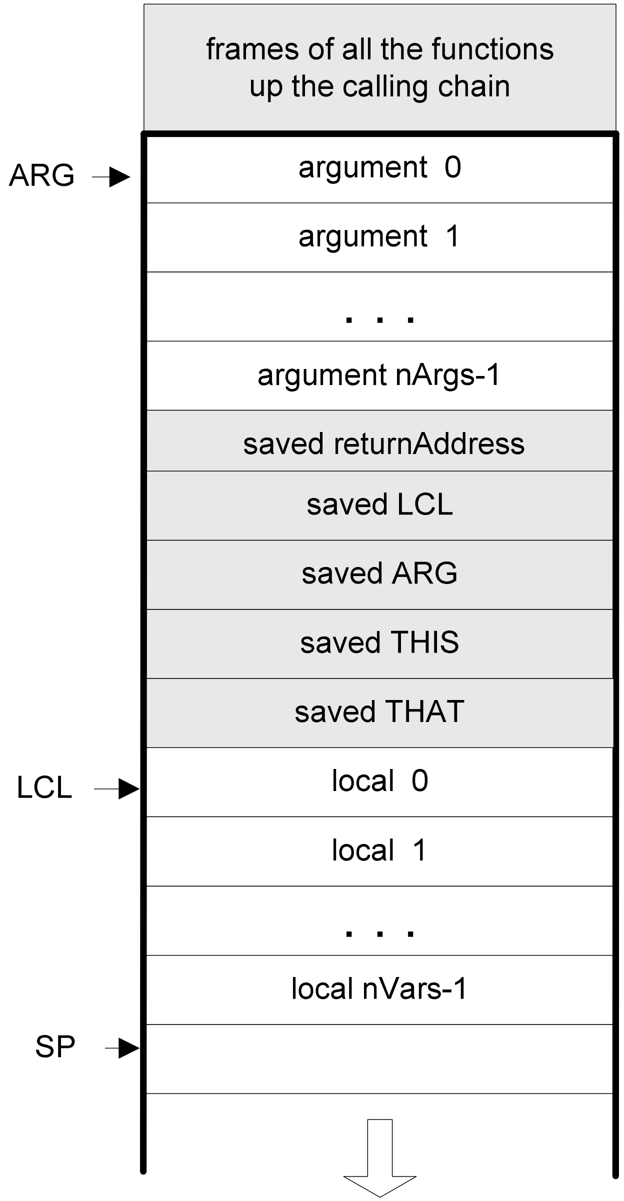


<u>Implementation:</u> If the VM is implemented as a program that translates VM code into assembly code, the translator must emit the above logic in assembly.

Implementing the `return` command

`return`

```
// In the course of implementing the code of g,
// we arrive to the command return.
// We assume that a return value has been pushed
// onto the stack.
// We effect the following logic:
  frame = LCL          // frame is a temp. variable
  retAddr = *(frame-5) // retAddr is a temp. variable
  *ARG = pop           // repositions the return value
                       // for the caller
  SP=ARG+1             // restores the caller's SP
  THAT = *(frame-1)    // restores the caller's THAT
  THIS = *(frame-2)    // restores the caller's THIS
  ARG = *(frame-3)     // restores the caller's ARG
  LCL = *(frame-4)     // restores the caller's LCL
  goto retAddr         // goto returnAddress
```



<u>Implementation:</u> If the VM is implemented as a program that translates VM code into assembly code, the translator must emit the above logic in assembly.

Bootstrapping

A high-level jack *program* (aka *application*) is a set of class files.
By a Jack convention, one class must be called `Main`, and this class must have at least one function, called `main`.

<u>The contract:</u> when we tell the computer to execute a Jack program, the function `Main.main` starts running

<u>Implementation:</u>

- After the program is compiled, each class file is translated into a `.vm` file
- The operating system is also implemented as a set of `.vm` files (aka "libraries") that co-exist alongside the program's `.vm` files
- One of the OS libraries, called `Sys.vm`, includes a method called `init`. The `Sys.init` function starts with some OS initialization code (we'll deal with this later, when we discuss the OS), then it does `call Main.main`
- Thus, to bootstrap, the VM implementation has to effect (e.g. in assembly), the following operations:

```
SP = 256        // initialize the stack pointer to 0x0100
call Sys.init   // call the function that calls Main.main
```

VM implementation over the Hack platform

- Extends the VM implementation described in the last lecture (chapter 7)
- The result: a single assembly program file with lots of agreed-upon symbols:

Symbol	*Usage*
`SP`, `LCL`, `ARG`, `THIS`, `THAT`	These predefined symbols point, respectively, to the stack top and to the base addresses of the virtual segments `local`, `argument`, `this`, and `that`.
`R13` - `R15`	These predefined symbols can be used for any purpose.
`Xxx.j`	Each static variable `j` in a VM file `Xxx.vm` is translated into the assembly symbol `Xxx.j`. In the subsequent assembly process, these symbolic variables will be allocated RAM space by the Hack assembler.
`functionName$label`	Each `label b` command in a VM function `f` should generate a globally unique symbol "`f$b`" where "`f`" is the function name and "`b`" is the label symbol within the VM function's code. When translating `goto b` and `if-goto b` VM commands into the target language, the full label specification "`f$b`" must be used instead of "`b`".
`(FunctionName)`	Each VM function f should generates a symbol "`f`" that refers to its entry point in the instruction memory of the target computer.
return-address	Each VM function call should generate and insert into the translated code a unique symbol that serves as a return address, namely the memory location (in the target platform's memory) of the command following the function call.

Proposed API

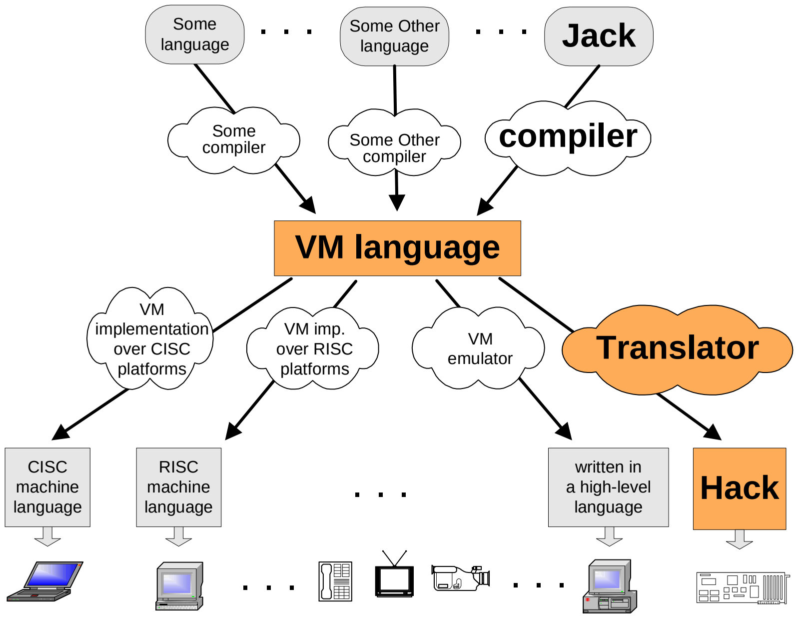
CodeWriter: Translates VM commands into Hack assembly code. The routines listed here should be added to the `CodeWriter` module API given in chapter 7.

Routine	**Arguments**	**Returns**	**Function**
`writeInit`	--	--	Writes the assembly code that effects the VM initialization, also called *bootstrap code*. This code must be placed at the beginning of the output file.
`writeLabel`	`label` (string)	--	Writes the assembly code that is the translation of the `label` command.
`writeGoto`	`label` (string)	--	Writes the assembly code that is the translation of the `goto` command.
`writeIf`	`label` (string)	--	Writes the assembly code that is the translation of the `if-goto` command.
`writeCall`	`functionName` (string) `numArgs` (int)	--	Writes the assembly code that is the translation of the `call` command.
`writeReturn`	--	--	Writes the assembly code that is the translation of the `return` command.
`writeFunction`	`functionName` (string) `numLocals` (int)	--	Writes the assembly code that is the trans. of the given `function` command.

Perspective

Benefits of the VM approach

- Code transportability: compiling for different platforms requires replacing only the VM implementation
- Language inter-operability: code of multiple languages can be shared using the same VM
- Common software libraries
- Code mobility: Internet
- Some virtues of the modularity implied by the VM approach to program translation:
  - Improvements in the VM implementation are shared by all compilers above it
  - Every new digital device with a VM implementation gains immediate access to an existing software base
  - New programming languages can be implemented easily using simple compilers



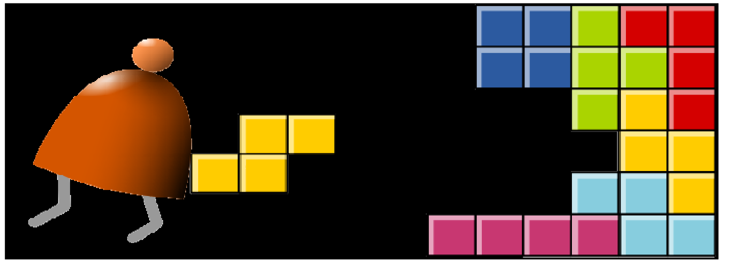
Benefits of managed code:

- Security
- Array bounds, index checking, …
- Add-on code
- Etc.

VM Cons

- Performance.

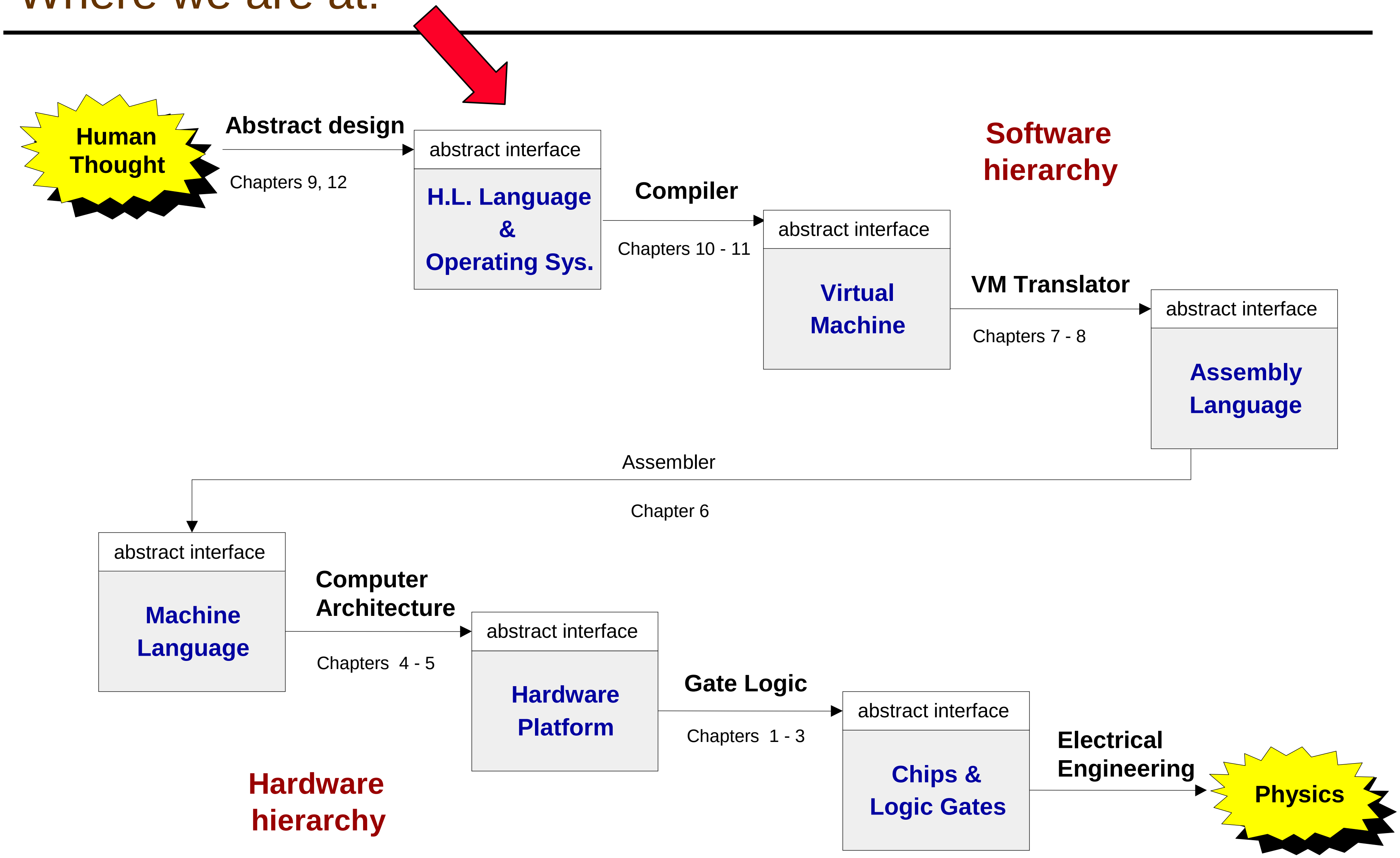
High-Level Language



Building a Modern Computer From First Principles

www.nand2tetris.org

Where we are at:



A brief evolution of object-oriented languages

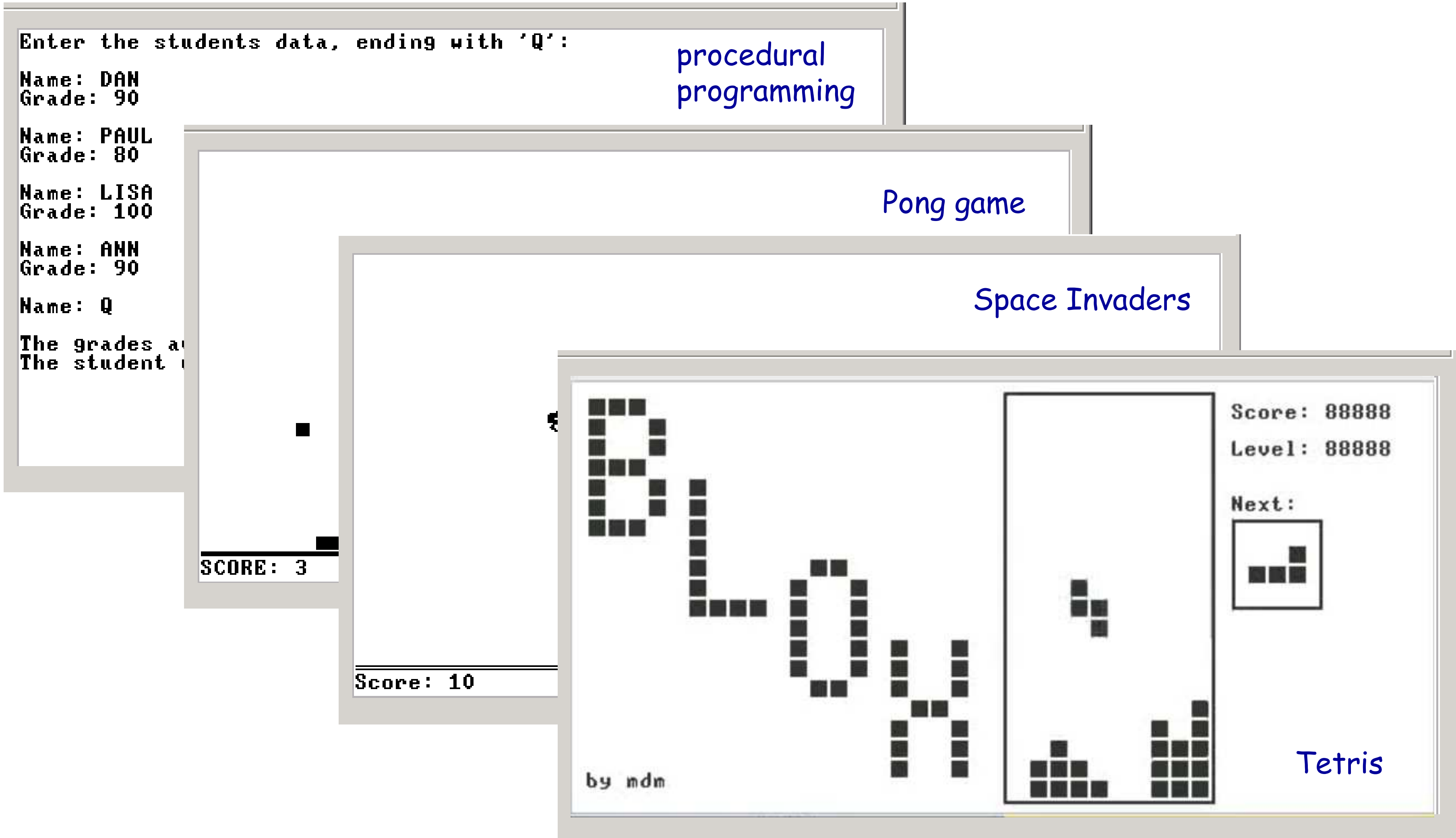
- Machine language (binary code)
- Assembly language (low-level symbolic programming)
- Simple procedural languages, e.g. `Fortran`, `Basic`, `Pascal`, `C`
- Simple object-based languages (without inheritance), e.g. early versions of `Visual Basic`, `JavaScript`

  Jack
- Fancy object-oriented languages (with inheritance): `C++`, `Java`, `C#`

The Jack programming language

Jack: a simple, object-based, high-level language with a Java-like syntax

Some sample applications written in Jack:



Disclaimer

Although Jack is a real programming language, we don't view it as an *end*

Rather, we use Jack as a *means* for teaching:

- How to build a compiler
- How the compiler and the language interface with the operating system
- How the topmost piece in the software hierarchy fits into the big picture

Jack can be learned (and un-learned) in one hour.

Hello world

```
/** Hello World program. */
class Main {
    function void main () {
        // Prints some text using the standard library
        do Output.printString("Hello World");
        do Output.println();      // New line
        return;
    }
}
```

Some observations:

- Java-like syntax
- Typical comments format
- Standard library
- Language-specific peculiarities.

Typical programming tasks in Jack

Jack can be used to develop any app that comes to my mind, for example:

- Procedural programming: a program that computes 1 + 2 + ... + n
- Object-oriented programming: a class representing bank accounts
- Abstract data type representation: a class representing fractions (like 2/5)
- Data structure representation: a class representing linked lists
- Etc.

We will now discuss the above app examples

As we do so, we'll begin to unravel how the magic of a high-level object-based language is delivered by the compiler and by the VM

These insights will serve us in the next lectures, when we build the Jack compiler.

Procedural programming example

```
class Main {

  /** Sums up 1 + 2 + 3 + ... + n */
  function int sum (int n) {
    var int sum, i;
    let sum = 0;
    let i = 1;
    while (~(i > n)) {
      let sum = sum + i;
      let i = i + 1;
    }
    return sum;
  }

  function void main () {
    var int n;
    let n = Keyboard.readInt("Enter n: ");
    do Output.printString("The result is: ");
    do Output.printInt(sum(n));
    return;
  }
}
```

Jack program = a collection of one or more classes

Jack class = a collection of one or more subroutines

Execution order: when we execute a Jack program, `Main.main()` starts running.

Jack subroutine:

- `method`
- `constructor`
- `function` (static method)
- (the example on the left has functions only, as it is "object-less")

Standard library: a set of OS services (methods and functions) organized in 8 supplied classes: `Math`, `String`. `Array`, `Output`, `Keyboard`, `Screen`, `Memory`, `Sys` (OS API in the book).

