- **Nand2Tetris**
  - before the class
  - Nand2Hack
    - Boolean Function and Gate Logic
      - Unit 1.1：Boolean Logic（布尔逻辑）
      - Unit 1.2：Boolean Functions Synthesis（布尔函数合成）
      - Unit 1.3：Logic Gates（逻辑门）
      - Unit 1.4：Hardware Description Language（硬件描述语言HDL）
      - Unit 1.5：Hardware Simulation（硬件仿真）
      - Unit 1.6：Multi-Bit Buses
      - Unit 1.7 Project 1 Overview
      - solution
    - Boolean Arithmetic（布尔运算） and the ALU
      - Unit 2.1: Binary Numbers
      - Unit 2.2: Binary Addition
      - Unit 2.3: Negative Numbers
      - Unit 2.4: Arithmetic Logic Unit
      - Unit 2.5: Project2 Overview
      - Q & A
      - Solution
    - Memory
      - Unit 3.1: Sequential Logic
      - Unit 3.2: Flip Flops（触发器）
      - Unit 3.3: Memory Units
      - Unit 3.4: Counters
      - Unit 3.5: Project3 Overview
      - Q & A
      - Solution
    - Machine Language
      - Unit 4.1: Machine Languages: Overview
      - Unit 4.2: Machine Languages: Elements
      - Unit 4.3: The Hack Computer and Machine Language
      - Unit 4.4: Hack Language Specification
      - Unit 4.5: Input/Output
      - Unit 4.6: Hack Programming, Part1
      - Unit 4.7: Hack Programming, Part2
      - Unit 4.8: Hack Programming, Part3
      - Unit 4.9: Project4 Overview
      - Q&A
      - Solution
    - Computer Architecture
      - Unit 5.1: Von Neumann Architechture
      - Unit 5.2: The Fetch-Execute Cycle
      - Unit 5.3: Central Processing Unit
      - Unit 5.4: The Hack Computer
      - Unit 5.5: Project 5 Overview
      - Q & A
      - Solution
    - Assembler（汇编器）
      - Unit 6.1: Assembly Languages and Assemblers
      - Unit 6.2: The Hack Assembly Language
      - Unit 6.3: The Assembly Process - Handling Instructions
      - Unit 6.4: The Assembly Process - Handling Symbols

Nand2Tetris

before the class

source of the class

- **课程网站：** Home | nand2tetris
- **授课平台：** Nand2Tetris I, Nand2Tetris II
- **所需软件：** Software | nand2tetris

structure

- From Nand to Hack
- From Hack to Trtris

the folder

- Tools
  - Nand2Tetris Part I
    - Hardware simulator
    - CPU emulator
    - Assembler
  - Nand2Tetris Part II
    - VM Emulator
    - Jack Complier
    - OS
- Projects
  - Nand2Tetris Part I
    - 00
    - 01
    - 02
    - 03
    - 04
    - 05
    - 06
  - Nand2Tetris Part II
    - 07
    - 08
    - 09
    - 10
    - 11
    - 12
    - 13

Weekly Projects for Part I

- Week 1 (Elementary logic gates基本逻辑门)
  - Nand (primitive)与非门
  - Not
  - And
  - Or
  - Xor
  - Mux
  - Dmux
  - Not16
  - And16
  - Or16
  - Mux16
  - Or8Way
  - Mux4Way16
  - Mux8Way16
  - DMux4Way
  - DMux8Way
- Week 2 (Arithmetic Logic Unit)
  - HalfAdder
  - FullAdder
  - Add16
  - Inc16
  - ALU
- Week 3 (Register and memory)
  - Bit
  - Register
  - RAM8
  - RAM64
  - RAM512
  - RAM4K
  - RAM16K
  - PC
- Week 4 (Writing low-level programs)
- Week 5 (Computer architecture)
  - Memory
  - CPU
  - Computer
- Week 6 (Developing an assembler)

Nand2Hack

Boolean Function and Gate Logic

Read:

- Chapter 1 of *The Elements of Computing Systems*.
- HDL Guide (except for A.2.4)

- Hack Chip Set (when writing your HDL programs, you can copy-paste chip-part signatures from here)
- FAQ, containing frequently asked questions by students. If you get an error message you can't solve, you should try looking here

Unit 1.1：Boolean Logic（布尔逻辑）

Two ways to describe the same Boolean function

- formula（逻辑表达式）
- Truth Table（真值表）

Boolean Identities：

- Commutative Laws（交换律）
  - $x\ AND\ y = y\ AND x$
  - $x\ OR\ y = y\ OR\ x$
- Associative Laws（结合律）
  - $(x\ AND\ (y\ AND\ z)) = ((x\ AND\ y)\ AND\ z)$
  - $(x\ OR (y\ OR\ z)) = ((x\ OR\ y)\ OR\ z)$
- Distributive Laws（分配率）
  - $(x\ AND (y\ OR\ z)) = (x\ AND\ y)\ OR\ (x\ AND\ z)$
  - $(x\ OR\ (y\ AND\ z)) = (x\ OR\ y)\ AND\ (x\ OR\ z)$
- De Morgan Laws（德摩根律）
  - $NOT(x\ AND\ y) = NOT(x)\ OR\ NOT(y)$
  - $NOT(x\ OR\ y) = NOT(x)\ AND\ NOT(y)$

Unit 1.2：Boolean Functions Synthesis（布尔函数合成）

- the problem to solve:

how to construct Boolean functions from more primitive operations(truth table)?

construct a disjunctive normal form formula for it, and it goes like this. We actually go row by row in the truth table. We focus only on the rows that have a value of 1.

- 能否得到最简单结果?

how do you actually find the shortest or most efficient formula that's equivalent to the one we've just derived? Well, that's not an easy problem in general. It's not easy for humans, nor is there any algorithm that can do that efficiently. In fact, this is **an NP-hard problem to actually find the shortest expression** that's equivalent to a given one, or even to verify if the expression that you're given is just a constant 0 or 1.

- **Theorem**
  - Any Boolean function can be represented using an expression containing AND,OR and NOT operations
  - Any Boolean function can be represented using an expression containing AND and NOT operations
    - **Proof:** $x\ OR\ y = NOT(NOT(x)\ AND\ NOT(y))$
  - Any Boolean function can be represented using an expression containing **only NAND** operations
    - **Proof:**

- $NOT(x) = (x\ NAND\ x)$
- $(x\ AND\ y) = NOT(x\ NAND\ y)$

Unit 1.3：Logic Gates（逻辑门）

Gate Logic：

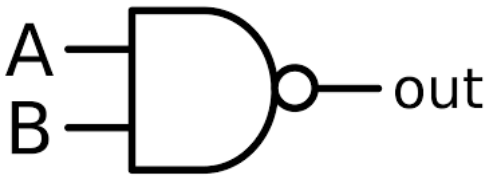
- A technique for implementing Boolean functions using logic gates
- Logic gates:is a stand alone chip which is

  designed to deliver a well-defined functionality
  - Elementary（基本逻辑门）
    - Nand,And,Or,Not,...
  - Composite（复合门）
    - Mux,Adder,...

Elementary logic gates：Nand

---

gate diagram :



functional specification :

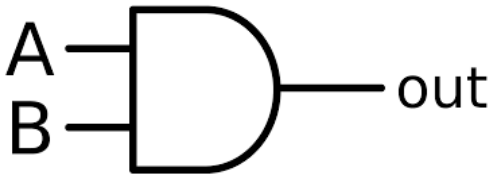
```
if (A==1 and B==1)
then out=0 else out=1
```

truth table :

A	B	out
0	0	1
0	1	1
1	0	1
1	1	0

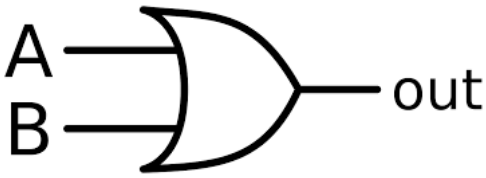
Elementary logic gates：And,Or,Not

And gate diagram :



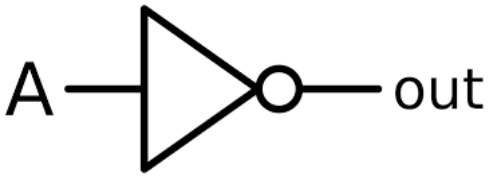
```
if (A==1 and B==1)
then out=1 else out=0
```

Or gate diagram :




```
if (A==1 or B==1)
then out=1 else out=0
```

Not gate diagram :




```
if (A==0)
then out=1 else out=0
```

notion of interface and implementation：

---

gate **interface** is gate **abstraction**, this is how the user thinks about what the gate is supposed to do. Interface **answers** the question： **what**. At the same time, if you want to understand how the chip is doing, what it's doing, you have to, go deeper and you have to go to another level of detail in which you actually,the black box opens up and you see how the chip is actually constructed. Or you do it yourself, you know, if, if your job is to be the person who is actually realizing this, obstruction.

- **The interface of the gate is unique.** There's only one way to describe the gate's functionality.

Otherwise, you know, if there's more than one way, either you're not describing it well or you're confusing the user because, you know, there should be only one unique way to describe what the gate is supposed to do.

- At the same time, there may be **several different implementations that realize the same obstruction.**

And some implementations may be more elegant, may use less energy, they maybe less or more expensive, and so on.

So one obstruction, many different implementations. This is very typical in in computer science, whenever you build a large system, you have this this duality.

Circuit implementations（电路实现）：

---

using light and relays

we have problems in computer science to worry about. So we are not going to worry at all about physical implementations.We are going to take existing logic gates beginning with Nand, and, and beginning with Nand only, and we're going to piece them together in some clever ways in order to generate and, and produce the required functionality.

Unit 1.4：Hardware Description Language（硬件描述语言HDL）

- the problem to solve

how to bulid and implement the logic gates using a formalism called Hardware Description Language or HDL

Design：from requirements to interface

```
/** Xor gate: out = (a And Not(b)) Or (Not(a) And b)*/
CHIP Xor {
    IN a, b;
    Out out;

    PARTS:
    // Implementation missing
}
```

- procedure
    - start the HDL program or the HDL file with some **documentation**, free-form, you can write whatever you want there, which describes what the gate is supposed to do
    - then we **specify the name** of the chip. The names of the inputs of the chips and the names of chip's output. And all this information, by the way, the name of the chip and the names of its inputs and outputs is typically given to you. You know, **it's not something that you decide**
- **gate interface implemented as an HDL stub file**

Design: from requirements to gate diagram

---

let us assume that we've already built an and gate an or gate and, and not gate

Design: from gate diagram to HDL

- procedure
    - **draw the boundary** of the chip diagram or the or the gate diagram,what remains outside the boundary is the user's view of this gate. In other word, the, the gate interface
    - notice that we're using some off the shelf gates and not and or. Now whenever you use an off, off the shelf gate, you are bound to use the names of the gate's input and output as advertised, so to speak. In other words, when you take the gate, gate off the shelf, the gate comes along with what can be called the gate's signature or the gate's API. So we have no degrees of freedom here. We have to **use the official names of the the inputs and outputs** of, of every one of our chip parts
    - name all the internal connections in our architecture

- move on to describe this diagram in HDL

```
/** Xor gate: out = (a And Not(b)) Or (Not(a) And b)*/
CHIP Xor {
    IN a, b;
    Out out;
    //上面为interface，下面为implementation
    PARTS:
    Not (in=a, out=nota);
    Not (in=b, out=notb);
    And (a=a, b=notb, out=aAndNotb);
    And (a=nota, b=b, out=notaAndb);
    Or  (a=aAndNotb, b=notaAndb, out=out);
}
```

①for each one of the **chip(A gate for me is simply a simple chip)** parts that we have, we write a single HDL statement that describes the chip along with all it connections

②So this HDL diagram is nothing more than a textural description of the gate diagram

HDL: some comments

- 与其他编程语言（python，java）的共同点：
  - good documentation
  - good descriptive names
    - both to the chips and connections
  - use indentation（缩进）
- 特殊性
  - **HDL is a functional or declarative language**.There is no procedure going on. There's no program execution going on. It is nothing more than a static description of the gate diagram
  - **The order of HDL statements is insignificant.**it is typically it's customary to begin to describe your diagram from left to right and this also makes the code more readable
  - **Before using a chip part,you must know its interface.**For example:

`Not(in= ,out=), And(a=, b=, out=), Or(a= ,b= ,out= )`

  - Connections like partName(a=a,...) and partName(...,out=out) are common

Hardware description languages

Common HDLs:

- VHDL
- Verilog
- Many more HDLs...

Our HDL

- Similar in spirit to other HDLs
- Minimal and simple
- Provides all your need for this course
- HDL Documentation:

- Textbook/Appendix A
- www.nand2tetris.org/HDL Survival Guide

Unit 1.5：Hardware Simulation（硬件仿真）

- the goal to reach

learn how we can take an HDL program and **verify** to the best of our ability that the program or the HDL file delivers the intended functionality of the underlying chip.

Hardware simulation in a nutshell（简言之）

Simulation options

- Interactive
- script-based simulation
- with/without output and compare file

Interactive simulation

Simulation process

- Load the HDL file into the hardware simulator
- Enter values(0's and 1's)into the chip's input pins(e.g. a and b)
- Evaluate the chip's logic
- Inspect the resulting values of:
  - The output pins (e.g. out)
  - The internal pins (e.g. nota,notb,aAndNotb,notaAndb)

Script-based simulation

---

eg:Xor.tst

```
load Xor.hdl,
output-file Xor.out,
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;
```

Xor.out

```
| a | b |out|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |
```

test script=series of commands to the simulator

*Output File,*created by the test script as a side-effect of the simulation process

Benefits:

- "Automatic" testing
- Replicable testing

The logic of a typical test script

- Initialize:
  - Load an HDL file
  - Create an empty out file
  - List the names of the pins whose values will be written to the output file
- Repeat
  - set-eval-output

Hardware simulators

- There are many of them!

