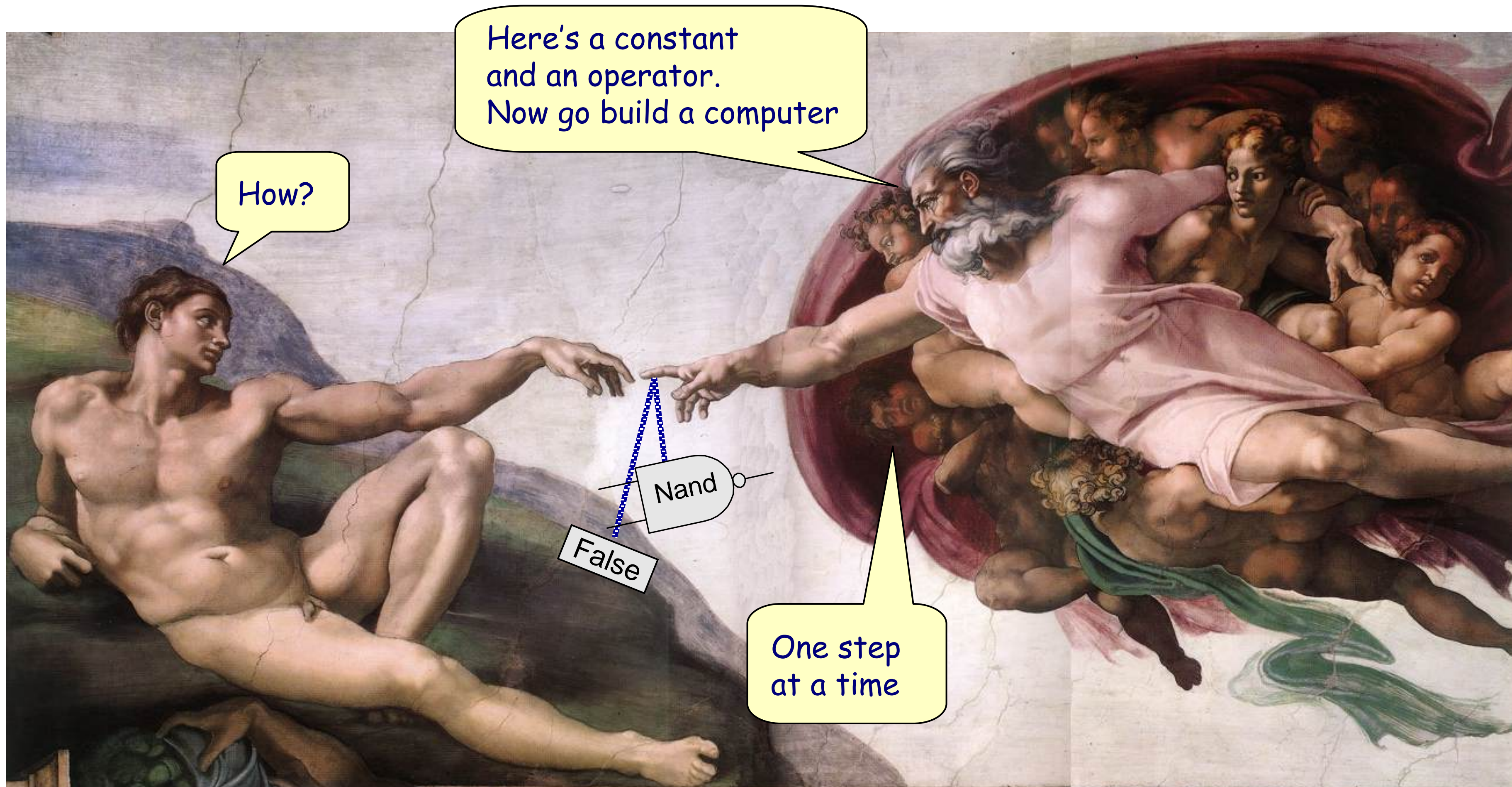
From Nand to Tetris in 12 Steps

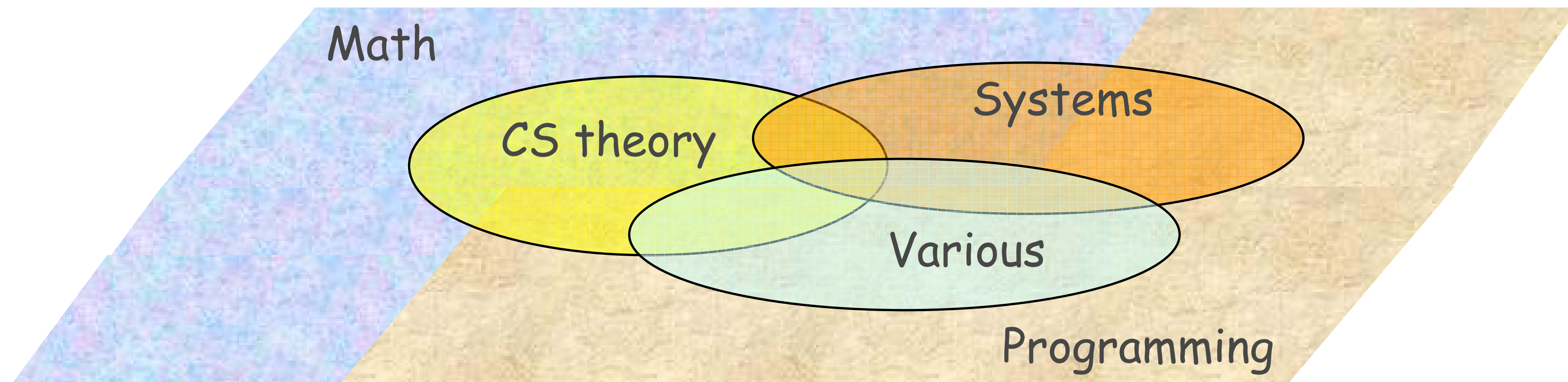


The Elements of Computing Systems:

Building a Modern Computer From First Principles

Noam Nisan and Shimon Schocken, MIT Press, 2005

The Computer Science Curriculum



Some Open Issues:

- Lack of integration
- Lack of comprehension

Our Solution:

- A new, integrative caspstone course
- Textbook: *The Elements of Computing Systems*
  Nisan and Schocken, MIT Press 2005.

Course Contents

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

Hardware

- **Architecture:** ALU/CPU design and implementation, machine code, assembly language programming, addressing modes, memory-mapped input-output (I/O).
- **Data structures and algorithms:** Stacks, h ion, arithmetic algorithms, geometric algorithm ations.

Algorithms

- **Programming Languages:** Object-based design and programming, abstract data types, scoping rules, syntax and semantics, references, OS libraries.
- **Compilers:** Lexical analysis, top-down parsing, symbol tables, virtual stack-based machine, code generation, implement cts.

Systems

- **Software Engineering:** Modular design, the ion paradigm, API design and documentation, proactive test planning, quality assurance, programming at the large.

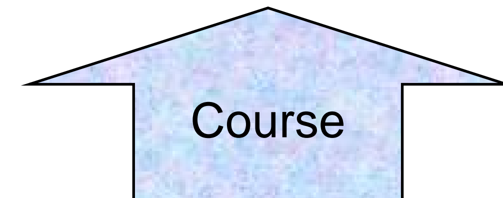
The Course Theme: Let’s Build a Computer

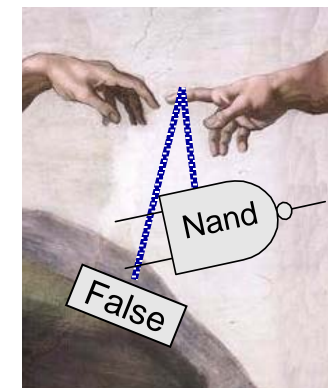
Course Goals

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

Course Methodology

- Constructive: do-it-yourself
- Self-contained: only requirement is programming
- Guided: all "plans" are given
- Focused: no optimization,
  no exceptions,
  no advanced features.





Sample Applications

```
Enter the students data, ending with 'Q':

Name: DAN
Grade: 90

Name: PAUL
Grade: 80

Name: LISA
Grade: 100

Name: ANN
Grade: 90

Name: Q

The grades average is 90
The student with the highest grade is LISA
```

Sample Applications

SCORE: 3

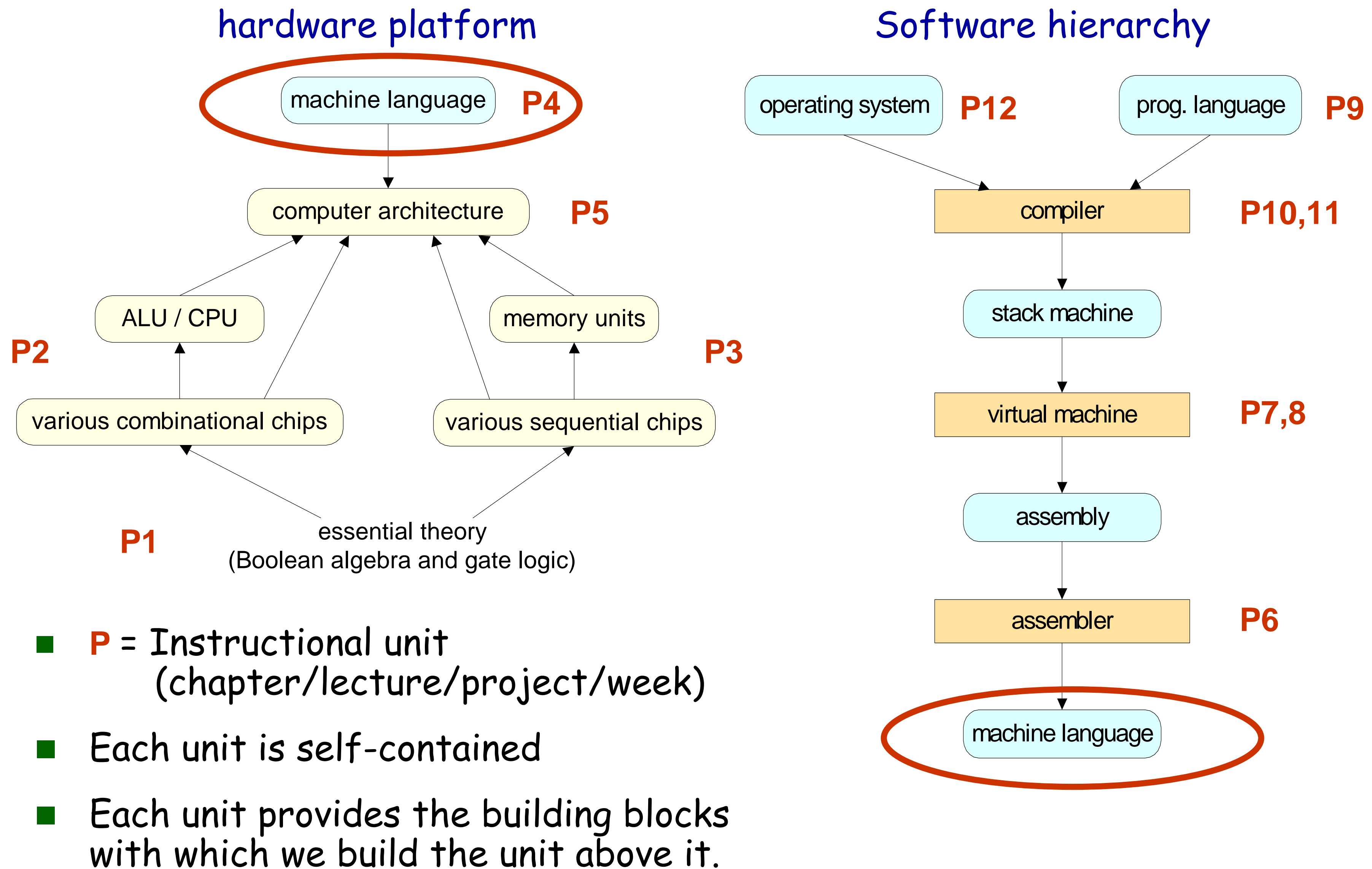
Sample Applications

Score: 10 Level: 1 Lives: 1

Sample Applications

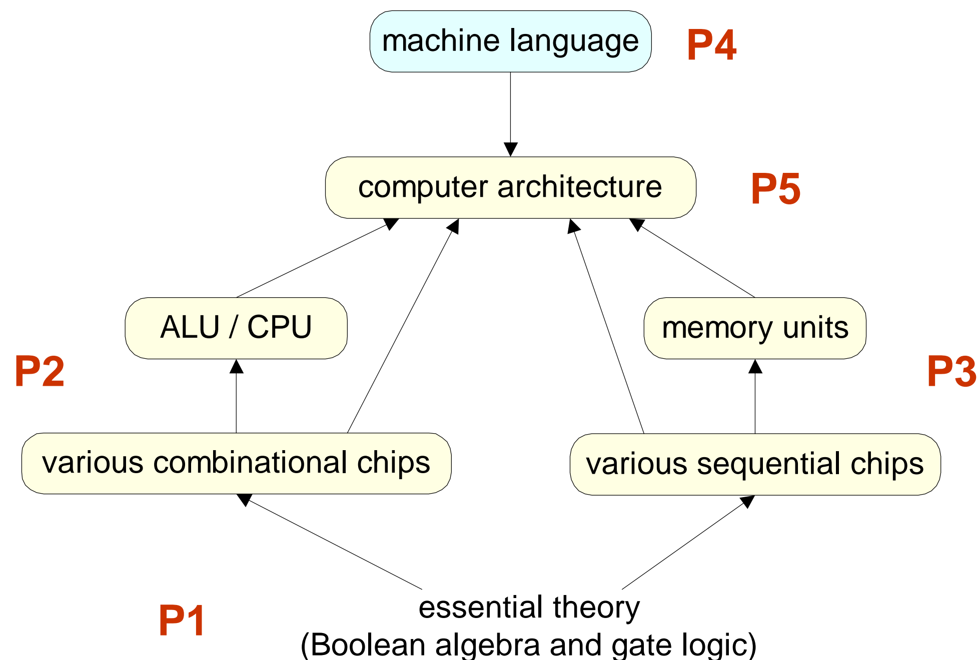
```
PLEASE TYPE A NUMBER BETWEEN 1 - 9

█ █ █ █ █ █         A
█ A █ █ █ █         T
█ A █ █ █ █         K
█ A █ █ █ █         D
█ A D █ █ █         R
█ A D R █ █         O
█ A D R █ █         L
█ A D R █ █         F
█ A D R █ █         M
M A D R █ █         I
M A D R I █         Y
M A D R I █         S
M A D R I █         B
M A D R I █

 YOU LOST   :-(
```