Object-oriented programming example

The `BankAccount` class API (method sigantures)

```
/** Represents a bank account.
    A bank account has an owner, an id, and a balance.
    The id values start at 0 and increment by 1 each
    time a new account is created. */

class BankAccount {

    /** Constructs a new bank account with a 0 balance. */
    constructor BankAccount new(String owner)

    /** Deposits the given amount in this account. */
    method void deposit(int amount)

    /** Withdraws the given amount from this account. */
    method void withdraw(int amount)

    /** Prints the data of this account. */
    method void printInfo()

    /** Disposes this account. */
    method void dispose()

}
```

Object-oriented programming example (continues)

```
/** Represents a bank account. */

class BankAccount {
  // class-level variable
  static int newAcctId;

  // Private variables (aka fields / properties)
  field int id;
  field String owner;
  field int balance;

  /** Constructs a new bank account */
  constructor BankAccount new (String owner) {
      let id = newAcctId;
      let newAcctId = newAcctId + 1;
      let this.owner = owner;
      let balance = 0;
      return this;   // 2
  }

  // More BankAccount methods.

}
```

```
// Code in any other class:
var int x;
var BankAccount b;              // 1
let b = BankAccount.new("joe"); // 3
```

Explain: `return this`

The constructor returns the RAM base address of the memory block that stores the data of the newly created BankAccount object

Explain: `b = BankAccount.new("joe")`

Calls the constructor (which creates a new BankAccount object), then stores a pointer to the object's base memory address in variable b

Behind the scene (following compilation):

```
// b = BankAccount.new("joe")
push "joe"
call BankAccount.new
pop b
```

Explanation: the VM code calls the constructor; the constructor creates a new object, pushes its base address onto the stack, and returns;

The calling code then pops the base address into a variable that will now point to the new object.

Object-oriented programming example (continues)

```
class BankAccount {
  static int nAccounts;

  field int id;
  field String owner;
  field int balance;

  // Constructor ... (omitted)

  /** Handles deposits */
  method void deposit (int amount) {
      let balance = balance + amount;
      return;
  }

  /** Handles withdrawls */
  method void withdraw (int amount){
      if (~(amount > balance)) {
          let balance = balance - amount;
      }
      return;
  }

  // More BankAccount methods.

}
```

```
...
var BankAccount b1, b2;
...
let b1 = BankAccount.new("joe");
let b2 = BankAccount.new("jane");
do b1.deposit(5000);
do b1.withdraw(1000);
...
```

Explain: `do b1.deposit(5000)`

- In Jack, `void` methods are invoked using the keyword `do` (a compilation artifact)
- The object-oriented method invocation style `b1.deposit(5000)` is a fabcy way to expres the procedural semantics `deposit(b1,5000)`

<u>Behind the scene</u> (following compilation):

```
// do b1.deposit(5000)
push b1
push 5000
call BanAccount.deposit
```

Object-oriented programming example (continues)

```
class BankAccount {
  static int nAccounts;

  field int id;
  field String owner;
  field int balance;

  // Constructor ... (omitted)

  /** Prints information about this account. */
  method void printInfo () {
      do Output.printInt(id);
      do Output.printString(owner);
      do Output.printInt(balance);
      return;
  }

  /** Disposes this account. */
  method void dispose () {
      do Memory.deAlloc(this);
      return;
  }

  // More BankAccount methods.

}
```

```
// Code in any other class:
...
var int x;
var BankAccount b;

let b = BankAccount.new("joe");
// Manipulates b...
do b.printInfo();
do b.dispose();
...
```

Explain `do b.dispose()`

Jack has no garbage collection; The programmer is responsible for explicitly recycling memory resources of objects that are no longer needed.

Abstract data type example

The `Fraction` class API (method sigantures)

```
/** Represents a fraction data type.
    A fraction consists of a numerator and a denominator, both int values */

class Fraction {

    /** Constructs a fraction from the given data */
    constructor Fraction new(int numerator, int denominator)

    /** Reduces this fraction, e.g. changes 20/100 to 1/5. */
    method void reduce()

    /** Accessors
    method int getNumerator()
    method int getDenominator()

    /** Returns the sum of this fraction and the other one */
    method Fraction plus(Fraction other)

    /** Returns the product of this fraction and the other one */
    method Fraction product(Fraction other)

    /** Prints this fraction */
    method void print()

    /** Disposes this fraction */
    method void dispose()
}
```

Abstract data type example (continues)

```
/** Represents a fraction data type.
    A fraction consists of a numerator and a denominator, both int values */
class Fraction {
    field int numerator, denominator;

    constructor Fraction new (int numerator, int denominator) {
        let this.numerator = numerator;
        let this.denominator = denominator;
        do reduce() // Reduces the new fraction
        return this
    }

    /** Reduces this fraction */
    method void reduce () {
        // Code omitted
    }

    // A static method that computes the greatest common denominator of a and b.
    function int gcd (int a, int b) {
        // Code omitted
    }

    method int getNumerator () {
        return numerator;
    }

    method int getDenominator () {
        return denominator;
    }

    // More Fraction methods follow.
```

```
// Code in any other class:
...
var Fraction a, b;
let a = Fraction.new(2,5);
let b = Fraction.new(70,210);
do b.print() // prints "1/3"
...
// (print method in next slide)
```

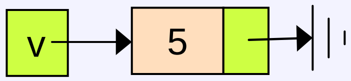
Abstract data type example (continues)

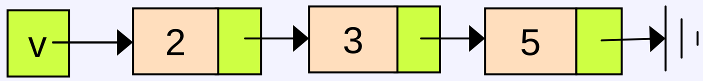
```
/** Represents a fraction data type.
    A fraction consists of a numerator and a denominator, both int values */
class Fraction {
    field int numerator, denominator;

    // Constructor and previously defined methods omitted

    /** Returns the sum of this fraction the other one */
    method Fraction plus (Fraction other) {
        var int sum;
        let sum = (numerator * other.getDenominator()) +
                  (other.getNumerator() * denominator());
        return Fraction.new(sum , denominator * other.getDenominator());
    }

    // Similar fraction arithmetic methods follow, code omitted.

    /** Prints this fraction */
    method void print () {
        do Output.printInt(numerator);
        do Output.printString("/");
        do Output.printInt(denominator);
        return
    }

 }
```

```
// Code in any other class:
var Fraction a, b, c;
let a = Fraction.new(2,3);
let b = Fraction.new(1,5);
// computes c = a + b
let c = a.plus(b);
do c.print(); // prints "13/15"
```

Data structure example

```
/** Represents a sequence of int values, implemented as a linked list.
    The list consists of an atom, which is an int value,
    and a tail, which is either a list or a null value.  */
class List {
    field int data;
    field List next;

    /* Creates a new list */
    constructor List new (int car, List cdr) {
        let data = car;
        let next = cdr;
        return this;
    }

    /* Disposes this list by recursively disposing its tail. */
    method void dispose() {
        if (~(next = null)) {
            do next.dispose();
        }
        do Memory.deAlloc(this);
        return;
    }
    ...
}  // class List.
```






```
// Code in any other class:
...
// Creates a list holding the numbers 2,3, and 5:
var List v;
let v = List.new(5 , null);
let v = List.new(2 , List.new(3,v));
...
```

Jack language specification

- Syntax
- Data types
- Variable kinds
- Expressions
- Statements
- Subroutine calling
- Program structure
- Standard library

(for complete language specification, see the book).

Jack syntax

White space and comments	Space characters, newline characters, and comments are ignored.
	The following comment formats are supported:
	`// Comment to end of line`
	`/* Comment until closing */`
	`/** API documentation comment */`

Symbols	`( )` Used for grouping arithmetic expressions and for enclosing parameter-lists and argument-lists `[ ]` Used for array indexing; `{ }` Used for grouping program units and statements; `,` Variable list separator; `;` Statement terminator; `=` Assignment and comparison operator; `.` Class membership; `+ - * / & | ~ < >` Operators.
Reserved words	`class, constructor, method, function` Program components `int, boolean, char, void` Primitive types `var, static, field` Variable declarations `let, do, if, else, while, return` Statements `true, false, null` Constant values `this` Object reference

Jack syntax (continues)

Constants	*Integer* constants must be positive and in standard decimal notation, e.g., `1984`. Negative integers like `-13` are not constants but rather expressions consisting of a unary minus operator applied to an integer constant. *String* constants are enclosed within two quote (") characters and may contain any characters except *newline* or *double-quote*. (These characters are supplied by the functions `String.newLine()` and `String.doubleQuote()` from the standard library.) *Boolean* constants can be `true` or `false`. The constant `null` signifies a null reference.
Identifiers	Identifiers are composed from arbitrarily long sequences of letters (`A-Z`, `a-z`), digits (`0-9`), and "_". The first character must be a letter or "_". The language is case sensitive. Thus `x` and `X` are treated as different identifiers.

Jack data types

Primitive types (Part of the language; Realized by the compiler):

- `int` 16-bit 2's complement (from -32768 to 32767)
- `boolean` `0` and `–1`, standing for `true` and `false`
- `char` unicode character ('a', 'x', '+', '%', ...)

Abstract data types (Standard language extensions; Realized by the OS / standard library):

- `String`
- `Array`

... (extensible)

Application-specific types (User-defined; Realized by user applications):

- `BankAccount`
- `Fraction`
- `List`
- `Bat / Ball`

. . . (as needed)

Jack variable kinds and scope

Variable kind	Definition / Description	Declared in	Scope
Static variables	**static** *type name1*, *name2*, ... ; Only one copy of each static variable exists, and this copy is shared by all the object instances of the class (like *private static variables* in Java)	Class declaration.	The class in which they are declared.
Field variables	**field** *type name1*, *name2*, ... ; Every object instance of the class has a private copy of the field variables (like *private object variables* in Java)	Class declaration.	The class in which they are declared, except for functions.
Local variables	**var** *type name1*, *name2*, ... ; Local variables are allocated on the stack when the subroutine is called and freed when it returns (like *local variables* in Java)	Subroutine declaration.	The subroutine in which they are declared.
Parameter variables	*type name1*, *name2*, ... Used to specify inputs of subroutines, for example: `function void drive (Car c, int miles)`	Appear in parameter lists as part of subroutine declarations.	The subroutine in which they are declared.

Jack expressions

A Jack *expression* is any one of the following:

- A constant
- A variable name in scope (the variable may be static, field, local, or a parameter)
- The keyword `this`, denoting the current object
- An array element using the syntax *arrayName*[*expression*],
  where *arrayNname* is a variable name of type `Array` in scope
- A subroutine call that returns a non-void type
- An *expression* prefixed by one of the unary operators - or ~ :

-*expression*	(arithmetic negation)
~*expression*	(logical negation)

- An expression of the form *expression op expression* where *op* is one of the following:

+ - * /	(integer arithmetic operators)
& |	(boolean and and or operators, bit-wise)
< > =	(comparison operators)

- (*expression*) (an expression within parentheses)

Jack Statements

```
let varName = expression;
or
let varName[expression] = expression;
```

```
if (expression) {
    statements
}
else {
    statements
}
```

```
while (expression) {
    statements
}
```

```
do function-or-method-call;
```

```
return expression;
or
return;
```

Jack subroutine calls

General syntax: *subroutineName(arg0, arg1, …)*

where each argument is a valid Jack expression

Parameter passing is *by-value* (primitive types) or *by-reference* (object types)

Example 1:

Consider the function (static method): `function int sqrt(int n)`

This function can be invoked as follows:

```
sqrt(17)
sqrt(x)
sqrt((b * b) – (4 * a * c))
sqrt(a * sqrt(c - 17) + 3)
```

Etc. In all these examples the argument value is computed and passed by-value

Example 2:

Consider the method: `method Matrix plus (Matrix other);`

If `u` and `v` were variables of type `Matrix`, this method can be invoked using: `u.plus(v)`

The `v` variable is passed by-reference, since it refers to an object.

Noteworthy features of the Jack language

- The (cumbersome) `let` keyword, as in `let x = 0;`
- The (cumbersome) `do` keyword, as in `do reduce();`
- No operator priority:

  `1 + 2 * 3` yields `9`, since expressions are evaluated left-to-right;

  To effect the commonly expected result, use `1 + (2 * 3)`
- Only three primitive data types: `int`, `boolean`, `char`;
  In fact, each one of them is treated as a 16-bit value
- No casting; a value of any type can be assigned to a variable of any type
- Array declaration: `Array x;` followed by `x = Array.new();`
- Static methods are called `function`
- Constructor methods are called `constructor`;
  Invoking a constructor is done using the syntax *ClassName*`.new(`*argsList*`)`

<u>Q</u>: Why did we introduce these features into the Jack language?

<u>A</u>: To make the writing of the Jack compiler easy!

Any one of these language features can be modified, with a reasonable amount of work, to make them conform to a more typical Java-like syntax.

Jack program structure

```
class ClassName {

    field variable declarations;

    static variable declarations;

    constructor type { parameterList ) {
        local variable declarations;
        statements
    }

    method type { parameterList ) {
        local variable declarations;
        statements
    }

    function type { parameterList ) {
        local variable declarations;
        statements
    }
}
```

About this spec:

- Every part in this spec can apper 0 or more times
- The order of the `field` / `static` declarations is arbitrary
- The order of the subroutine declarations is arbitrary
- Each *type* is either `int`, `boolean`, `char`, or a class name.

A Jack program:

- Each class is written in a separate file (compilation unit)
- Jack program = collection of one or more classes, one of which must be named `Main`
- The `Main` class must contain at least one method, named `main()`

Jack standard library aka language extensions aka Jack OS

```
class Math {
}
```

```
Class String {
}
```

```
Class Array {
}
```

```
class Output {
}
```

```
Class Screen {
}
```

```
class Memory {
}
```

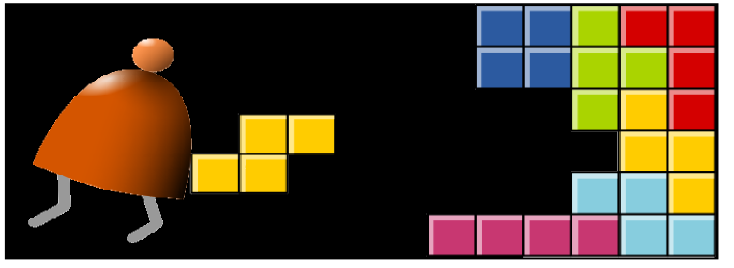
```
Class Keyboard {
}
```

```
Class Sys {
    function void halt():
    function void error(int errorCode)
    function void wait(int duration)
}
```

Perspective

- Jack is an object-based language: no inheritance
- Primitive type system
- Standard library
- Our hidden agenda: gearing up to learn how to develop the ...
  - Compiler (projects 10 and 11)
  - OS (project 12).

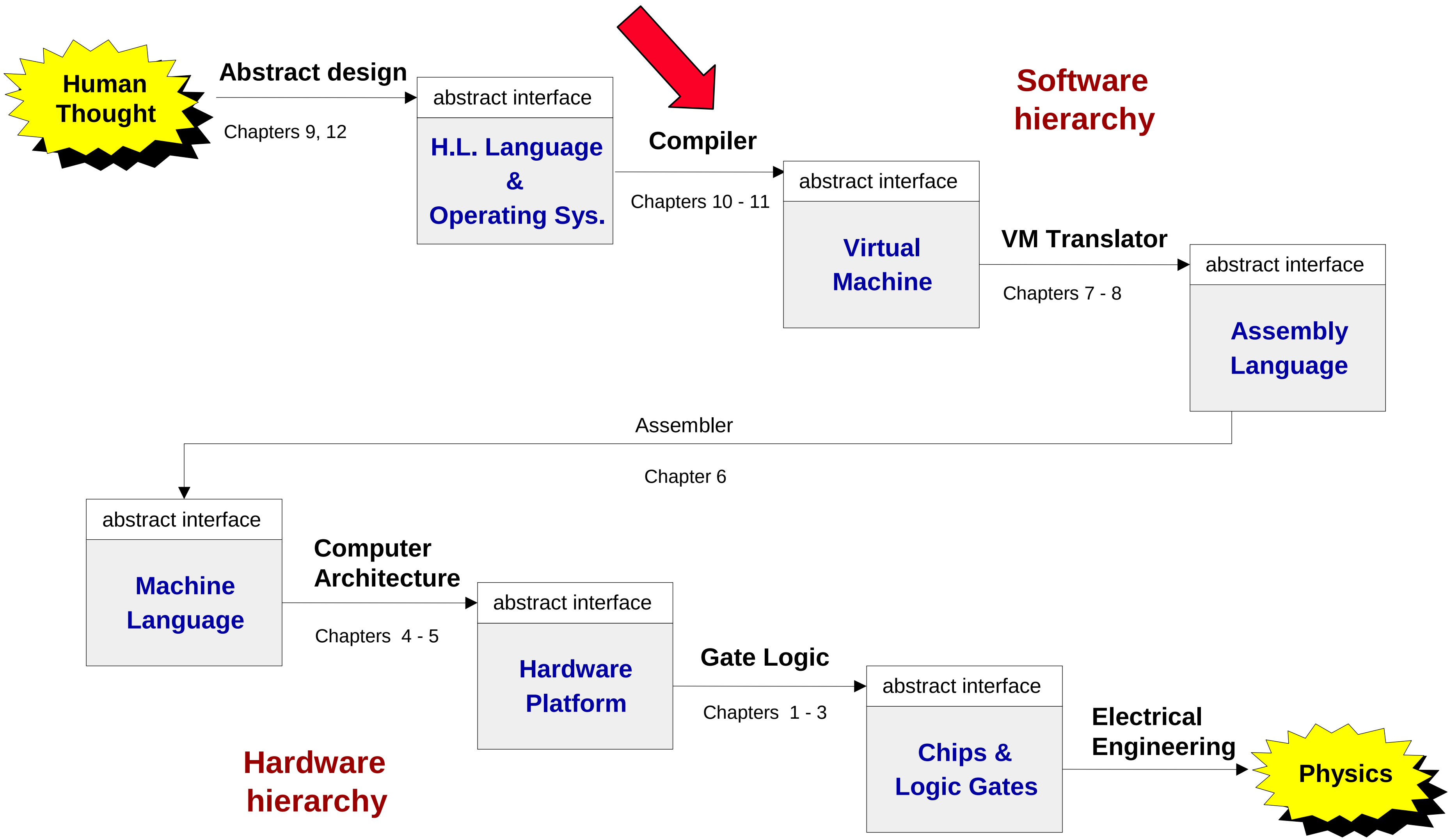
Compiler I: Syntax Analysis



Building a Modern Computer From First Principles

www.nand2tetris.org

Course map



Human Thought
Abstract design
Chapters 9, 12
abstract interface
H.L. Language & Operating Sys.
Compiler
Chapters 10 - 11
abstract interface
Virtual Machine
Software hierarchy
VM Translator
Chapters 7 - 8
abstract interface
Assembly Language
Assembler
Chapter 6
abstract interface
Machine Language
Computer Architecture
Chapters 4 - 5
abstract interface
Hardware Platform
Gate Logic
Chapters 1 - 3
abstract interface
Chips & Logic Gates
Electrical Engineering
Physics
Hardware hierarchy

Motivation: Why study about compilers?