Our hardware simulator

- minamal and simple
- provides all you need for this course
- hardware simulator documentation
  - www.nand2tetris.org/Hardware Simulator Tutorial

Simulation-with-compare-file logic

- When each output command is executed,the outputted line is compared to the corresponding line in the compare file
- If the two lines are not the same,the simulator throws a comparison error.

Behavioral simulation

- The chip logic can be implemented in some high-level language
- Enables high-level planning and testing of a hardware architectures before writing any HDL code

Hardware construction projects

- The players(firse approximation):
  - System architects
  - Developers
- The system architect decides which chips are
- For each chip,the architect creates
  - A chip API
  - A test script
  - A compare file
- Given these resources, the developers can build the chips.

The developer's view

- Taken together,the three files provide a convenient specification of:
  - The chip interface(.hdl)
  - What the chip is supposed to(.cmp)
  - How to test the chip(.tst)
- Missing: chip implementation

Unit 1.6：Multi-Bit Buses

Arrays of Bits

- sometimes we manipulate "together" an array of bits
- It it conceptually convenient to think about such a group of bits as a single entity,sometimes termed "bus"
- HDLs will usually provide some convenient notation(符号) for handling these buses

Buses in HDL

```
/*
 * Adds two 16-bit values
 */
CHIP Add16{
    IN a[16], b[16];
    Out out[16];

    PARTs:
        ...
}
```

Using Buses

```
/*
 * Adds three 16-bit values.
 */
CHIP Add3Way16{
    IN first[16], second[16], third[16];
    OUT out[16];

    PARTS:

      Add16(a=first, b=second, out=temp);
      Add16(a=temp, b=third, out=out);
}
```

Working with Bits in Buses——multi-way chips（多通道门）

```
/*
 * ANDs together all 4 bits of the input
 */
CHIP And4Way{
    IN a[4];
    OUT out;

    PARTS:
        AND(a=a[0], b=a[1], out=t01);
        AND(a=t01, b=a[2], out=t012);
        AND(a=t012, b=a[3], out=out);
}
```

Working with Bits in Buses——multi-bit chips（多位门）

```
/*
 * Computes a bit-wise and of its two 4-bit
 * input buses
 */
CHIP And4{
    IN a[4], b[4];
    OUT out[4];

    PARTS:
        AND(a=a[0], b=b[0], out=out[0]);
        AND(a=a[1], b=b[1], out=out[1]);
        AND(a=a[2], b=b[2], out=out[2]);
        AND(a=a[3], b=b[3], out=out[3]);
}
```

Sub-buses

- Buses can be composed from(and broken into) sub-buses

```
...
IN lsb[8], msb[8], ...
...
Add16(a[0..7]=lsb, a[8..15]=msb, b=..., out=...);
Add16(...,out[0..3]=t1,out[4..15]=t2)
```

- Some syntactic choices of our HDL
    - Overlaps（重叠） of sub-buses are allowed on output buses of parts
    - Width of internal pins is deduced automatically
    - "false" and "true" may be used as buses of any width

Unit 1.7 Project 1 Overview

Project 1

Given：Nand

Goal：Bulid the following gates:

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

Why these 15 particular gates?

Because...

- They are commonly used gates
- They comprise all the elementary logic gates needed to build our computer

Elementary logic gates

- Not
- And
- Or
- Xor
- Mux
- DMux

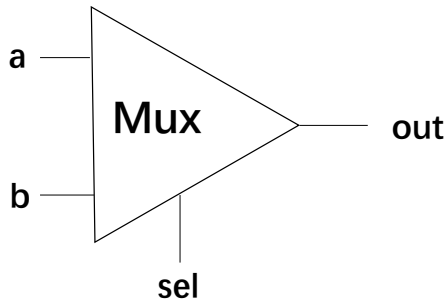
16-bit variants

- Not16
- And16
- Or16
- Mux16

Multi-way variants

- Or8Way
- Mux4Way16
- Mux8Way16
- DMux4Way
- DMux8Way

Multiplexor（数据选择器）

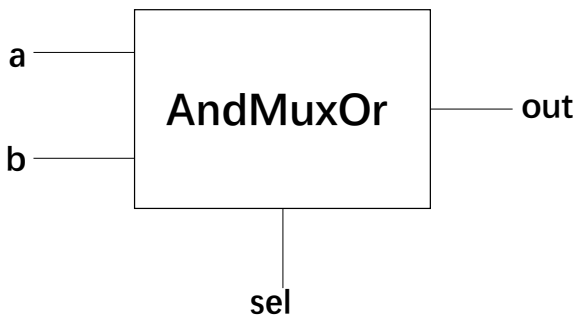


features:

```
if (sel==0)
    out=a
else
    out=b
```

- A 2-way multiplexor enables selecting and outputting,one out of two possible inputs
- Widely used in
    - Digital design
    - Communications networks
- Implementation tip:can be implemented with And,Or,and Not gates

Example: using mux logic to build a programble gate

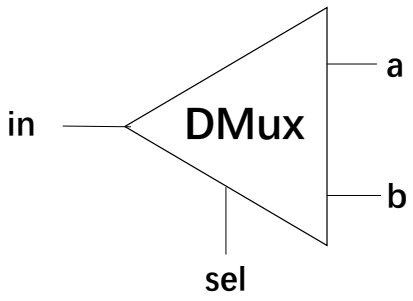


```
if (sel==0)
    out = (a And b)
else
    out = (a Or b)
```

Mux.hdl

```
CHIP AndMuxOr{
    IN a,b,sel;
    OUT out;

    PARTS:
    And (a=a, b=b, out=andOut);
    Or  (a=a, b=b, out=orOut);
    Mux (a=andOut, b=orOut, sel=sel, out=out);
}
```

Demultiplexor（多路分配器）

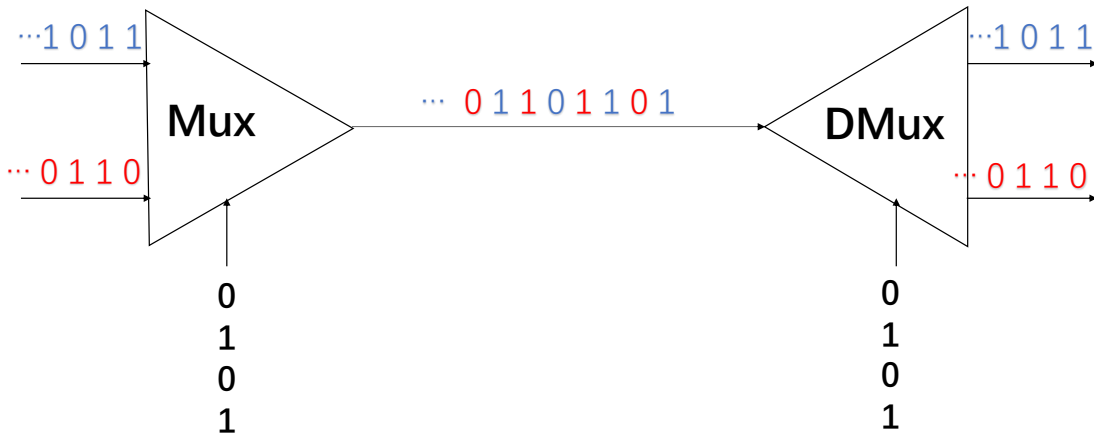



```
if (sel==0)
    {a,b}={in,0}
else
    {a,b}={0,in}
```

- Acts like the "inverse" of a multiplexor
- Distributes the single input value into one of two possible destinations

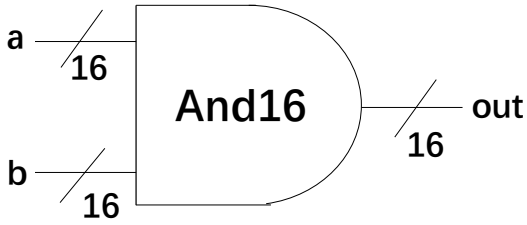
Example:Multiplexing/demultiplexing in communications networks

---



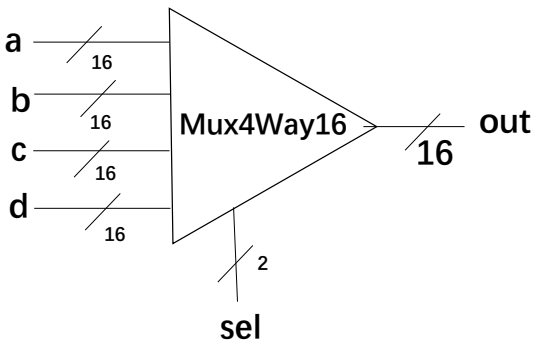
- Each sel bit is connected to an oscillator that produces a repetitive train of alternating 0 and 1 signals
- Enables transmitting multiple messages on a single.shared communications line
- A common use of multiplexing/demultiplexing logic,unrelated to this course

And16



- A straightforward 16-bit extension of the elementary And gate
- See Unit 1.6 for more details on working with multi-bit buses

16-bit,4-way multiplexor



sel[1]	sel[0]	out
0	0	a
0	1	b
1	0	c
1	1	d

- Implementation tips:Can be bulit from several Mux16 gates

Build-in chips

- If you do't implement some chips from the Hack chipset,you can still use them as chip-parts of other chips
- Just rename their given stub files to,say,ChipName.hdl1
- This will cause the simulator to use bulit-in chip implementation instead of the HDL implementation

Best practice advice

- Try to implement the chips in the given order
- If you don't implement some chips,you can still use them as chip-parts in other chips(use bulit-in implementations)
- You can invent new,"helper chips";however,this is not required:you can build any chip using previously-bulit chips only
- Strive to use as few chip-parts as possible

Implementation end-notes

- Create and edit your *.hdl files using a text editor
- A chip cannot be used in its own implementation

- When running an HDL program in the hradware simulator,errors are reported in red,at the left-bottom corner
- Multi-bit busses(unit1.6) are indexed right to left:If A is a 16-bit bus,then A[0] is the right-most bit,and A[15] is the left-most bit
- Use the supplied resources:
    - Hardware simulator tutorial
    - HDL Appendix(book)
    - HDL Survival Guide(by Mark Armbrust)

solution

- Not.hdl

```
CHIP Not {
    IN in;
    OUT out;

    PARTS:
    // Put your code here:
    Nand(a = in,b = true,out = out);
}
```

- And.hdl

```
CHIP And {
    IN a, b;
    OUT out;

    PARTS:
    // Put your code here:
    Nand(a=a, b=b, out=w1);
    Not(in=w1, out=out);
}
```

- Or.hdl

```
CHIP Or {
    IN a, b;
    OUT out;

    PARTS:
    // Put your code here:
    Not(in=a, out=nota);
    Not(in=b, out=notb);
    Nand(a=nota, b=notb, out=out);
}
```

- Xor.hdl

```
CHIP Xor {
    IN a, b;
    OUT out;

    PARTS:
    Not (in=a, out=nota);
    Not (in=b, out=notb);
    And (a=a, b=notb, out=w1);
    And (a=nota, b=b, out=w2);
    Or  (a=w1, b=w2, out=out);
}
```

- Mux.hdl

```
CHIP Mux {
    IN a, b, sel;
    OUT out;

    PARTS:
    // Put your code here:
    Not(in=sel, out=notsel);
    And(a=b, b=sel, out=w1);
    And(a=notsel, b=a, out=w2);
    Or(a=w1, b=w2, out=out);
}
```

- DMux.hdl

```
CHIP DMux {
    IN in, sel;
    OUT a, b;

    PARTS:
    // Put your code here:
    And(a=in, b=sel, out=b);
    Not(in=sel, out=notsel);
    And(a=in, b=notsel, out=a);
}
```

- And16.hdl

```
CHIP And16 {
    IN a[16], b[16];
    OUT out[16];

    PARTS:
    // Put your code here:
    And(a=a[0], b=b[0], out=out[0]);
    And(a=a[1], b=b[1], out=out[1]);
    And(a=a[2], b=b[2], out=out[2]);
    And(a=a[3], b=b[3], out=out[3]);
    And(a=a[4], b=b[4], out=out[4]);
    And(a=a[5], b=b[5], out=out[5]);
```

```
    And(a=a[6], b=b[6], out=out[6]);
    And(a=a[7], b=b[7], out=out[7]);
    And(a=a[8], b=b[8], out=out[8]);
    And(a=a[9], b=b[9], out=out[9]);
    And(a=a[10], b=b[10], out=out[10]);
    And(a=a[11], b=b[11], out=out[11]);
    And(a=a[12], b=b[12], out=out[12]);
    And(a=a[13], b=b[13], out=out[13]);
    And(a=a[14], b=b[14], out=out[14]);
    And(a=a[15], b=b[15], out=out[15]);
}
```

- Not16.hdl

```
CHIP Not16 {
    IN in[16];
    OUT out[16];

    PARTS:
    // Put your code here:
    Not(in=in[0], out=out[0]);
    Not(in=in[1], out=out[1]);
    Not(in=in[2], out=out[2]);
    Not(in=in[3], out=out[3]);
    Not(in=in[4], out=out[4]);
    Not(in=in[5], out=out[5]);
    Not(in=in[6], out=out[6]);
    Not(in=in[7], out=out[7]);
    Not(in=in[8], out=out[8]);
    Not(in=in[9], out=out[9]);
    Not(in=in[10], out=out[10]);
    Not(in=in[11], out=out[11]);
    Not(in=in[12], out=out[12]);
    Not(in=in[13], out=out[13]);
    Not(in=in[14], out=out[14]);
    Not(in=in[15], out=out[15]);
}
```