Course map



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

Hardware projects



Hardware projects:

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

Tools:

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

Project 1: Elementary logic gates

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

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

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

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

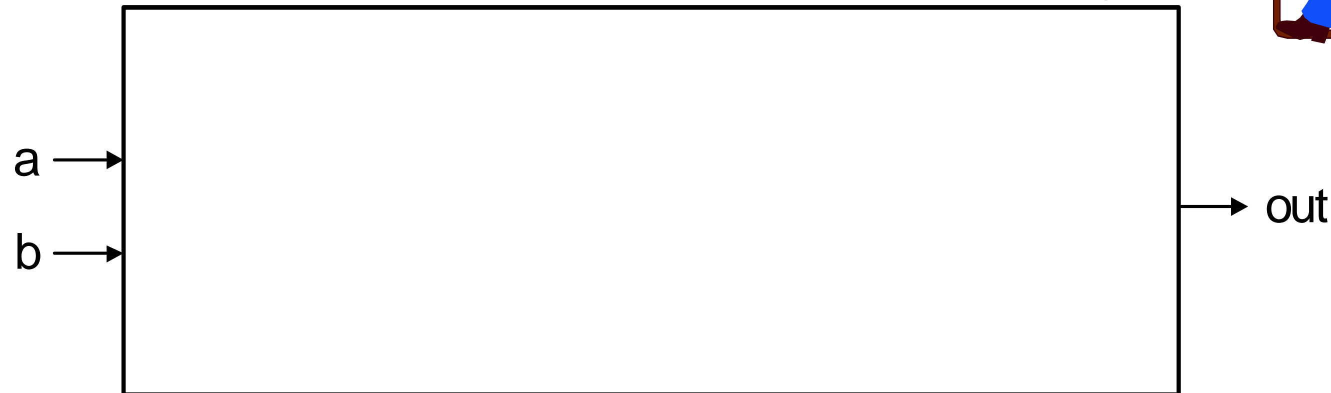
Interface: And(a,b) = 1 exactly when a=b=1

And.hdl

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

Building an **`And`** gate

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

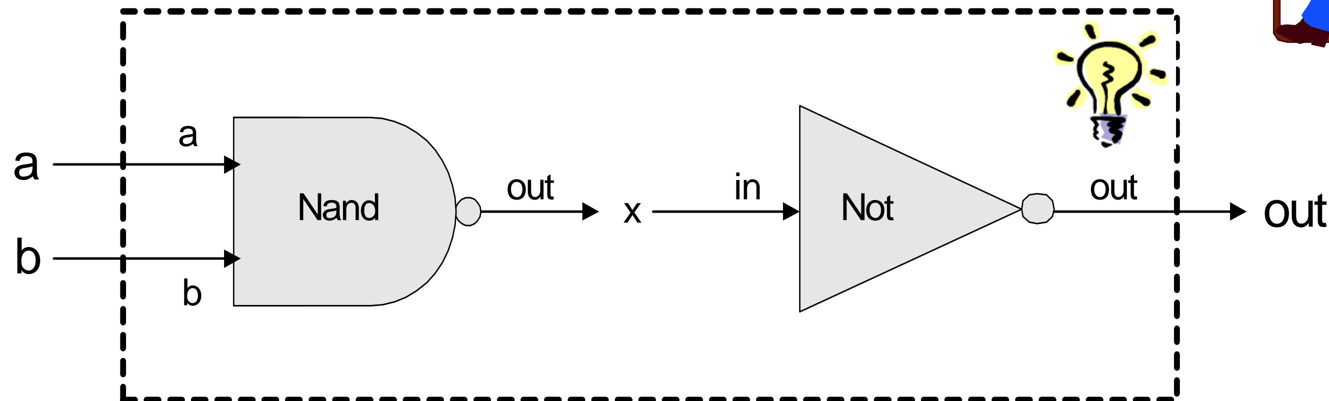


And.hdl

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

Building an **`And`** gate

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

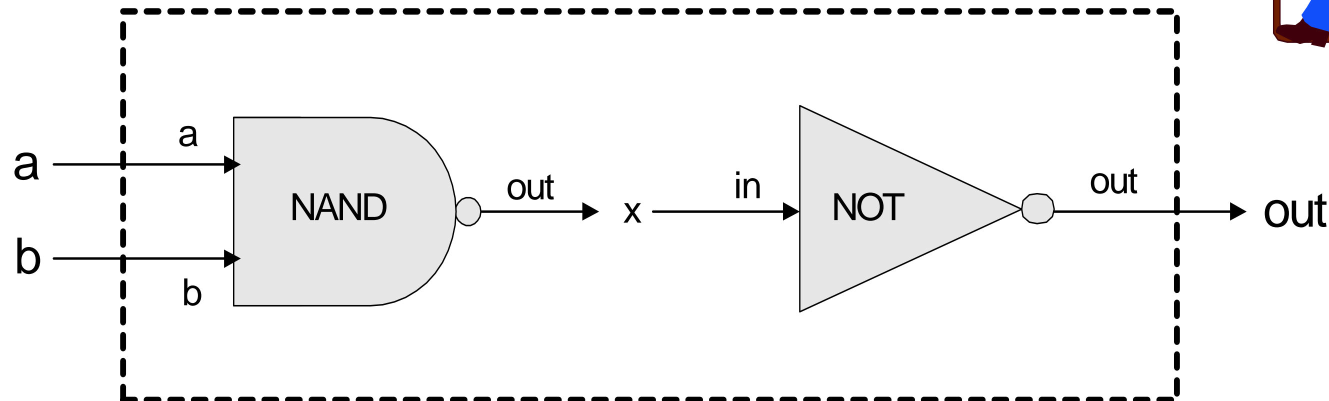


And.hdl

```
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

Hardware Simulator - D:\hack\Chips\Project 1\Xor.hdl

File View Run Help

Animate: Program flow   Format: Decimal   View: Script

Slow Fast

Chip Name :   Time : 0

Input pins

Name	Value
a	0
b	0

Output pins

Name	Value
out	0

HDL

```
// Xor (exclusive or) gate
// if a<>b out=1 else out=0
CHIP Xor {
   IN a,b;
   OUT out;
   PARTS:
   Not (in=a,out=nota);
   Not (in=b,out=notb);
   And (a=a,b=notb,out=x);
   And (a=nota,b=b,out=y);
   Or  (a=x,b=y,out=out);
}
```

Internal pins

Name	Value
nota	1
notb	1
x	0
y	0

```
load Xor,
output-file Xor.out,
compare-to Xor.cmp,
output-list a%B3.1.3 b%B3.1.3 out%B3.1.3;

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

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

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

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

HDL program

test script

gate diagram

a b And And Or out

Script restarted

Hardware simulator

Hardware Simulator - D:\hack\Chips\Project 1\Xor.hdl

File View Run Help

Slow Fast Animate: Program flow Format: Decimal View: Script

Chip Name : Time : 0

Input pins

Name	Value
a	0
b	0

Output pins

Name	Value
out	0

HDL program

HDL

```
// Xor (exclusive or) gate
// if a<>b out=1 else out=0
CHIP Xor {
   IN a,b;
   OUT out;
   PARTS:
   Not (in=a,out=nota);
   Not (in=b,out=notb);
   And (a=a,b=notb,out=x);
   And (a=nota,b=b,out=y);
   Or  (a=x,b=y,out=out);
}
```

Internal pins

Name	Value
nota	1
notb	1
x	0
y	0

```
load Xor,
output-file Xor.out,
compare-to Xor.cmp,
output-list a%B3.1.3 b%B3.1.3 out%B3.1.3;

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

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

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

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

Script restarted

Hardware simulator

Hardware Simulator - D:\hack\Chips\Project 1\Xor.hdl

File View Run Help

Slow Fast | Animate: Program flow | Format: Decimal | View: Script (Script / Output / Compare / None)

Chip Name : Xor Time : 0

Input pins

Name	Value
a	1
b	1

Output pins

Name	Value
out	0

HDL program

HDL

```
// Xor (exclusive or) gate
// if a<>b out=1 else out=0
CHIP Xor {
   IN a,b;
   OUT out;
   PARTS:
   Not (in=a,out=nota);
   Not (in=b,out=notb);
   And (a=a,b=notb,out=x);
   And (a=nota,b=b,out=y);
   Or  (a=x,b=y,out=out);
}
```

Internal pins

Name	Value
nota	0
notb	0
x	0
y	0

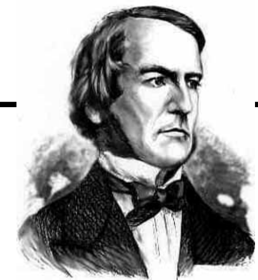
```
load Xor,
output-file Xor.out,
compare-to Xor.cmp,
output-list a%B3.1.3 b%B3.1.3 out%B3.1.3;