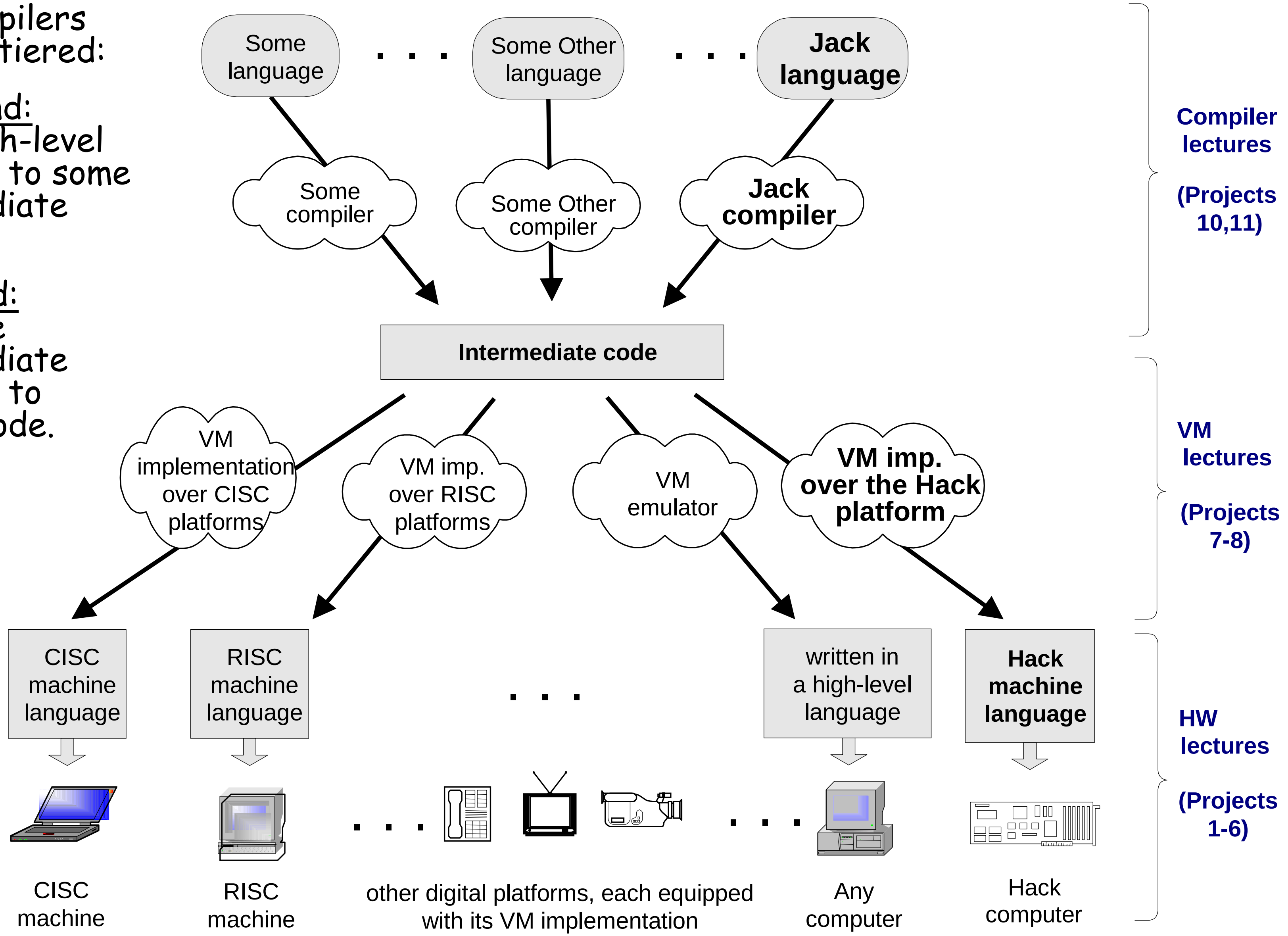
<u>Because Compilers …</u>

- Are an essential part of applied computer science
- Are very relevant to computational linguistics
- Are implemented using classical programming techniques
- Employ important software engineering principles
- Train you in developing software for transforming one structure to another (programs, files, transactions, …)
- Train you to think in terms of "description languages".

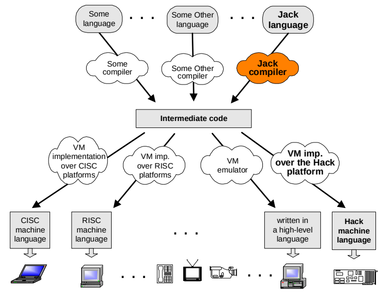
The big picture

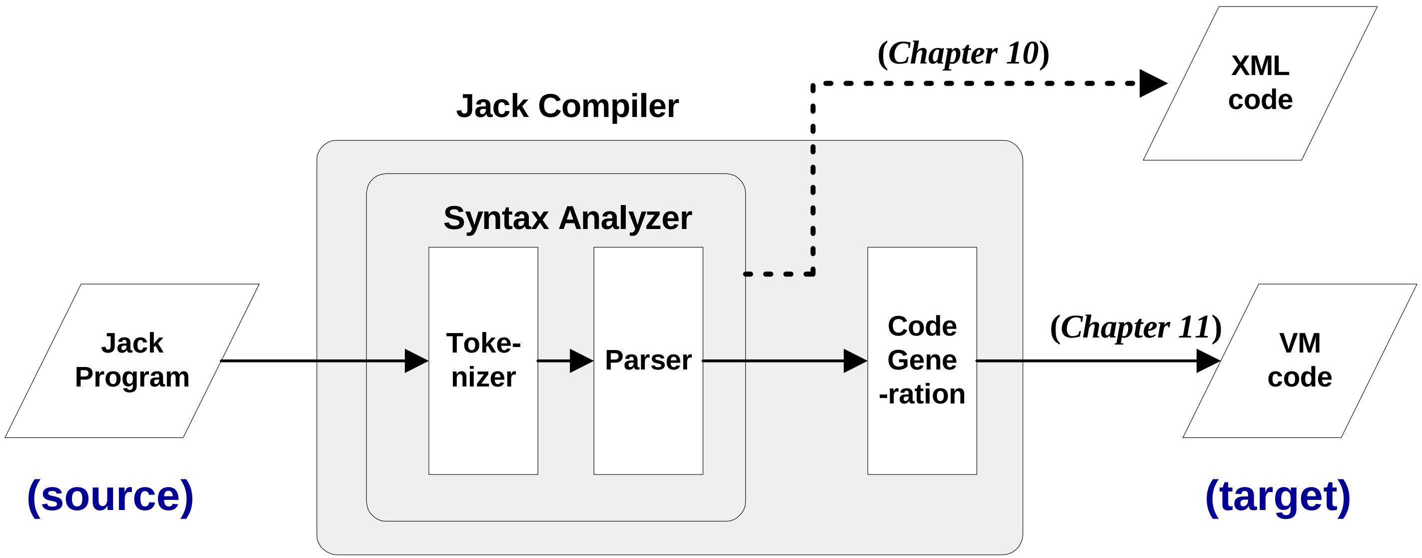
Modern compilers are two-tiered:

- Front-end: from high-level language to some intermediate language
- Back-end: from the intermediate language to binary code.



Compiler architecture (front end)





- Syntax analysis: understanding the semantics implied by the source code
  - Tokenizing: creating a stream of "atoms"
  - Parsing: matching the atom stream with the language grammar

  XML output = one way to demonstrate that the syntax analyzer works

- Code generation: reconstructing the semantics using the syntax of the target code.

Tokenizing / Lexical analysis

Code fragment

```
while (count<=100) { /** demonstration */
      count++;
      // body of while continues
      ...
```

Tokens

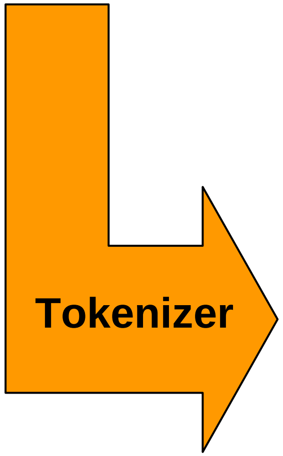
```
while
(
count
<=
100
)
{
count
++
;
...
```

- Remove white space
- Construct a token list (language atoms)
- Things to worry about:
  - Language specific rules:
    e.g. how to treat "++"
  - Language-specific classifications:
    keyword, symbol, identifier, integerCconstant, stringConstant,...
- While we are at it, we can have the tokenizer record not only the token, but also its lexical classification (as defined by the source language grammar).

Jack Tokenizer

Source code

```
if (x < 153) {let city = ”Paris”;}
```



Tokenizer's output

```
<tokens>
  <keyword> if </keyword>
  <symbol> ( </symbol>
  <identifier> x </identifier>
  <symbol> &lt; </symbol>
  <integerConstant> 153 </integerConstant>
  <symbol> ) </symbol>
  <symbol> { </symbol>
  <keyword> let </keyword>
  <identifier> city </identifier>
  <symbol> = </symbol>
  <stringConstant> Paris </stringConstant>
  <symbol> ; </symbol>
  <symbol> } </symbol>
</tokens>
```

Parsing

- The tokenizer discussed thus far is part of a larger program called *parser*
- Each language is characterized by a *grammar.*
  The parser is implemented to recognize this grammar in given texts
- The parsing process:
  - A text is given and tokenized
  - The parser determines weather or not the text can be generated from the grammar
  - In the process, the parser performs a complete structural analysis of the text
- The text can be in an expression in a :
  - Natural language (English, …)
  - Programming language (Jack, …).

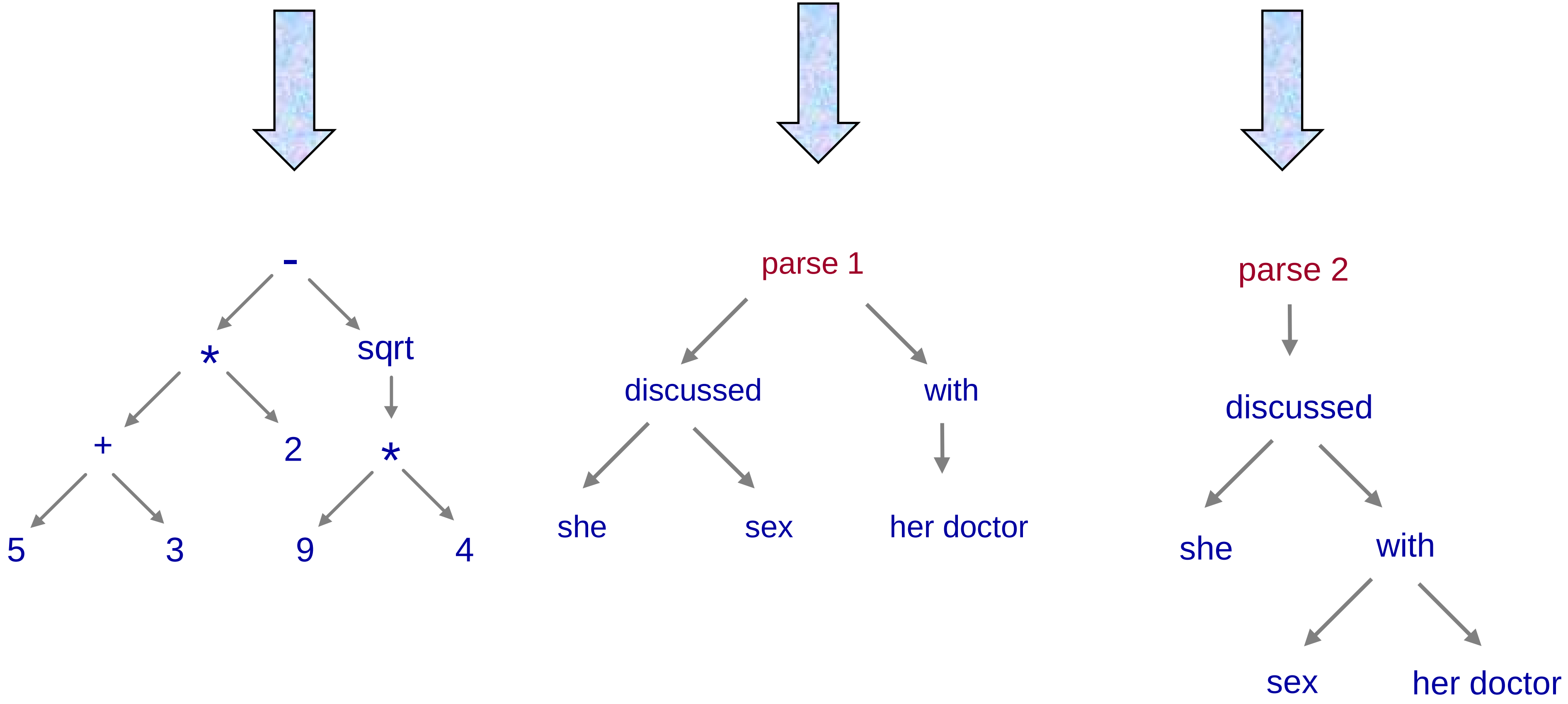
Parsing examples

Jack

`(5+3)*2 – sqrt(9*4)`

English

`she discussed sex with her doctor`



More examples of challenging parsing

Time flies like an arrow

We gave the monkeys the bananas because they were hungry

We gave the monkeys the bananas because they were over-ripe

I never said she stole my money

<u>I</u> never said she stole my money

I <u>never</u> said she stole my money

I never <u>said</u> she stole my money

I never said <u>she</u> stole my money

I never said she <u>stole</u> my money

I never said she stole <u>my</u> money

I never said she stole my <u>money</u>

A typical grammar of a typical C-like language

Grammar

```
program:         statement;

statement:         whileStatement
                 | ifStatement
                 | // other statement possibilities ...
                 | '{' statementSequence '}'

whileStatement: 'while' '(' expression ')' statement

ifStatement:       simpleIf
                 | ifElse

simpleIf:        'if' '(' expression ')' statement

ifElse:          'if' '(' expression ')' statement
                 'else' statement

statementSequence:    ''    // null, i.e. the empty sequence
                    | statement ';' statementSequence

expression:       // definition of an expression comes here

// more definitions follow
```

Code sample

```
while (expression) {
  if (expression)
     statement;
     while (expression) {
        statement;
        if (expression)
           statement;
     }
  while (expression) {
     statement;
     statement;
  }
}

if (expression) {
   statement;
   while (expression)
      statement;
      statement;
   }
   if (expression)
      if (expression)
         statement;
}
```

- Simple (terminal) forms / complex (non-terminal) forms
- Grammar = set of rules on how to construct complex forms from simpler forms
- Highly recursive.

Parse tree

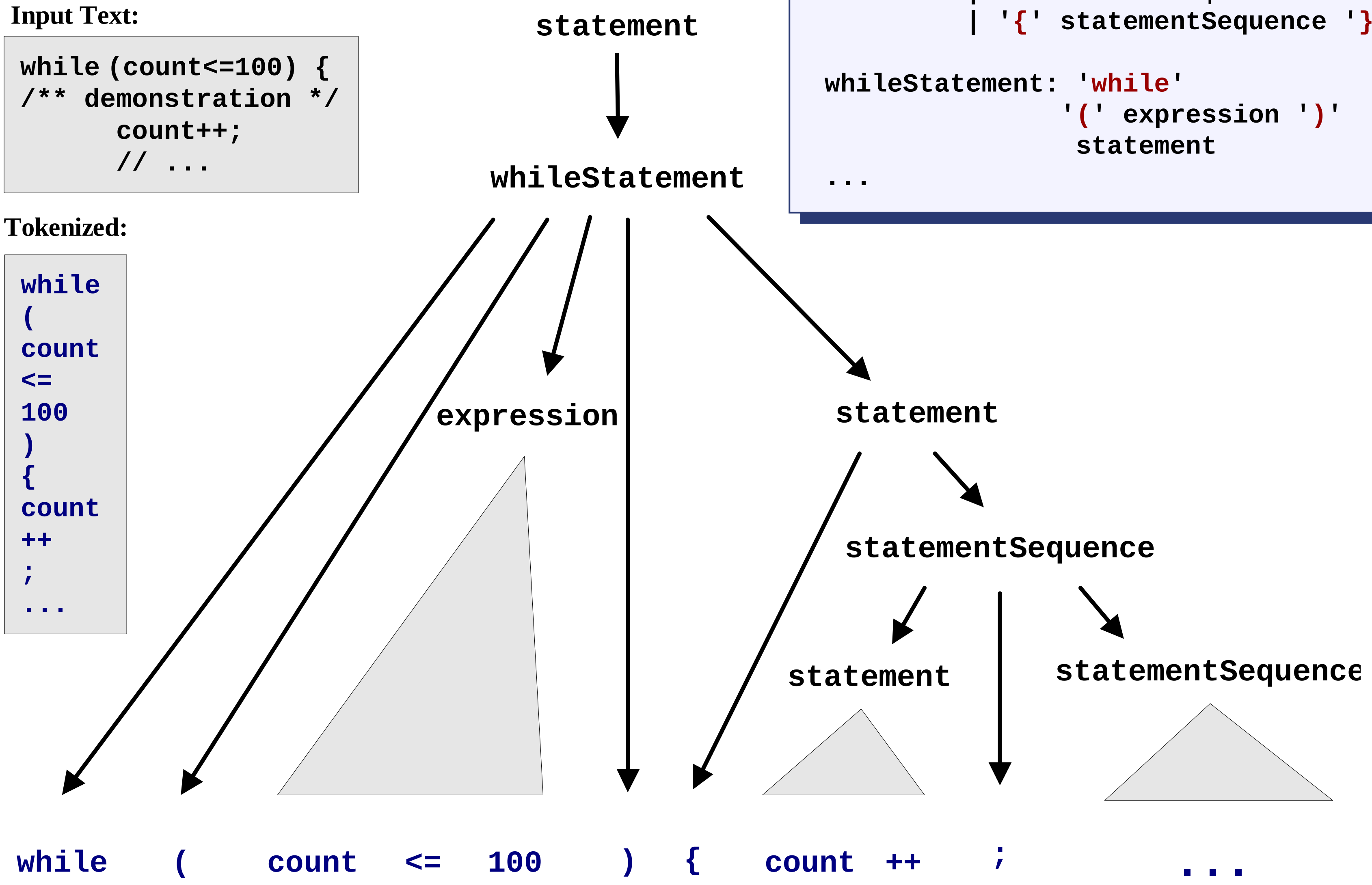
```
program:  statement;

statement: whileStatement
         | ifStatement
         | // other statement possibilities ...
         | '{' statementSequence '}'

whileStatement: 'while'
                '(' expression ')'
                statement
...
```

Input Text:

```
while (count<=100) {
/** demonstration */
      count++;
      // ...
```

Tokenized:

```
while
(
count
<=
100
)
{
count
++
;
...
```



Recursive descent parsing

```
...
statement:    whileStatement
            | ifStatement
            | ...            // other statement possibilities follow
            | '{' statementSequence '}'

whileStatement: 'while' '(' expression ')' statement

ifStatement: ...             // if definition comes here

statementSequence:  ''      // null, i.e. the empty sequence
                  | statement ';' statementSequence

expression: ...    // definition of an expression comes here

...                // more definitions follow
```

code sample

```
while (expression) {
    statement;
    statement;
    while (expression) {
        while (expression)
            statement;
            statement;
    }
}
```

- Highly recursive
- LL(0) grammars: the first token determines in which rule we are
- In other grammars you have to look ahead 1 or more tokens
- Jack is almost LL(0).

