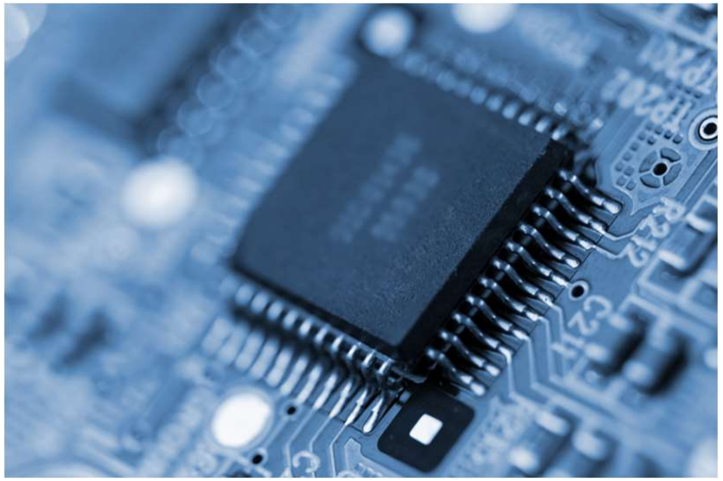
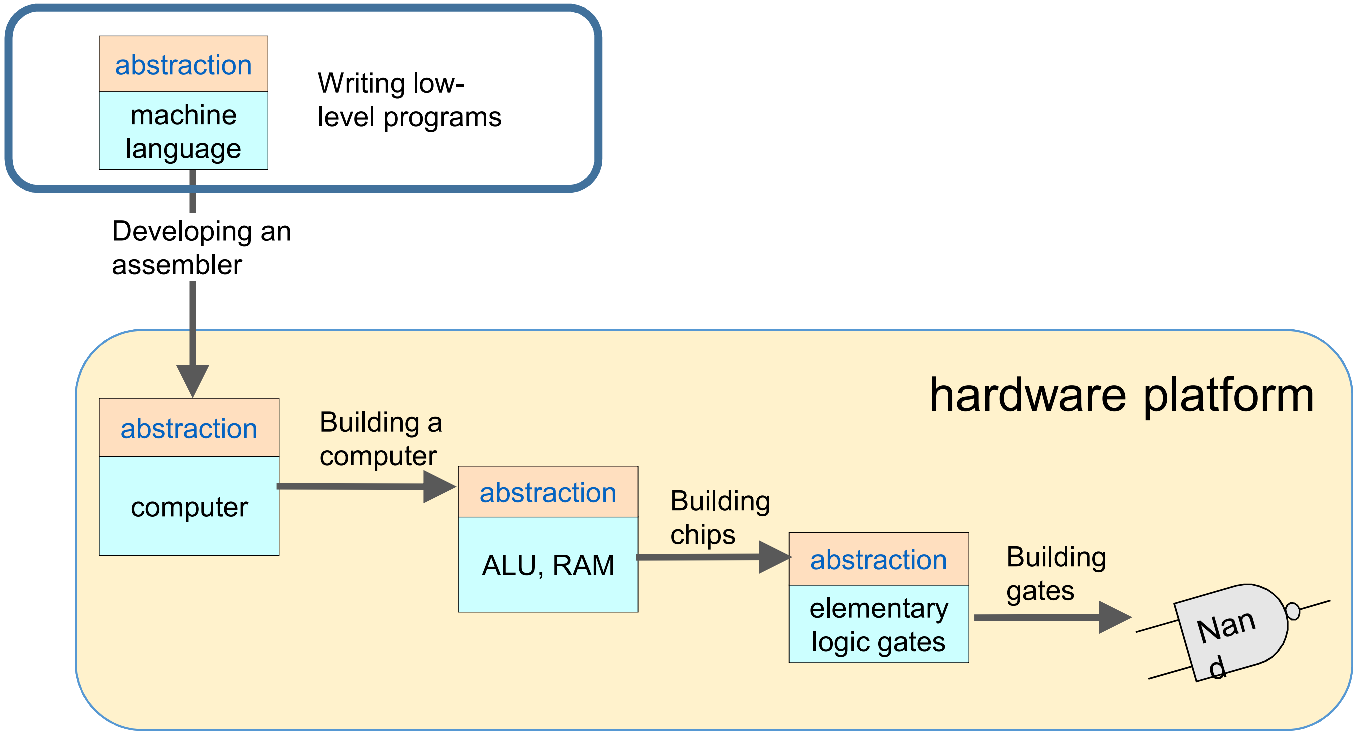
UNIVERSITY OF LEEDS