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

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

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

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

output file

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

End of script - Comparison ended successfuly

Chip anatomy

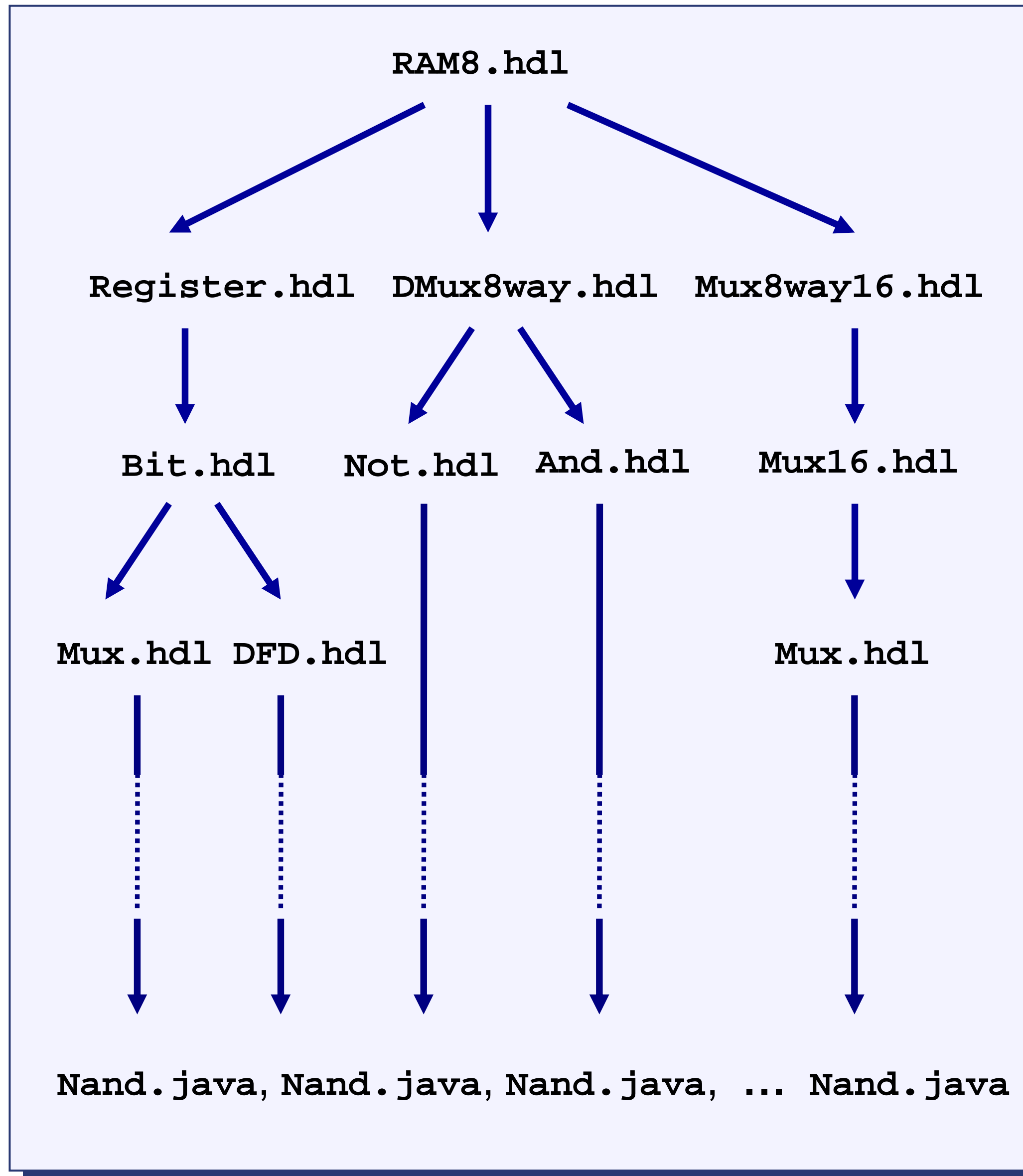


George Boole
1815-1864

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

Claude Shannon, 1916-2001

Chip anatomy



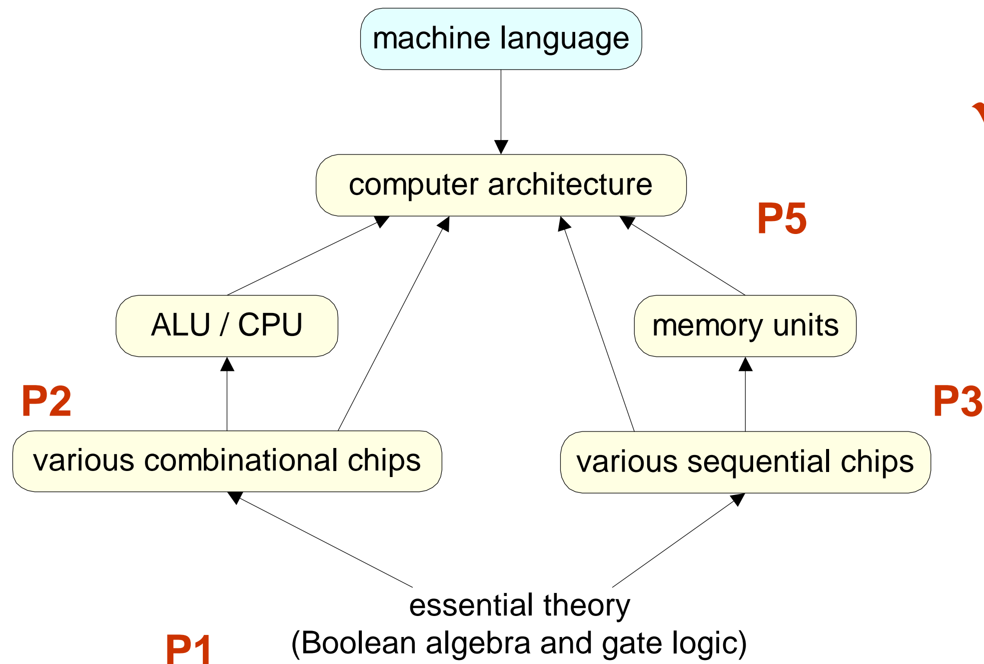
Simulator logic:

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

Benefits:

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

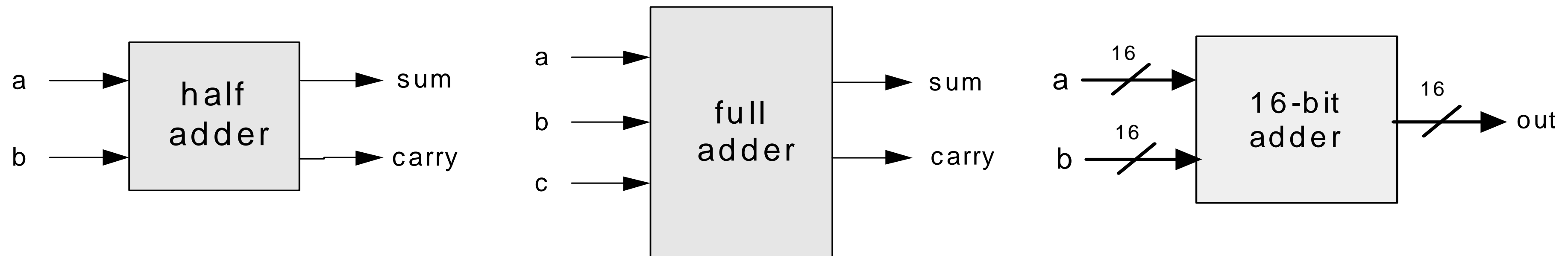
Hardware projects

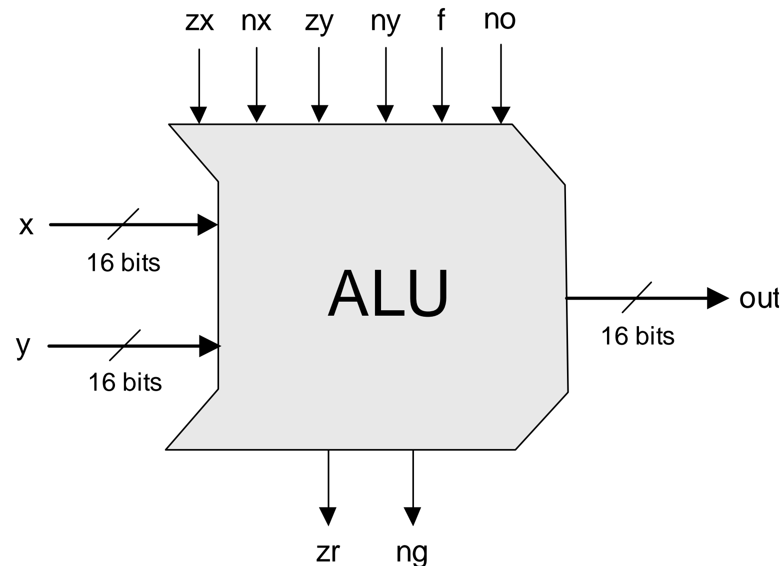


Hardware projects:

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

Project 2: Combinational chips





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,

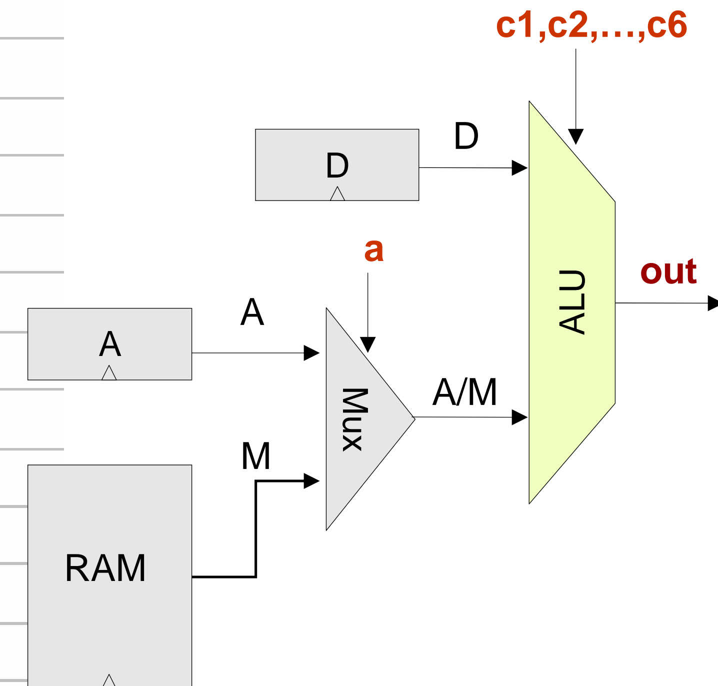
x&y, x|y

ALU logic

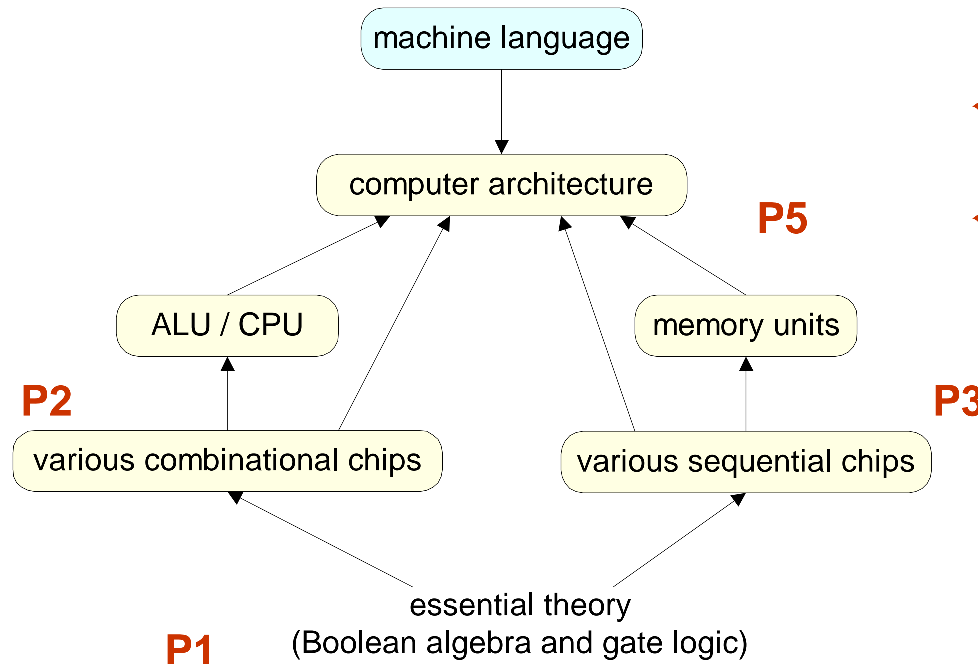
These bits instruct how to pre-set the x input		These bits instruct how to pre-set the y input		This bit selects between + / And	This bit inst. how to post-set out	Resulting ALU output
zx	nx	zy	ny	f	no	out=
if zx then x=0	if nx then x=!x	if zy then y=0	if ny then y=!y	if f then out=x+y else out=x And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

A glimpse ahead:

out (when a=0)	c1	c2	c3	c4	c5	c6	out (when a=1)
0	1	0	1	0	1	0	
1	1	1	1	1	1	1	
-1	1	1	1	0	1	0	
D	0	0	1	1	0	0	
A	1	1	0	0	0	0	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



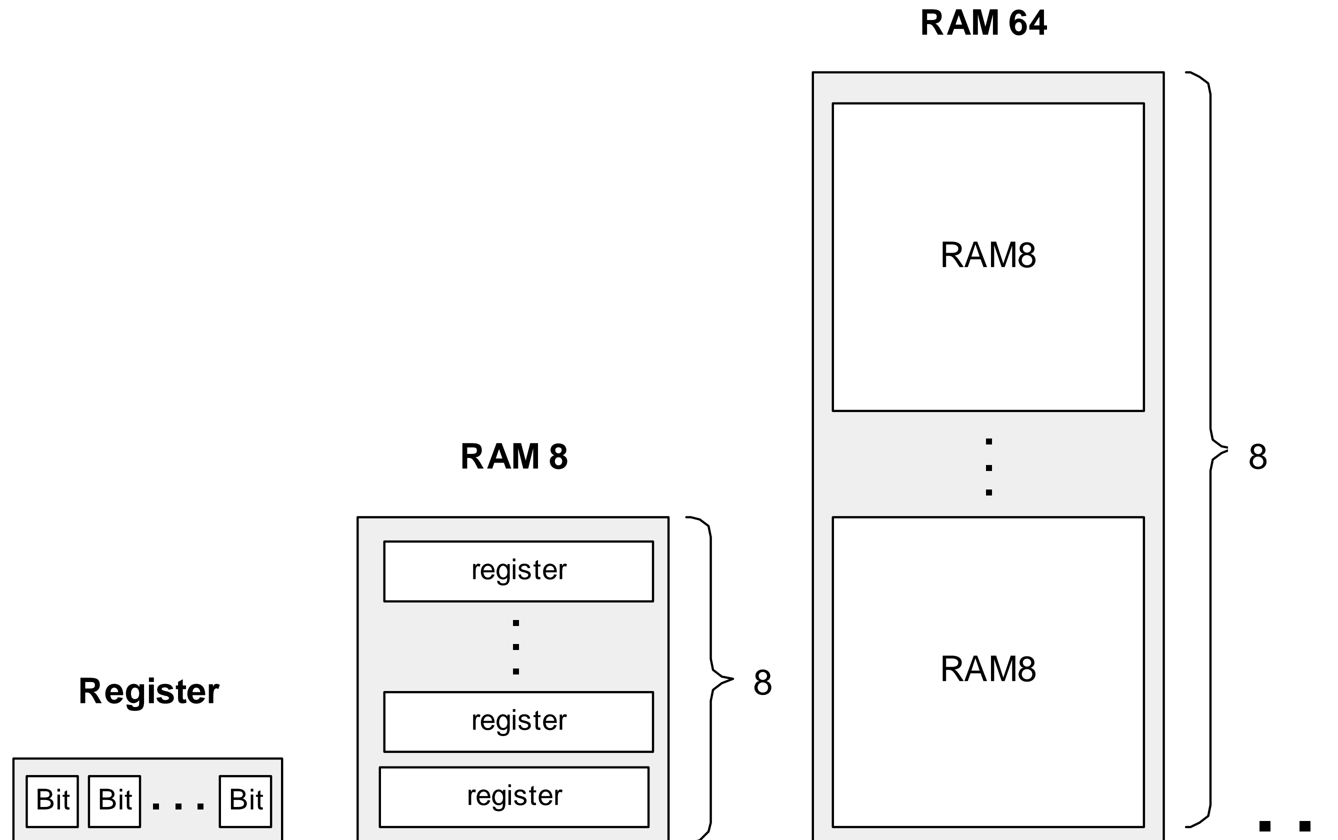
Hardware projects



Hardware projects:

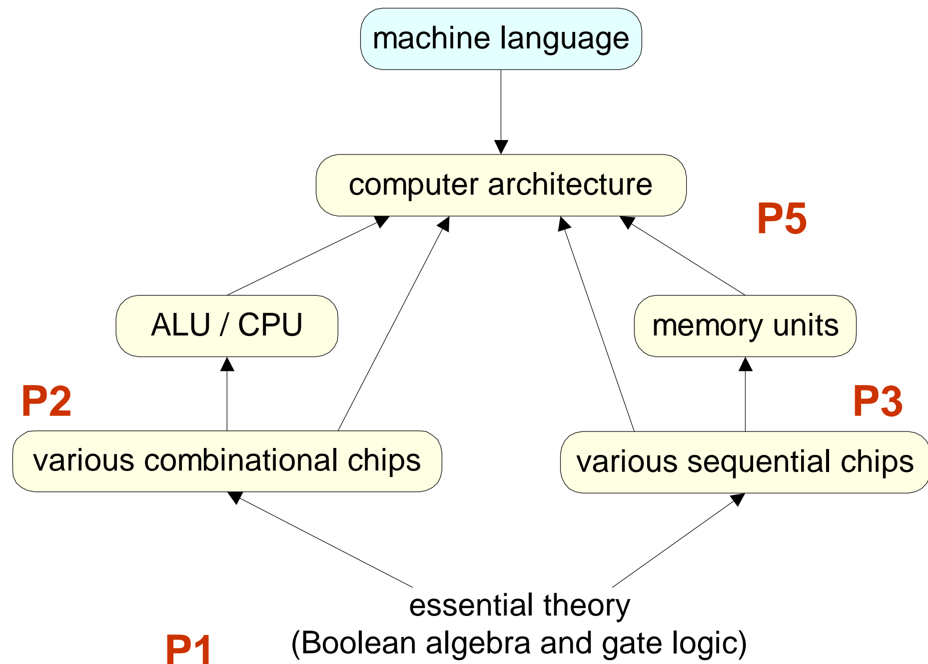
- P1: Elementary logic gates
- P2: Combinational gates (ALU)
- P3: Sequential chips (memory)
- P4: Machine language
- P5: Computer architecture

Project 3: Sequential chips



- DFF > Bit > Register > RAM8 > RAM64 > ... > RAM32K

Hardware projects



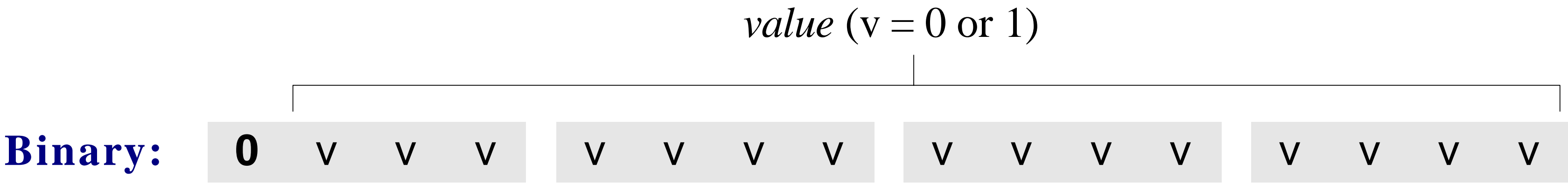
Hardware projects:

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

Machine Language: *A*-instruction

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

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



Machine Language: *C*-instruction

```
dest = comp ; jump      // If dest is null, the "=" is ommited
                        // If jump is null, the ";" is ommited
```

`comp` is one of:

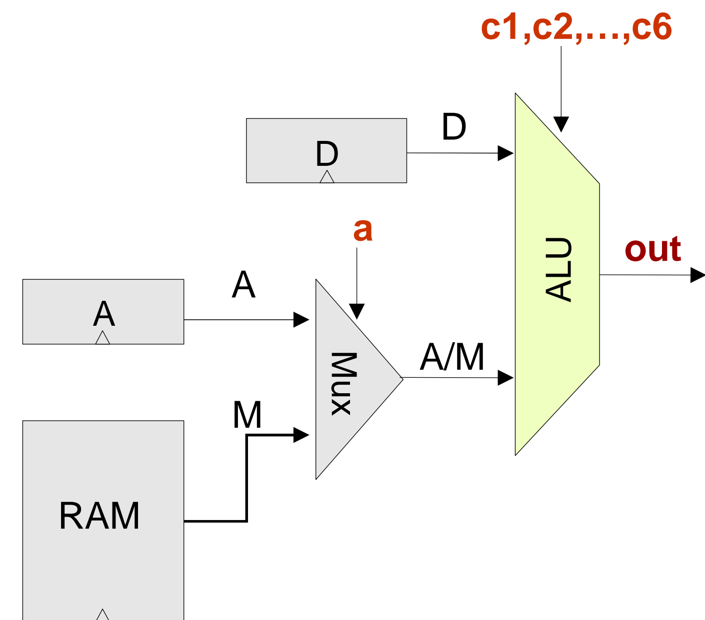
```
0,1,-1,D,A,!D,!A,-D,-A,D+1,A+1,D-1,A-1,D+A,D-A,A-D,D&A,D|A,
          M,  !M,  -M,   M+1,   M-1,D+M,D-M,M-D,D&M,D|M
```

`dest` is one of:

```
Null, M, D, MD, A, AM, AD, AMD
```

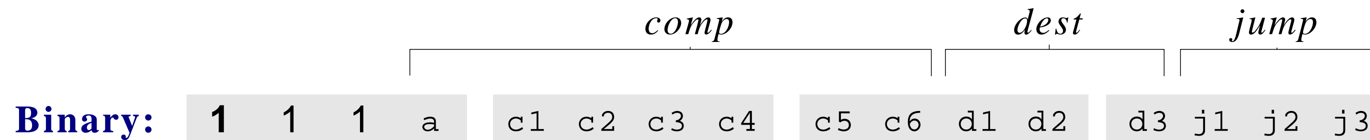
`jump` is one of:

```
Null, JGT, JEQ, JGE,JLT, JNE, JLE,JMP
```



Machine Language: *C*-instruction (cont.)

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.

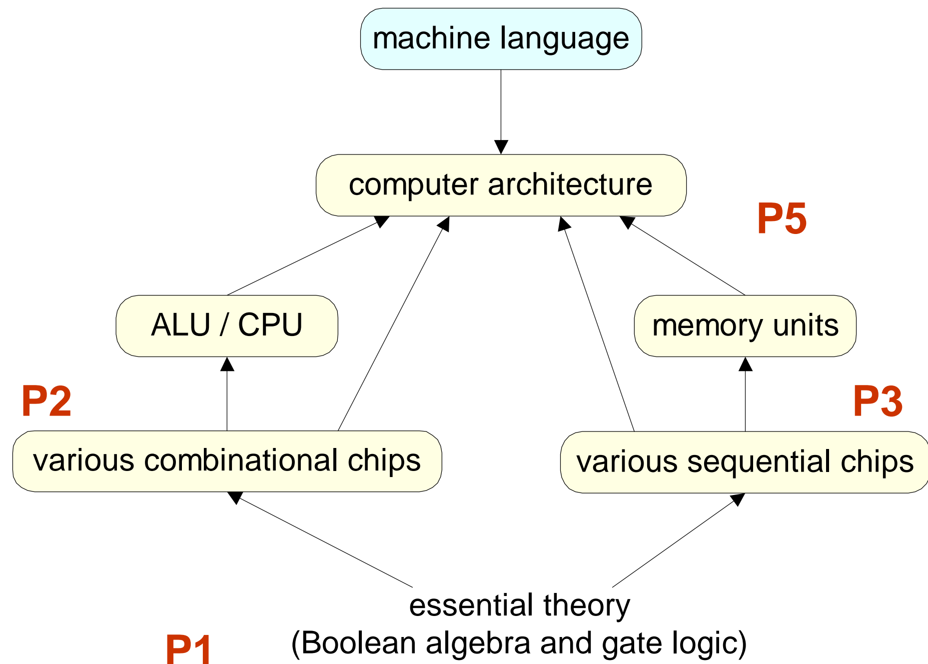


(when a=0) *comp*	c1	c2	c3	c4	c5	c6	(when a=1) *comp*
0	1	0	1	0	1	0	
1	1	1	1	1	1	1	
-1	1	1	1	0	1	0	
D	0	0	1	1	0	0	
A	1	1	0	0	0	0	M
!D	0	0	1	1	0	1	
!A	1	1	0	0	0	1	!M
-D	0	0	1	1	1	1	
-A	1	1	0	0	1	1	-M
D+1	0	1	1	1	1	1	
A+1	1	1	0	1	1	1	M+1
D-1	0	0	1	1	1	0	
A-1	1	1	0	0	1	0	M-1
D+A	0	0	0	0	1	0	D+M
D-A	0	1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

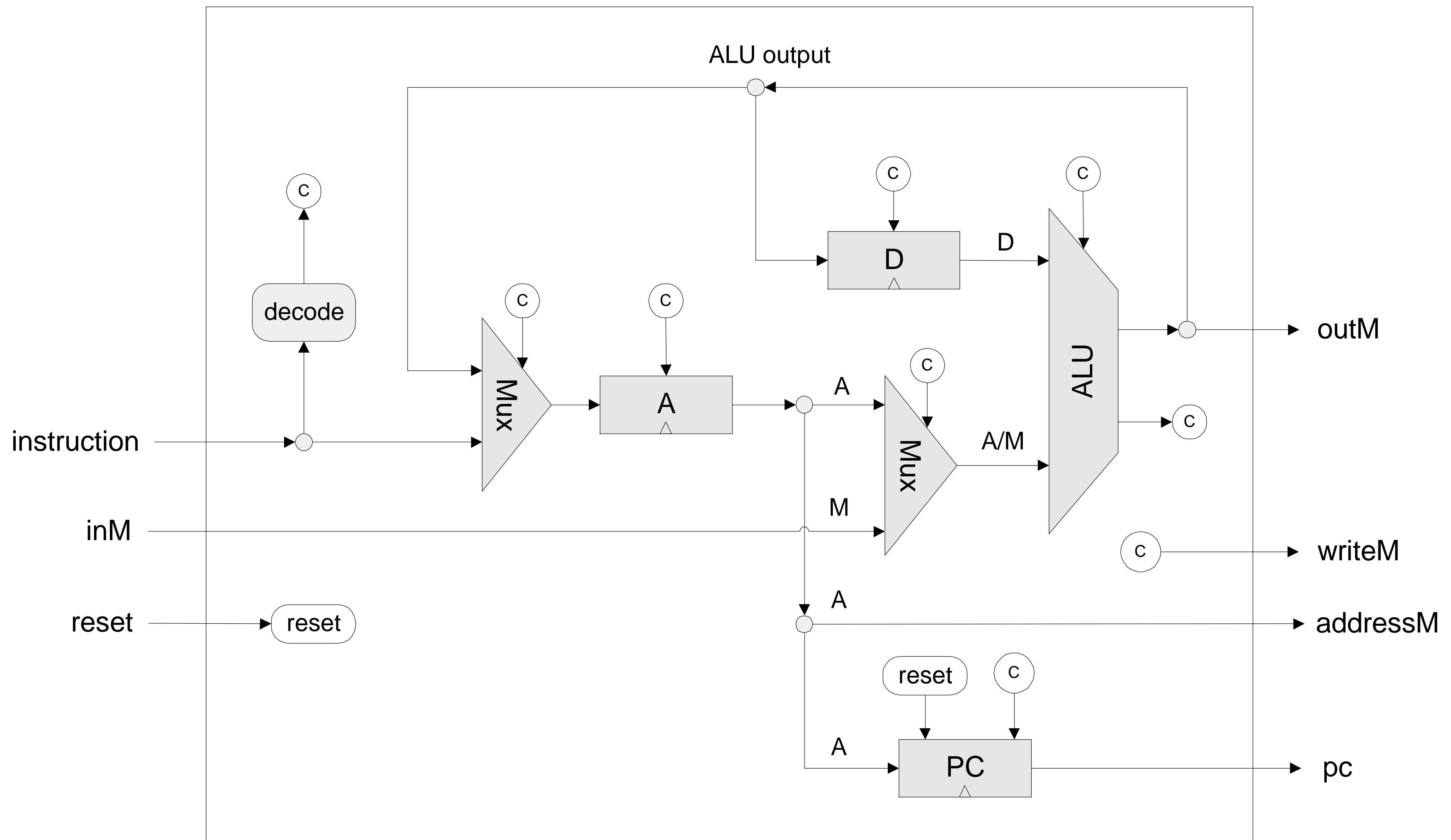
Hardware projects



Hardware projects:

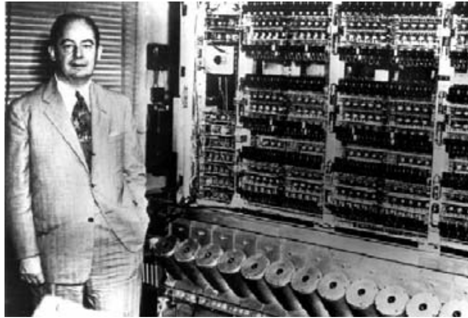
- P1: Elementary logic gates
- P2: Combinational gates (ALU)
- P3: Sequential gates (memory)
- P4: Machine language
- P5: Computer architecture