- Or16.hdl

```
CHIP Or16 {
    IN a[16], b[16];
    OUT out[16];

    PARTS:
    // Put your code here:
    Or(a=a[0], b=b[0], out=out[0]);
    Or(a=a[1], b=b[1], out=out[1]);
    Or(a=a[2], b=b[2], out=out[2]);
    Or(a=a[3], b=b[3], out=out[3]);
    Or(a=a[4], b=b[4], out=out[4]);
    Or(a=a[5], b=b[5], out=out[5]);
    Or(a=a[6], b=b[6], out=out[6]);
    Or(a=a[7], b=b[7], out=out[7]);
```

```
    Or(a=a[8], b=b[8], out=out[8]);
    Or(a=a[9], b=b[9], out=out[9]);
    Or(a=a[10], b=b[10], out=out[10]);
    Or(a=a[11], b=b[11], out=out[11]);
    Or(a=a[12], b=b[12], out=out[12]);
    Or(a=a[13], b=b[13], out=out[13]);
    Or(a=a[14], b=b[14], out=out[14]);
    Or(a=a[15], b=b[15], out=out[15]);
}
```

- Mux16.hdl

```
CHIP Mux16 {
    IN a[16], b[16], sel;
    OUT out[16];

    PARTS:
    // Put your code here:
    Mux(a=a[0], b=b[0], sel=sel, out=out[0]);
    Mux(a=a[1], b=b[1], sel=sel, out=out[1]);
    Mux(a=a[2], b=b[2], sel=sel, out=out[2]);
    Mux(a=a[3], b=b[3], sel=sel, out=out[3]);
    Mux(a=a[4], b=b[4], sel=sel, out=out[4]);
    Mux(a=a[5], b=b[5], sel=sel, out=out[5]);
    Mux(a=a[6], b=b[6], sel=sel, out=out[6]);
    Mux(a=a[7], b=b[7], sel=sel, out=out[7]);
    Mux(a=a[8], b=b[8], sel=sel, out=out[8]);
    Mux(a=a[9], b=b[9], sel=sel, out=out[9]);
    Mux(a=a[10], b=b[10], sel=sel, out=out[10]);
    Mux(a=a[11], b=b[11], sel=sel, out=out[11]);
    Mux(a=a[12], b=b[12], sel=sel, out=out[12]);
    Mux(a=a[13], b=b[13], sel=sel, out=out[13]);
    Mux(a=a[14], b=b[14], sel=sel, out=out[14]);
    Mux(a=a[15], b=b[15], sel=sel, out=out[15]);
}
```

- Or8Way.hdl

```
CHIP Or8Way {
    IN in[8];
    OUT out;

    PARTS:
    // Put your code here:
    Or(a=in[0], b=in[1], out=o1);
    Or(a=o1, b=in[2], out=o2);
    Or(a=o2, b=in[3], out=o3);
    Or(a=o3, b=in[4], out=o4);
    Or(a=o4, b=in[5], out=o5);
    Or(a=o5, b=in[6], out=o6);
    Or(a=o6, b=in[7], out=out);
}
```

- Mux4Way16.hdl

```
CHIP Mux4Way16 {
    IN a[16], b[16], c[16], d[16], sel[2];
    OUT out[16];

    PARTS:
    // Put your code here:
    Mux16(a=a, b=b, sel=sel[0], out=amux16b);
    Mux16(a=c, b=d, sel=sel[0], out=cmux16d);
    Mux16(a=amux16b, b=cmux16d, sel=sel[1], out=out);
}
```

- Mux8Way16.hdl

```
CHIP Mux8Way16 {
    IN a[16], b[16], c[16], d[16],
       e[16], f[16], g[16], h[16],
       sel[3];
    OUT out[16];

    PARTS:
    // Put your code here:
    Mux16(a=a, b=b, sel=sel[0], out=amux16b);
    Mux16(a=c, b=d, sel=sel[0], out=cmux16d);
    Mux16(a=e, b=f, sel=sel[0], out=emux16f);
    Mux16(a=g, b=h, sel=sel[0], out=gmux16h);
    Mux4Way16(a=amux16b, b=cmux16d, c=emux16f, d=gmux16h, sel=sel[1..2],
out=out);
}
```

- DMux4Way.hdl

```
CHIP DMux4Way {
    IN in, sel[2];
    OUT a, b, c, d;

    PARTS:
    // Put your code here:
    DMux(in=in, sel=sel[0], a=ac, b=bd);
    DMux(in=ac, sel=sel[1], a=a, b=c);
    DMux(in=bd, sel=sel[1], a=b, b=d);
}
```

- DMux8Way.hdl

```
CHIP DMux8Way {
    IN in, sel[3];
    OUT a, b, c, d, e, f, g, h;

    PARTS:
    // Put your code here:
    DMux(in=in, sel=sel[2], a=abcd, b=efgh);
    DMux4Way(in=abcd,sel=sel[0..1], a=a, b=b, c=c, d=d);
    DMux4Way(in=efgh,sel=sel[0..1], a=e, b=f, c=g, d=h);
}
```

Boolean Arithmetic（布尔运算） and the ALU

Using the chipset that we've built in the previous module, we will now proceed to build a family of adders -- chips designed to add numbers. We will then take a big step forward and build an Arithmetic Logic Unit. The ALU, which is designed to perform a whole set of arithmetic and logical operations, is the computer's calculating brain. Later in the course we will use this ALU as the centerpiece chip from which we will build the computer's Central Processing Unit, or CPU. Since all these chips operate on binary numbers (0's and 1's), we will start this module with a general overview of binary arithmetic, and only then delve into building the ALU

Unit 2.1: Binary Numbers

Binary→Decimal：

$b_n b_{n-1} b_{n-2} \dots b_1 b_0 = \sum_i b_i \cdot 2^i$

Maximum with k bits is : $1 + 2 + 4 + \dots + 2^{k-1} = 2^k - 1$

Fixed word size

We will use a fixed number of bits.

Unit 2.2: Binary Addition

The whole point of representing something is if we want to manipulate it:

- Addition
- Subtraction
- Which is Greater?
- Multiplication
- Division

What we will really be talking about is how to **add**.Once we do that we'll basically get the whole rest of other operations almost for free:

- once we understand how to represent negative numbers,we will be able to get subtraction for free and to understand which of two numbers is greater for free
- Multiplication and divisions are more complicated, but nicely enough, we can actually postpone them to software. We will not build in hardware any multiplication or division eh, circuitry. But rather, we will actually let software do it and things are much easier to do in software because you just have to write little programs rather than actually connect stupid little devices.

A question of overflow：

Suppose that we were somehow unlucky and the, and the two left-most bits of our, the two numbers we were adding were 1.

So what is the problem with that?

The problem is that, that when we add them, we have a carry that needs to go to the left of the word size. And there is no place to carry, to put that carry bit because we finished our word size.

So what would we do?

Will we raise some warning or anything like that? Well, the answer is very simple. What is usually done in computer systems is nothing. **We just ignore any carry bit that does not fit into the word.**

What does that mean, really?

So if you try to look at it from a mathematical point of view, what it means is that the addition that we're actually doing in our hardware is not real integer addition, because we cannot go beyond the numbers that fit inside the word size. Instead, what we have is really an addition module 2 to the width of the word size if you look at it mathematically. In other words, the answer is correct, but may, except for the case that it may be off by exactly 2 to the n where n is the word size. If the result was more than 2 to the n, the hardware automatically decreases 2 to the n, which is basically the carry that we just threw because there was an overflow. So that's what they usually do, and the rest of anyone using a computer, anyone using computer and software, needs to remember that if he exceeds the word size, then the result that you get, that you get is not the true integer result of the integer addition, but rather the truncated result after the overflow was already disposed of.

Building an Adder

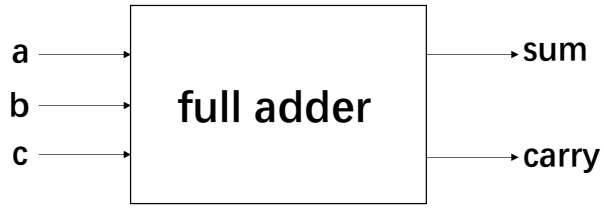
1. Half Adder - adds two bits
2. Full Adder - adds three bits
3. Adder - Adds two numbers

Half Adder:

truth table:

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

Full Adder:

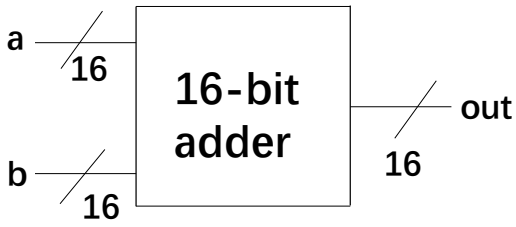


truth table:

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

16-bit Adder

out = a+b, as 16-bit integers(overflow ignored)



Unit 2.3: Negative Numbers

Representing Numbers in 4 Bits

With n bits can represent the positive integers in the range $0 \ldots 2^n - 1$

Negative Numbers - Sign bit

First bit is -/+. All other bits represent a positive number

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

B	D
1110	-6
1111	-7

This way,if it starts with 0,it's going to be just positive numbers,if the first bit is 1,then the next three bits,and it's going to be a negative number.that is represented by the next three bits

This would be one possibility of representing negative numbers. A possibility that is not very popular .Why?There are a bunch of problems with it. One thing that you may immediately notice, this is very inelegant. We have negative 0. What is this negative 0 and why is it different than 0. What we learned in math was that 0 equals negative 0. So of course, we may in principle decide to have two representations of 0 in our computer, but that is inelegant and probably means that there's going to be trouble.**Usually, if you have something that's not elegant, it's going to bite you.** And in fact here, if you actually try to manipulate these kind of negative numbers. Using some kind of hardware, you'll get into trouble. You'll need to explicitly deal with pluses and minuses, and the whole thing will be a mess.

in other words:

Complications:

- -0 ?
- Implementations need to handle different cases

So hardly anyone uses this anyti, anymore.

Below is what people use instead:

2's complement（2-补码）/ radix complement（基补码）

Represent Negative number $-x$ using the positive number：$2^n - x$

Which is going to be a positive number and you are going to present it like we've seen so far

B	D
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	-8 $(2^4 - 8)$
1001	-7 $(2^4 - 7)$

B	D
1010	-6 (2^4-6)
1011	-5 (2^4-5)
1100	-4 (2^4-4)
1101	-3 (2^4-3)
1110	-2 (2^4-2)
1111	-1 (2^4-1)

Positive Numbers in the range：$0 \dots 2^{n-1}-1$

Negative Numbers in the range：$-1 \dots -2^{n-1}$

the main thing that's nice about this trick is that we will basically get our addition and subtraction and almost all the operation that we need to do with numbers almost for free.

Addition in 2's Complement (for Free)

$(-2)+(-3)=(-5)$

$(14)+(13)=(11)$

$(1110)+(1101)=(11011)=(1011)$

$11011=27_{ten}$

$1011=11_{ten}$

Now how does that happen? Why does this magic happen? Will it always happen?

Well as we saw in the last unit, our addition is anyway modulo 2 to the n. （modulo:取余，即mod，$2\ mod\ 16=2$）That is because we throw the overflow bits.

The result that we get is correct up to an additive 2^n at the additive factor.

And our representation is also modulo 2 to the n （负数的余数应该取正的，如：$-3\ mod\ 16=13$）in the sense that we represent two numbers as equal. -3 and 13 are equal up to an addition of exactly the same 2 to the n. And since both the representation and addition have the same convention, then they exactly fit and we don't need to do anything else.

But immediately our hardware that was designed as previously just for positive numberage, just works like it is

Computing -x

Input: x

Output: -x（in 2s complement)

If we solve this when we know how to substract!：$y-x=y+(-x)$

Idea: $2^n-x=1+(2^n-1)-x$

其中2^n-1在二进制中为全由1组成的位，故2^n-1-x即为not x，

not x-1等于$2^n - 1 - (x - 1) = 2^n - x = -x$

eg.

Input: 4

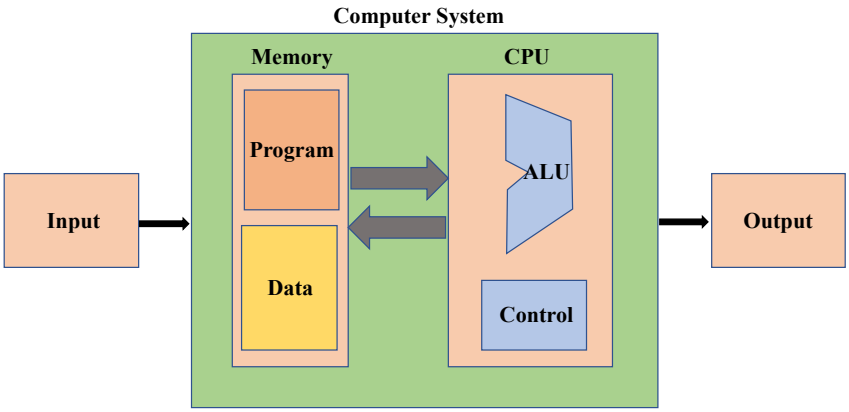
0100->$1111 - 0100$=1011->$1011 + 1$=1100

To add one: Flip the bits from right to left,stopping the first time 0 is filpped to 1

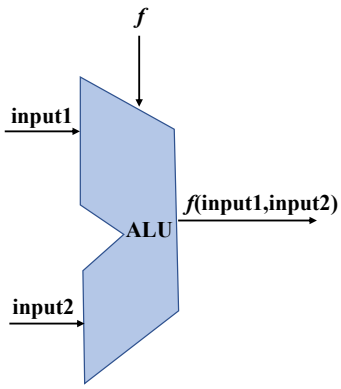
Output: -4 (In 2's complement 12_{ten}: 1100)

Unit 2.4: Arithmetic Logic Unit

Von Neumann Architecture



The Arithmetic Logic Unit



- Abstract
  - receive two multi-bit inputs
  - the third input is the function that has to be computed

The ALU computes a function on the two inputs, and outputs the result

f: one out of a family of pre-defined arithmetic and logical functions

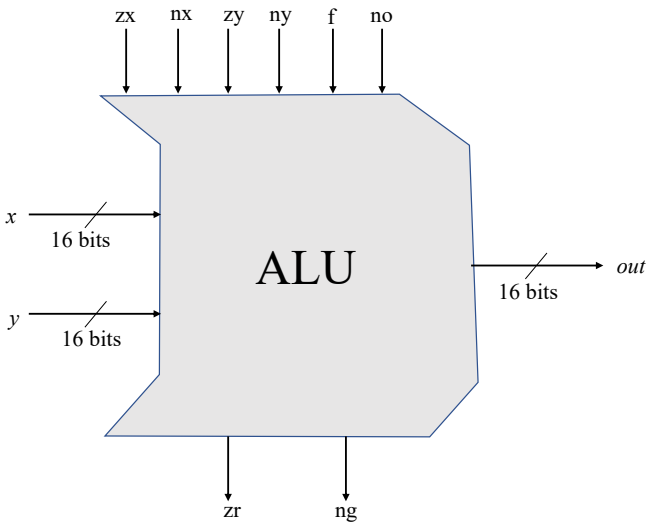
- Arithmetic operations: integer addition,multiplication, division,...
- logical operations: And,Or,Xor,...