Parser implementation: a set of parsing methods, one for each rule:

- **parseStatement()**
- **parseWhileStatement()**
- **parseIfStatement()**
- **parseStatementSequence()**
- **parseExpression().**

A linguist view on parsing

Parsing:

One of the mental processes involved in sentence comprehension, in which the listener determines the syntactic categories of the words, joins them up in a tree, and identifies the subject, object, and predicate, a prerequisite to determining who did what to whom from the information in the sentence.

(Steven Pinker,
The Language Instinct)

The Jack grammar

Lexical elements:	The Jack language includes five types of terminal elements (tokens):
keyword:	'**class**' | '**constructor**' | '**function**' | '**method**' | '**field**' | '**static**' | '**var**' | '**int**' | '**char**' | '**boolean**' | '**void**' | '**true**' | '**false**' | '**null**' | '**this**' | '**let**' | '**do**' | '**if**' | '**else**' | '**while**' | '**return**'
symbol:	'**{**' | '**}**' | '**(**' | '**)**' | '**[**' | '**]**' | '**.**' | '**,**' | '**;**' | '**+**' | '**-**' | '*****' | '**/**' | '**&**' | '**|**' | '**<**' | '**>**' | '**=**' | '**~**'
integerConstant:	A decimal number in the range 0 .. 32767.
StringConstant	'"' A sequence of Unicode characters not including double quote or newline '"'
identifier:	A sequence of letters, digits, and underscore ('_') not starting with a digit.
Program structure:	A Jack program is a collection of classes, each appearing in a separate file. The compilation unit is a class. A class is a sequence of tokens structured according to the following context free syntax:
class:	'**class**' className '**{**' classVarDec* subroutineDec* '**}**'
classVarDec:	('**static**' | '**field**') type varName ('**,**' varName)* '**;**'
type:	'**int**' | '**char**' | '**boolean**' | className
subroutineDec:	('**constructor**' | '**function**' | '**method**') ('**void**' | type) subroutineName '**(**' parameterList '**)**' subroutineBody
parameterList:	((type varName) ('**,**' type varName)*)?
subroutineBody:	'**{**' varDec* statements '**}**'
varDec:	'**var**' type varName ('**,**' varName)* '**;**'
className:	identifier
subroutineName:	identifier
varName:	Identifier

'x': x appears verbatim
x: x is a language construct
x?: x appears 0 or 1 times
x*: x appears 0 or more times
x|y: either x or y appears
(x,y): x appears, then y.

The Jack grammar (cont.)

Statements:

statements: statement*

statement: letStatement | ifStatement | whileStatement | doStatement | returnStatement

letStatement: '**let**' varName ('**[**' expression '**]**')? '=' expression '**;**'

ifStatement: '**if**' '**(**' expression '**)**' '**{**' statements '**}**' ('**else**' '**{**' statements '**}**')?

whileStatement: '**while**' '**(**' expression '**)**' '**{**' statements '**}**'

doStatement: '**do**' subroutineCall '**;**'

ReturnStatement '**return**' expression? '**;**'

Expressions:

expression: term (op term)*

term: integerConstant | stringConstant | keywordConstant | varName | varName '**[**' expression '**]**' | subroutineCall | '**(**' expression '**)**' | unaryOp term

subroutineCall: subroutineName '**(**' expressionList '**)**' | (className | varName) '**.**' subroutineName '**(**' expressionList '**)**'

expressionList: (expression ('**,**' expression)*)?

op: '+' | '-' | '*****' | '**/**' | '**&**' | '**|**' | '<' | '>' | '='

unaryOp: '-' | '~'

KeywordConstant: '**true**' | '**false**' | '**null**' | '**this**'

'x': x appears verbatim

x: x is a language construct

x?: x appears 0 or 1 times

x*: x appears 0 or more times

x|y: either x or y appears

(x,y): x appears, then y.

Jack syntax analyzer in action

```
Class Bar {
   method Fraction foo(int y) {
     var int temp; // a variable
     let temp = (xxx+12)*-63;
      ...
   ...
```

Syntax analyzer

```
<varDec>
  <keyword> var </keyword>
  <keyword> int </keyword>
  <identifier> temp </identifier>
  <symbol> ; </symbol>
</varDec>
<statements>
  <letStatement>
    <keyword> let </keyword>
    <identifier> temp </identifier>
    <symbol> = </symbol>
    <expression>
       <term>
         <symbol> ( </symbol>
         <expression>
           <term>
             <identifier> xxx </identifier>
           </term>
           <symbol> + </symbol>
           <term>
             <int.Const.> 12 </int.Const.>
           </term>
    </expression>
    ...
```

Syntax analyzer

- Using the language grammar, a programmer can write a syntax analyzer program (parser)
- The syntax analyzer takes a source text file and attempts to match it on the language grammar
- If successful, it can generate a parse tree in some structured format, e.g. XML.

The syntax analyzer's algorithm shown in this slide:

- If xxx is non-terminal, output:

  <xxx>
  Recursive code for the body of **xxx**
  </xxx>

- If xxx is terminal (keyword, symbol, constant, or identifier) , output:

  <xxx>
  xxx value
  </xxx>

JackTokenizer: a tokenizer for the Jack language (proposed implementation)

JackTokenizer: Removes all comments and white space from the input stream and breaks it into Jack-language tokens, as specified by the Jack grammar.

Routine	**Arguments**	**Returns**	**Function**
Constructor	input file / stream	--	Opens the input file/stream and gets ready to tokenize it.
`hasMoreTokens`	--	Boolean	Do we have more tokens in the input?
`advance`	--	--	Gets the next token from the input and makes it the current token. This method should only be called if *hasMoreTokens()* is true. Initially there is no current token.
`tokenType`	--	`KEYWORD`, `SYMBOL`, `IDENTIFIER`, `INT_CONST`, `STRING_CONST`	Returns the type of the current token.
`keyWord`	--	`CLASS`, `METHOD`, `FUNCTION`, `CONSTRUCTOR`, `INT`, `BOOLEAN`, `CHAR`, `VOID`, `VAR`, `STATIC`, `FIELD`, `LET`, `DO`, `IF`, `ELSE`, `WHILE`, `RETURN`, `TRUE`, `FALSE`, `NULL`, `THIS`	Returns the keyword which is the current token. Should be called only when `tokenType()` is `KEYWORD`.

JackTokenizer (cont.)

`symbol`	--	Char	Returns the character which is the current token. Should be called only when `tokenType()` is `SYMBOL`.
`identifier`	--	String	Returns the identifier which is the current token. Should be called only when `tokenType()` is `IDENTIFIER`
`intVal`		Int	Returns the integer value of the current token. Should be called only when `tokenType()` is `INT_CONST`
`stringVal`		String	Returns the string value of the current token, without the double quotes. Should be called only when `tokenType()` is `STRING_CONST`.

CompilationEngine: a recursive top-down parser for Jack

The CompilationEngine effects the actual compilation output.

It gets its input from a `JackTokenizer` and emits its parsed structure into an output file/stream.

The output is generated by a series of `compilexxx()` routines, one for every syntactic element `xxx` of the Jack grammar.

The contract between these routines is that each `compilexxx()` routine should read the syntactic construct `xxx` from the input, `advance()` the tokenizer exactly beyond `xxx`, and output the parsing of `xxx`.
Thus, `compilexxx()`may only be called if indeed `xxx` is the next syntactic element of the input.

In the first version of the compiler, which we now build, this module emits a structured printout of the code, wrapped in XML tags (defined in the specs of project 10). In the final version of the compiler, this module generates executable VM code (defined in the specs of project 11).

In both cases, the parsing logic and module API are exactly the same.

CompilationEngine (cont.)

Routine	**Arguments**	**Returns**	**Function**
Constructor	Input stream/file Output stream/file	--	Creates a new compilation engine with the given input and output. The next routine called must be `compileClass()`.
`CompileClass`	--	--	Compiles a complete class.
`CompileClassVarDec`	--	--	Compiles a static declaration or a field declaration.
`CompileSubroutine`	--	--	Compiles a complete method, function, or constructor.
`compileParameterList`	--	--	Compiles a (possibly empty) parameter list, not including the enclosing "`()`".
`compileVarDec`	--	--	Compiles a `var` declaration.

CompilationEngine (cont.)

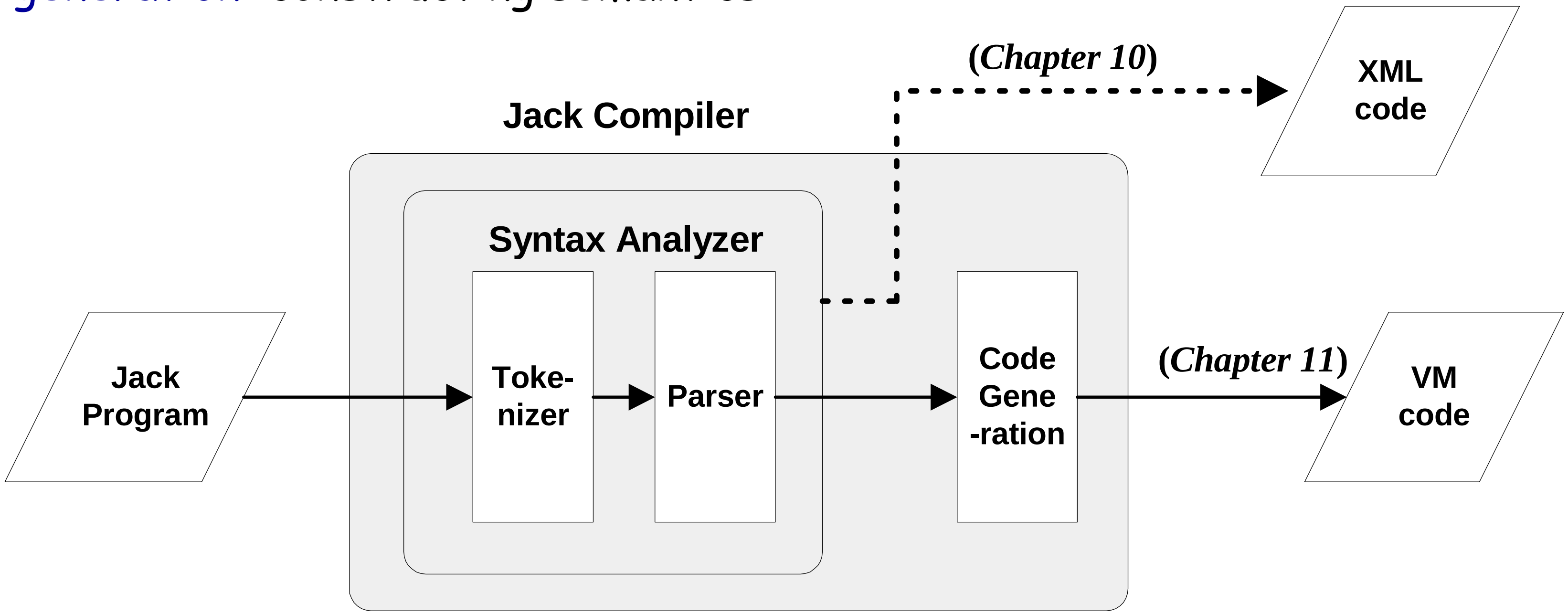
`compileStatements`	--	--	Compiles a sequence of statements, not including the enclosing "{}".
`compileDo`	--	--	Compiles a `do` statement.
`compileLet`	--	--	Compiles a `let` statement.
`compileWhile`	--	--	Compiles a `while` statement.
`compileReturn`	--	--	Compiles a `return` statement.
`compileIf`	--	--	Compiles an `if` statement, possibly with a trailing `else` clause.

CompilationEngine (cont.)

`CompileExpression`	--	--	Compiles an expression.
`CompileTerm`	--	--	Compiles a *term*. This routine is faced with a slight difficulty when trying to decide between some of the alternative parsing rules. Specifically, if the current token is an identifier, the routine must distinguish between a variable, an array entry, and a subroutine call. A single look-ahead token, which may be one of "[", "(", or "." suffices to distinguish between the three possibilities. Any other token is not part of this term and should not be advanced over.
`CompileExpressionList`	--	--	Compiles a (possibly empty) comma-separated list of expressions.

Summary and next step

- Syntax analysis: understanding syntax
- Code generation: constructing semantics



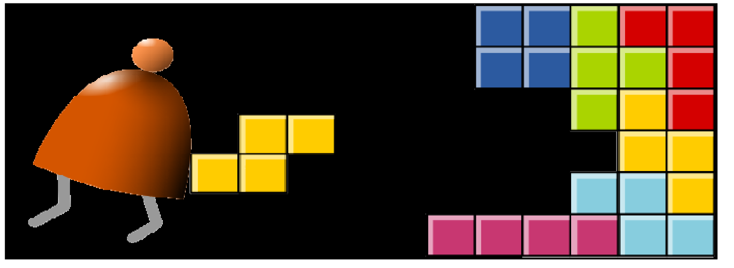
<u>The code generation challenge:</u>

- Extend the syntax analyzer into a full-blown compiler that, instead of generating passive XML code, generates executable VM code
- Two challenges: (a) handling data, and (b) handling commands.

Perspective

- The parse tree can be constructed on the fly
- Syntax analyzers can be built using:
  - **Lex** tool for tokenizing
  - **Yacc** tool for parsing
  - Do everything from scratch (our approach ...)
- The Jack language is intentionally simple:
  - Statement prefixes: **let**, **do**, ...
  - No operator priority
  - No error checking
  - Basic data types, etc.
- Richer languages require more powerful compilers
- The Jack compiler: designed to illustrate the key ideas that underlie modern compilers, leaving advanced features to more advanced courses
- Industrial-strength compilers:
  - Have good error diagnostics
  - Generate tight and efficient code
  - Support parallel (multi-core) processors.

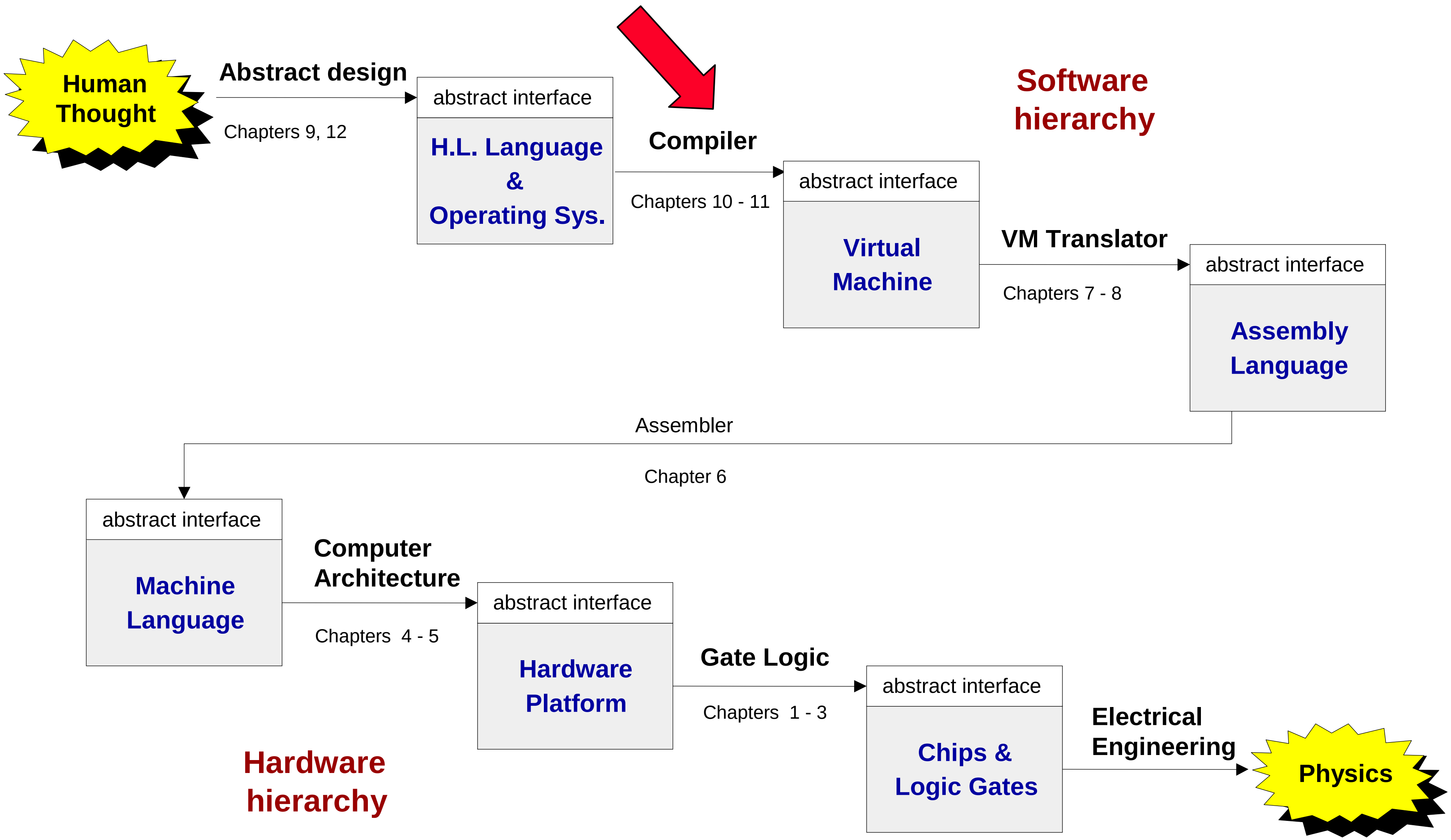
Compiler II: Code Generation



Building a Modern Computer From First Principles

www.nand2tetris.org

Course map


Human Thought
Abstract design
Chapters 9, 12
abstract interface
H.L. Language & Operating Sys.
Compiler
Chapters 10 - 11
Software hierarchy
abstract interface
Virtual Machine
VM Translator
Chapters 7 - 8
abstract interface
Assembly Language
Assembler
Chapter 6
abstract interface
Machine Language
Computer Architecture
Chapters 4 - 5
abstract interface
Hardware Platform
Gate Logic
Chapters 1 - 3
abstract interface
Chips & Logic Gates
Electrical Engineering
Physics
Hardware hierarchy