Project 5: CPU



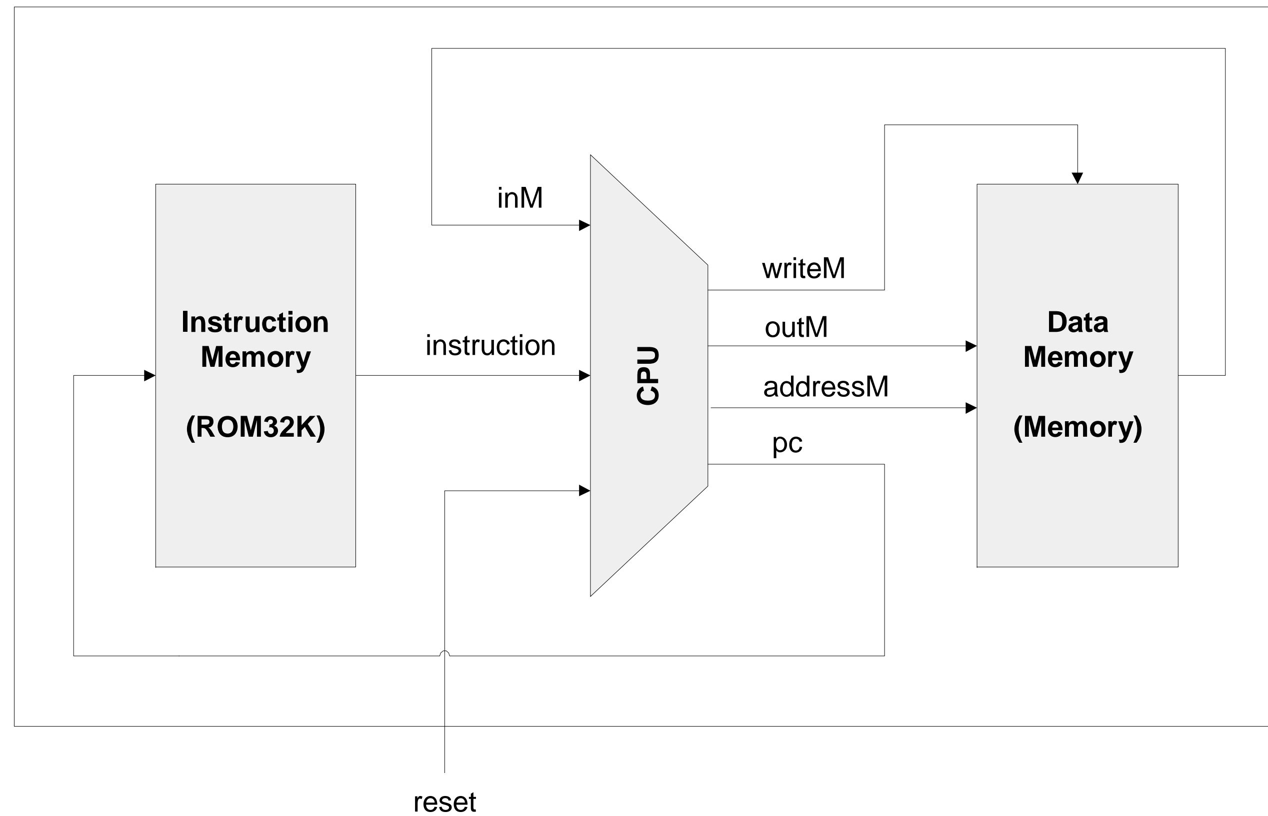
Project 5: Computer

- Many other computer models can be built.

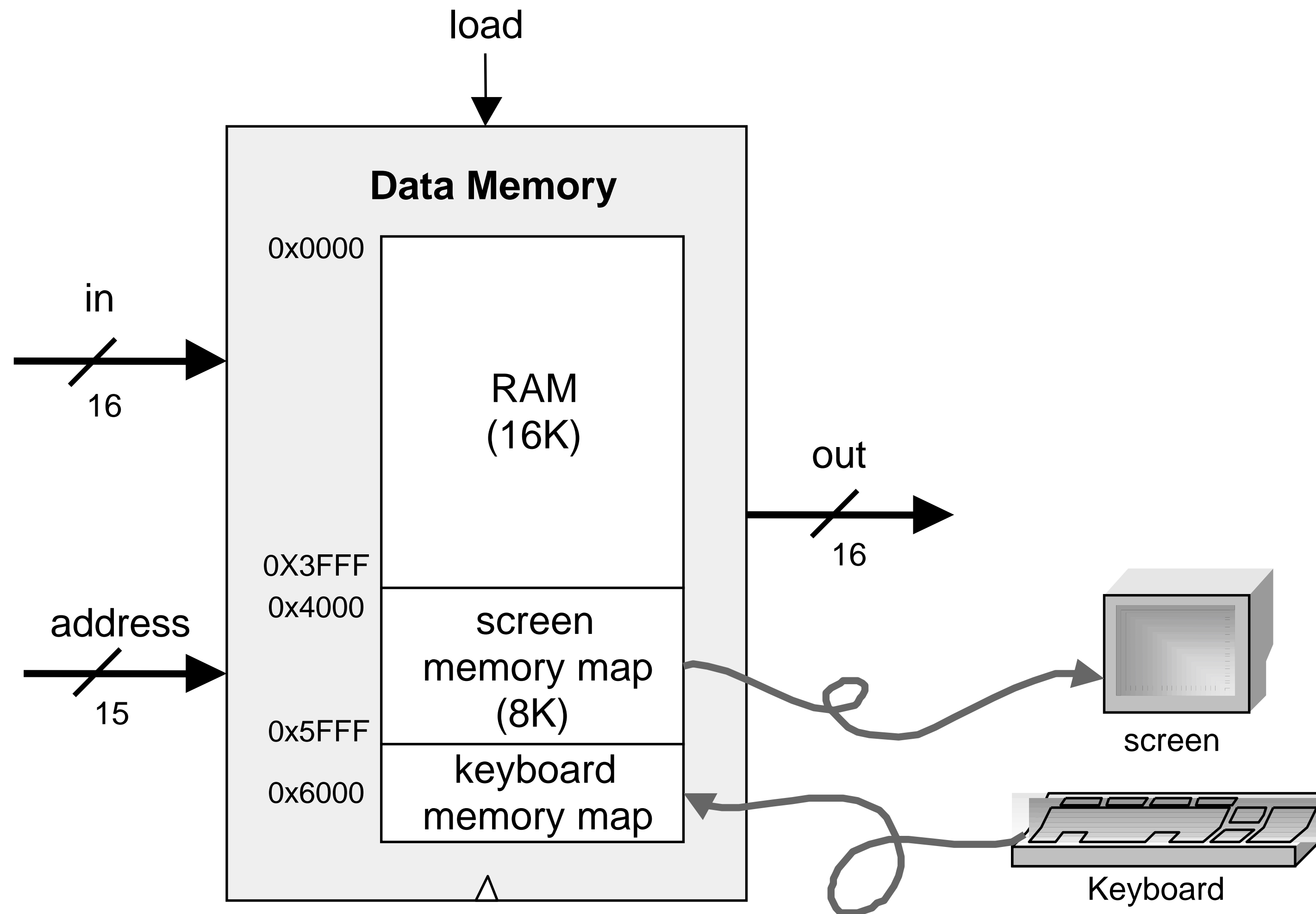


J. Von Neumann

(1903-1957)



Input / Output Devices



Recap: the Hack chip-set and hardware platform

Elementary logic gates

(**Project 1**):

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

Combinational chips

(Project 2):

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

Sequential chips

(**Project 3**):

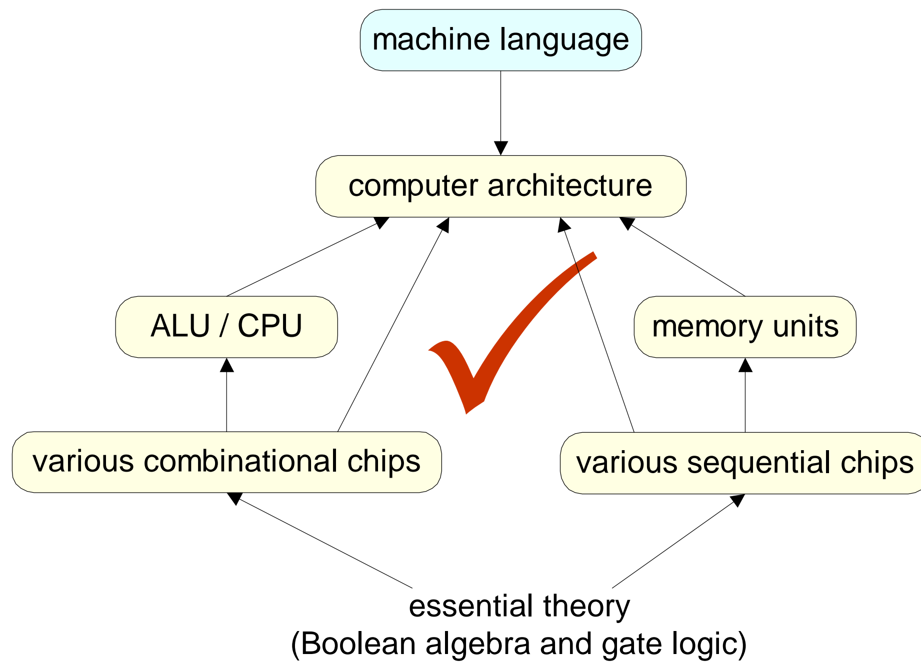
- `DFF` (primitive)
- `Bit`
- `Register`
- `RAM8`
- `RAM64`
- `RAM512`
- `RAM4K`
- `RAM16K`
- `PC`

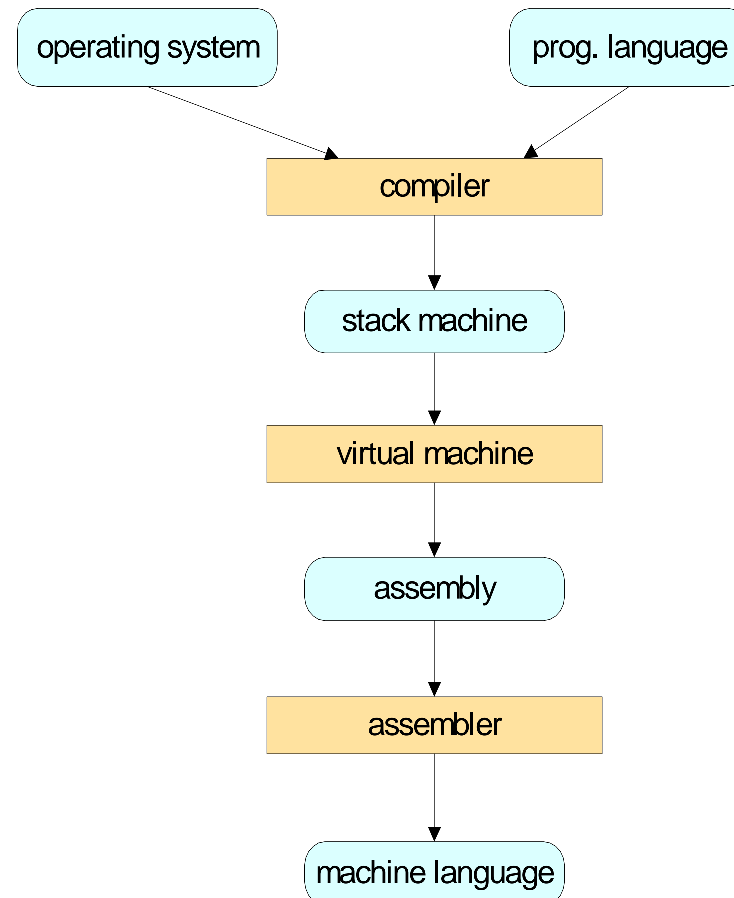
Computer Architecture

(**Project 5**):

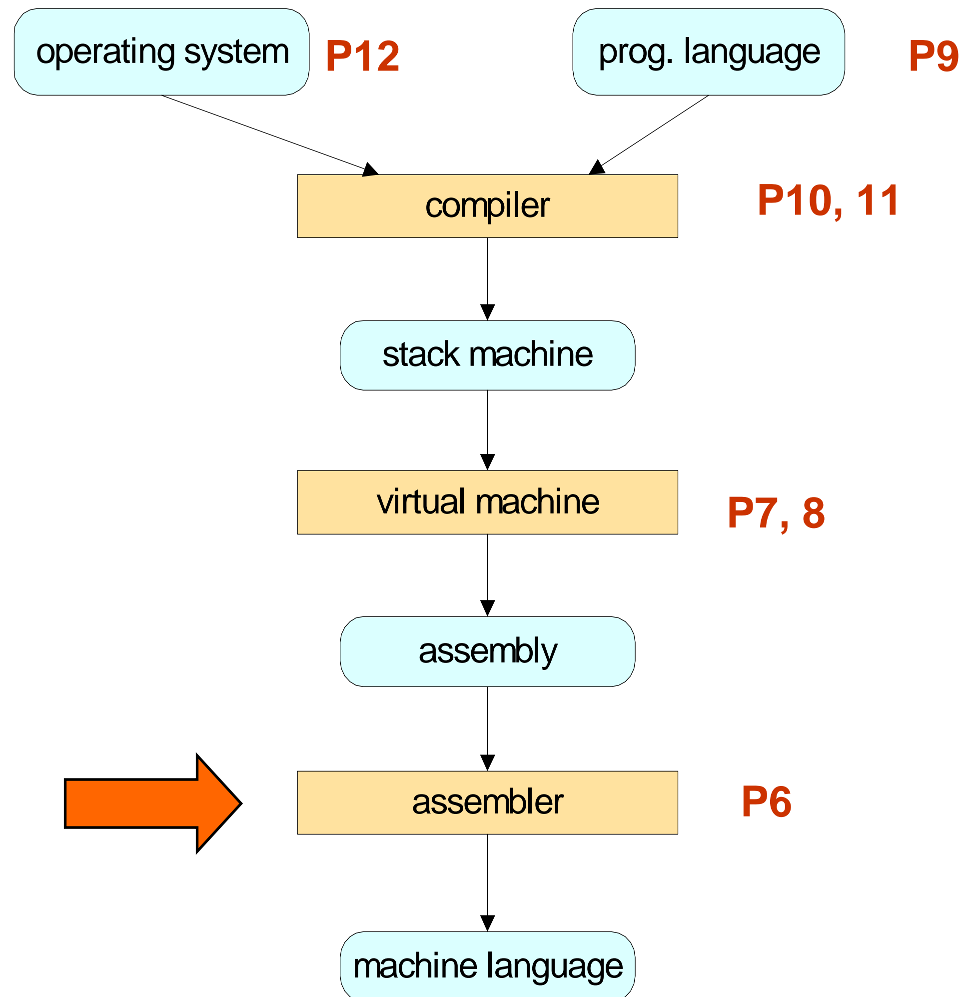
- `Memory`
- `CPU`
- `Computer`

Course map





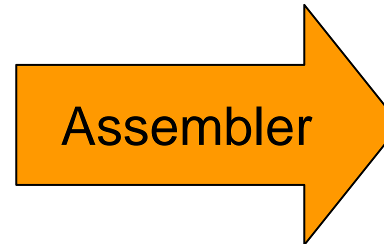
Software projects



Project 6: Assembler

Sum.asm

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



Sum.bin

```
0000000000010000
1110111111001000
0000000000010001
1110101010001000
0000000000010000
1111110000010000
0000000001100100
1110010011010000
0000000000010010
1110001100000001
0000000000010000
1111110000010000
0000000000010001
1111000010001000
0000000000010000
1111110111001000
0000000000000100
1110101010000111
```