Computer Processors

Machine Language



Lecture 11


abstraction
machine language
Writing low-level programs
Developing an assembler
hardware platform
abstraction
computer
Building a computer
abstraction
ALU, RAM
Building chips
abstraction
elementary logic gates
Building gates
Nand

Machine Language

UNIVERSITY OF LEEDS

- Up to now the hardware has been a concrete implementation
- Defining a computer abstractly by specifying its machine language
- Machine languages define the interface between programming and logic gates
- Machine languages define low level operations
    - Manipulating memory
    - Basic logic operations
- Rarely used but gives a glimpse into computer architecture

Machine Language

- A *machine language* is an agreed upon formalism, designed to code low level programs as a series of machine instructions, such as
  - Manipulating memory
  - Computing a logical operation

Memory

UNIVERSITY OF LEEDS

- We constructed memory from logic gates in Lecture 9
- Memory is a collection of hardware implementations that store *data* and *instructions*
- Memory can be seen as a continuous array of *words* of a fixed width (16-bit)
- Each location has a unique address
- Using C like syntax we refer to the word in memory using the shorthand RAM[address]

Processor

UNIVERSITY OF LEEDS

- The processor, or Central Processing Unit (CPU), is a device capable of performing a set of basic boolean functions
- Operations typically include
  - arithmetic/logic operations
  - Memory access
  - Flow control (branching)

Registers

UNIVERSITY OF LEEDS

- Registers are ‘special’ memory locations in a Processor
- Memory access is slow
- Machine instructions, unless multiple words are used, can’t reference the entire address space
- Registers are located close to the CPU (Faster)
- Few registers - as few as 4, although modern processors may have many KiloBytes
- All computation is done on registers rather than RAM

Machine Language

- A machine language program is a series of coded instructions
- An instruction is a binary sequence of some specified length (16-bits)

$$\underbrace{0\ 1\ 0\ 1}_{\text{Operation}}\ \underbrace{1\ 1\ 0\ 0}_{\text{Operand 0}}\ \underbrace{1\ 0\ 1\ 0}_{\text{Operand 1}}\ \underbrace{1\ 1\ 0\ 0}_{\text{Operand 2}}$$

- Such an operation might sum operand 1 to operand 2 storing the result in operand 0

Machine Language

UNIVERSITY OF LEEDS

- The operation code (*opcode*) corresponds to operations defined in the decoder circuitry
- Opcodes are often given mnemonics such as **`ADD`** or **`AND`**
- Operands are values stored in registers
- Registers have mnemonics such as **`R1`**, **`R2`**…
- The instruction from before might then look like

```
ADD R2,R1,R0
```

Looks a lot more friendly than

0101110010101100

Assembly

UNIVERSITY OF LEEDS

- Taking the mnemonic abstraction further we can write problems in that format
- Programs can be defined as lists of mnemonics
- A mnemonic is read & translated into the underlying binary sequence that represents a machine instruction
- Some mnemonics might be sequences of machine instructions
- This is the next level past machine language programming, the mnemonics are called *assembly*
- Assembly is translated into machine language by an *Assembler*

Assembly Commands

UNIVERSITY OF LEEDS

- Arithmetic and Logic Operations
    - Addition, Subtraction, boolean operations bit-wise operations

```
ADD R2, R1, R0 // R2 = R1 + R0          R0,R1,R2 are registers
ADD R2, R1, foo // R2 = R1 + foo        R1,R2 and foo are registers
AND R1, R2, R0 // R1 is equal to bit-wise R2 and R0
```