Which operations should the ALU perform?

A hardware/software tradeoff

The Hack ALU

- Operates on two 16-bit,two's complement values
- Outputs a 16-bit,two's complement values
- Which function to compare is set by six 1-bit inputs
- Computes one out of a family of 18 functions
- Also outputs two 1-bit values



- out
  - 0
  - 1
  - -1
  - x
  - y
  - !x
  - !y
  - -x
  - -y
  - x+1
  - y+1
  - x-1

    - y-1
    - x+y
    - x-y
    - y-x
    - x&y
    - x|y
- truth table

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&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

- zr and ng

if out==0 then zr = 1,else zr=0

if out<0 then ng = 1,else ng=0

These two control bits will come into play when we build the complete computer's architecture

Perspective

The Hack ALU is：

- Simple
- Elegant
- Easy to implement:
    - Set a 16-bit value to 0000000000000000
    - Set a 16-bit value to 1111111111111111
    - Negate a 16-bit value(bit-wise)
    - Compute + or & on two 16-bit values
    - That's it!

Unit 2.5: Project2 Overview

Project 2

Given: All the chips built in Project1

Goal: Build the following chips:

- HalfAdder
- FullAdder
- Add16
- Inc16
- ALU

A family of combinational chips, from simple adders to an Arithmetic Logic Unit

Half Adder

Implementation tips

Can be built using two very elementary gates.

sum:Xor ; carry:And

Full Adder

Implementation tips

Can be built from two half-adders（这也是半加器这个叫法的来源）

16-bit adder

Implementation tips

- An n-bit adder can be built from n full-adder chips
- The carry bit is "piped" from right to left
- The MSB carry bit is ignored

16-bit incrementor

Implementation tip

The single-bit 0 and 1 values are represented in HDL as false and true

ALU

Implementation tips

- Building blocks: Add16,and various chips built in Project 1
- Can be built with less than 20 lines of HDL code

Best practice advice

- you will have to use chips that you've implemented in Project1.The best practice is to use their built-in versions（只要不把项目1中自己实现的放在项目2的相同目录下，仿真器自己变回调用built-in版本）

Q & A

- *we've already built about 20 chips in this course, are these chips standard? Namely, are they typically being used in many other computer systems, or are they specialized to the computer that you built in this particular course?*

Most of the gates that we build in the course are completely standard. The half adder, the full adder, the adder. These gates are completely standard, as are the gates we built last week, like the multiplexor the or Xor gates and so on. What is not completely standard is our ALU, which is extremely simple. In this course we clearly emphasize simplicity over almost anything else, because we want to fit everything into one course. So for that reason our ALU is extremely simplified in its implementation, and in the respect it's a bit unique among usual ALUs.

- *how come the allele that we built does not feature more operations like multiplication and division.*

Well the answer is that indeed there is no problem to write HDL code that specifies chips that carry out multiplication and division by operating directly on zeroes and ones, on, on bits. And in fact, there are some very elegant and very nice algorithms to do just that.But in general, when you build a computer system the overall functionality that the system provides is divided between the hardware and the operating system that runs on top of it. So, it is the designers freedom to decide how much functionality to put in each one of these layers. When we design the hack computer which is the computer that will continue to build in the course. We decided, as Noam explained before, that the ALU will be extremely simple. And that operations like division and multiplication will be delegated to the software that runs on top of this computer. And indeed, in the second part of this course, in Nand to Tetris part two, we are going to design among other things an operating system, and the operating system will have several libraries and one of these libraries is gone, is going to be called math. And the math library will feature all sorts of of very useful mathematical operations including multiplication and division. So at the end of the day the programmer who writes programs that have to run on this computer will not feel the difference. The programmer will not really care if certain algebraic operation is being done by the operating system or by the hardware. It will be completely transparent for the high level programmer. But of course there's there's some tradeoff here. And typically when you design an operation in hardware typically it runs much faster but it's costly to design and it also costs money to manufacture the the more complex hardware unit. So once again it's a matter of tradeoff, cost effectiveness. And that's how we decided to build the the heck computer. Simple ALU and many

extensions later when we build the operating system. We hope that we convinced you that the ALU that we designed in this course is indeed simple.

- *Is this ALU actually efficient*

Almost everything that we did in the construction is completely efficient, so you really can't say much more.

But there is one component which is where some important optimization is still possible, and it's probably worthwhile to talk for one second about the kinds of optimizations that we're talking about, and this is the adder：adder有多个全加器组成，每个全加器的carry都需要传给下一个全加器的输入，这导致很大的延迟，优化方法：carry look ahead

- *why do you recommend using built-in chips in project two instead of the chips that we actually built in project one?*

the most important of these reasons is the notion of we can call it local failures. And the idea is that if you build, I'm sorry, if you use built-in chips as your chip arts, and some and some problem raises its ugly head in the present project, then you are guaranteed that this problem can be attribute to bugs and problems that were created in this project only, and not in previous projects. So, this is also sometimes called the notion of unit testing. You test each unit separately from the rest of the system. And this principle of unit testing goes hand in hand with other very important principles like abstraction and modularity. And taken together, you know, one of these things that these principles imply is that once you've finished building a certain module you can put it away. You can stop worrying about its implementation and use only the interface or the API of this module when you build more complex functionality. This really is the only way to manage complex projects. And by adhering to these principles which we find extremely important by doing so, we can really take this super-ambitious project of building a modern computer from first principles and do it only in seven weeks.

we should also confess that our simulator is not that efficient, especially when we get to more complex projects, if you actually layer the chips that were constructed in previous projects, our simulator will have to face a huge number or arrays and will simply be slow. If you do it using just the, the finished end of the previous chips, just the, the specifications of the previous chips, then our simulator will be fast and nice to work with.

Solution

- HalfAdder.hdl

```
CHIP HalfAdder {
    IN a, b;    // 1-bit inputs
    OUT sum,    // Right bit of a + b
        carry;  // Left bit of a + b

    PARTS:
    // Put you code here:
    And(a=a, b=b, out=carry);
    Xor(a=a, b=b, out=sum);
}
```

- FullAdder.hdl

```
CHIP FullAdder {
    IN a, b, c;  // 1-bit inputs
    OUT sum,     // Right bit of a + b + c
        carry;   // Left bit of a + b + c

    PARTS:
    // Put you code here:
    HalfAdder(a=a, b=b, sum=halftohalf, carry=halftoor);
    HalfAdder(a=c, b=halftohalf, sum=sum, carry=half2or);
    Or(a=half2or, b=halftoor, out=carry);
}
```

- Add16.hdl

```
CHIP Add16 {
    IN a[16], b[16];
    OUT out[16];

    PARTS:
   // Put you code here:
   FullAdder(a=a[0], b=b[0], c=false, sum=out[0], carry=w1);
   FullAdder(a=a[1], b=b[1], c=w1, sum=out[1], carry=w2);
   FullAdder(a=a[2], b=b[2], c=w2, sum=out[2], carry=w3);
   FullAdder(a=a[3], b=b[3], c=w3, sum=out[3], carry=w4);
   FullAdder(a=a[4], b=b[4], c=w4, sum=out[4], carry=w5);
   FullAdder(a=a[5], b=b[5], c=w5, sum=out[5], carry=w6);
   FullAdder(a=a[6], b=b[6], c=w6, sum=out[6], carry=w7);
   FullAdder(a=a[7], b=b[7], c=w7, sum=out[7], carry=w8);
   FullAdder(a=a[8], b=b[8], c=w8, sum=out[8], carry=w9);
   FullAdder(a=a[9], b=b[9], c=w9, sum=out[9], carry=w10);
   FullAdder(a=a[10], b=b[10], c=w10, sum=out[10], carry=w11);
   FullAdder(a=a[11], b=b[11], c=w11, sum=out[11], carry=w12);
   FullAdder(a=a[12], b=b[12], c=w12, sum=out[12], carry=w13);
   FullAdder(a=a[13], b=b[13], c=w13, sum=out[13], carry=w14);
   FullAdder(a=a[14], b=b[14], c=w14, sum=out[14], carry=w15);
   FullAdder(a=a[15], b=b[15], c=w15, sum=out[15]);
}
```

- Inc16.hdl

```
CHIP Inc16 {
    IN in[16];
    OUT out[16];

    PARTS:
   // Put you code here:
   Add16(a=in, b[0]=true, out=out);
}
```

- ALU.hdl

```
CHIP ALU {
    IN
```

```
        x[16], y[16],  // 16-bit inputs
        zx, // zero the x input?
        nx, // negate the x input?
        zy, // zero the y input?
        ny, // negate the y input?
        f,  // compute out = x + y (if 1) or x & y (if 0)
        no; // negate the out output?

    OUT
        out[16], // 16-bit output
        zr, // 1 if (out == 0), 0 otherwise
        ng; // 1 if (out < 0),  0 otherwise

    PARTS:
   // Put you code here:
   Mux16(a=x, b=false, sel=zx, out=muxx0);
   Mux16(a=y, b=false, sel=zy, out=muxy0);
   Not16(in=muxx0, out=nmuxx0);
   Not16(in=muxy0, out=nmuxy0);
   Mux16(a=muxx0, b=nmuxx0, sel=nx, out=x0muxnx0);
   Mux16(a=muxy0, b=nmuxy0, sel=ny, out=y0muxny0);
   And16(a=x0muxnx0, b=y0muxny0, out=mxandmy);
   Add16(a=x0muxnx0, b=y0muxny0, out=mxaddmy);
   Mux16(a=mxandmy, b=mxaddmy, sel=f, out=muxf);
   Not16(in=muxf, out=nmuxf);
   Mux16(a=muxf, b=nmuxf, sel=no,
out=out,out[0..7]=zout1,out[8..15]=zout2,out[15]=ng);
   Or8Way(in=zout1, out=w1);
   Or8Way(in=zout2, out=w2);
   Or(a=w1, b=w2, out=w3);
   Mux(a=true, b=false, sel=w3, out=zr);
}
```

Memory

Having built the computer's ALU, this module we turn to building the computer's main memory unit, also known as Random Access Memory, or RAM. This will be done gradually, going bottom-up from elementary **flip-flop**（触发器） gates to one-bit registers to n-bit registers to a family of RAM chips. Unlike the computer's processing chips, which are based on combinational logic, the computer's memory logic requires a clock-based **sequential logic**（时序逻辑）. We will start with an overview of this theoretical background, and then move on to build our memory chipset.

> 触发器（flip-flop）是具有记忆功能的最小记忆单元，它能够存储一个基本的比特位（bit），我们在这里称之为"寄存器（register）"。可以同时存储若干比特位（设w位）的寄存器我们称之为"w-bit寄存器（w-bit register）"。可以利用一组w-bit寄存器进一步构成内存（memory），需要注意的是，这里所谓"寄存器（register）"与后面讨论的CPU内部使用的寄存器虽然在英文术语中都是"register"，但从物理构成上和功能上是不一样的。

Unit 3.1: Sequential Logic

Combinatorial Logic（组合逻辑）

- So far we ignored the issue of time

- The inputs were just "there" - fixed and unchanging
- The output was just a function of the input
    - Not of anything that happened "previously"
- The output was computed "instantaneously"
- This is sometimes called "Combinatorial Logic"

Hello, Time

- Use the same hardware over time
    - Inputs change and outputs should follow
    - eg:

```
For i = 1 ... 100:
    a[i] = b[i]+c[i]
```

- Remember "State"
    - Memory
    - Counters
    - eg:

```
For i = 1 ... 100:
    sum = sum + i
```

- Deal with speed
    - sweep it under the rug in a satisfactory manner

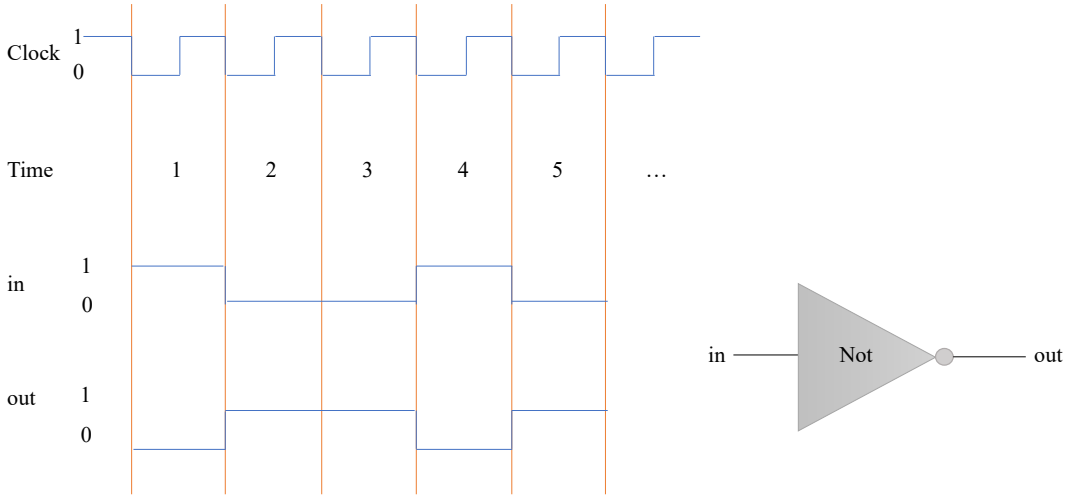
The clock

Physical Time:

in the physical world, time is a continuous arrow of time. Starts at time negative infinite, maybe the big bang, and keep that advancing in seconds, milliseconds, microseconds, picoseconds. Maybe, to some limit of quantum time, but maybe not.