Ada Lovelace
(1815-1852)

Assembler in action

Assembler - D:\hack\shimon progs\sumto100.asm

File Run Help

Source

```
// computes sum=1+...+100
  @1
  M=1  // count=1
  @sum // allocated at RAM[16]
  M=0
(LOOP)
  @1
  D=M
  @100
  D=D-A // if count=100 ...
  @END
  D;JGT // goto end
  @1
  D=M
  @sum
  M=D+M // sum=sum+count
  @1
  M=M+1  // count=count+1
  @LOOP
  0;JMP
(END)  // infinite loop
  @END
  0;JMP
```

Destination

```
0000000000000001
1110111111001000
0000000000010000
1110101010001000
0000000000000001
1111110000010000
0000000001100100
1110010011010000
0000000000010010
1110001100000001
0000000000000001
1111110000010000
0000000000010000
1111000010001000
0000000000000001
1111110111001000
0000000000000100
1110101010000111
0000000000010010
1110101010000111
```

File compilation succeeded

Project 6:

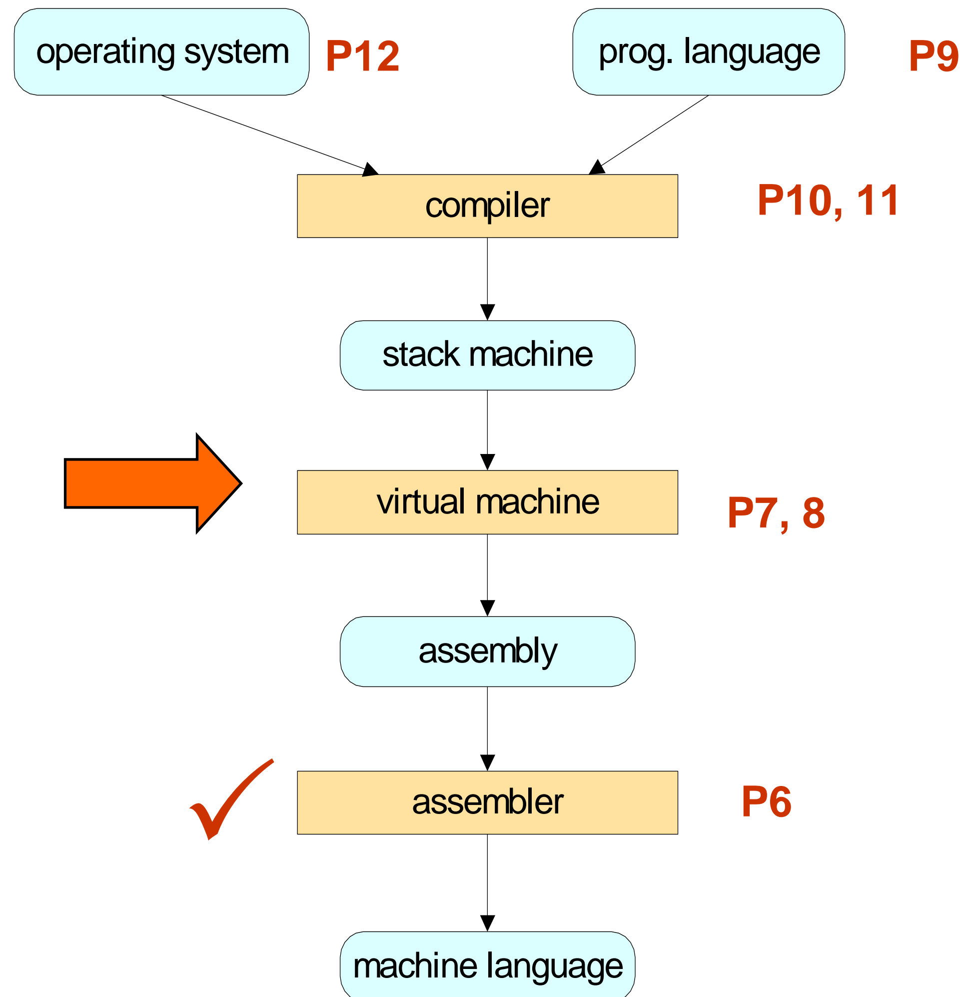
Build an assembler

CPU Emulator

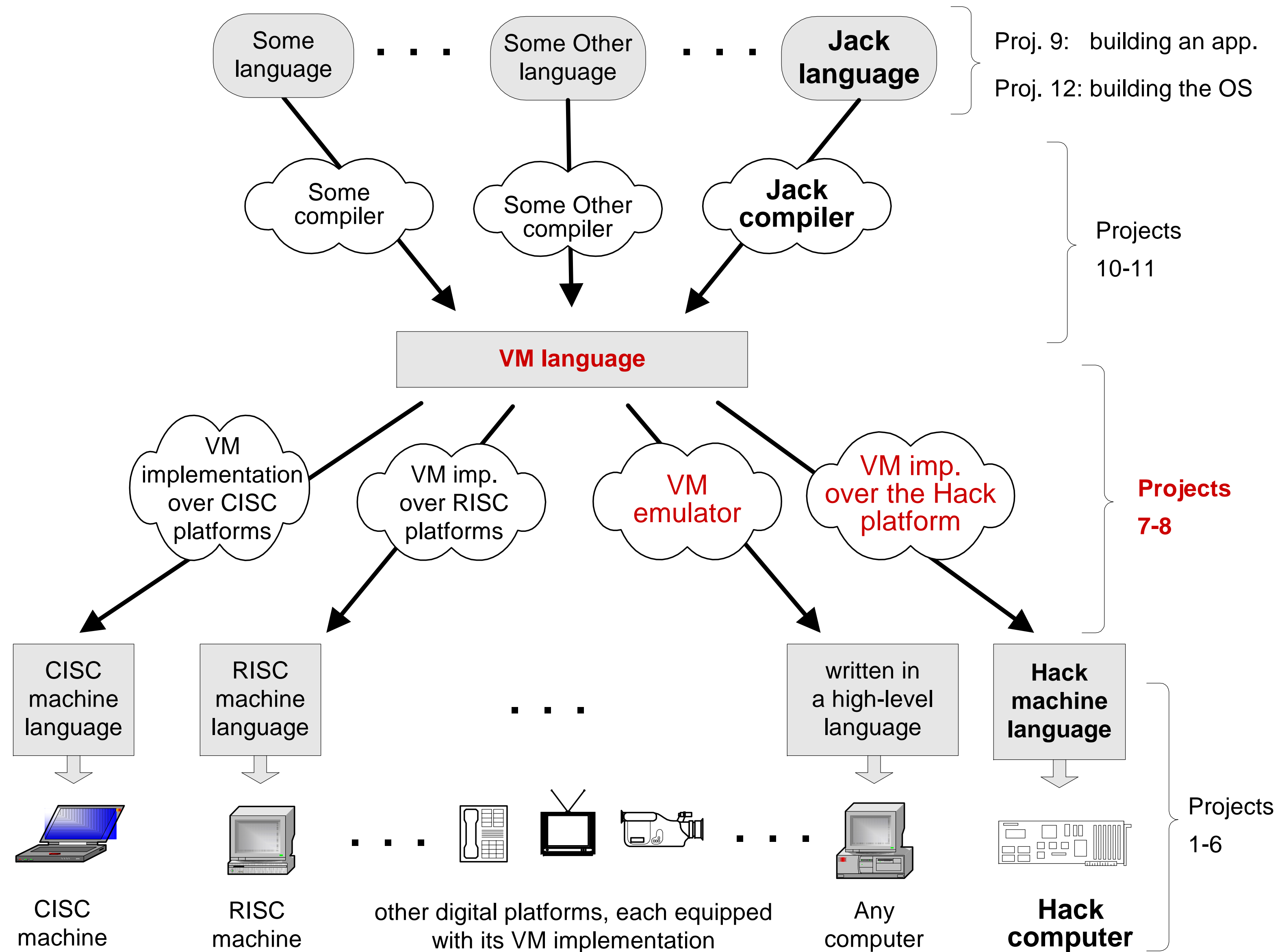
CPU Emulator - D:\hack\instructor\Examples\rect\rect.asm

- instruction memory
- data memory
- 256 by 512 pixels simulated screen
- keyboard enabler
- Data register
- program counter
- address register
- ALU

Software projects



The Big Picture



The VM language

Arithmetic commands

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

Memory access commands

pop *segment i*

push *segment i*

Program flow commands

label *symbol*

goto *symbol*

if-goto *symbol*

Function calling commands

function *funcationName nLocals*

call *functionName nArgs*

return

The VM abstraction: a Stack Machine

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;
}
```

Stack machine code

```
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 Emulator

Virtual Machine Emulator (1.3b1) - D:\hack\IT projects\project 4\solution\add

File View Run Help

Slow Fast Animate: Program flow View: Script Format: Decimal

Program

0	function	Main.add 3
1	push	constant 15
2	pop	local 0
3	push	constant 7
4	pop	local 1
5	push	local 1
6	push	constant 1
7	add	
8	pop	local 1
9	push	local 0
10	push	local 1
11	add	
12	pop	local 0
13	push	local 1
14	push	local 0

Working Stack

15
8

Call Stack

Sys.init
Main.main
Main.add

Static

0	0
1	0
2	0
3	0
4	0

Local

0	15
1	8
2	0
3	15
4	8

Argument

0	19
1	261
2	256
3	0
4	0

This

0	271
1	266
2	261
3	0
4	0

That

0	271
1	266

Temp

0	0
1	0

```
Repeat {
  VMStep;
}
```

Stack

263	256
264	0
265	0
266	15
267	8
268	0
269	15
270	8
271	0
272	0
273	0
274	0
275	0
276	0
277	0

RAM

SP:	0	271
LCL:	1	266
ARG:	2	261
THIS:	3	0
THAT:	4	0
Temp0:	5	0
Temp1:	6	0
Temp2:	7	0
Temp3:	8	0
Temp4:	9	0
Temp5:	10	0
Temp6:	11	0
Temp7:	12	0
R13:	13	0
R14:	14	0

VM code

working stack

virtual memory segments

default test script

global stack

host RAM

(the RAM is not part of the VM)

