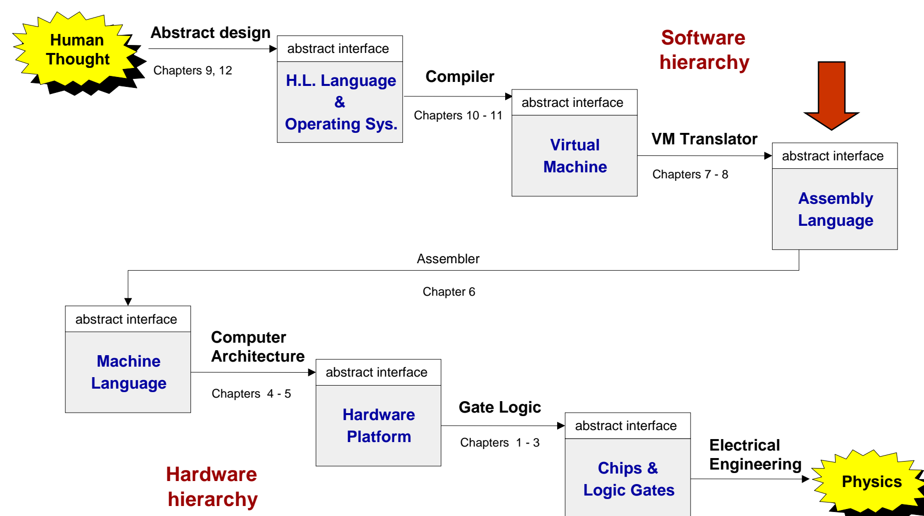
Machine Language

Shimon Schocken

Spring 2005

Where we are at:



Copyright © Shimon Schocken

Machine language is "the soul of the machine"

Duality:

- Machine language (= instruction set) can be viewed as an abstract description of the hardware platform
- The hardware can be viewed as a means for realizing an abstract machine language

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*
- Varies across different hardware platforms.

Binary and symbolic notation

```
1010 0011 0001 1001
```

```
ADD R3, R1, R9
```

- Machine instructions are typically broken into *fields*
- Evolution:
  - Symbolic documentation (*op-codes* and memory references)
  - Symbolic writing
  - Requires a *translator*.

Ada Lovelace
(1815-1852)

Copyright © Shimon Schocken

Lecture plan

- Introduction to machine languages
- The Hack machine language:
  - Symbolic version
  - Binary version

Arithmetic and logical operations

```
ADD R2,R1,R3   // R2←R1+R3 where R1,R2,R3 are registers

ADD R2,R1,foo  // R2←R1+foo where foo stands for the value of the
               // memory location pointed at by the user-defined
               // label foo.

AND R1,R1,R2   // R1←bit wise And of R1 and R2
```


Copyright © Shimon Schocken

Memory access operations

Direct addressing:

```
LOAD R1,67     //  R1←Memory[67]

// Or, assuming that bar refers to memory address 67:

LOAD R1,bar    //  R1←Memory[67]
```

Immediate addressing:

```
LOADI R1,67    // R1←67
```

Indirect addressing:

```
// Translation of x=foo[j] or x=*(foo+j):
ADD R1,foo,j    // R1←foo+j
LOAD* R2,R1     // R2←memory[R1]
STR R2,x        // x←R2
```


Flow of control operations

High level:

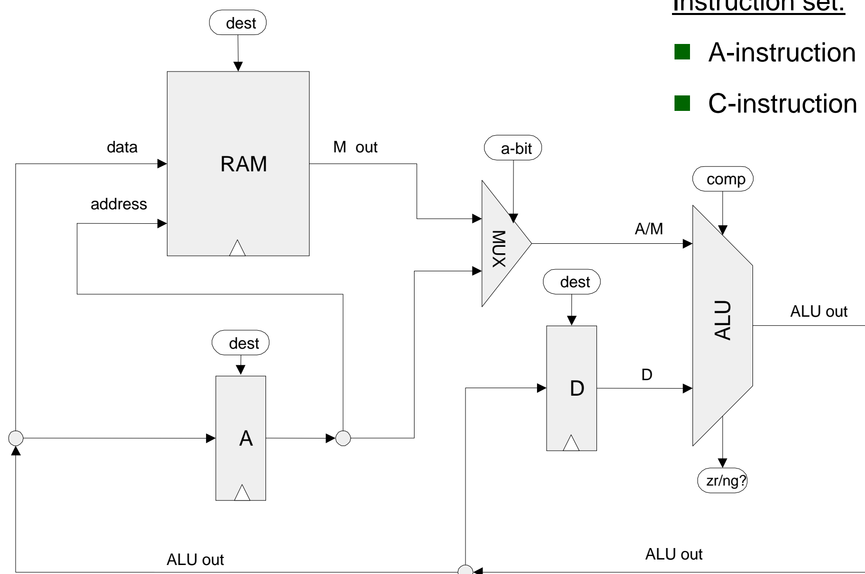
```
// A while loop:
while (R1>=0) {
   code segment 1
}
code segment 2
```

Low level:

```
// Typical translation:
beginWhile:
  JNG R1,endWhile // If R1<0 goto endWhile
  // Translation of code segment 1 comes here
  JMP beginWhile     // Goto beginWhile
endWhile:
  // Translation of code segment 2 comes here
```


Copyright © Shimon Schocken

The Hack hardware platform (first approximation, without ROM and PC)



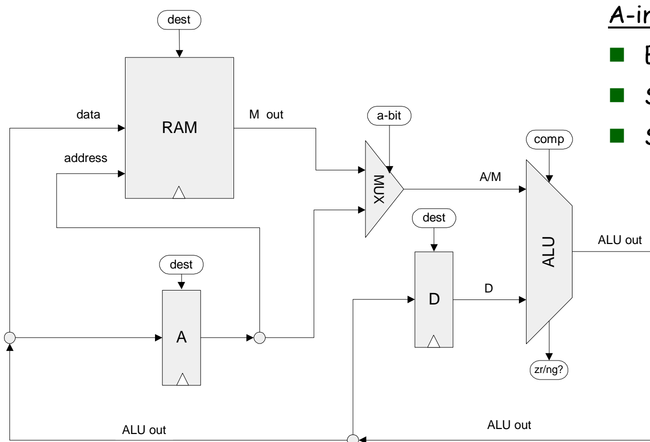
Instruction set:

- A-instruction
- C-instruction

A-instruction

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

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



A-instructions are used for:

- Entering constants
- Selecting a RAM location
- Selecting a ROM location.

Copyright © Shimon Schocken

C-instruction

```
dest = comp ; jump      // comp is mandatory
                        // dest and jump are optional
```

Where:

`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
```


Some coding examples

- Set A to 17
- Set D to A - 1
- Set both A and D to A + 1
- Compute –1
- Set D to 19
- Set RAM[53] to 171
- Set both A and D to A + D
- Set RAM[5034] to D - 1
- Add 1 to RAM[7], and also store the result in D.

```
@value  // set A to value
```

```
dest = comp ; jump
```


Copyright © Shimon Schocken

Coding high-level operations (examples)

Symbol table:

y	3
j	17
sum	18
q	19
x	2048
end	507
next	112

- `sum = 12`
- `j = j + 1`
- `q = sum + 12 – j`
- `x[j] = 15`
- `Etc.`

Coding branching operations

Symbol table:

y	3
j	17
sum	18
q	19
x	2048
end	507
next	112

Low level:

- If D = 0 goto 112
- If D – 12 < 0 goto 507
- If D – 1 > RAM[12] goto 112

High level:

- If sum > 0 goto end
- If x[i] – 12 <= y goto next

Copyright © Shimon Schocken

Coding branching operations (cont.)

Logic:

```
if RAM[3]=5 then
   goto 100
else goto 200
```

Implementation:

```
@3
D=M     // D=RAM[3]
@5
D=D-A   // D=D-5
@100
D;JEQ   // If D=0 goto 100
@200
0;JMP   // Goto 200
```

- To prevent conflicting use of the A register, in well-written programs a *C*-instruction that includes a jump directive should not contain a reference to M, and vice versa.

Flow of control operations: IF

High level:

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

Low level:

```
     @L1
     // set D to not(condition)
     D;jeq
     // Translation of code segment 1
     @L2
     0;jmp
(L1)
     // Translation of code segment 2
(L2)
     // Translation code segment 3
```


Copyright © Shimon Schocken

Flow of control operations: WHILE

High level:

```
while condition {
   code segment 1
}
code segment 2
```

Low level:

```
(L1)
     @L2
     // set D to not(condition)
     D;jeq
     // Translation of code segment 1
     @L1
     0;jmp
(L2)
     // Translation of code segment 2
```


Complete program example

C:

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

Hack:

```
// Adds 1+...+100.
      @i      // i refers to some mem. location
      M=1     // i=1
      @sum    // sum refers to some mem. 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=sum+i
      @i
      M=M+1    // i=i+1
      @LOOP
      0;JMP    // Goto LOOP
 (END)
      @END
      0;JMP    // Infinite loop
```


Copyright © Shimon Schocken

Lecture plan

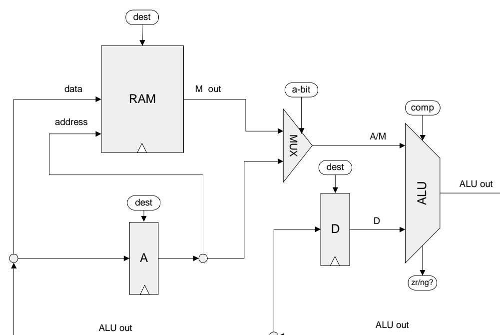
- Symbolic machine language
- Binary machine language

A-instruction

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

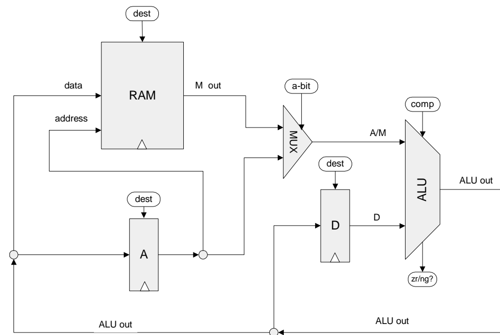


Copyright © Shimon Schocken

C-instruction

Symbolic: *dest=comp;jump* // Either the *dest* or *jump* fields may be empty.

Binary: `1 1 1 a c1 c2 c3 c4 c5 c6 d1 d2 d3 j1 j2 j3` (comp = a c1–c6; dest = d1 d2 d3; jump = j1 j2 j3)



C-instruction (cont.)

Symbolic: *dest=comp;jump* // Either the *dest* or *jump* fields may be empty.

Binary: `1 1 1 a c1 c2 c3 c4 c5 c6 d1 d2 d3 j1 j2 j3` (comp = a c1–c6; dest = d1 d2 d3; jump = 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* ≥ 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* ≤ 0 jump
1	1	1	JMP	Jump

Copyright © Shimon Schocken

Symbols

- Predefined:
  - Virtual registers: **r0**, **r1**, ... , **r15**
  - Predefined pointers: the symbols **SP**, **LCL**, **ARG**, **THIS**, and **THAT** are pre-defined to refer to RAM addresses 0 to 4, respectively.
  - I/O pointers: The symbols **SCREEN** and **KBD** are predefined to refer to RAM addresses 16384 and 24576, respectively (base addresses of the *screen* and *keyboard* memory maps)
- Label symbols: User-defined 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 predefined and is not defined elsewhere using the "**(Xxx)**" command is treated as a variable, and is assigned a unique memory address by the assembler, starting at RAM address 16.

Some more program examples

- Write a program that effects:

  r2 = r0 + r1

- Write a program that effects:

  if r0 > r1 set r2 = 1 else set r2 = 0

Copyright © Shimon Schocken

Perspective

- Hack is a simple language
- User friendly syntax: `D=D+A` instead of `ADD D,D,A`
- Hack is a "½-address machine"
- A Macro-language can be easily developed
- Assembler.

Copyright © Shimon Schocken