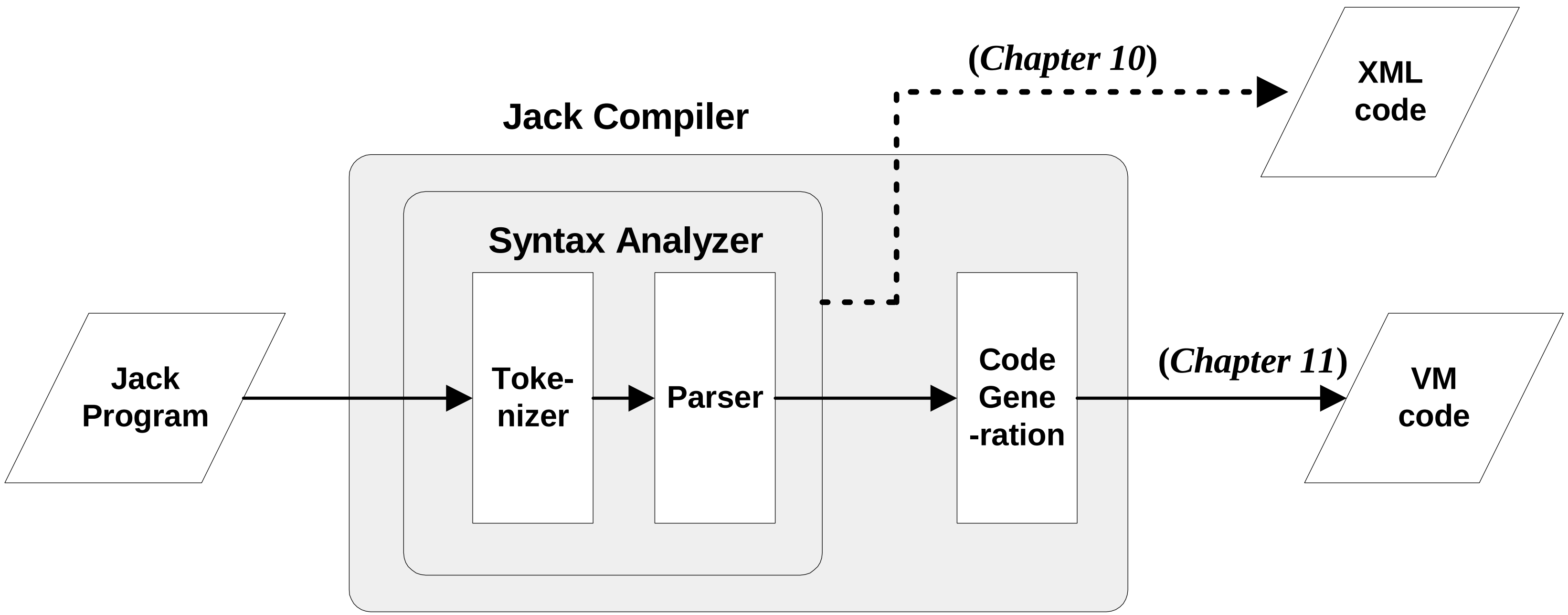
The big picture

1. Syntax analysis: extracting the semantics from the source code

previous lecture

2. Code generation: expressing the semantics using the target language

this lecture



Syntax analysis (review)

```
Class Bar {
    method Fraction foo(int y) {
        var int temp; // a variable
        let temp = (xxx+12)*-63;
        ...
    ...
```

Syntax analyzer

```
<varDec>
  <keyword> var </keyword>
  <keyword> int </keyword>
  <identifier> temp </identifier>
  <symbol> ; </symbol>
</varDec>
<statements>
  <letStatement>
    <keyword> let </keyword>
    <identifier> temp </identifier>
    <symbol> = </symbol>
    <expression>
       <term>
         <symbol> ( </symbol>
         <expression>
           <term>
             <identifier> xxx </identifier>
           </term>
           <symbol> + </symbol>
           <term>
             <int.Const.> 12 </int.Const.>
            </term>
    </expression>
    ...
```

<u>The code generation challenge:</u>

- Program = a series of operations that manipulate data
- Compiler: converts each "understood" (parsed) source operation and data item into corresponding operations and data items in the target language
- Thus, we have to generate code for
  - handling data
  - handling operations
- Our approach: morph the syntax analyzer (project 10) into a full-blown compiler: instead of generating XML, we'll make it generate VM code.

Memory segments (review)

VM memory Commands:

```
pop segment i
push segment i
```

Where *i* is a non-negative integer and *segment* is one of the following:

`static`: holds values of global variables, shared by all functions in the same class

`argument`: holds values of the argument variables of the current function

`local`: holds values of the local variables of the current function

`this`: holds values of the private ("object") variables of the current object

`that`: holds array values (silly name, sorry)

`constant`: holds all the constants in the range `0` … `32767` (pseudo memory segment)

`pointer`: used to anchor `this` and `that` to various areas in the heap

`temp`: fixed 8-entry segment that holds temporary variables for general use; Shared by all VM functions in the program.

Code generation example

```
method int foo() {
  var int x;
  let x = x + 1;
  ...
```

Syntax analysis

```
<letStatement>
  <keyword> let </keyword>
  <identifier> x </identifier>
  <symbol> = </symbol>
  <expression>
    <term>
      <identifier> x </identifier>
      </term>
        <symbol> + </symbol>
      <term>
        <constant> 1 </constant>
       </term>
  </expression>
</letStatement>
```

Code generation

```
push local 0
push constant 1
add
pop local 0
```

(note that x is the first local variable declared in the method)

Handling variables

When the compiler encounters a variable, say x, in the source code, it has to know:

What is x's *data type*?

Primitive, or ADT (class name) ?

(Need to know in order to properly allocate RAM resources for its representation)

What *kind* of variable is x?

`local, static, field, argument` ?

(We need to know in order to properly allocate it to the right memory segment; this also implies the variable's life cycle).

Handling variables: mapping them on memory segments (example)

```
class BankAccount {
   // Class variables
   static int nAccounts;
   static int bankCommission;
   // account properties
   field int id;
   field String owner;
   field int balance;

   method void transfer(int sum, BankAccount from, Date when) {
      var int i, j;    // Some local variables
      var Date due;    // Date is a user-defined type
      let balance = (balance + sum) - commission(sum * 5);
      // More code ...
   }
```

- The target language uses 8 memory segments
- Each memory segment, e.g. `static`, is an indexed sequence of 16-bit values that can be referred to as `static 0`, `static 1`, `static 2`, etc.

When compiling this class, we have to create the following mappings:

The class variables	`nAccounts` , `bankCommission`	are mapped on	`static 0,1`
The object fields	`id`, `owner`, `balance`	are mapped on	`this 0,1,2`
The argument variables	`sum`, `bankAccount`, `when`	are mapped on	`arg 0,1,2`
The local variables	`i`, `j`, `due`	are mapped on	`local 0,1,2`.

Handling variables: symbol tables

```
class BankAccount {
   // Class variables
   static int nAccounts;
   static int bankCommission;
   // account properties
   field int id;
   field String owner;
   field int balance;

   method void transfer(int sum, BankAccount from, Date when) {
      var int i, j;     // Some local variables
      var Date due;     // Date is a user-defined type
      let balance = (balance + sum) - commission(sum * 5);
      // More code ...
   }
```

Class-scope symbol table

Name	Type	Kind	#
nAccounts	int	static	0
bankCommission	int	static	1
id	int	field	0
owner	String	field	1
balance	int	field	2

Method-scope (transfer) symbol table

Name	Type	Kind	#
this	BankAccount	argument	0
sum	int	argument	1
from	BankAccount	argument	2
when	Date	argument	3
i	int	var	0
j	int	var	1
due	Date	var	2

How the compiler uses symbol tables:

- The compiler builds and maintains a linked list of hash tables, each reflecting a single scope nested within the next one in the list
- Identifier lookup works from the current symbol table back to the list's head

  (a classical implementation).

Handling variables: managing their life cycle

Class-scope symbol table

Name	Type	Kind	#
nAccounts	int	static	0
bankCommission	int	static	1
id	int	field	0
owner	String	field	1
balance	int	field	2

Method-scope (transfer) symbol table

Name	Type	Kind	#
this	BankAccount	argument	0
sum	int	argument	1
from	BankAccount	argument	2
when	Date	argument	3
i	int	var	0
j	int	var	1
due	Date	var	2

Variables life cycle

`static` variables: single copy must be kept alive throughout the program duration

`field` variables: different copies must be kept for each object

`local` variables: created on subroutine entry, killed on exit

`argument` variables: similar to `local` variables.

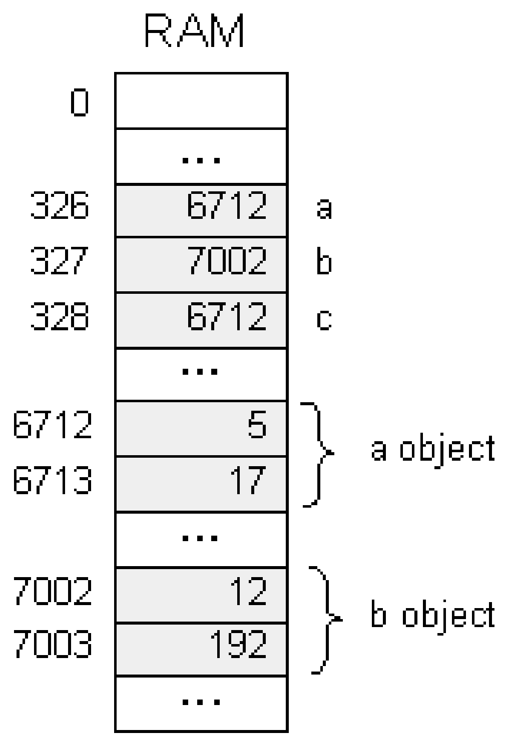
Good news: the VM implementation already handles all these details !

Handling objects: construction / memory allocation

Java code

```
class Complex {
    // Fields (properties):
    int re;  // Real part
    int im;  // Imaginary part
    ...
    /** Constructs a new Complex number */
    public Complex (int re, int im) {
        this.re = re;
        this.im = im;
    }
    ...
}

class Foo {
    public void bla() {
        Complex a, b, c;
        ...
        a = new Complex(5,17);
        b = new Complex(12,192);
        ...
        c = a; // Only the reference is copied
        ...
  }
```

Following compilation:



How to compile:

`foo = new ClassName(…)` ?

The compiler generates code affecting:

`foo = Memory.alloc(n)`

Where `n` is the number of words necessary to represent the object in question, and `Memory.alloc` is an OS method that returns the base address of a free memory block of size `n` words.

Handling objects: accessing fields

Java code

```
class Complex {
    // Properties (fields):
    int re;  // Real part
    int im;  // Imaginary part
    ...
    /** Constructs a new Complex number */
    public Complex(int re, int im) {
        this.re = re;
        this.im = im;
    }
    ...
    /** Multiplies this Complex number
        by the given scalar */
    public void mult (int c) {
        re = re * c;
        im = im * c;
    }
    ...
}
```

<u>How to compile:</u>

`im = im * c` **?**

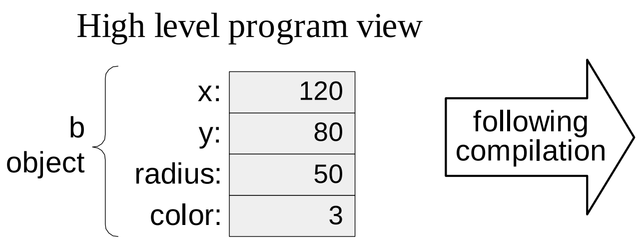
1. look up the two variables in the symbol table
2. Generate the code:

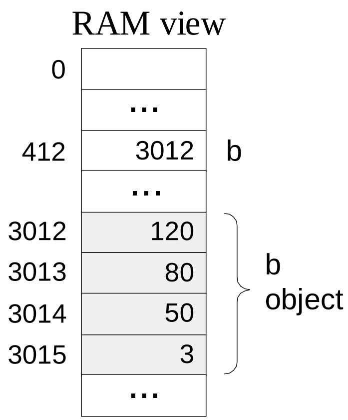
```
*(this + 1) = *(this + 1)
              times
              (argument 0)
```

This pseudo-code should be expressed in the target language.

Handling objects: establishing access to the object's fields

<u>Background:</u> Suppose we have an object named `b` of type `Ball`. A `Ball` has `x`,`y` coordinates, a `radius`, and a `color`.





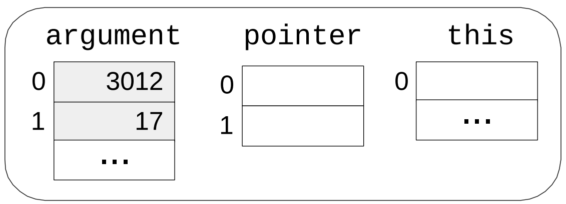
(Actual RAM locations of program variables are run-time dependent, and thus the addresses shown here are arbitrary examples.)

Assume that `b` and `r` were passed to the function as its first two arguments.

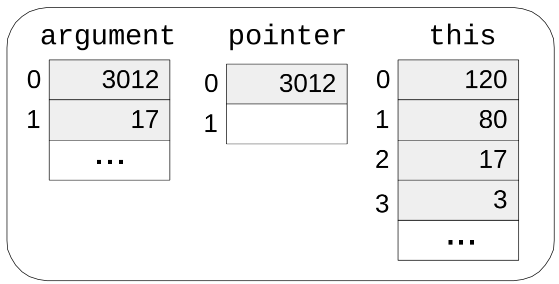
<u>How to compile (in Java):</u>

```
b.radius = r  ?

// Get b's base address:
push argument 0
// Point the this segment to b:
pop pointer 0
// Get r's value
push argument 1
// Set b's third field to r:
pop this 2
```

Virtual memory segments just before the operation `b.radius=17`:



Virtual memory segments just after the operation `b.radius=17`:



(`this 0` is now alligned with `RAM[3012]`)

Handling objects: method calls

Java code

```
class Complex {
    // Properties (fields):
    int re;  // Real part
    int im;  // Imaginary part
    ...
    /** Constructs a new Complex object. */
    public Complex(int re, int im) {
        this.re = re;
        this.im = im;
    }
    ...
}

class Foo {
    ...
    public void bla() {
        Complex x;
        ...
        x = new Complex(1,2);
        x.mult(5);
        ...
   }
}
```

How to compile:

`x.mult(5)` ?

This method call can also be viewed as:

`mult(x,5)`

Generate the following code:

```
push x
push 5
call mult
```

General rule: each method call

`foo.bar(v1,v2,...)`

is translated into:

```
push foo
push v1
push v2
...
call bar
```

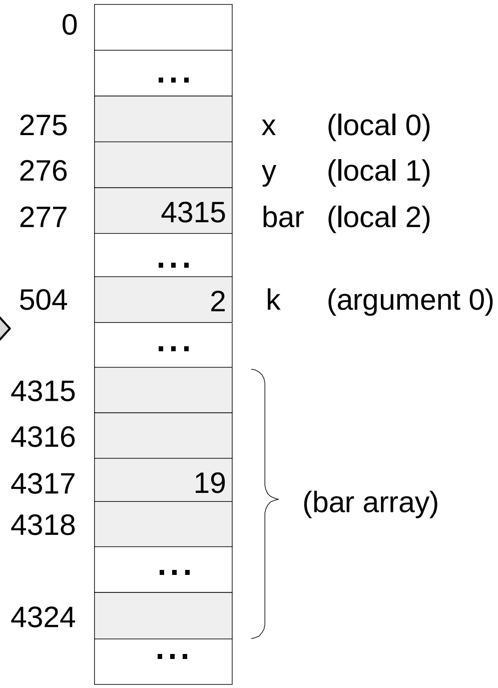
Handling arrays: declaration / construction

Java code

```
class Bla {
 ...
 void foo(int k) {
   int x, y;
   int[] bar; // declare an array
   ...
   // Construct the array:
   bar = new int[10];
   ...
   bar[k]=19;
 }
 ...
 Main.foo(2); // Call the foo method
 ...
```

Following compilation:

RAM state



<u>How to compile:</u>

`bar = new int(n)` ?

Generate code affecting:

`bar = Memory.alloc(n)`

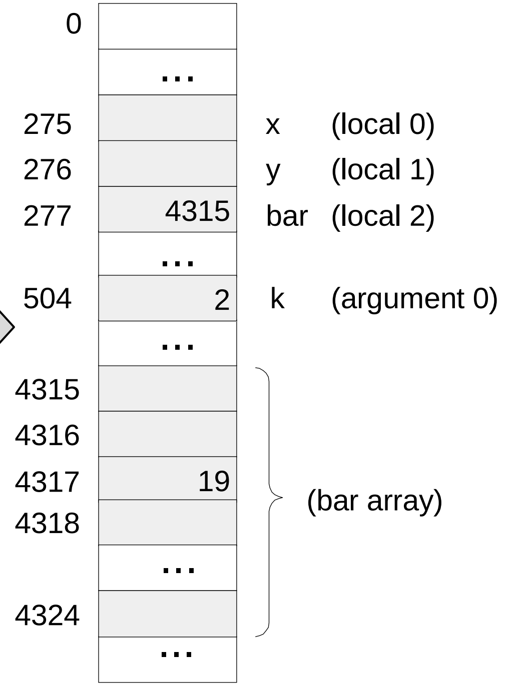
Handling arrays: accessing an array entry by its index

Java code

```
class Bla {
 ...
 void foo(int k) {
   int x, y;
   int[] bar; // declare an array
   ...
   // Construct the array:
   bar = new int[10];
   ...
   bar[k]=19;
 }
 ...
 Main.foo(2); // Call the foo method
 ...
```

Following compilation:

RAM state, just after executing `bar[k] = 19`



<u>How to compile:</u> `bar[k] = 19` ?