Projects 7,8: Implement the VM over the Hack platform

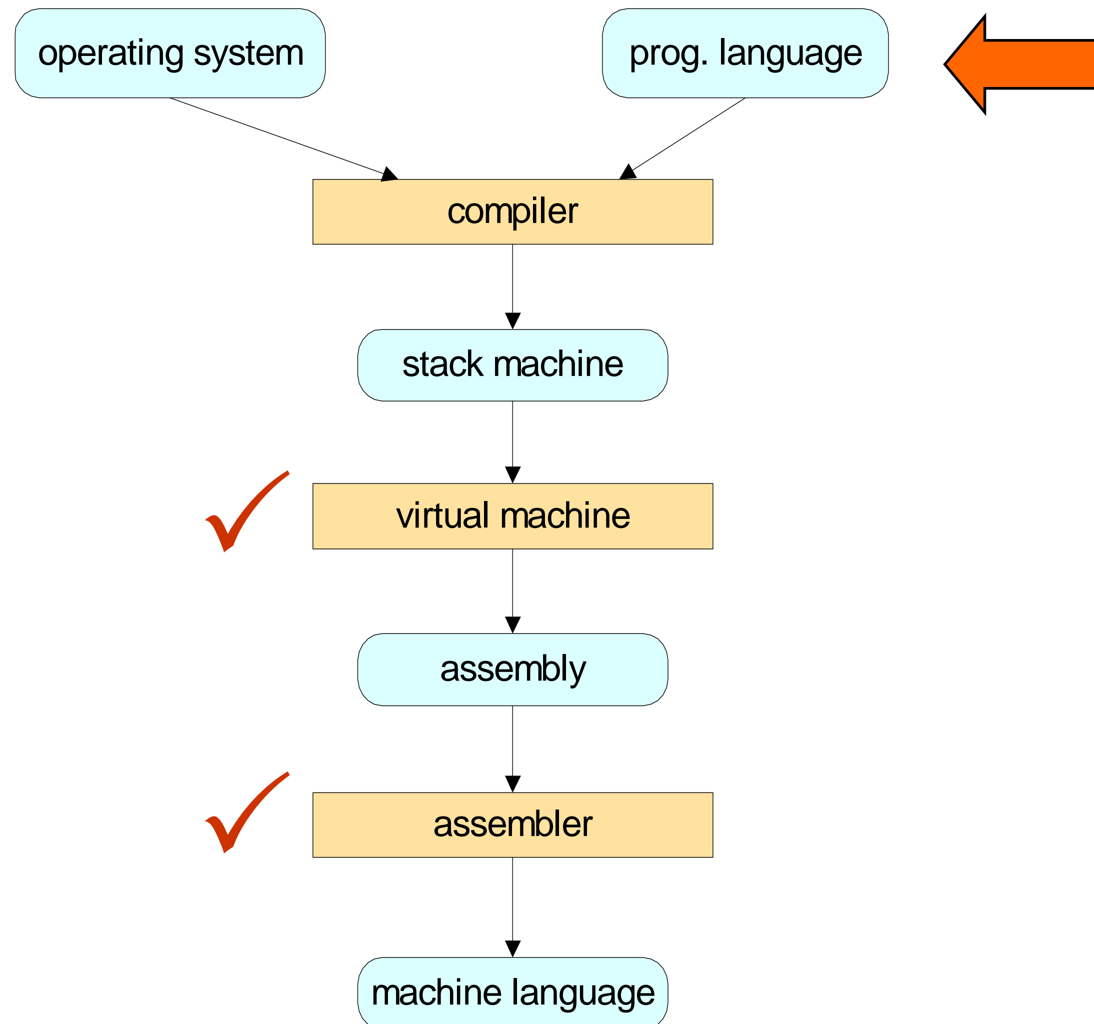
Mult.vm

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

Mult.asm

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

Software projects



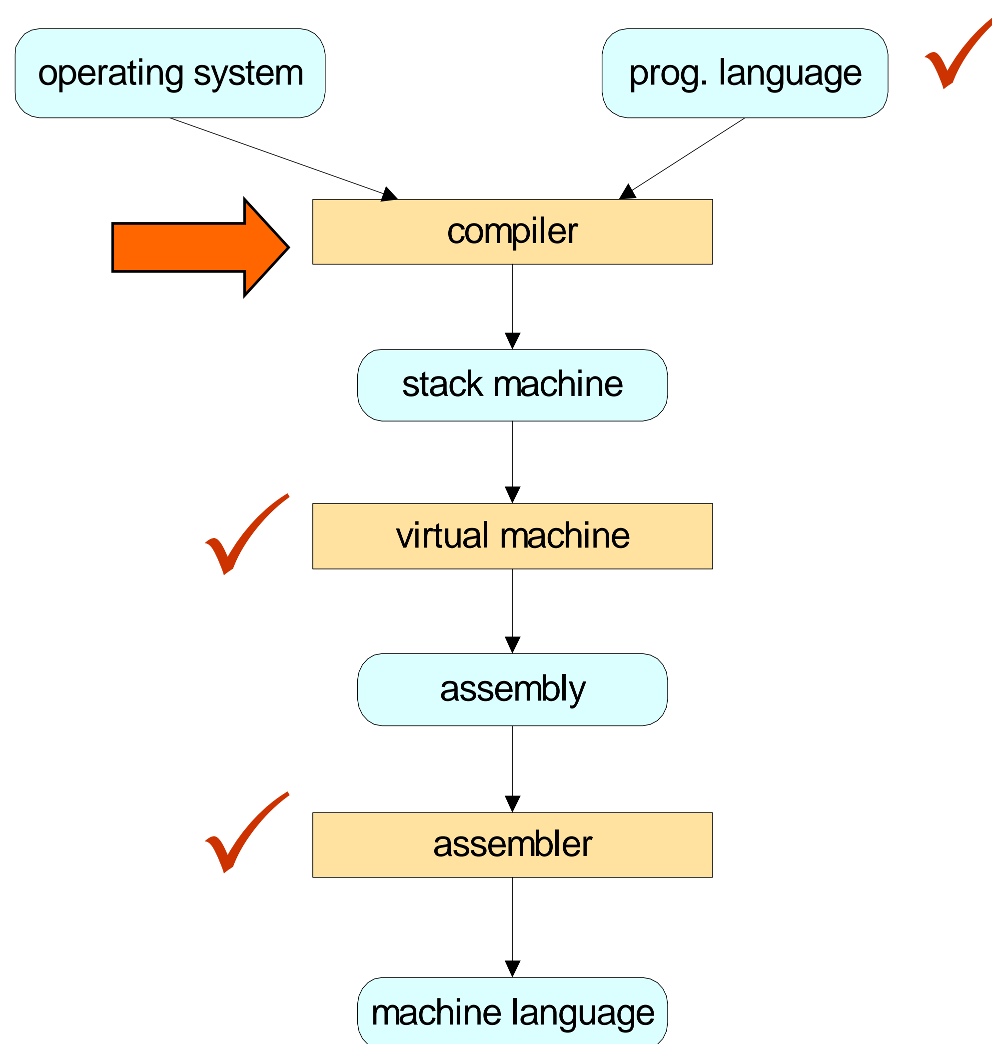
Jack Language

```
class Math {

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

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

Project 9: Write a Jack program

Demo Pong

Software projects



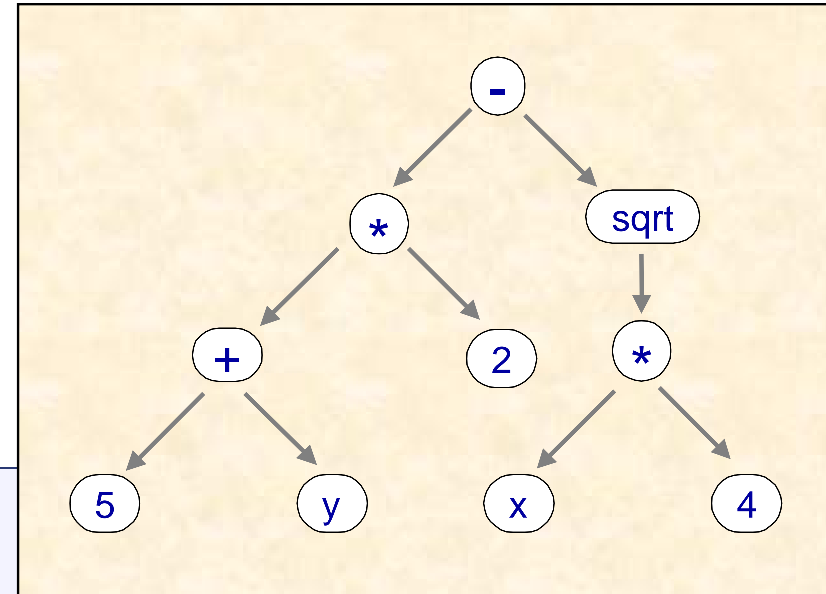
Compiler project I: Syntax Analysis

Prog.jack

```
(5+y)*2 – sqrt(x*4)
```

Jack Grammar

Syntax Analyzer



Prog.xml

```
<expression>
  <term>
    <symbol> ( </symbol>
      <expression>
        <term>
          <integerConstant> 5 </integerConstant>
        </term>
        <symbol> + </symbol>
        <term>
          <identifier> y </identifier >
        </term>
   ...
</expression>
```

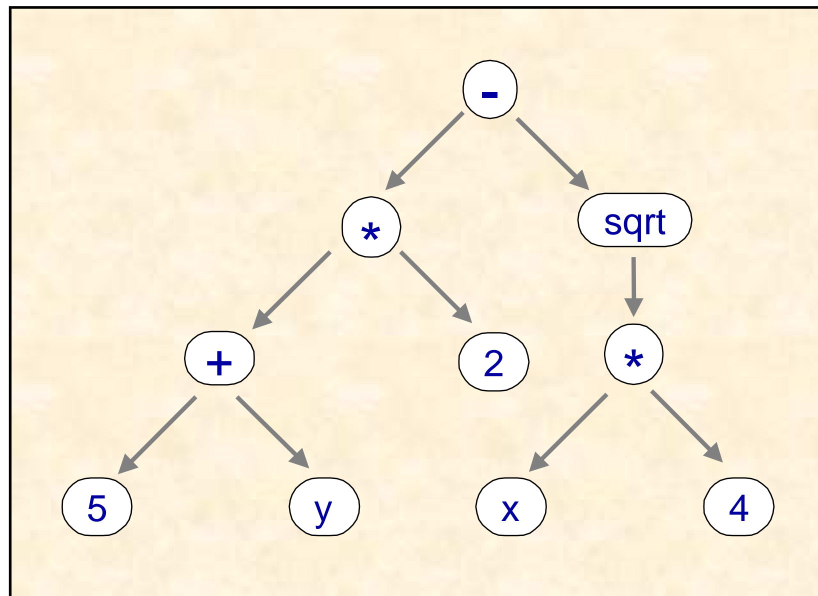
Compiler project II: Code Generation

Prog.jack

```
(5+y)*2 – sqrt(x*4)
```

Syntax analyzer

Project 9



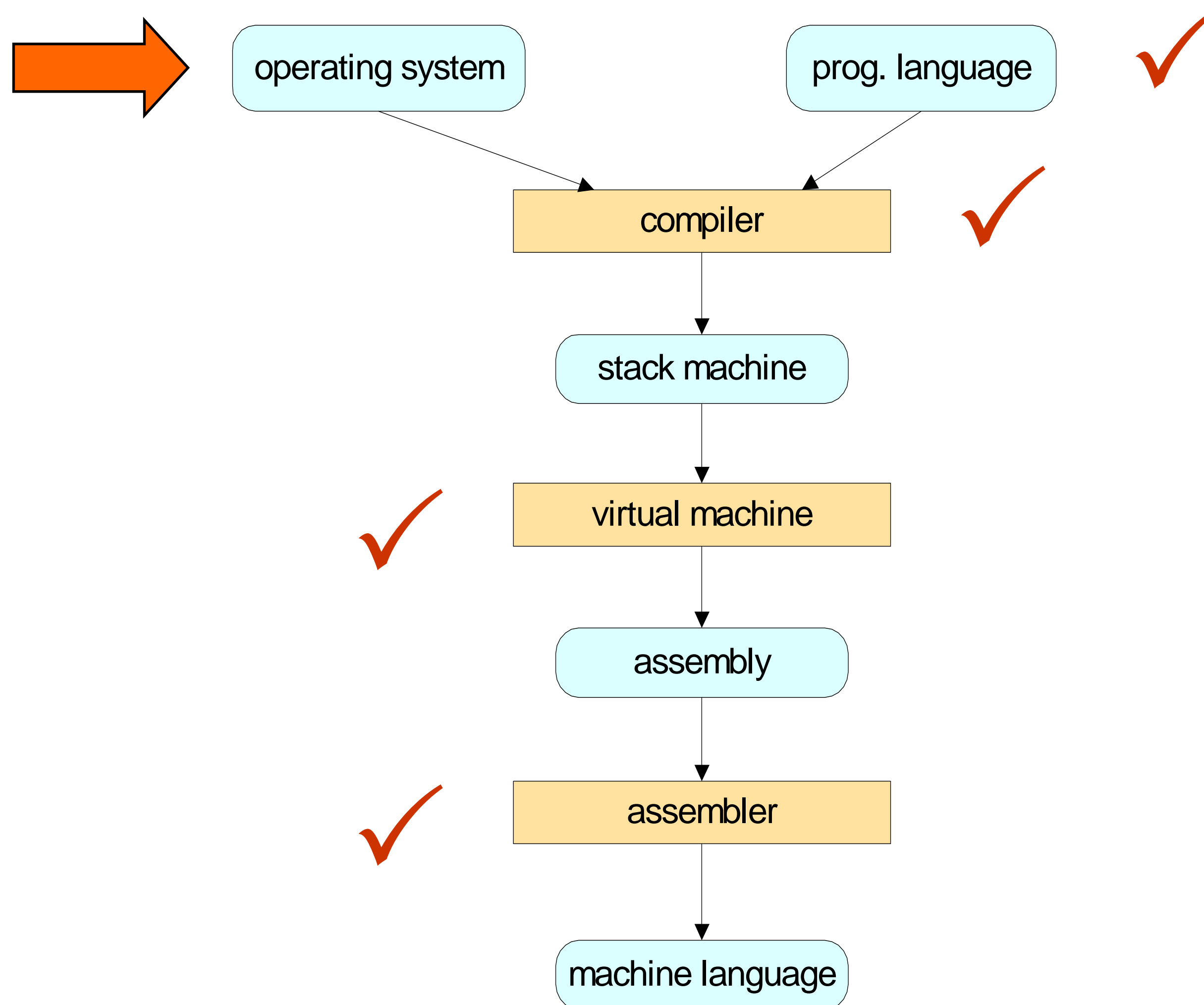
Project 10

Code generator

Prog.vm

```
push 5
push y
add
push 2
call mult
push x
push 4
call mult
call sqrt
sub
```

Software projects



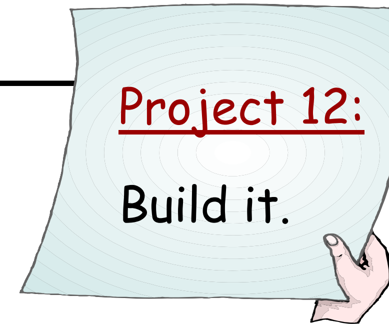
Typical Jack code

```
/** Computes the average of a sequence of integers. */
class Main {
  function void main() {
    var Array a;
    var int length;
    var int i, sum;

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

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

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

OS Libraries

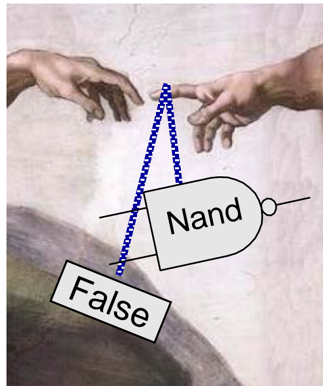
- **Math**: Provides basic mathematical operations;
- **String**: Implements the `String` type and string-related operations;
- **Array**: Implements the `Array` type and array-related operations;
- **Output**: Handles text output to the screen;
- **Screen**: Handles graphic output to the screen;
- **Keyboard**: Handles user input from the keyboard;
- **Memory**: Handles memory operations;
- **Sys**: Provides some execution-related services.