What we're going to do what everyone does in computer science we're going to actually **convert this continuous physical time**, which is very complicated to think about **into discreet time（离散时间）**

in IT clock:



We're going to have what's called a **clock**, which is some kind of of oscillator going up and down at a certain fixed rate. And, each cycle of the clock we're going to treat as one digital integer time unit. So, once we have this clock, it will basically break up our physical continuous time into a sequence of time equals one, time equals two, time equals three, and so on.

Within each time unit, we're going to deal as a time it was in each time unit as though it was one indivisible thing. Nothing changes within a time unit, within an integer time unit.

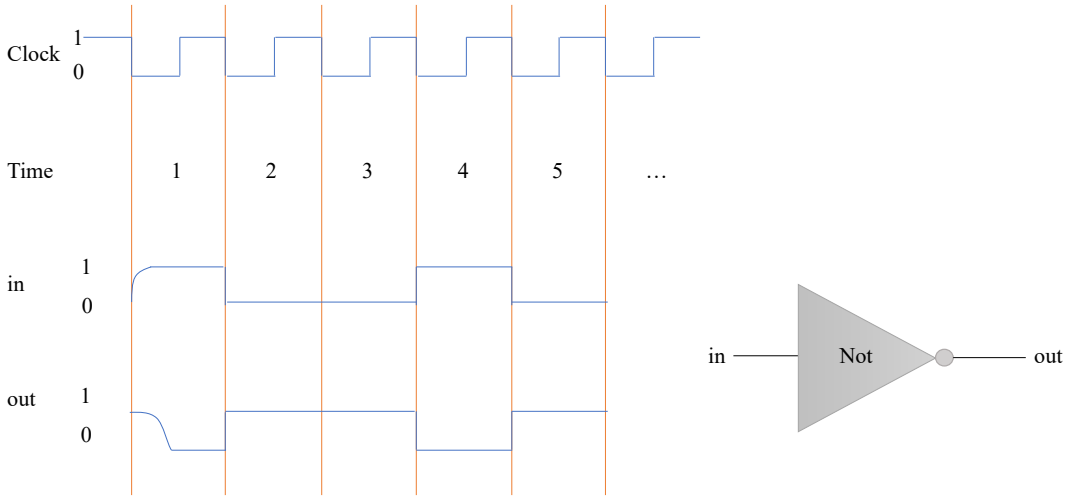
So for example, if we have a not gate, If we look at it's input and it's output, at every different time unit, it can have a different input. And, at that time unit, it will compute the output from that input in an instantaneous manner, as we think about it.

Every time unit the input could change, and then the output would follow. So, if you look at the diagram we see here, we see that in time 1 the input is 1. The output is then of course 0 because that's what an out gate does. At time 2, the input was redu, was go, went down to 0. For example, it could be state one, but if it went out of zero, immediately the output goes up to one and so on.

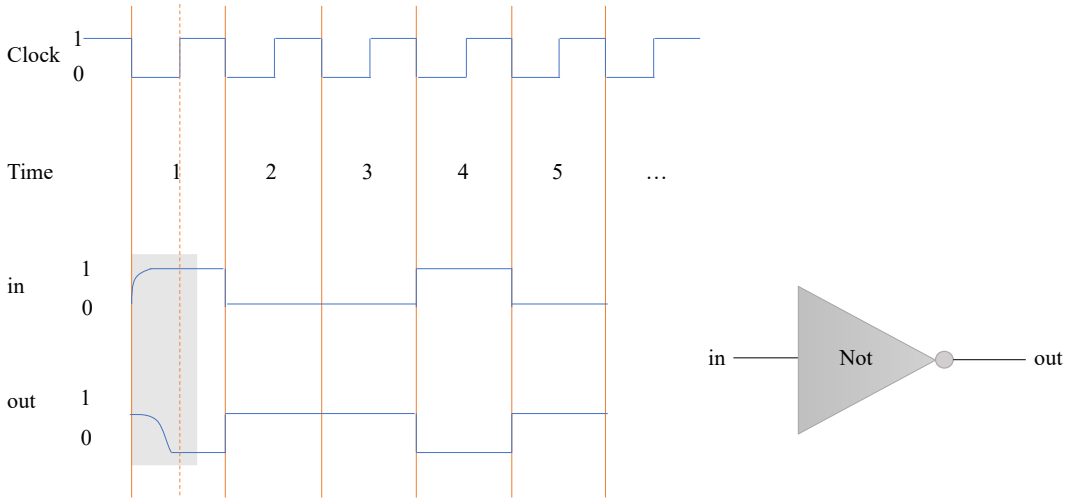
how to sweep under the rug the issue of delays?

When you really look at a physical signal that's really implemented somehow maybe, probably with some electrical signals. It doesn't change instantaneously in zero time between time, logical time one and logical time two, between logical time zero and logical time one. Really, the little course current builds slowly and the voltage may be that's how we present the actual bit. It changes slowly. And then, what we really see if we look at the actual analog signal inside our implementation of the gate, it's going to be some kind of way form that we see here in unit time unit one.

It takes time for the input to reach it's final stage.And then, it will also take some time for the output to reach the final stage. Probably, it will take more time than it takes for the input because there's an additional delay, the delay of the gate self:



the whole point of this logical we break time into digital, into integer units is the fact that they won't want to think about these delays. As long as our clock cycle is not too fast, as long as we take, give ourselves enough time between consecutive time units,we can ignore everything that happened at the beginning of the cycle. All the grey area here:



as long as by the end of the gray area, all the signals reach their true and final and consistent state we're all done. In fact, the way **we choose the cycle of the clock is to make sure that all the hardware there really stabilizes**.

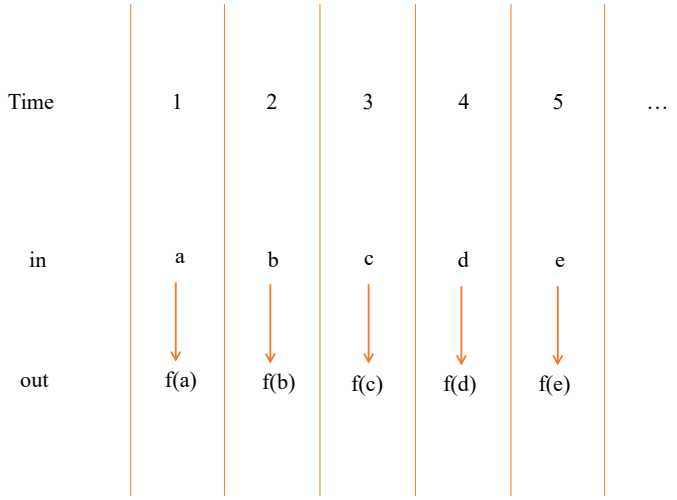
And, the implementations give you the logical operations by the end of the gray unit, and the clock need to be a little bit wider than that. So really, in reality, there is going to be all these gray areas where voltages changes and the system basically try's to stabilize into the new logical eh, situations.

And, what happens is at the end of the clock cycle is what we view as the real logical state of the system. And then, we can simply ignore these inconsistencies because now, whatever happened before the gray area, we don't need to worry about it because we know it's gone by the end of the clock cycle.So, that ends our, the way that we sweep under the rug the issue of delays. And, that really justifies the way that we can think about time in integer steps, one after another.

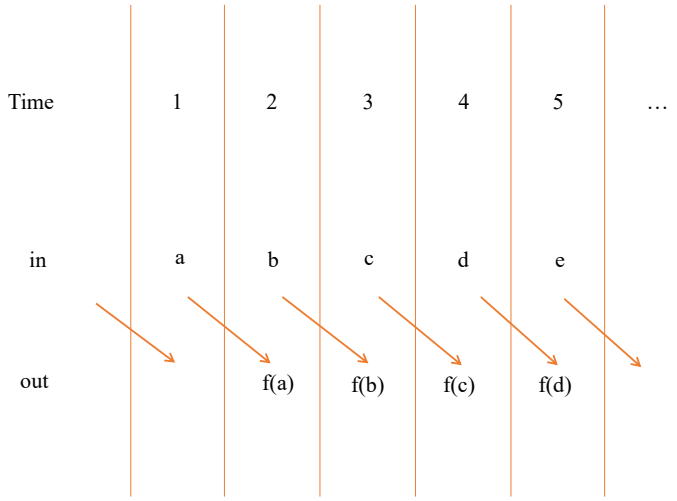
So now, we've reached the situation that we have integer time units and we know that something can happen at every different time unit.

Combinatorial Logic vs. Sequential Logic

- Combinatorial: `out[t] = function( in[t] )`

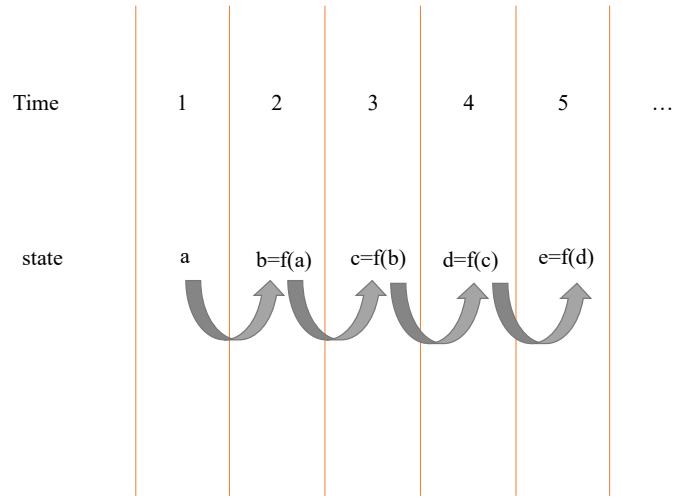


- Sequential: `out[t] = function( in[t-1] )`



We can actually have the input and output be the same but, bits. The same busses the same location than our hardware.

We and, we can now recall it to state. As long as now what's our, our value at time t depends on the previous value at time t minus 1, we can have these values live in the same wires in the circuit:

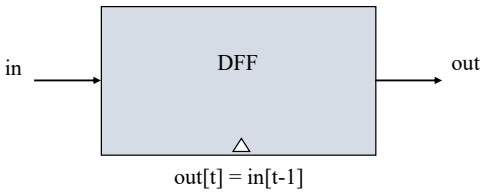


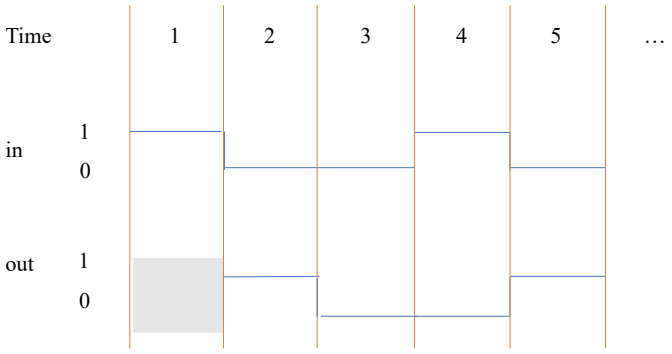
Unit 3.2: Flip Flops（触发器）

Remembering State

- Missing ingredient: remember one bit of information from time t-1 so it can be used at time t
- At the "end of time" t-1, such an ingredient can be at either of two states: "remembering o" or "remembering 1"
- This ingredient remembers by "flipping" between these possible states
- Gates that can flip between two states are called Flip-Flops

The "Clocked Data Flip Flop（数据触发器/DFF/D触发器）"