VM Code (pseudo)

```
// bar[k]=19, or *(bar+k)=19
push bar
push k
add
// Use a pointer to access x[k]
pop addr // addr points to bar[k]
push 19
pop *addr // Set bar[k] to 19
```

VM Code (actual)

```
// bar[k]=19, or *(bar+k)=19
push local 2
push argument 0
add
// Use the that segment to access x[k]
pop pointer 1
push constant 19
pop that 0
```

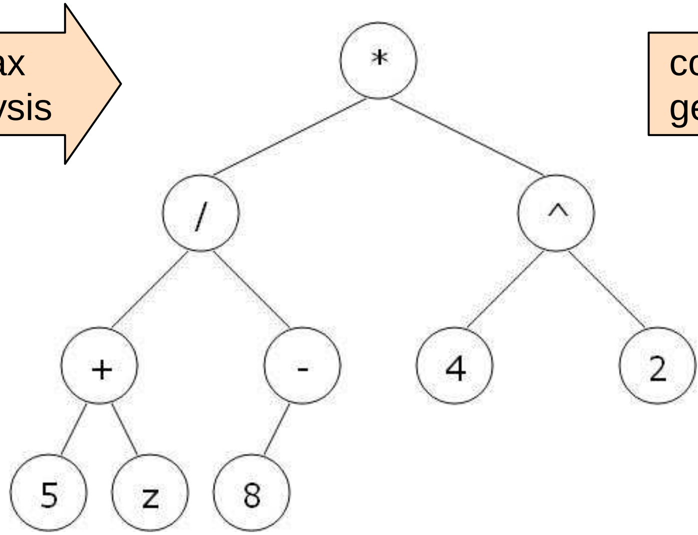
Handling expressions

High-level code

```
((5+z)/-8)*(4^2)
```

syntax analysis

parse tree



code generation

VM code

```
push 5
push z
add
push 8
neg
call div
push 4
push 2
call power
call mult
```

To generate VM code from a parse tree *exp*, use the following logic:

<u>The `codeWrite(`*exp*`)` algorithm</u>:

if *exp* is a constant *n* then output `"push n"`

if *exp* is a variable *v* then output `"push v"`

if *exp* is $op(exp_1)$ then `codeWrite(`exp_1`);` output `"op";`

if *exp* is $(exp_1\ op\ exp_2)$ then `codeWrite(`exp_1`);` `codeWrite(`exp_2`);` output `"op";`

if *exp* is $f\ (exp_1, ..., exp_n)$ then `codeWrite(exp1); ... codeWrite(exp1);` output `"call f";`

Handling program flow

High-level code

```
if (cond)
   s1
else
   s2
...
```

code generation

VM code

```
   VM code to compute and push !(cond)
   if-goto L1
   VM code for executing s1
   goto L2
label L1
   VM code for executing s2
label L2
   ...
```

High-level code

```
while (cond)
   s
...
```

code generation

VM code

```
label L1
   VM code to compute and push !(cond)
   if-goto L2
   VM code for executing s
   goto L1
label L2
   ...
```

Final example

High level code (BankAccount.jack class file)

```
/* Some common sense was sacrificed in this banking example in order
   to create a non trivial and easy-to-follow compilation example. */
class BankAccount {
   // Class variables
   static int nAccounts;
   static int bankCommission;  // As a percentage, e.g., 10 for 10 percent
   // account properties
   field int id;
   field String owner;
   field int balance;

   method int commission(int x) { /* Code omitted */ }

   method void transfer(int sum, BankAccount from, Date when) {
      var int i, j;   // Some local variables
      var Date due;   // Date is a user-defined type
      let balance = (balance + sum) - commission(sum * 5);
      // More code ...
      return;
   }
   // More methods ...
}
```

Class-scope symbol table

Name	Type	Kind	#
nAccounts	int	static	0
bankCommission	int	static	1
id	int	field	0
owner	String	field	1
balance	int	field	2

Method-scope (transfer) symbol table

Name	Type	Kind	#
this	BankAccount	argument	0
sum	int	argument	1
from	BankAccount	argument	2
when	Date	argument	3
i	int	var	0
j	int	var	1
due	Date	var	2

Pseudo VM code

```
function BankAccount.commission
  // Code omitted
function BankAccount.trasnfer
  // Code for setting "this" to point
  // to the passed object (omitted)
  push balance
  push sum
  add
  push this
  push sum
  push 5
  call multiply
  call commission
  sub
  pop balance
  // More code ...
  push 0
  return
```

Final VM code

```
function BankAccount.commission 0
  // Code omitted
function BankAccount.trasnfer 3
  push argument 0
  pop pointer 0
  push this 2
  push argument 1
  add
  push argument 0
  push argument 1
  push constant 5
  call Math.multiply 2
  call BankAccount.commission 2
  sub
  pop this 2
  // More code ...
  push 0
  return
```

Perspective

Jack simplifications that are challenging to extend:

- Limited primitive type system
- No inheritance
- No public class fields, e.g. must use `r = c.getRadius()`
  rather than `r = c.radius`

Jack simplifications that are easy to extend: :

- Limited control structures, e.g. no `for`, `switch`, …
- Cumbersome handling of `char` types, e.g. cannot use `let x='c'`

Optimization

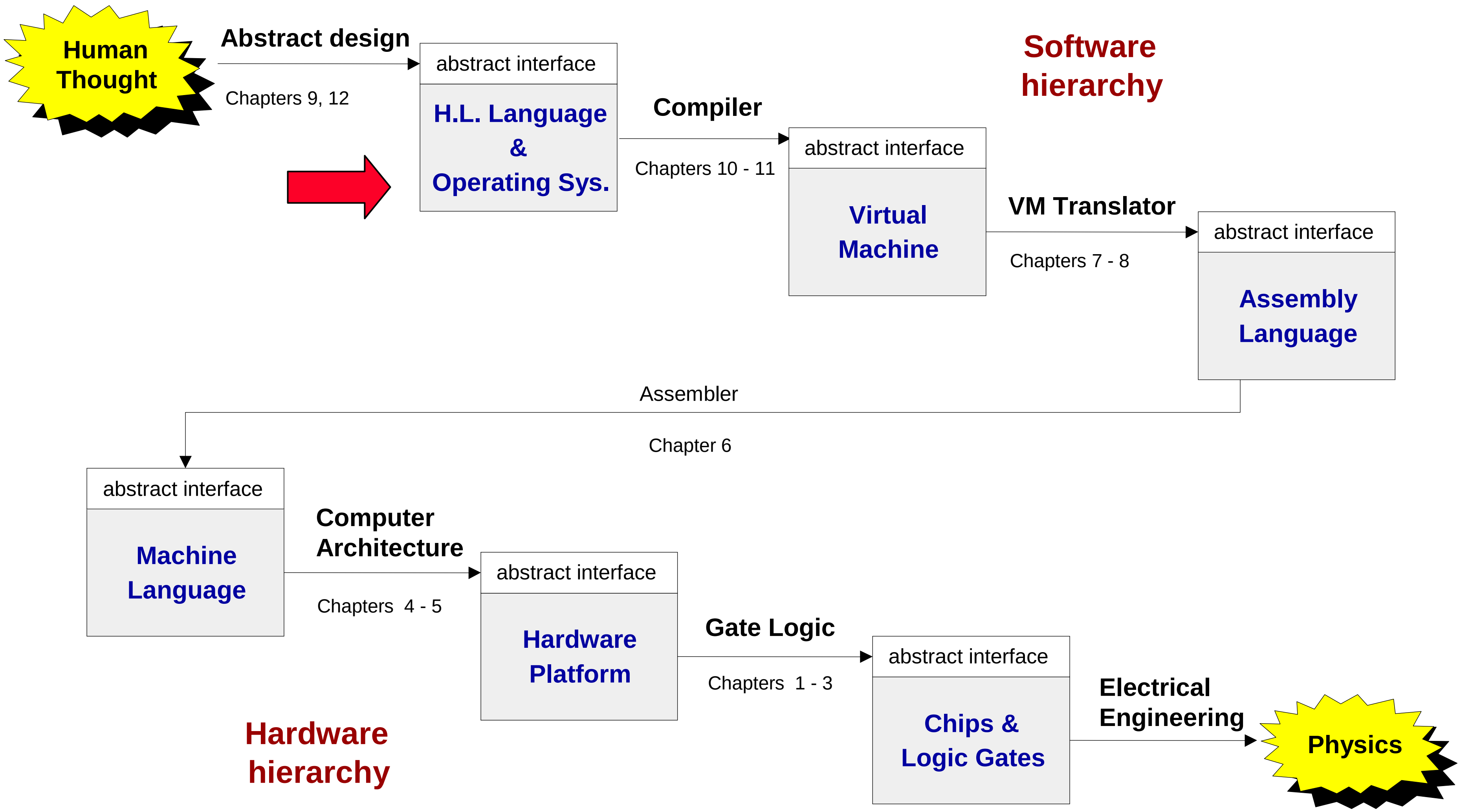
- For example, `c=c+1` is translated inefficiently into `push c`, `push 1`, add, `pop c`.
- Parallel processing
- Many other examples of possible improvements …

Operating Systems

Building a Modern Computer From First Principles

www.nand2tetris.org

Where we are at:



Jack revisited

```
/** 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;
  }
}
```

Jack revisited

```
/** 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;
  }
}
```

Typical OS functions

Language extensions / standard library

- Mathematical operations (**abs**, **sqrt**, ...)
- Abstract data types (**String**, **Date**, ...)
- Output functions (**printChar**, **printString** ...)
- Input functions (**readChar**, **readLine** ...)
- Graphics functions (**drawPixel**, **drawCircle**, ...)
- And more ...

System-oriented services

- Memory management (objects, arrays, ...)
- I/O device drivers
- Mass storage
- File system
- Multi-tasking
- UI management (shell / windows)
- Security
- Communications
- And more ...

The Jack OS

- **`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.

Jack OS API

```
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)
}
```

A typical OS:

- Is modular and scalable
- Empowers programmers (language extensions)
- Empowers users (file system, GUI, ...)
- Closes gaps between software and hardware
- Runs in “protected mode”
- Typically written in some high level language
- Typically grows gradually, assuming more and more functions
- Must be efficient.

Efficiency

We have to implement various operations on *n*-bit binary numbers (n = 16, 32, 64, ...).

<u>For example, consider *multiplication*</u>

- Naïve algorithm: to multiply x*y: { for i = 1 ... y do sum = sum + x }

  Run-time is proportional to *y*

  In a 64-bit system, *y* can be as large as 2^{64}.

  Multiplications can take years to complete

- Algorithms that operate on *n*-bit inputs can be either:

  - <u>Naïve</u>: run-time is proportional to the <u>*value*</u> of the *n*-bit inputs

  - <u>Good</u>: run-time is proportional to *n, the input's <u>size.</u>*

Example I: multiplication

The "steps"

		1	0	1	1	=	11
			1	0	1	=	5
		1	0	1	1		
	0	0	0	0			
1	0	1	1				
1	1	0	1	1	1	=	55

The algorithm explained (first 4 of 16 iteration)

x:	0	0	0	1	0	1	1	
y:	0	0	0	0	1	0	1	j'th bit of y
	0	0	0	**1**	**0**	**1**	**1**	1
	0	0	**1**	**0**	**1**	**1**	0	0
	0	**1**	**0**	**1**	**1**	0	0	1
	1	**0**	**1**	**1**	0	0	0	0
$x \cdot y$:	0	1	1	0	1	1	1	sum

multiply(x, y):

// Where $x, y \geq 0$

$sum = 0$

$shiftedX = x$

for $j = 0 \ldots (n-1)$ do

  if (j-th bit of y) $= 1$ then

    $sum = sum + shiftedX$

  $shiftedX = shiftedX * 2$

- Run-time: proportional to *n*
- Can be implemented in SW or HW
- Division: similar idea.

Example II: square root

The square root function has two convenient properties:

- It's inverse function is computed easily
- Monotonically increasing

Functions that have these two properties can be computed by binary search:

sqrt(x):

// Compute the integer part of $y = \sqrt{x}$. Strategy:

// Find an integer y such that $y^2 \le x < (y+1)^2$ (for $0 \le x < 2^n$)

// By performing a binary search in the range $0 \ldots\ 2^{n/2} - 1$.

$y = 0$

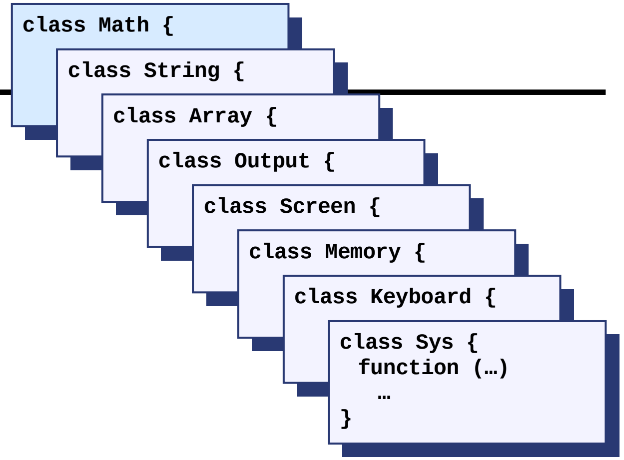
for $j = n/2 - 1 \ldots 0$ do

  if $(y + 2^j)^2 \le x$ then $y = y + 2^j$

return y

Number of loop iterations is bounded by n/2, thus the run-time is O(n).

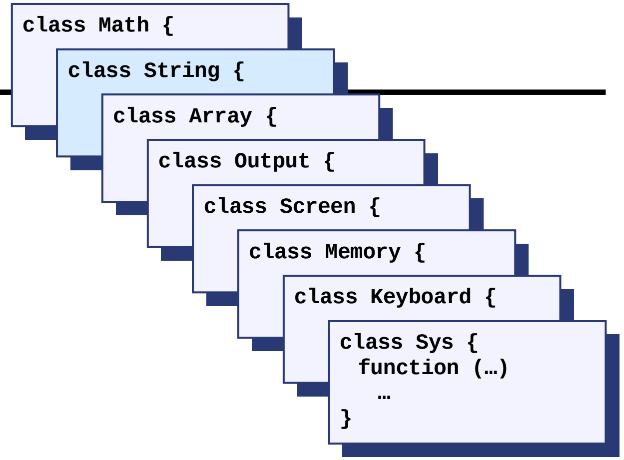
Math operations (in the Jack OS)




```
class Math {

    function void init()

    function int abs(int x)

    function int multiply(int x, int y)

    function int divide(int x, int y)

    function int min(int x, int y)

    function int max(int x, int y)

    function int sqrt(int x)

}
```

The remaining functions are simple to implement.

String processing (in the Jack OS)




```
Class String {

   constructor String new(int maxLength)

   method void   dispose()

   method int    length()

   method char   charAt(int j)

   method void   setCharAt(int j, char c)

   method String appendChar(char c)

   method void   eraseLastChar()

   method int    intValue()

   method void   setInt(int j)

   function char backSpace()

   function char doubleQuote()

   function char newLine()

}
```

Single digit ASCII conversions

Character:	'0'	'1'	'2'	'3'	'4'	'5'	'6'	'7'	'8'	'9'
ASCII code:	48	49	50	51	52	53	54	55	56	57

- asciiCode(digit) == digit + 48
- digit(asciiCode) == asciiCode - 48

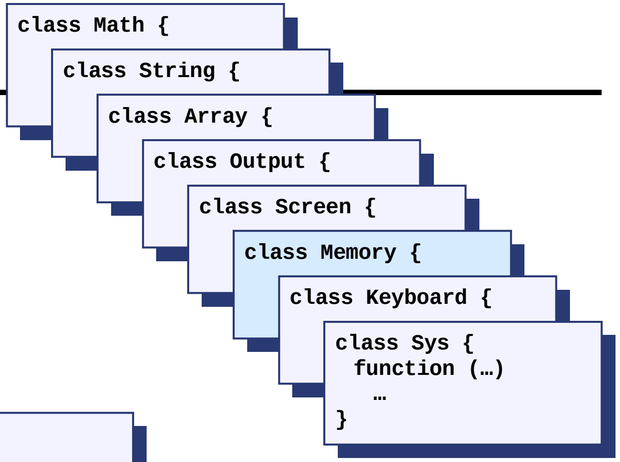
Converting a number to a string

- SingleDigit-to-character conversions: done
- Number-to-string conversions:

```
// Convert a non-negative number to a string
int2String(n):
    lastDigit = n % 10
    c = character representing lastDigit
    if n < 10
        return c  (as a string)
    else
        return int2String(n / 10).append(c)
```

```
// Convert a string to a non-negative number
string2Int(s):
    v = 0
    for i = 1 ... length of s do
        d = integer value of the digit s[i]
        v = v * 10 + d
    return v
    // (Assuming that s[1] is the most
    //  significant digit character of s.)
```

Memory management (in the Jack OS)




```
class Memory {

    function int peek(int address)

    function void poke(int address, int value)

    function Array alloc(int size)

    function void deAlloc(Array o)

}
```

Memory management (naive)

- When a program constructs (destructs) an object, the OS has to allocate (de-allocate) a RAM block on the heap:
  - `alloc(size)`: returns a reference to a free RAM block of size `size`
  - `deAlloc(object)`: recycles the RAM block that `object` refers to

Initialization: $free = heapBase$

// Allocate a memory block of *size* words.

alloc(*size*):

  $pointer = free$

  $free = free + size$

  return *pointer*

// De-allocate the memory space of a given *object*.

deAlloc(*object*):

  do nothing

- The data structure that this algorithm manages is a single pointer: free.

Memory management (improved)

Initialization:

$freeList = heapBase$

$freeList.length = heapLength$

$freeList.next = \text{null}$

// Allocate a memory space of *size* words.

alloc(*size*):

Search *freeList* using best-fit or first-fit heuristics
to obtain a segment with *segment.length* > *size*

If no such segment is found, return failure
(or attempt defragmentation)

block = needed part of the found segment
(or all of it, if the segment remainder is too small)

Update *freeList* to reflect the allocation

$block[-1] = size + 1$ // Remember block size, for de-allocation

Return *block*

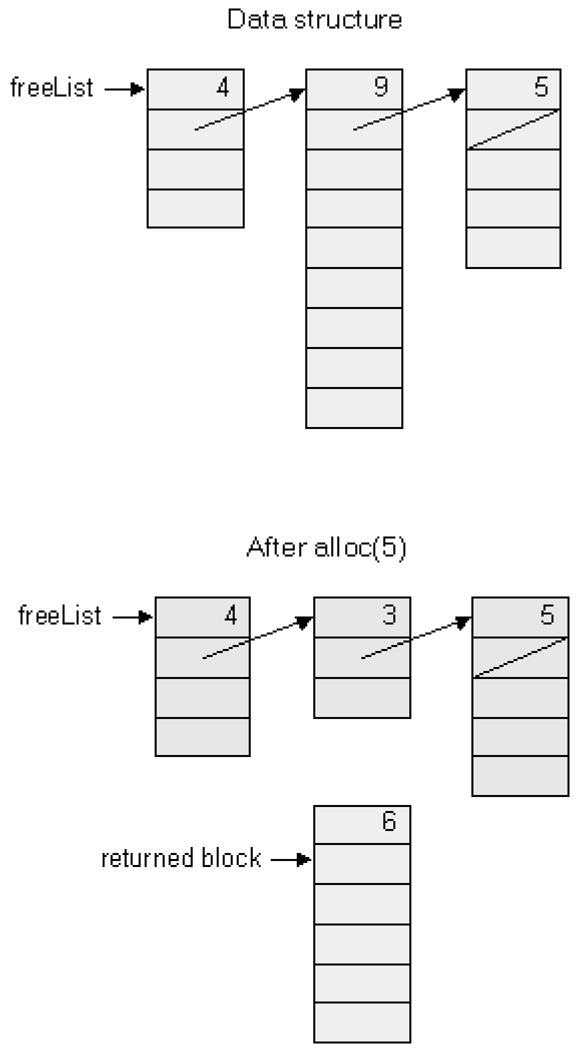
// Deallocate a decommissioned *object*.

deAlloc(*object*):

$segment = object - 1$

$segment.length = object[-1]$

Insert *segment* into the *freeList*



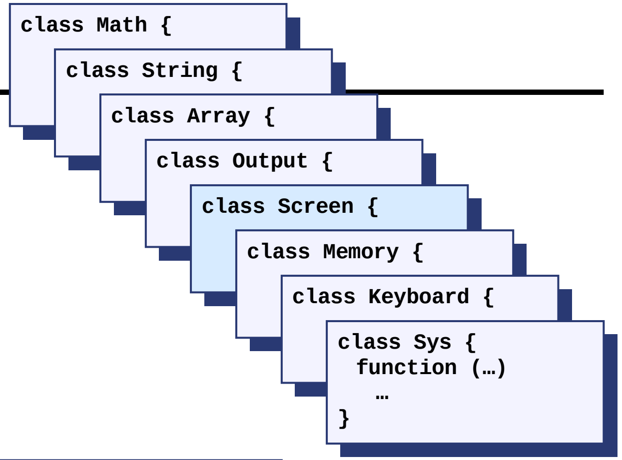
Peek and poke

```
class Memory {

    function int peek(int address)

    function void poke(int address, int value)

    function Array alloc(int size)

    function void deAlloc(Array o)

}
```

- Implementation: based on our ability to exploit exotic casting in Jack:

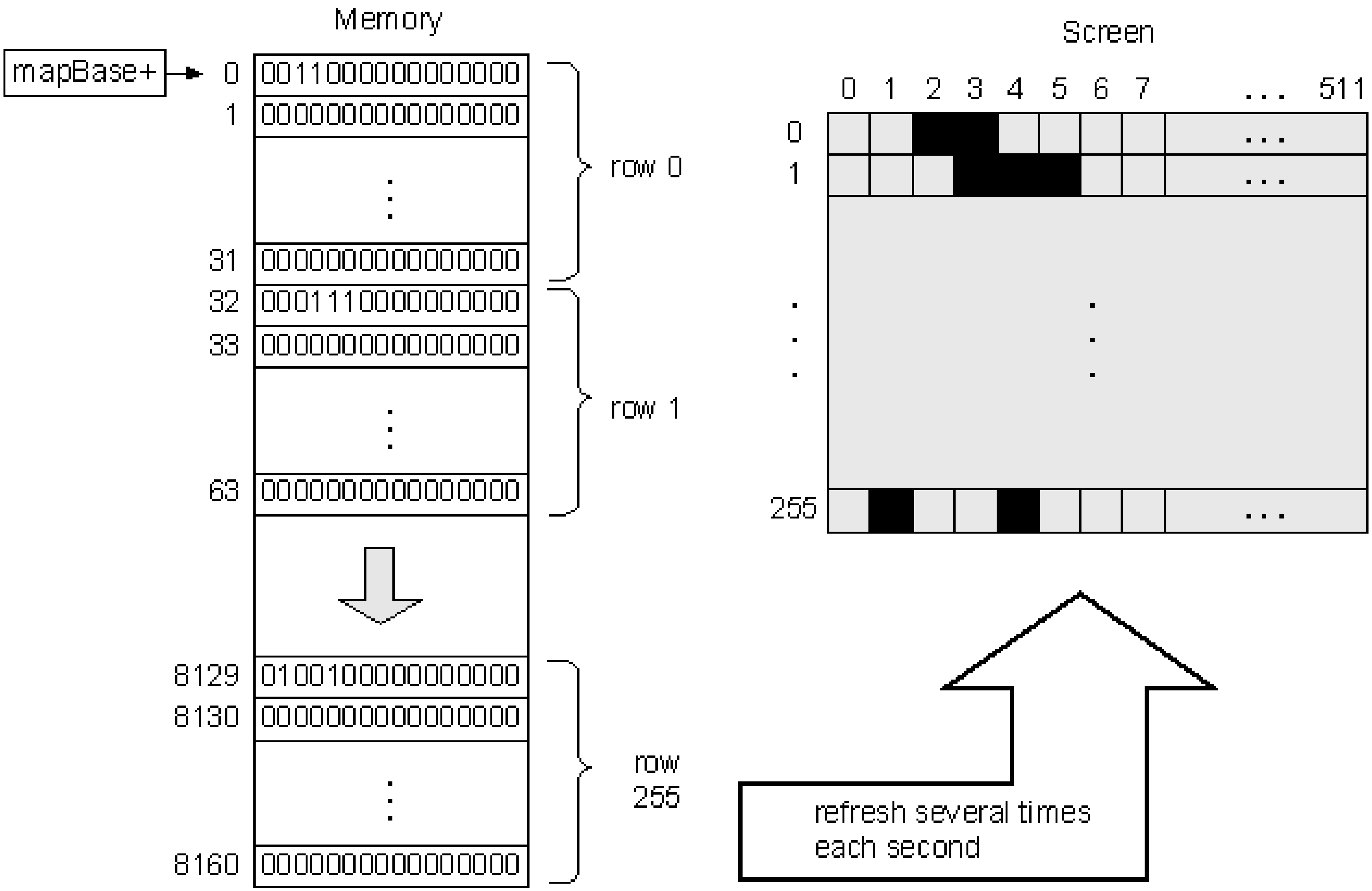
```
// To create a Jack-level "proxy" of the RAM:
var Array memory;
let memory = 0;
// From this point on we can use code like:
let x = memory[j]   // Where j is any RAM address
let memory[j] = y   // Where j is any RAM address
```

Graphics primitives (in the Jack OS)




```
Class Screen {

    function void clearScreen()

    function void setColor(boolean b)

    function void drawPixel(int x, int y)

    function void drawLine(int x1, int y1, int x2, int y2)

    function void drawRectangle(int x1, int y1,int x2, int y2)

    function void drawCircle(int x, int y, int r)

}
```

Memory-mapped screen



Pixel drawing

drawPixel (x, y):

// Hardware-specific.

// Assuming a memory mapped screen:

Write a predetermined value in the RAM location corresponding to screen location (x, y).

- Implementation: using poke(address,value)

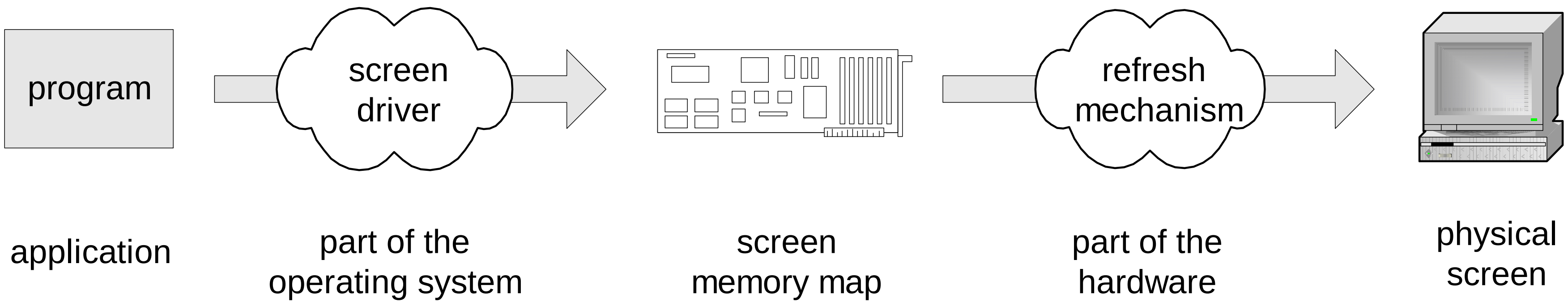
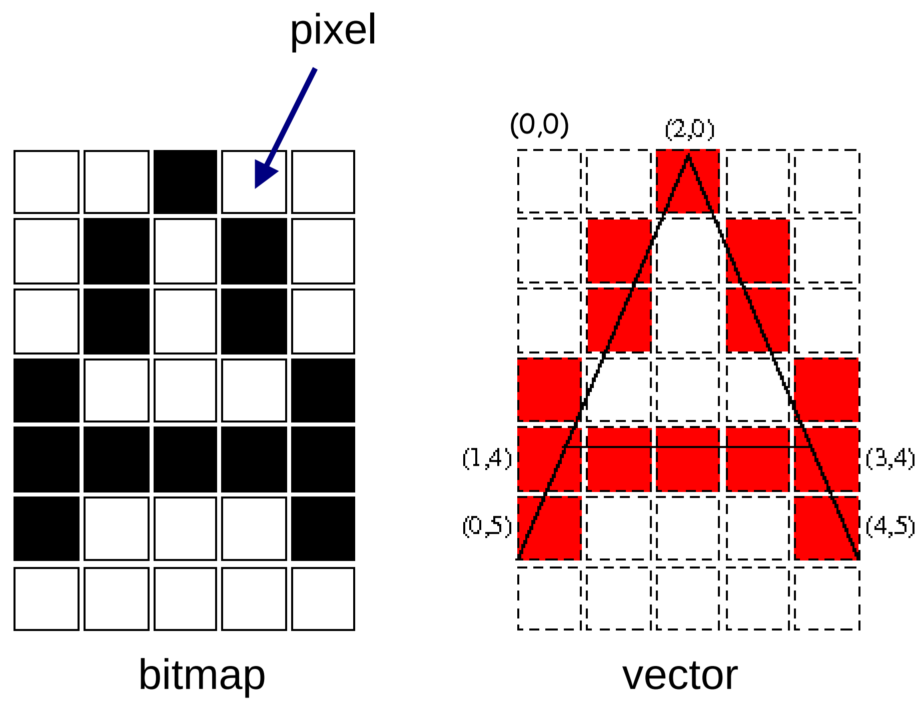
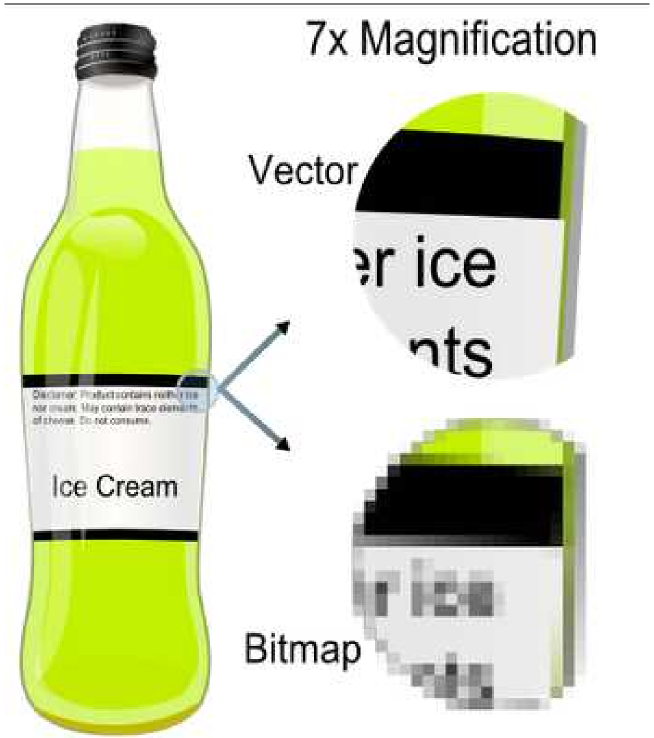


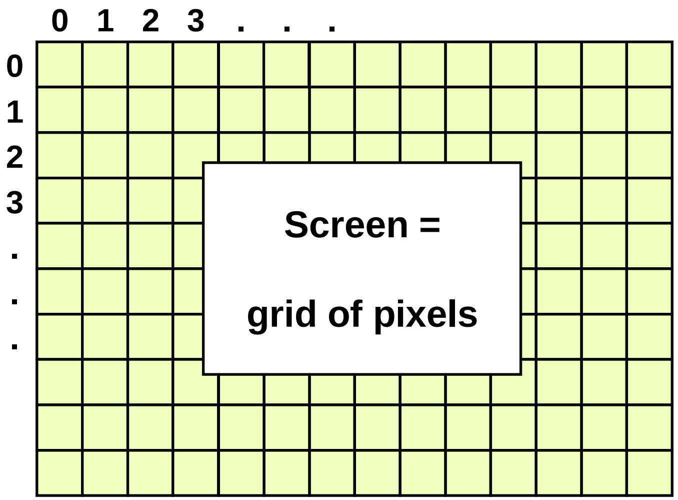
Image representation: bitmap versus vector graphics

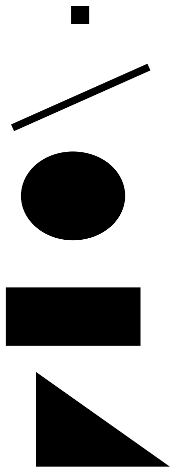




- Bitmap file: 00100, 01010,01010,10001,11111,10001,00000, . . .
- Vector graphics file: drawLine(2,0,0,5), drawLine(2,0,4,5), drawLine(1,4,3,4)
- Pros and cons of each method.

Vector graphics: basic operations





drawPixel(x,y) (Primitive operation)

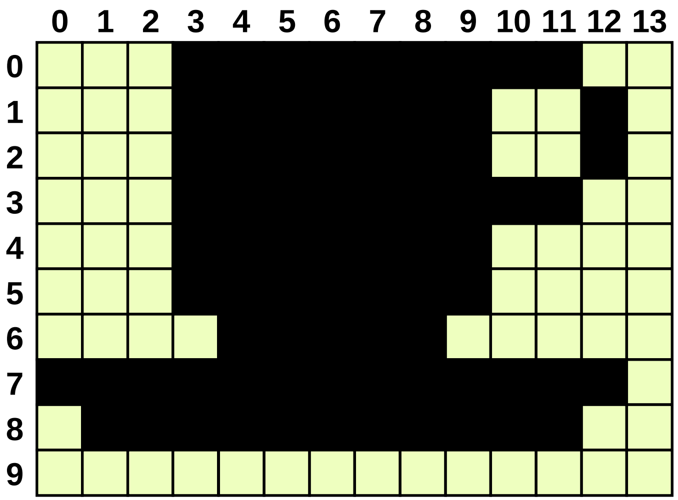
drawLine(x1,y1,x2,y2)

drawCircle(x,y,r)

drawRectangle(x1,y1,x2,y2)

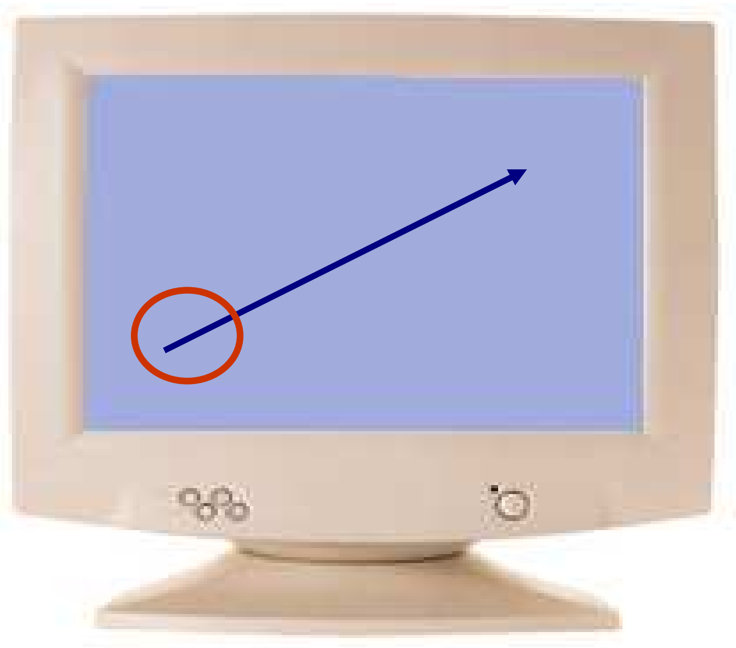
drawTriangle(x1,y1,x2,y2,x3,y3)

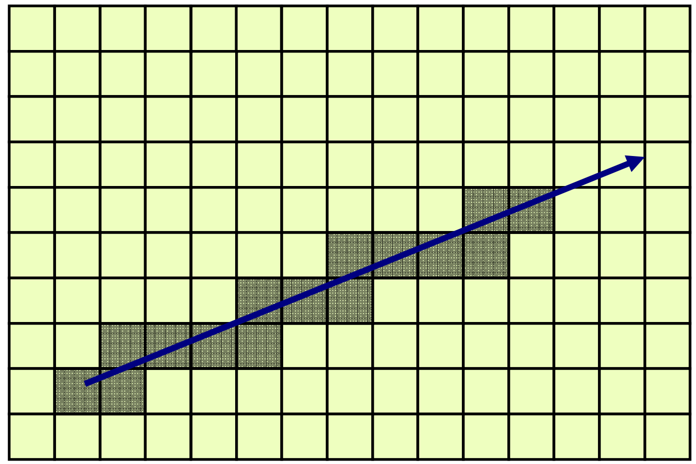
etc. (a few more similar operations)



```
drawLine(0,3,0,11)
drawRectangle(1,3,5,9)
drawLine(1,12,2,12)
drawLine(3,10,3,11)
drawLine(6,4,6,9)
drawLine(7,0,7,12)
drawLine(8,1,8,12)
```

How to draw a line?



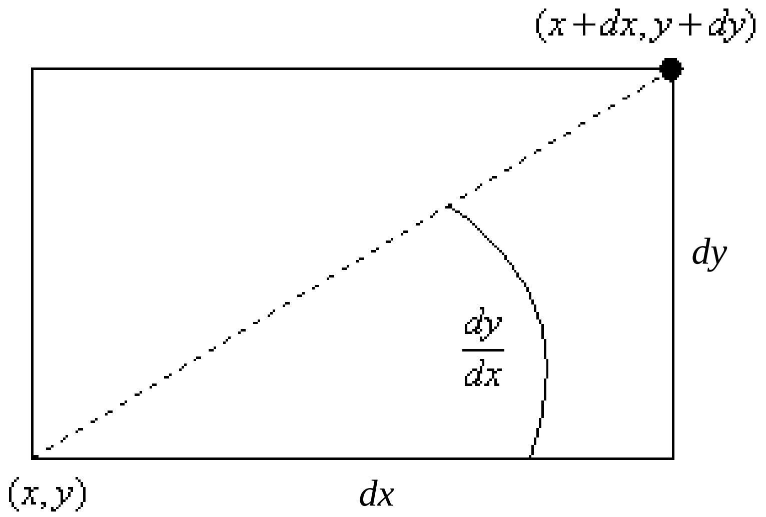


drawLine(x1,y1,x2,y2)

- <u>Basic idea:</u> drawLine is implemented through a sequence of drawPixel operations
- <u>Challenge 1:</u> which pixels should be drawn ?
- <u>Challenge 2:</u> how to draw the line *fast* ?
- <u>Simplifying assumption:</u> the line that we are asked to draw goes north-east.