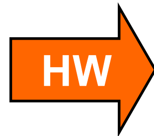
OS API

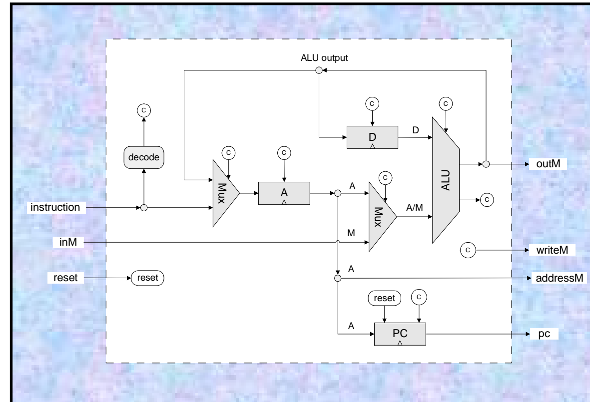


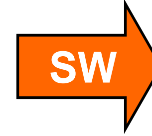

```
class Math {
}

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)
}
```

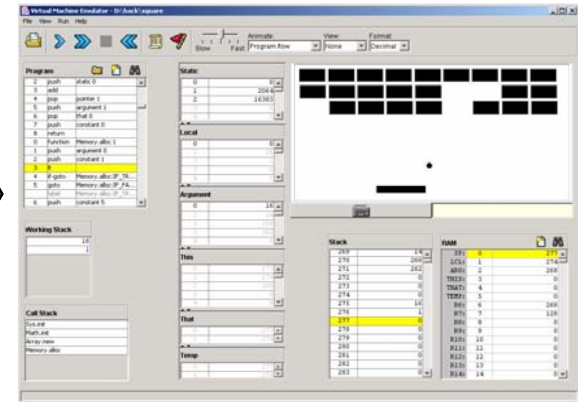
Recap

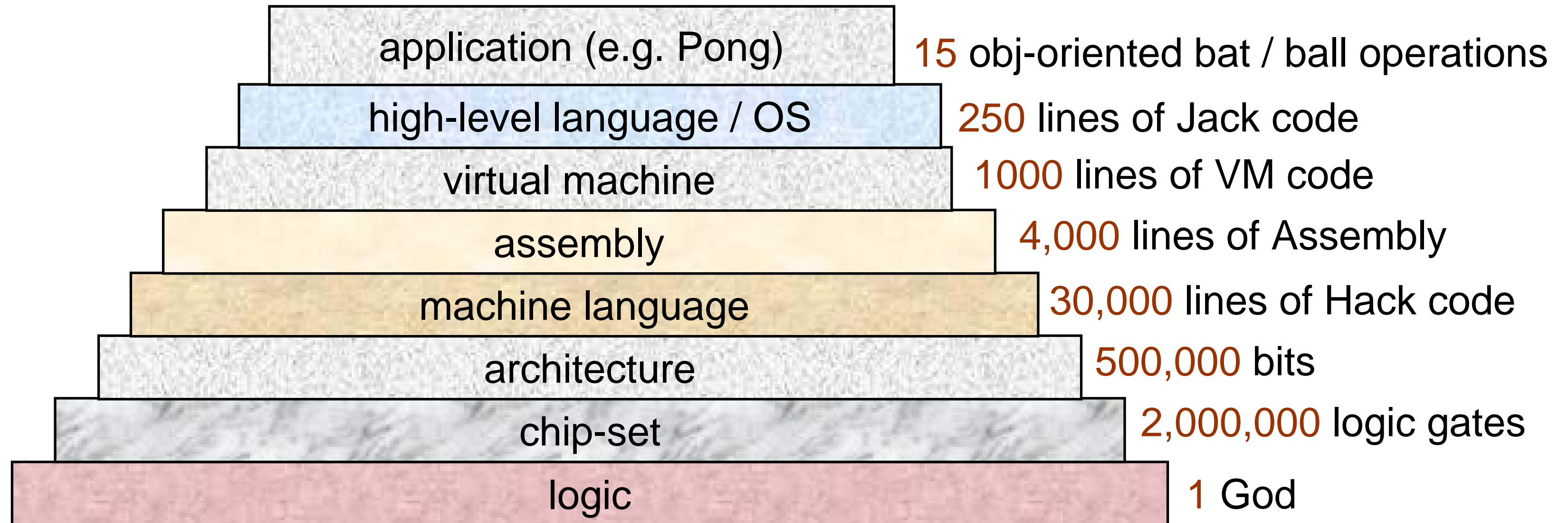




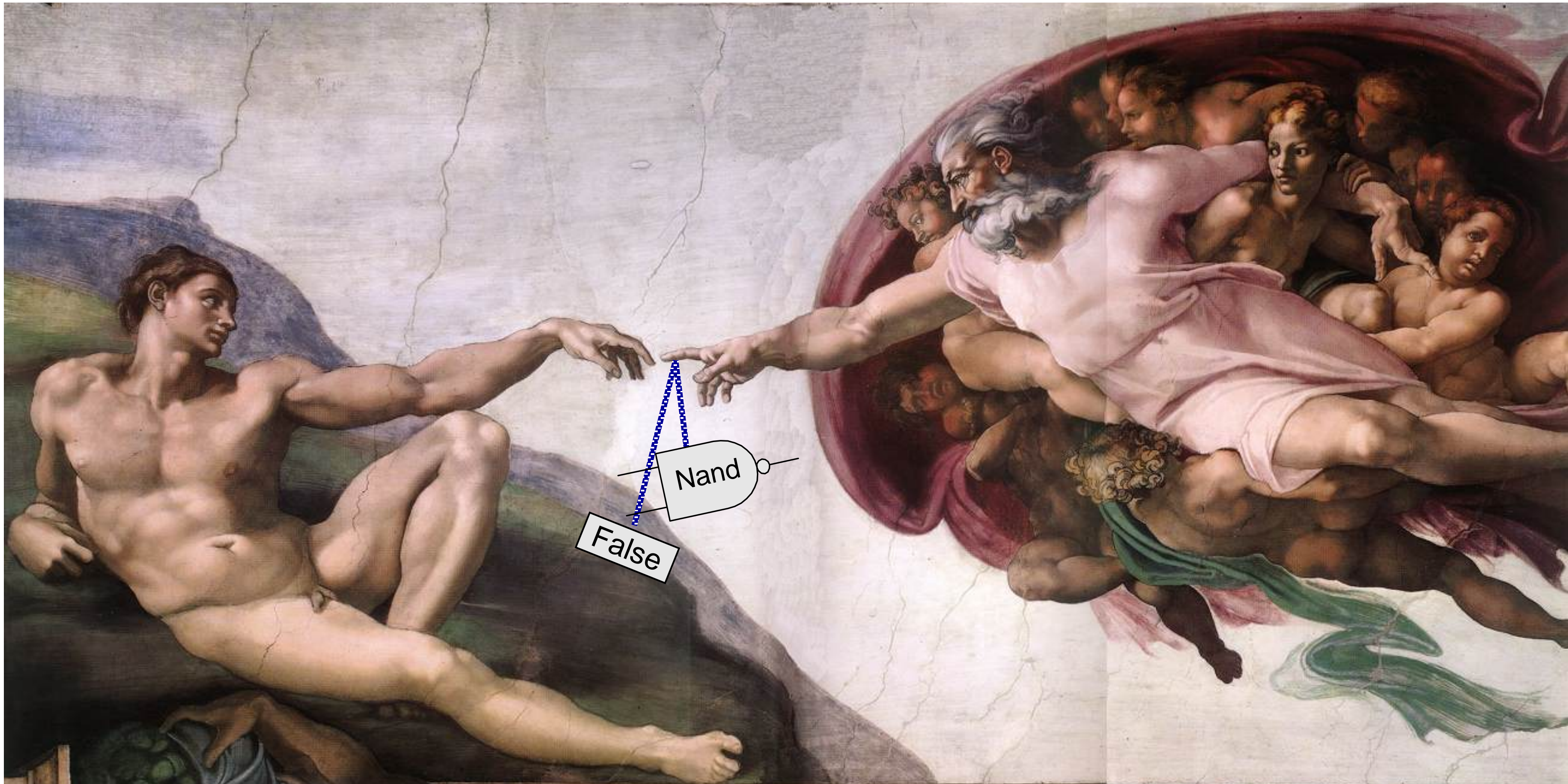






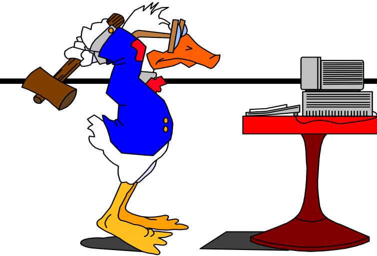


God gave us 0 and Nand



Everything else was done by humans.

Why the approach works



Take home

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

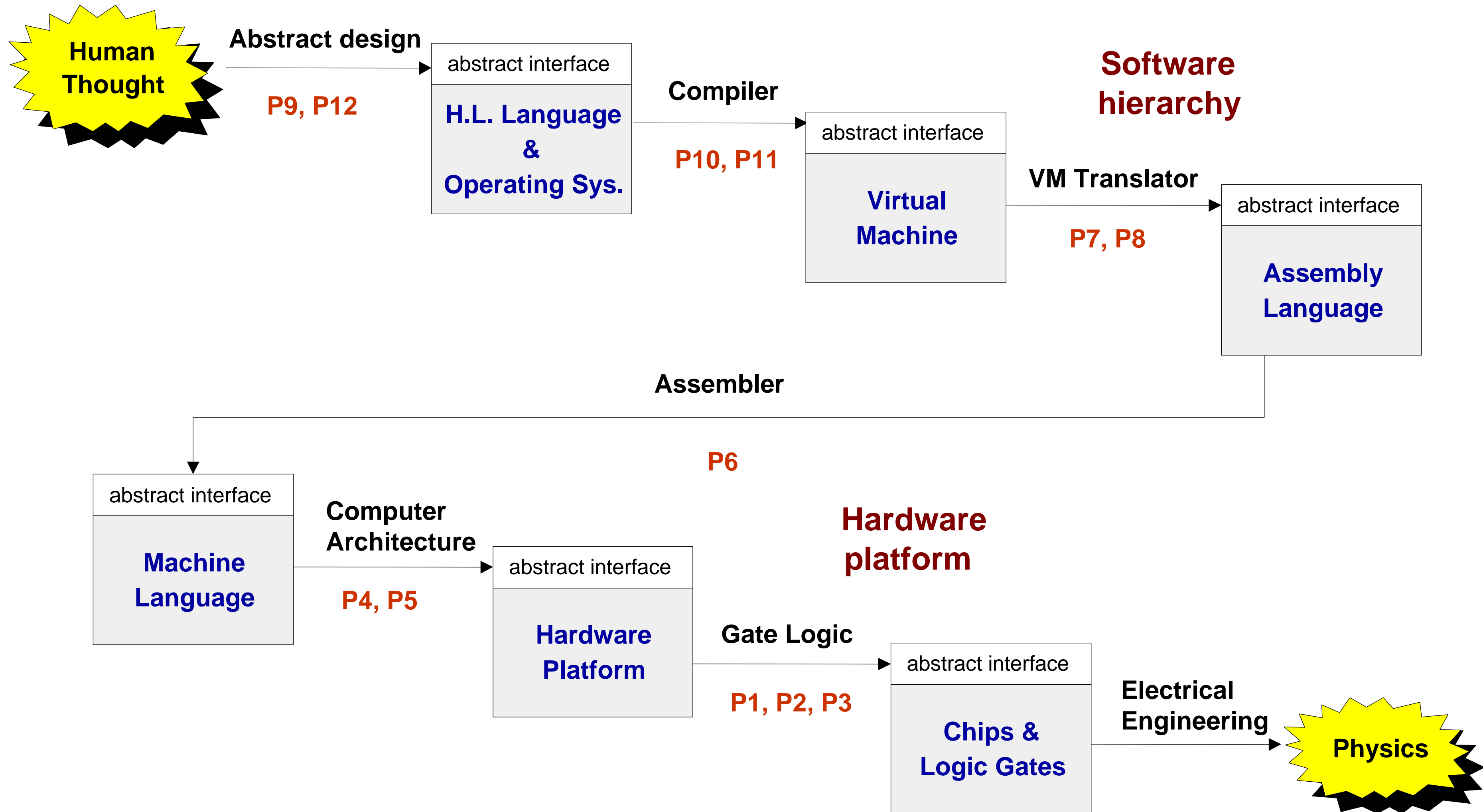
Occam razor

- No optimization
- No advanced features
- No exceptions

Highly-Managed

- Detailed API's are given
- Hundreds of test files and programs
- Modular projects, unit-testing.

Abstraction–Implementation Paradigm



Course / book site

www.idc.ac.il/tecs