- There would normally be at least one opcode for each ALU function

Assembly Commands

- Memory access
    - Direct addressing
    - Immediate addressing
    - Indirect addressing

LOAD R1, 67 // **R1=Memory[67]** **Direct Addressing**

LOADI R1, 67 // **R1=67** **Immediate Addressing**

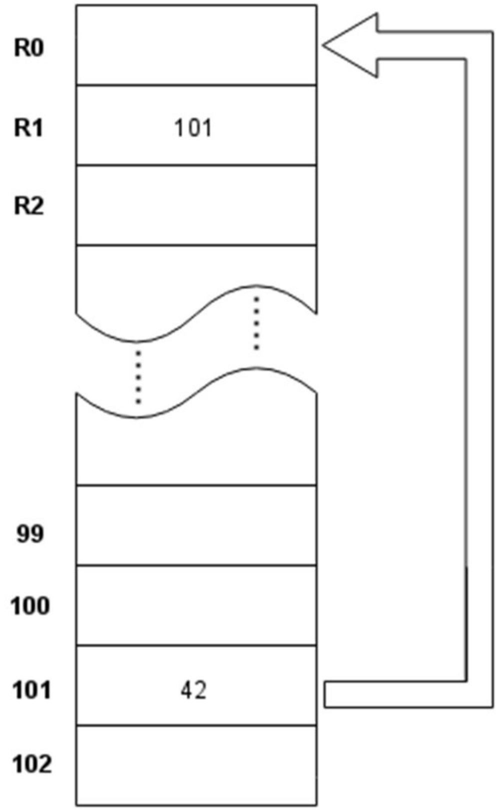
Assembly Commands

UNIVERSITY OF LEEDS

- Indirect addressing
    - Method of implementing pointers
    - Loads the value of the memory address referenced by the value of another register

```
LOADI R1, 101 // R1=101

LOAD* R0, R1        // R0=Memory[R1]
```



Assembly Commands

- Programs normally execute in a linear fashion, one instruction after another
- For programming constructs like **if** statements and **for** loops we want to be able to specify a location to jump to

Execution flow

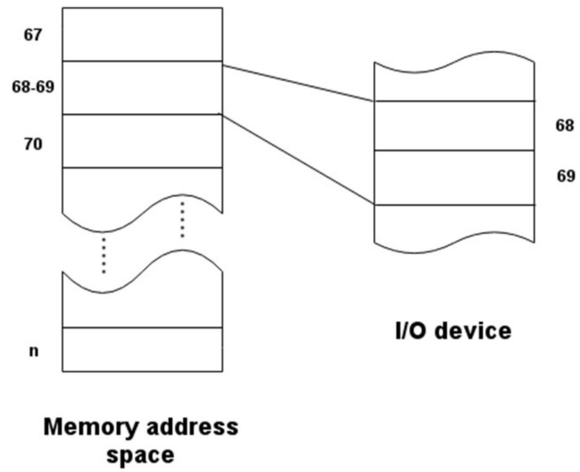
```
for(int i = 0; i <= 10; i++){
        if(i % 2 == 0){
        // output i is even
        }
}
```

i % 2 != 0

While i <=10

Hack Machine language

- Hack is a 16-bit von Neumann architecture
  - A 16-bit CPU
  - Two 32K 16-bit memory modules (instruction memory, data memory)
  - Two memory mapped Input/Output (details in lecture 12)
    - Keyboard
    - Screen



Hack Machine Registers

UNIVERSITY OF LEEDS

- The Hack Machine has two 16-bit registers called **D** and **A**.
- The registers can be manipulated via the instructions
- Register **D** is exclusively used to store data
- Register **A** can be used as a data register or an address register
- There is an implicit label called **M** which refers to the value of the memory location pointed to by register **A**
- Jump instructions use register **A** to specify the destination to jump to
- To load an immediate value into register **A** the following notation is used:

```
@value // load value into register A
```

A-instructions

UNIVERSITY OF LEEDS

- *A*-instructions manipulate register **A**

$$\underbrace{\mathbf{0}}_{A\text{-instruction}}\ \underbrace{\mathbf{0\,0\,0\,0\,0\,0\,0\,0\,0\,1\,0\,0\,1\,0\,1}}_{\text{15-bit value}}$$

is equivalent to:

@37

- *A*-instructions are identified by their leading ‘**0**’
- The following 15-bits are a binary value
- The **@** symbol can be preceded by a symbol referencing a number

A-instructions

UNIVERSITY OF LEEDS

- A-instructions have different purposes
  - To load a constant into the computer under program control
  - To set up for a subsequent *C*-instruction designed to manipulate a certain memory location by first setting the **A** register to the address of the location
  - To set up for a subsequent *C*-instruction design to specify a jump by first loading the destination address into register **A** then executing an instruction which may result in a jump

C-instructions

UNIVERSITY OF LEEDS

- *C*-instructions execute some computation operation
- What to compute, where to store the result and what to do next
- The command format is

```
dest=comp;jump
```

- If either the destination or jump field are empty then they might be omitted

```
M=D+M
D;JGT
0;JMP
```

All valid *C*-instructions

C-instructions

- *C*-instructions execute some computation operation

```
dest=comp;jump
```

- *C*-instructions are encoded in machine language as follows

$$\underbrace{1}_{C\text{-instruction}}\ 1\ 1\ a\ \underbrace{c1\ c2\ c3\ c4\ c5\ c6}_{\text{computation}}\ \underbrace{d1\ d2\ d3}_{\text{destination}}\ \underbrace{j1\ j2\ j3}_{\text{jump}}$$

- *C*-instructions start with a ‘1’, the next two bits aren’t used, the following three fields correspond to the three parts in the symbolic representation

C-instructions: computation

UNIVERSITY OF LEEDS

- The a-bit and the 6 c-bits control the computation executed by a C-instruction

comp mnemonic (when $a = 0$)	c1	c2	c3	c4	c5	c5	comp mnemonic (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
$\neg D$	0	0	1	1	0	1	
$\neg A$	1	1	0	0	0	1	$\neg M$
$-D$	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	$M-D$
$D \wedge A$	0	0	0	0	0	0	$D \wedge M$
$D \vee A$	0	1	0	1	0	1	$D \vee M$

C-instructions: computation

UNIVERSITY OF LEEDS

- If we wanted to compute D-1 the command would be

  1 1 1 **0 0 0 1 1 1 0** d1 d2 d3 j1 j2 j3

- If we wanted to compute D+A the command would be

  1 1 1 **0 0 0 0 0 1 0** d1 d2 d3 j1 j2 j3

- If we wanted to compute the constant 0

  1 1 1 **0 1 0 1 0 1 0** d1 d2 d3 j1 j2 j3

C-instructions: destination

- The value computed by a *C*-instruction can be saved in a number of places (destinations)
- Stored in **D**, **A** or **M**
- The three bits of the destination part of the *C*-instruction indicate what to do with the result

d1	d2	d3	Mnemonic	Description
0	0	0	**null**	the value is not store
0	0	1	**M**	store in **M** (Memory[**A**])
0	1	0	**D**	store in register **D**
0	1	1	**MD**	store in **M** and register **D**
1	0	0	**A**	store in register **A**
1	0	1	**MA**	store in **M** and register **A**
1	1	0	**AD**	store in registers **A** and **D**
1	1	1	**AMD**	store in **M** and registers **A** and **D**

C-instructions: jump

UNIVERSITY OF LEEDS

- The *jump* field instructs the computer which instruction to execute next
- Defaults to executing the next instruction, can be made to fetch and execute any instruction from program memory
- The criteria for a jump is specified by the three jump bits in the instruction and the result of the last computation

j1	j2	j3	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

File specifications

UNIVERSITY OF LEEDS

- Machine language programs by convention have the file extension `.hack`
- Each line contains either an *A*-instruction or a *C*-instruction
- Contract states that line **n** of a `.hack` file will be loaded into program memory address **n**

Summary

- Introduced machine language
- Introduced addressing types
- Introduced assembly and assemblers
- Introduced the specifics of the Hack machine language

Essential Reading - The elements of Computing Systems - Chapter 4