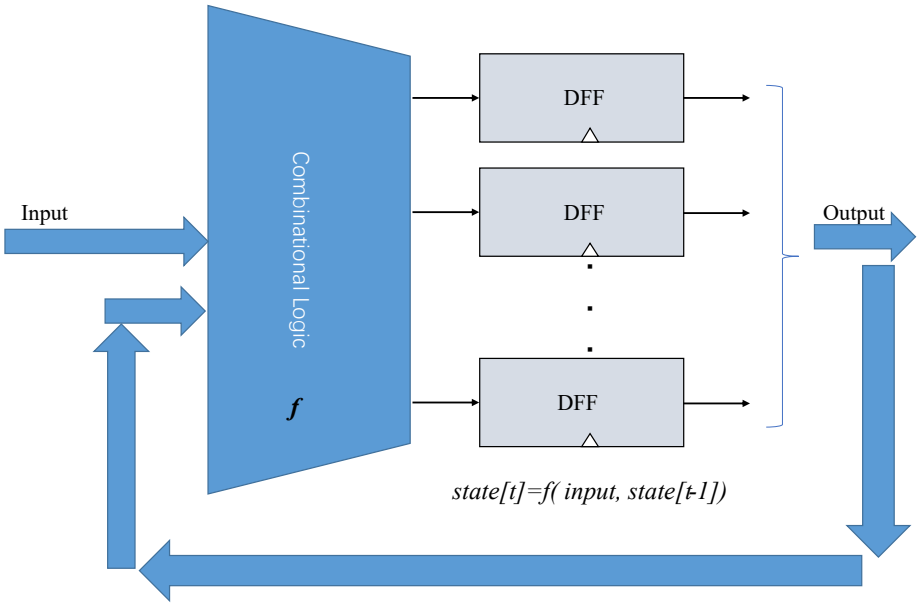


Implementation of the D Flip Flop

- In this course: it is a primitive

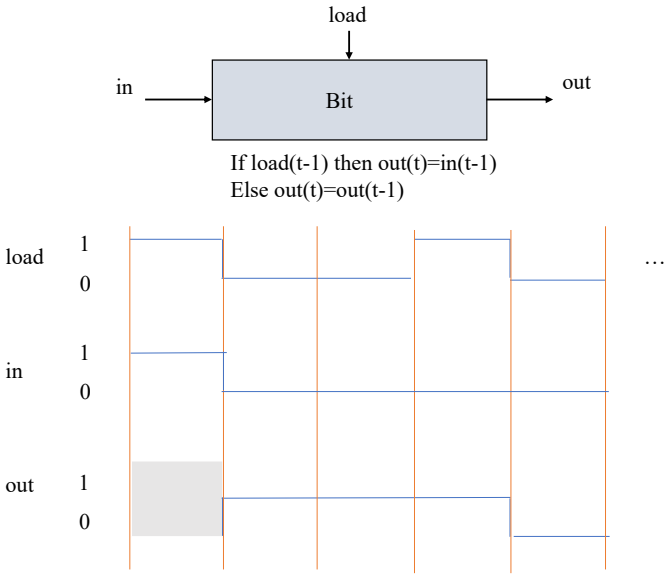
- In many physical implementation, it may be built from actual Nand gates:
  - Step1：create a "loop" achieving an "un-clocked" flip-flop
  - Step2：Isolation across time steps using a "master-slave" setup
- Very cute
  - But conceptually confusing
- Our Hardware Simulator forbids "combinatorial loops"
  - A cycle in the hardware connections is allowed only if it passes through a sequential gate

Sequential Logic Implementation

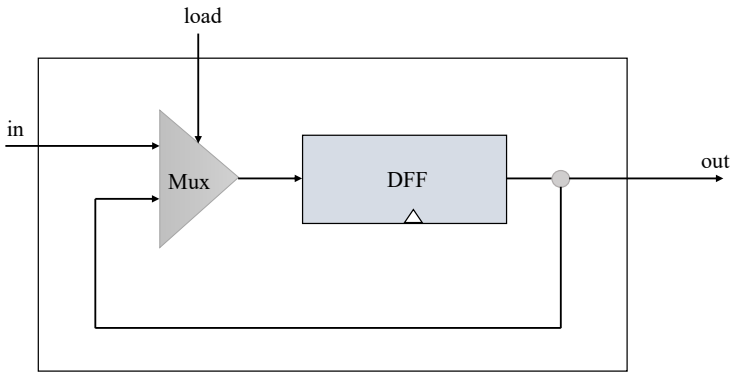


Remembering For Ever: 1-bit Register

- Goal: remember an input bit "forever":until requested to load a new value



Working Bit Implementation



Unit 3.3: Memory Units

when we say memory in computer hardware,we mean many different things:

First of all there is main memory, which consists of the memory that resides actually inside the computer and hard-wired into the computer's motherboard. And this main memory also divides into several different categories of memories. The most famous one is called RAM or the Random Access Memory.

And then there's secondary memory like hard-disks and memory sticks and so on.

And then there's also the distinction between volatile and non-volatile memory. For example, when you pull out the plug of the computer, the RAM is effectively erased. At the same time the information which is stored on the disks and on flash memory and so on, persists even when the computer is not connected to a power supply.

So, we have this distinction.The RAM is used to store both the data on which our programs operate as well as the instructions, which are the building blocks of the programs themselves.

And we will talk about this duality later on in the course when we talk about the overall computer architectureAnd finally, another comment that I want to make before we actually start this unit, is that like anything else when we talk about memory.

We can talk about it from a physical perspective, how we actually build the memory, what kind of the technology ,do we use in order to realize the memory. And we can also talk about the logical organization of the memory.

Now, in this unit, and in this course, in general, we always focus on logical considerations. And we focus in particular on the RAM unit, which is once again, the most important element of the computer's main memory.

Memory

---

memory:

- Main memory:RAM,...
- Secondary memory:disks,...

- Volatile/non-volatile

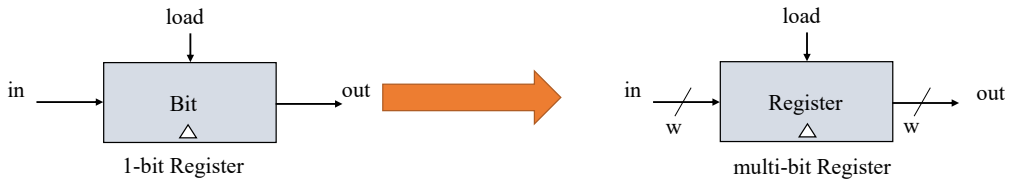
RAM:

- Data
- Instructions

Perspective

- Physical
- Logical

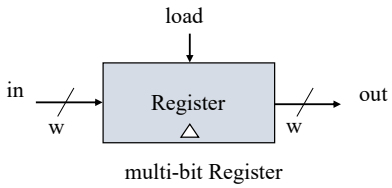
The most basic memory element : Register



- w(word width): 16-bit, 32-bit, 64-bit

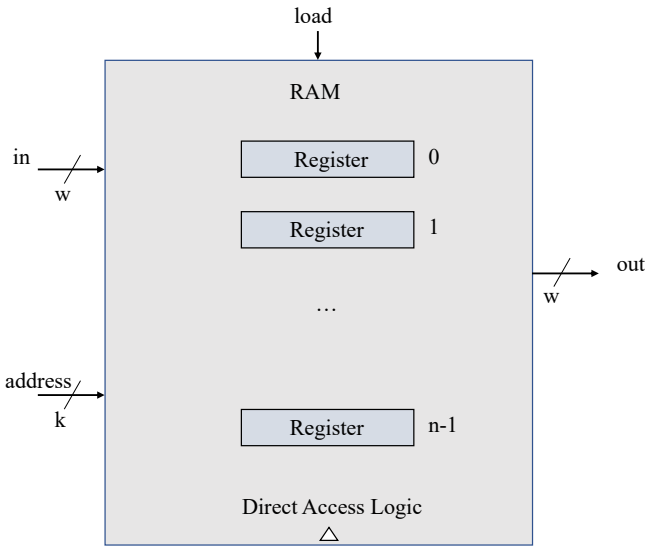
  (from now on we will talk about 16-bit registers,without loss if generality)
- Register's state: the value which is current stored inside the register

Register / read logic



- To read the Register
  - probe out
- Result
  - out emits the Register's state
- To set Register = v
  - set in=v ; set load=1
- Result
  - The Register's state becomes v
  - From the next cycle onward, out emits v

RAM unit



RAM abstraction:

A sequence of *n* addressable register,with addresses 0 to *n-1*

At any given point of time, only *one* register in the RAM is selected

k(width of address input):

$k = log_2 n$

w(word width):

No impact on the RAM logic

(Hack computer : w=16)

RAM is a sequence chip, with a clocked behavior

To read Register i :

set address= i

Result:

out emits the state of Register i

To set Register i to v :

set address = i

set in = v

set load = 1

Result:

- The state of Register i becomes v
- From the cycle onward, out emits v

A family of 16-bit RAM chips

---

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

- Why these particular RAM chips?
- Because that's what we need for the Hack computer
- Why "Random Access Memory"?
    - Because irrespective of the RAM size, every register can be accessed at the same time - instantaneously!

Unit 3.4: Counters

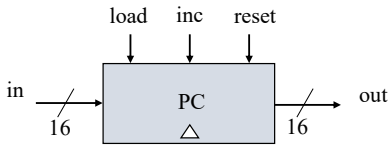
Where counters come play

- The computer must keep track of which instruction should be fetched and executed next
- This control mechanism can be realized by a Program Counter
- The PC contains the address of the instruction that will be fetched and executed next
- Three possible control settings:
  - Reset: fetch the first instruction `PC=0`
  - Next: fetch the next instruction `PC++`
  - Goto: fetch instruction n `PC=n`

Counter:

A chip that realizes the abstraction

Counter abstraction



Unit 3.5: Project3 Overview

Project 3

Given:

- All the chips built in Projects 1 and 2
- Data Flip-Flop(DFF gate)

Goal: Build the following chips:(A family of sequential chips,from a 1-bit register to a 16K memory unit)

- Bit
- Register
- RAM8

- RAM64
- RAM512
- RAM4K
- RAM16K
- PC

1-bit register

Implementation tip:

Can be built from a DFF and a multiplexor

16-bit register

Implementation tip:
Can be built from multiple 1-bit registers

8-Register RAM

Implementation tip:

- Feed the in value to all the register,simultaneously
- Use mux/demux chips to select the right register

RAM8,RAM64,...RAM16K

Implementation tip:

- A RAM device can be built by grouping smaller RAM-parts together
- Think about the RAM's address input as consisting of two field;
    - One field can be used to select a RAM-part;
    - The other field can be used to select a register within that RAM-part
- Use mux/demux logic to affect this address scheme

Q & A

is NAND gate the only basic technology for building memory systems today?

the truth is this is not the only way to construct flip flops. In many other cases, flip flops are constructed using some basic physical properties of the underlying solid state physics, that, of the devices that are used for this kind of storage. Now, this kind of details of such devices and how they remember anything is really physics or electrical engineering and not something that we touch in this course.

During this week, we built a memory device which you called RAM. Is this the only memory device that the computers use?

Well, the answer is definitely not. Computers use various kinds of memory devices of which the RAM is indeed the most important one. The RAM, which stands for random access memory stores both data and instructions. And, it is a volatile device, meaning that it depends on an external power supply. So, once you disconnect the computer from the power supply or turn it off, the contents of the RAM is effectively erased. So, in addition to the RAM unit, computers also typically

use another device called ROM, which stands for read only memory. So, the ROM is not only read-only, it is also a non-volatile device, unlike the RAM. Which means that it maintains its current contents a long time, over time and it does not depend on an external power supply. Which makes it very convenient because the ROM is where you want to put the programs that have to work when you turn on the computer. You know, this is what is known as the booting process. So, when you boot up the computer, the program, which is stored or pre stored in the ROM starts running. And, this program normally initializes all sorts of things, in the operating system and in the computer. Actually, not in the operating system, but in lower level code. And, the next thing that the ROM does is or the program that resides in the ROM does, it clones from the disc the startup code of the operating system. And then finally, we begin to see some windows on the screen and the computer sort of comes alive. Another technology which you've probably heard of is called flash memory. And, flash memory is is a technology which actually combines the good things of both the RAM and the ROM. On the one hand it's a read write memory. You can both read and modify its contents. And at the same time it does not depend on an external power supply like the ROM. So, once you turn off the computer. The contents of the flash memory remains intact. So, we talked about RAM, ROM, flash. Another kind of memory which which you normally encounter is called cache memory.

what is cache memory and why do we need it?

When one actually builds a computer, the memory is going to be a pretty costly part of the whole system. And as you can expect, there are many different technologies for building memories. And, the faster the memory, is the more expensive it usually is. The larger the memory, the more expensive it usually is. So, an architect, a computer architect is always faced with a trade off of that we want to put more money into the memory and make it larger and faster. Or, does we want to get a cheaper memory, and maybe put the money more in the processing unit? A usual tradeoff is to have a large, cheap memory, maybe slow also, and a very small, expensive, fast memory.And, try to make sure that what the, that does it is very often need, need used by the processor reside in the very fast memory, while the data that is only rarely used resides in the slow memory. because this way, you get the speed of a very fast memory, while you have the size of a very cheap memory.To do this correctly is a very intricate art, and today computers have whole hierarchies of caches. These are called caches, the small, fast memories called the cache. Hierarchies of such caches that are faster and faster and expensive, more expensive and more expensive and smaller and smaller, as they get closer to the processor. And, doing that correctly allows you to really get amazing eh, effective speed of a very fast memory, even though there is only a very small fast memory, and a very large slow memory.