Line Drawing

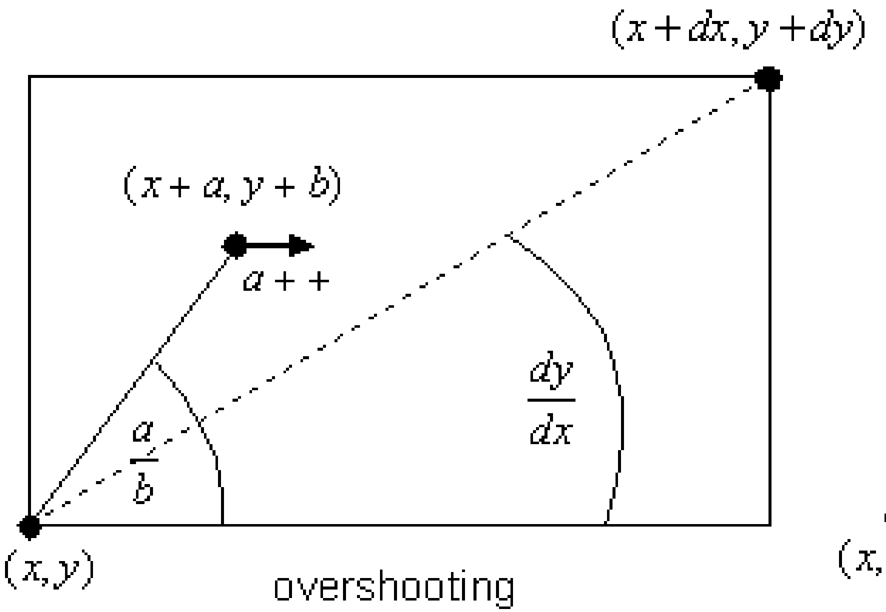
- Given: drawLine(x1,y1,x2,y2)
- Notation: x=x1, y=y1, dx=x2-x1, dy=y2-y1
- Using the new notation:
  We are asked to draw a line between (x,y) and (x+dx,y+dy)

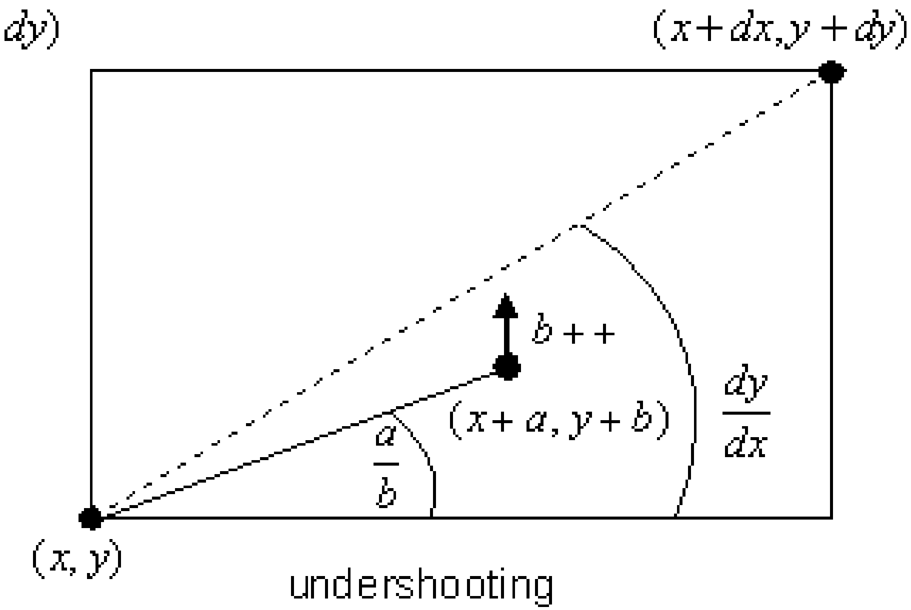



```
set (a,b) = (0,0)
while there is more work to do
    drawPixel(x+a,y+b)
    decide if you want to go right, or up
    if you decide to go right, set a=a+1;
     if you decide to go up, set b=b+1
```

```
set (a,b) = (0,0)
while (a ≤ dx) and (b ≤ dy)
    drawPixel(x+a,y+b)
    decide if you want to go right, or up
    if you decide to go right, set a=a+1;
     if you decide to go up, set b=b+1
```

Line Drawing algorithm





<u>drawLine(x,y,x+dx,y+dy)</u>

set (a,b) = (0,0)

while (a ≤ dx) and (b ≤ dy)

  drawPixel(x+a,y+b)

  decide if you want to go right, or up

  if you decide to go right, set a=a+1;
  if you decide to go up, set b=b+1

<u>drawLine(x,y,x+dx,y+dy)</u>

set (a,b) = (0,0)

while (a ≤ dx) and (b ≤ dy)

  drawPixel(x+a,y+b)

Oy Vey

  if b/a > dy/dx set a=a+1
  else set b=b+1

Line Drawing algorithm, optimized

```
drawLine(x,y,x+dx,y+dy)
set (a,b) = (0,0)
while (a ≤ dx) and (b ≤ dy)
      drawPixel(x+a,y+b)
      if b/a > dy/dx set a=a+1
       else          set b=b+1
```

Motivation

- When you draw polygons, e.g. in animation or video, you need to draw millions of lines
- Therefore, drawLine must be ultra fast
- Division is a very slow operation
- Addition is ultra fast (hardware based)

```
drawLine(x,y,x+dx,y+dy)
set (a,b) = (0,0),  diff = 0
while (a ≤ dx) and (b ≤ dy)
      drawPixel(x+a,y+b)
      if diff < 0 set a=a+1,  diff = diff + dx
       else       set b=b+1, diff = diff - dy
```

b/a > dy/dx   is the same as   a*dy < b*dx

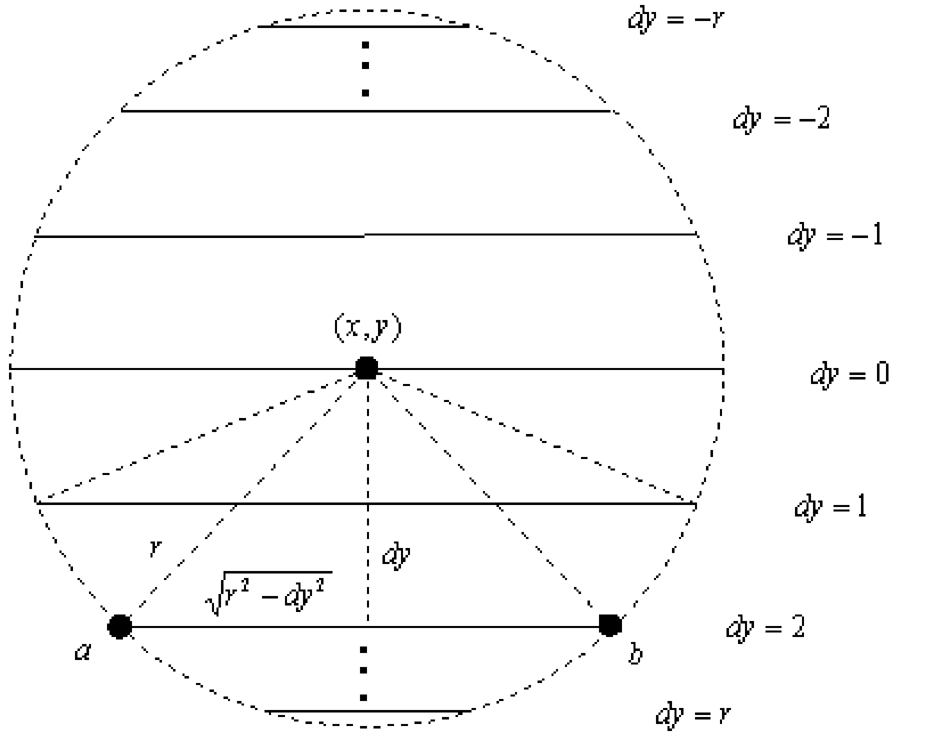
Define diff = a*dy – b*dx

Let's take a close look at this diff:

1. b/a > dy/dx is the same as diff < 0
2. When we set (a,b)=(0,0), diff = 0
3. When we set a=a+1, diff goes up by dy
4. When we set b=b+1, diff goes down by dx

Circle drawing

The screen origin (0,0) is at the top left.



point $a = (x - \sqrt{r^2 - dy^2}, y + dy)$

point $b = (x + \sqrt{r^2 - dy^2}, y + dy)$

drawCircle(x, y, r):

for each $dy \in -r \ldots r$ do

drawLine from $(x - \sqrt{r^2 - dy^2}, y + dy)$ to $(x + \sqrt{r^2 - dy^2}, y + dy)$

An anecdote about efficiency and design

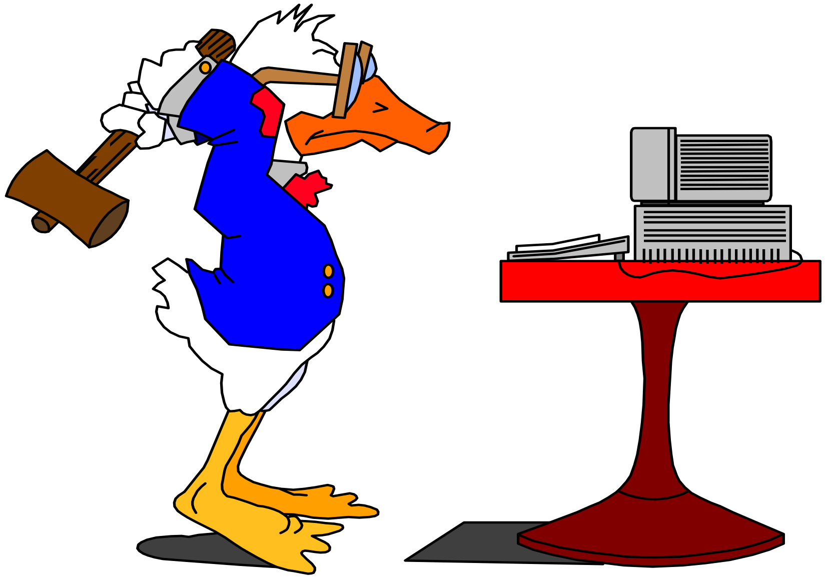
… Jobs obsessed about the look of what would appear on the screen. One day Bill Atkinson burst into his office all excited. He had just come up with a brilliant algorithm that could draw circles onscreen quickly. The math for making circles usually required calculating square roots, which the Motorola 68000 microprocessor didn't support. But Atkinson did a workaround based on the fact that the sum of a sequence of odd numbers produces a sequence of perfect squares (e.g. 1 + 3 = 4, 1 + 3 + 5 = 9, etc.)

When Atkinson fired up his demo, everyone was impressed except Jobs. "Well, circles are nice," he said, "but how about drawing rectangles with rounded corners?"

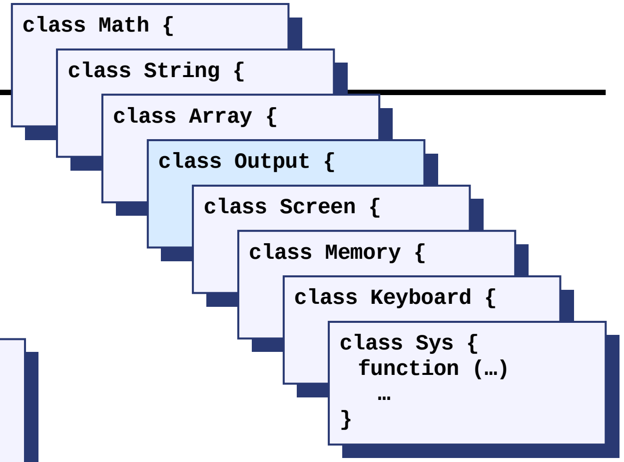
(*Steve Jobs*, by Walter Isaacson, 2012)

To sum up (vector graphics)…

- To do vector graphics (e.g. display a PPT file), you have to draw polygons
- To draw polygons, you need to draw lines
- To draw lines, you need to divide
- Division can be
  re-expressed as multiplication
- Multiplication can be
  reduced to addition
- Addition is easy.



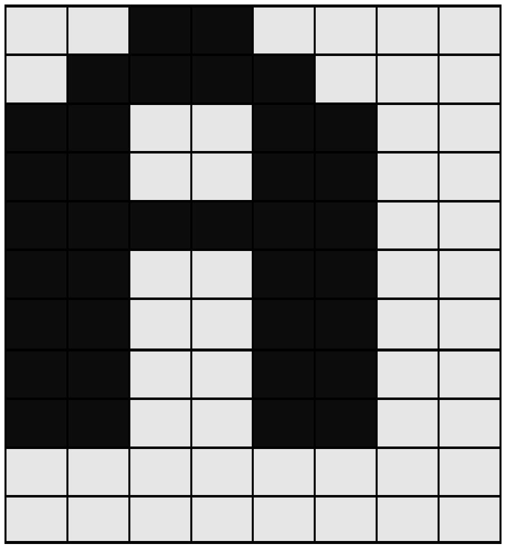
Character output primitives (in the Jack OS)




```
class Output {

    function void moveCursor(int i, int j)

    function void printChar(char c)

    function void printString(String s)

    function void printInt(int i)

    function void println()

    function void backSpace()

}
```

Character output

- Given display: a physical screen, say 256 rows by 512 columns
- We can allocate an 11 by 8 grid for each character
- Hence, our output package should manage a 23 lines by 64 characters screen
- Font: each displayable character must have an agreed-upon bitmap
- In addition, we have to manage a “cursor”.



Font implementation (in the Jack OS)

```
class Output {
   static Array charMaps;
   function void initMap() {
      let charMaps = Array.new(127);
      // Assign a bitmap for each character
      do Output.create(32,0,0,0,0,0,0,0,0,0,0,0);          // space
      do Output.create(33,12,30,30,30,12,12,0,12,12,0,0);  // !
      do Output.create(34,54,54,20,0,0,0,0,0,0,0,0);       // “
      do Output.create(35,0,18,18,63,18,18,63,18,18,0,0);  // #
      ...
      do Output.create(48,12,30,51,51,51,51,51,30,12,0,0); // 0
      do Output.create(49,12,14,15,12,12,12,12,12,63,0,0); // 1
      do Output.create(50,30,51,48,24,12,6,3,51,63,0,0);   // 2
      . . .
      do Output.create(65,0,0,0,0,0,0,0,0,0,0,0);          // A ** TO BE FILLED **
      do Output.create(66,31,51,51,51,31,51,51,51,31,0,0); // B
      do Output.create(67,28,54,35,3,3,3,35,54,28,0,0);    // C
      . . .
      return;
 }
```

```
// Creates a character map array
function void create(int index, int a, int b, int c, int d, int e,
                     int f, int g, int h, int i, int j, int k) {
    var Array map;
    let map = Array.new(11);
    let charMaps[index] = map;
    let map[0] = a;
    let map[1] = b;
    let map[2] = c;
    ...
    let map[10] = k;
    return; }
```

Keyboard primitives (in the Jack OS)

```
class Math {
class String {
class Array {
class Output {
class Screen {
class Memory {
class Keyboard {
class Sys {
  function (…)
    …
}
```

```
Class Keyboard {

    function char keyPressed()

    function char readChar()

    function String readLine(String message)

    function int readInt(String message)

}
```

Keyboard input

```
keyPressed():
  // Depends on the specifics of the keyboard interface
  if a key is presently pressed on the keyboard
    return the ASCII value of the key
  else
    return 0
```

- If the RAM address of the keyboard's memory map is known, the above logic can be implemented using a peek function
- Problem I: the elapsed time between a "key press" and key release" events is unpredictable
- Problem II: when pressing a key, the user should get some visible feedback (cursor, echo, ...).

A historic moment remembered

… Wozniak began writing the software that would get the microprocessor to display images on the screen. After a couple of month he was ready to test it. "I typed a few keys on the keyboard and I was shocked! The letters were displayed on the screen."

It was Sunday, June 29, 1975, a milestone for the personal computer. "It was the first time in history," Wozniak later said, "anyone had typed a character on a keyboard and seen it show up on their own computer's screen right in front of them"

(*Steve Jobs*, by Walter Isaacson, 2012)

Keyboard input (cont.)

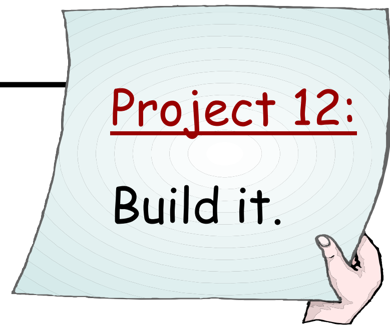
readChar():

```
// Read and echo a single character
display the cursor
while no key is pressed on the keyboard
    do nothing  // wait till the user presses a key
c = code of currently pressed key
while a key is pressed
    do nothing // wait for the user to let go
print c at the current cursor location
move the cursor one position to the right
return c
```

readLine():

```
// Read and echo a "line" (until newline)
s = empty string
repeat
    c = readChar()
    if c = newline character
        print newline
        return s
    else if c = backspace character
            remove last character from s
            move the cursor 1 position back
        else
            s = s.append(c)
return s
```

Jack OS recap

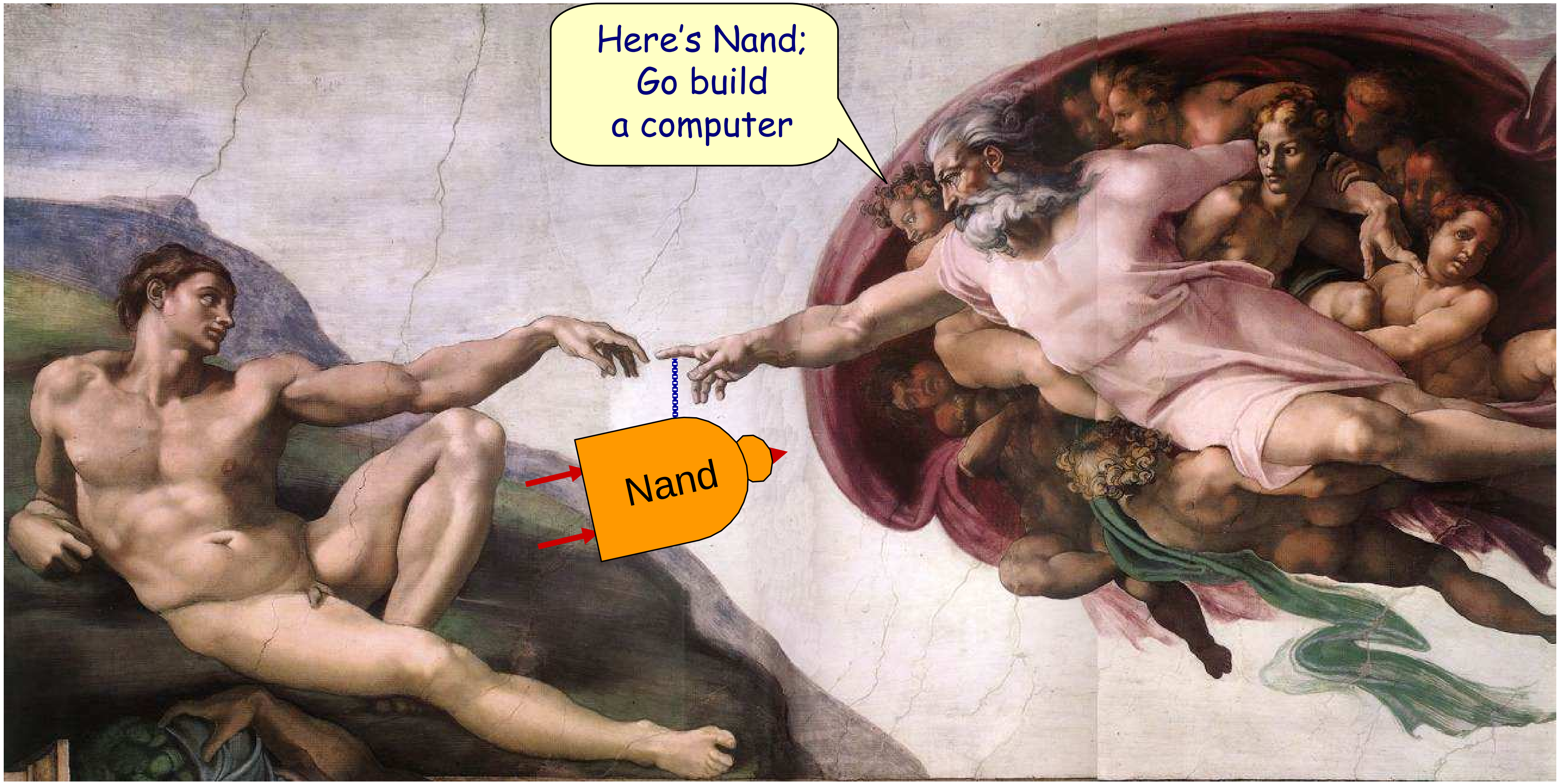



```
class Math {

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)
}
```

- Implementation: just like GNU Unix and Linux were built:
- Start with an existing system,
  and gradually replace it with a new system,
  one library at a time.

Perspective

- What we presented can be described as a:
  - mini OS
  - Standard library
- Many classical OS functions are missing
- No separation between user mode and OS mode
- Some algorithms (e.g. multiplication and division) are standard
- Other algorithms (e.g. line- and circle-drawing) can be accelerated with special hardware
- And, by the way, we've just finished building the computer.



In CS, God gave us Nand

Everything else was done by humans.

Some Final notes

- CS is a science
- What is science?
- Reductionism
- Life science: From Aristo (millions of rules) to Darwin (3 rules) to Watson and Crick (1 rule)
- Computer science: We *knew* in advance that we could build a computer from almost nothing. In this course we actually did it.
- Key lessons:
  - Elegance
  - Clarity
  - Simplicity
  - Playfulness.

a	b	Out
0	0	1
0	1	1
1	0	1
1	1	0