Solution

- Bit.cmp

```
CHIP Bit {
    IN in, load;
    OUT out;

    PARTS:
    // Put your code here:
    Mux(a=w2, b=in, sel=load, out=w1);
    DFF(in=w1, out=out, out=w2);
}
```

- Register.hdl

```
CHIP Register {
    IN in[16], load;
    OUT out[16];

    PARTS:
    // Put your code here:
    Bit(in=in[0], load=load, out=out[0]);
    Bit(in=in[1], load=load, out=out[1]);
    Bit(in=in[2], load=load, out=out[2]);
    Bit(in=in[3], load=load, out=out[3]);
    Bit(in=in[4], load=load, out=out[4]);
    Bit(in=in[5], load=load, out=out[5]);
    Bit(in=in[6], load=load, out=out[6]);
    Bit(in=in[7], load=load, out=out[7]);
    Bit(in=in[8], load=load, out=out[8]);
    Bit(in=in[9], load=load, out=out[9]);
    Bit(in=in[10], load=load, out=out[10]);
    Bit(in=in[11], load=load, out=out[11]);
    Bit(in=in[12], load=load, out=out[12]);
    Bit(in=in[13], load=load, out=out[13]);
    Bit(in=in[14], load=load, out=out[14]);
    Bit(in=in[15], load=load, out=out[15]);
}
```

- RAM8.hdl

```
CHIP RAM8 {
    IN in[16], load, address[3];
    OUT out[16];

    PARTS:
    // Put your code here:
    DMux8Way(in=true, sel=address, a=a, b=b, c=c, d=d, e=e, f=f, g=g, h=h);
    And(a=a, b=load, out=load0);
    Register(in=in, load=load0, out=a1);
    And(a=b, b=load, out=load1);
    Register(in=in, load=load1, out=b1);
    And(a=c, b=load, out=load2);
    Register(in=in, load=load2, out=c1);
    And(a=d, b=load, out=load3);
    Register(in=in, load=load3, out=d1);
    And(a=e, b=load, out=load4);
    Register(in=in, load=load4, out=e1);
    And(a=f, b=load, out=load5);
    Register(in=in, load=load5, out=f1);
    And(a=g, b=load, out=load6);
    Register(in=in, load=load6, out=g1);
    And(a=h, b=load, out=load7);
    Register(in=in, load=load7, out=h1);
    Mux8Way16(a=a1, b=b1, c=c1, d=d1, e=e1, f=f1, g=g1, h=h1, sel=address,
out=out);
}
```

- RAM64.hdl

```
CHIP RAM64 {
    IN in[16], load, address[6];
    OUT out[16];

    PARTS:
    // Put your code here:
    DMux8Way(in=true, sel=address[3..5], a=a, b=b, c=c, d=d, e=e, f=f, g=g,
h=h);
    And(a=a, b=load, out=load0);
    RAM8(in=in, load=load0, address=address[0..2], out=a1);
    And(a=b, b=load, out=load1);
    RAM8(in=in, load=load1, address=address[0..2], out=b1);
    And(a=c, b=load, out=load2);
    RAM8(in=in, load=load2, address=address[0..2], out=c1);
    And(a=d, b=load, out=load3);
    RAM8(in=in, load=load3, address=address[0..2], out=d1);
    And(a=e, b=load, out=load4);
    RAM8(in=in, load=load4, address=address[0..2], out=e1);
    And(a=f, b=load, out=load5);
    RAM8(in=in, load=load5, address=address[0..2], out=f1);
    And(a=g, b=load, out=load6);
    RAM8(in=in, load=load6, address=address[0..2], out=g1);
    And(a=h, b=load, out=load7);
    RAM8(in=in, load=load7, address=address[0..2], out=h1);
    Mux8Way16(a=a1, b=b1, c=c1, d=d1, e=e1, f=f1, g=g1, h=h1, sel=address[3..5],
out=out);
}
```

- PC.hdl

```
CHIP PC {
    IN in[16],load,inc,reset;
    OUT out[16];

    PARTS:
    // Put your code here:
    Mux(a=inc, b=true, sel=load, out=out1);
    Mux(a=out1, b=true, sel=reset, out=out2);
    Inc16(in=w, out=wadd1);
    Mux16(a=wadd1, b=in, sel=load, out=out3);
    Mux16(a=out3, b=false, sel=reset, out=out4);
    Register(in=out4, load=out2, out=out, out=w);

}
```

- RAM512.hdl

```
CHIP RAM512 {
    IN in[16], load, address[9];
    OUT out[16];

    PARTS:
    // Put your code here:
```

```
    DMux8Way(in=true, sel=address[6..8], a=a, b=b, c=c, d=d, e=e, f=f, g=g,
h=h);
    And(a=a, b=load, out=load0);
    RAM64(in=in, load=load0, address=address[0..5], out=a1);
    And(a=b, b=load, out=load1);
    RAM64(in=in, load=load1, address=address[0..5], out=b1);
    And(a=c, b=load, out=load2);
    RAM64(in=in, load=load2, address=address[0..5], out=c1);
    And(a=d, b=load, out=load3);
    RAM64(in=in, load=load3, address=address[0..5], out=d1);
    And(a=e, b=load, out=load4);
    RAM64(in=in, load=load4, address=address[0..5], out=e1);
    And(a=f, b=load, out=load5);
    RAM64(in=in, load=load5, address=address[0..5], out=f1);
    And(a=g, b=load, out=load6);
    RAM64(in=in, load=load6, address=address[0..5], out=g1);
    And(a=h, b=load, out=load7);
    RAM64(in=in, load=load7, address=address[0..5], out=h1);
    Mux8Way16(a=a1, b=b1, c=c1, d=d1, e=e1, f=f1, g=g1, h=h1, sel=address[6..8],
out=out);
}
```

- RAM4K.hdl

```
CHIP RAM4K {
    IN in[16], load, address[12];
    OUT out[16];

    PARTS:
    // Put your code here:
    DMux8Way(in=true, sel=address[9..11], a=a, b=b, c=c, d=d, e=e, f=f, g=g,
h=h);
    And(a=a, b=load, out=load0);
    RAM512(in=in, load=load0, address=address[0..8], out=a1);
    And(a=b, b=load, out=load1);
    RAM512(in=in, load=load1, address=address[0..8], out=b1);
    And(a=c, b=load, out=load2);
    RAM512(in=in, load=load2, address=address[0..8], out=c1);
    And(a=d, b=load, out=load3);
    RAM512(in=in, load=load3, address=address[0..8], out=d1);
    And(a=e, b=load, out=load4);
    RAM512(in=in, load=load4, address=address[0..8], out=e1);
    And(a=f, b=load, out=load5);
    RAM512(in=in, load=load5, address=address[0..8], out=f1);
    And(a=g, b=load, out=load6);
    RAM512(in=in, load=load6, address=address[0..8], out=g1);
    And(a=h, b=load, out=load7);
    RAM512(in=in, load=load7, address=address[0..8], out=h1);
    Mux8Way16(a=a1, b=b1, c=c1, d=d1, e=e1, f=f1, g=g1, h=h1, sel=address[9..11],
out=out);
}
```

- RAM16K.hdl

```

```

```
CHIP RAM16K {
    IN in[16], load, address[14];
    OUT out[16];

    PARTS:
    // Put your code here:
    DMux4Way(in=true, sel=address[12..13], a=a, b=b, c=c, d=d);
    And(a=a, b=load, out=load0);
    RAM4K(in=in, load=load0, address=address[0..11], out=a1);
    And(a=b, b=load, out=load1);
    RAM4K(in=in, load=load1, address=address[0..11], out=b1);
    And(a=c, b=load, out=load2);
    RAM4K(in=in, load=load2, address=address[0..11], out=c1);
    And(a=d, b=load, out=load3);
    RAM4K(in=in, load=load3, address=address[0..11], out=d1);
    Mux4Way16(a=a1, b=b1, c=c1, d=d1, sel=address[12..13], out=out);
}
```

Machine Language

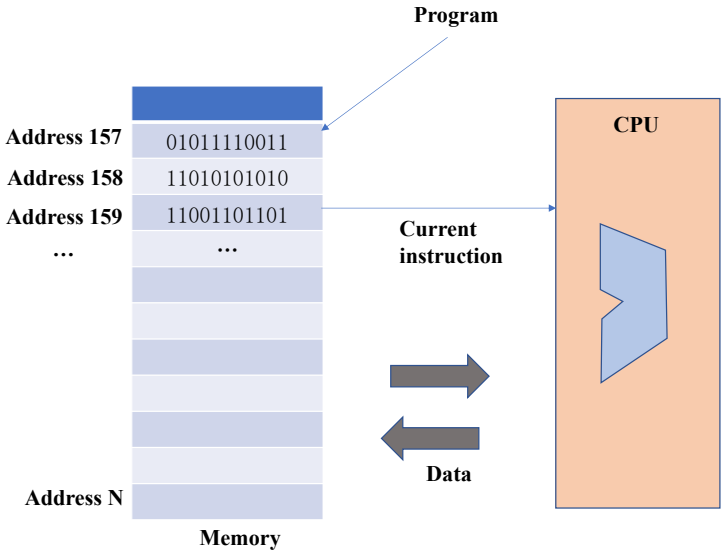
A critically important aspect of building a new computer system is designing the low-level *machine language*, or *instruction set*, with which the computer can be instructed to do various things. As it turns out, this can be done before the computer itself is actually built. For example, we can write a Java program that emulates the yet-to-be-built computer, and then use it to emulate the execution of programs written in the new machine language. Such experiments can give us a good appreciation of the bare bone "look and feel" of the new computer, and lead to decisions that may well change and improve both the hardware and the language designs. Taking a similar approach, in this module we assume that the Hack computer and machine language have been built, and write some low-level programs using the Hack machine language. We will then use a supplied CPU Emulator (a computer program) to test and execute our programs. This experience will give you a taste of low-level programming, as well as a solid hands-on overview of the Hack computer platform.

Unit 4.1: Machine Languages: Overview

Universality

- Same **Hardware** can run many different **Software** programs
    - Theory: Universal Turing Machine
    - Practice: von Neumann Architecture

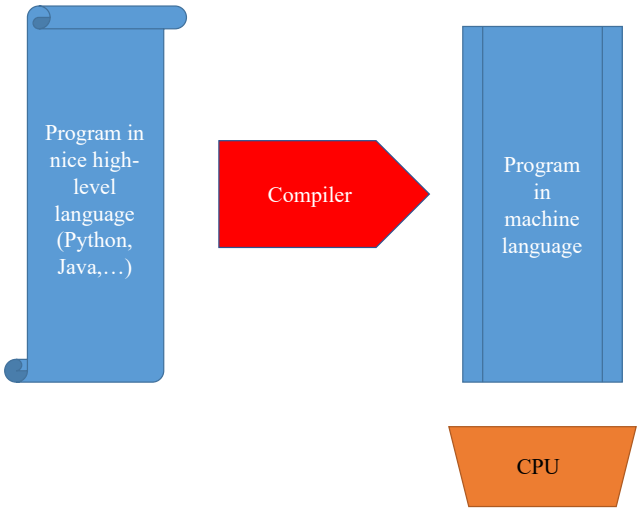
Machine Languages



Three elements of machine languages

- **Operations:** how are we going to specify the instructions?
- **Program Counter:** how do we know which instruction to perform at any given stage and time?
- **Addressing:** we're going to have to tell the hardware what to operate on

Compilation



Mnemonics

Instruction: `0100010|0011|0010` -> `ADD R3 R2`

Interpretation1: The "symbolic form" doesn't really exists but is just a convenient mnemonics to present machine language instructions to humans.

Interpretation2: We will allow humans to write machine language instructions using this "assembly language" and will have an "Assembler" program convert it to the bit-form.

Symbols

Machine Language: `1010000110000001`

Assembly Language: `ADD 1,Mem[129]`

(location 129 in memory holds the "index")

ADD 1,**index(A "Symbolic Assembler(符号汇编器)" can translate "index"->Mem[129])**

Unit 4.2: Machine Languages: Elements

Machine Language

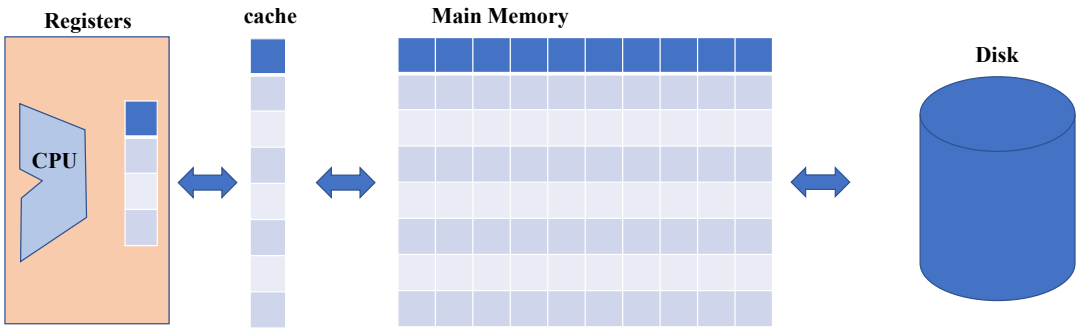
- Specification of the Hardware/Software Interface
  - What are the supported operations?
  - What do they operate on?
  - How is the program controlled?
- Usually in close correspondence to actual Hardware Architecture
  - Not necessarily so
- Cost-Performance（性价比）Tradeoff
  - Silicon Area（硅面积）
  - Time to Complete Instruction

Machine Operations

- Usually correspond to what's implemented in Hardware
  - Arithmetic Operations: add,substract,...
  - Logical Operations: and,or,...
  - Flow Control: "goto instruction X","if C then goto instruction Y"
- Differences between machine languages
  - Richness of set of operations(divisions?bulk copy?...)
  - Data types(width,floating point)

Memory Hierarchy

- Accessing a memory location is expensive
  - Need to supply a long address
  - Getting the memory contents into the CPU take time
- Solution: Memory Hierarchy



Registers

- CPUs usually contain a few,easily accessed, "registers"
- Their number and functions are a central part of the architecture
- Data Registers
  - Add R1,R2
- Address Registers
  - Store R1,@A

Once we have these registers, now we can think about, go back to the original question.

How do we decide which data to work upon? How do we tell the computer for us, an operation, let's say a simple add operation, what is it supposed to operate upon?

And there are a bunch of different possibilities. Here are base, here are four possibilities.

These are sometimes called **addressing modes（寻址方式）**, and there, some computers have other possibilities as well

Addressing Modes

- Register
  - Add R1, R2 // R2 <- R2+R1
- Direct
  - Add R1, M[200] // Mem[200] <- Mem[200]+R1
- Indirect
  - Add R1,@A // Mem[A] <- Mem[A]+R1
- Immediate
  - Add 73,R1 // R1 <- R1+73

Input/Output

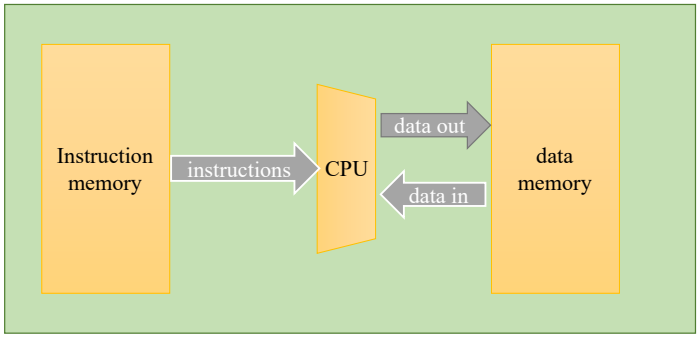
- Many types of Input and Output Devices
  - Keyboard,mouse,camera,sensors,printers,screen,sound...
- CPU needs some kind of protocol to talk to each of them
  - Software "Drivers" know these protocols
- One general method of interaction uses "memory mapping"
  - Memory Location 12345 holds the direction of the last movement of the mouse
  - Memory Location 45678 is not a real memory location but a way to tell the printer which paper to use

Flow Control

- Ususally the CPU executes machine instructions in sequence
- Sometimes we need to "jump" unconditionally to another location,e.g. so we can loop
- Sometimes we need to jump only if some condition is met

Unit 4.3: The Hack Computer and Machine Language

Hack computer: hardware



A 16-bit machine consisting of

- Data memory(RAM): a sequence of 16-bit registers: `RAM[0],RAM[1],RAM[2],...`
- Instruction memory(ROM): a sequence of 16-bit registers: `RAM[0],RAM[1],RAM[2],...`
- Central Processing Unit(CPU): performs 16-bit instructions
- Instruction bus/data bus/address bus

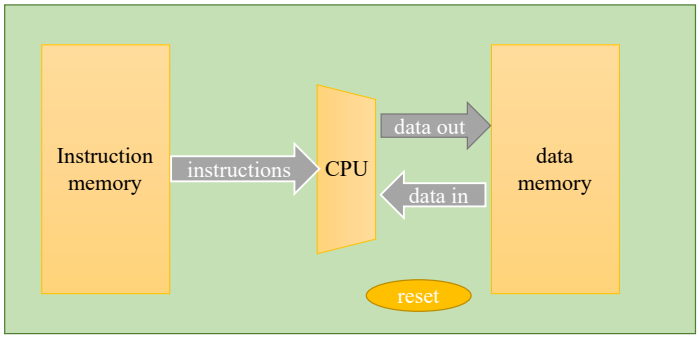
Hack computer: software

Hack machine languages:

- 16-bit A-instructions
- 16-bit C-instructions

Hack program = sequence of instructions written in the Hack machine language

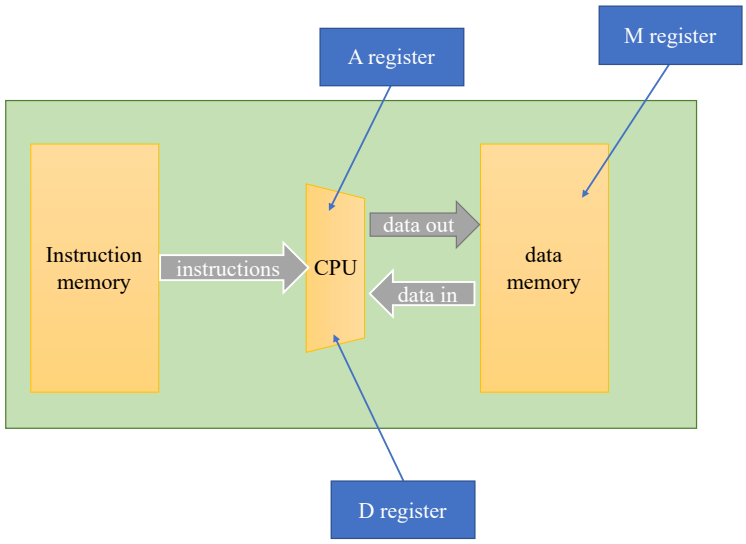
Hack computer: control



Control:

- The ROM is loaded with a Hack program
- The reset button is pushed
- The program starts running

Hack computer: registers



The Hack machine languges recognizes three registers:

- D holds a 16-bit value
- A holds a 16-bit value
- M represents the 16-bit RAM register addressed by A

The A-instruction

Syntax: `@value`

Where *value* is either:

- a non-negative decimal constant or
- a symbol referring to such a constant(later)

Semantics:

- Sets the A register to *value*
- Side effect: RAM[A] becomes the selected RAM register

Example: `@21`

Effect:

- Sets the A register to 21
- RAM[21] becomes the selected RAM register

Usage example:

```
// set RAM[100] to -1
@100    // A=100
M=-1    // RAM[100]=-1
```

The C-instruction

---

destination, computation and a jump directive

`dest = comp ; jump` (both *dest* and *jump* are optional)

how it works:

First of all, we compute something and then we can do one of two things. We can either store the result of the computation in some destination or we can use this computation to decide if we want to jump to some other instruction in the program. Now, this is the basic overall semantics of the C-instruction.

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 = `null, M, D, MD, A, AM, AD, AMD` M refers to **RAM[A]**

jump = `null, JGT, JEQ, JGE, JLT, JNE, JLE, JMP` if(*comp jump 0*) jump to execute the instruction in **ROM[A]**

Semantics:

- Compute the value of comp
- Stores the result in dest;

- If the Boolen expression (*comp jump 0*) is true, jumps to execute the the instruction stored in ROM[A]

Example:

```
// Set the D register to -1
D=-1
```

```
// Set RAM[300] to the value of the D register minus 1
@300   // A=300
M=D-1  // RAM[300]=D-1
```

```
// If (D-1==0) jump to execute the instruction stored in ROM[56]
@56    //A=56
D-1;JEQ    //if (D-1==0) goto 56
```

Unit 4.4: Hack Language Specification

The Hack machine language

Two ways to express the same semantics:

- Binary code
- Symbolic language

Symbolic:

```
@17
D+1;JLE
```

translate->

Binary:

```
0000000000010001
1110011111000110
```

execute

The A-instruction: symbolic and binary syntax

Semantics: Set the A register to value

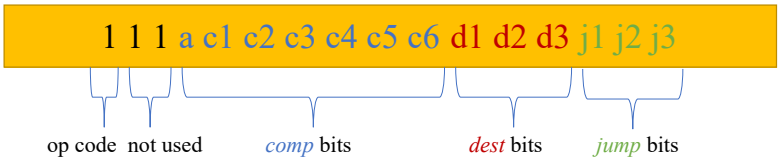
- Symbolic syntax:
  - `@value`
  - Where value is either
    - a non-negative decimal constant <= 32767 ($2^{15} - 1$) or
    - a symbol referring to such a constant (later)
  - example: `@21` Effect: sets the A register to 21
- Binary syntax:

- `0value`
- Where value is a 15-bit binary number
- example: `0000000000010101` Effect: sets the A register to 21

The C-instruction: symbolic and binary syntax

Symbolic syntax: `dest=comp;jump`

Binary syntax:



comp	comp	c1	c2	c3	c4	c5	c6
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	M	1	1	0	0	0	0
!D		0	0	1	1	0	1
!A	!M	1	1	0	0	0	1
-D		0	0	1	1	1	1
-A	-M	1	1	0	0	1	1
D+1		0	1	1	1	1	1
A+1	M+1	1	1	0	1	1	1
D-1		0	0	1	1	1	0
A-1	M-1	1	1	0	0	1	0

---

comp	comp	c1	c2	c3	c4	c5	c6
D+A	D+M	0	0	0	0	1	0
D-A	D-M	0	1	0	0	1	1
A-D	M-D	0	0	0	1	1	1
D&A	D&M	0	0	0	0	0	0
D|A	D|M	0	1	0	1	0	1
a=0	a=1						

dest	d1	d2	d3	effect: the value is stored in:
null	0	0	0	The value is not stored
M	0	0	1	RAM[A]
D	0	1	0	D register
MD	0	1	1	RAM[A] and D register
A	1	0	0	A register
AM	1	0	1	A register and RAM[A]
AD	1	1	0	A register and D register
AMD	1	1	1	A register, RAM[A], and D register

jump	j1	j2	j3	effect
null	0	0	0	no jump
JGT	0	0	1	if out>0 jump
JEQ	0	1	0	if out=0 jump
JGE	0	1	1	if out≥0 jump
JLT	1	0	0	if out<0 jump
JNE	1	0	1	if out≠0 jump
JLE	1	1	0	if out≤0 jump
JMP	1	1	1	unconditional jump

Hack program

```
//compute RAM[1]=1+...+RAM[0]
```

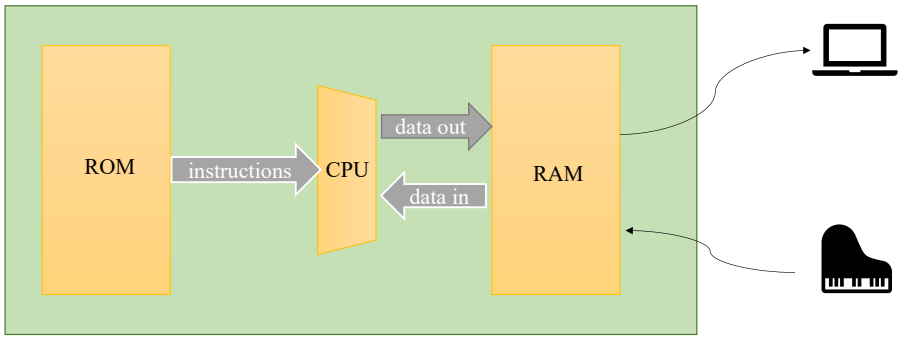
```
//usage: put a number in RAM[0]
@16 //RAM[16] represents i
M=1 //i=1
@17 //RAM[17] represents sum
M=0 //sum=01

@16
D=M
@0
D=D-M
@17 // if i>RAM[0] goto 17
D;JGT

@16
D=M
@17
D=D+M   // sum += i
@16
M=M+1   //i++
@4      //goto 4 (loop)
0;JMP

@17
D=M
@1
M=D     // RAM[1] = sum
@21     // program's end
0;JMP   // infinite loop
```

observations:

- A Hack program is a sequence of Hack instructions
- White space is permitted
- Comments are used to explain code
- There are better ways to write symbolic Hack programs:stay tuned

Unit 4.5: Input/Output

Hack computer platform



Peripheral I/O devices:

- Keyboard: used to enter inputs
- Screen: used to dispaly outputs

I/O devices are used for:

- Getting data from users
- Displaying data to users

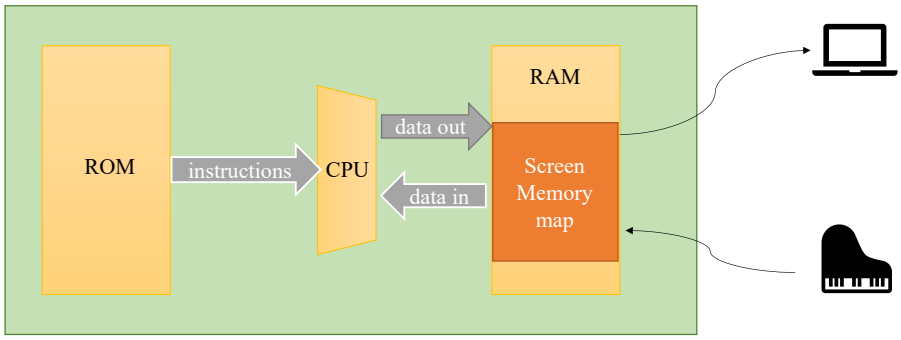
High-level approach:

Sophisticated software libraries enabling text, graphics, animation, audio, video, etc.

Low-level approach:

Bits

Hack computer platform: Output



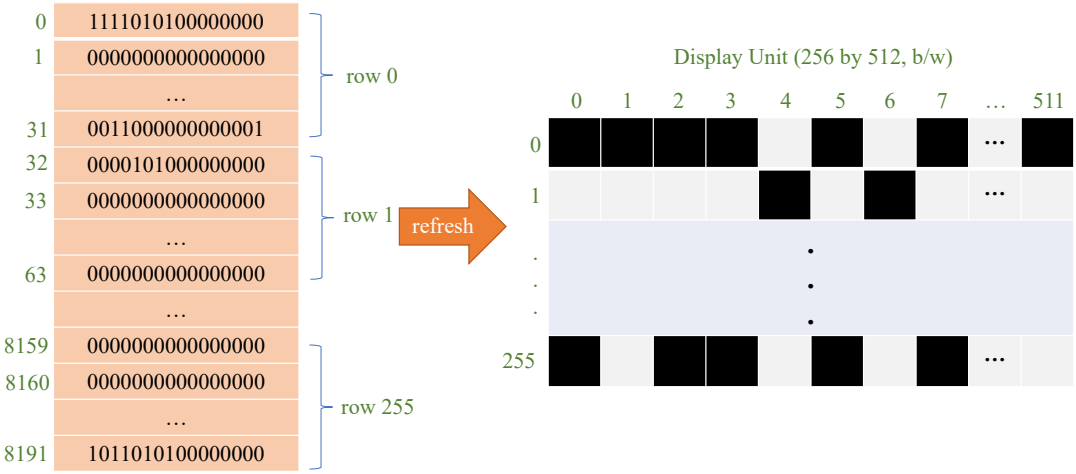
Screen memory map

A designated memory area, dedicated to manage a display unit

The physical display is continuously *refreshed* from the memory map, many times per second

Output is effected by writing code that manipulates the screen memory map

Screen memory map



To set pixel (*row*,*col*) on/off:

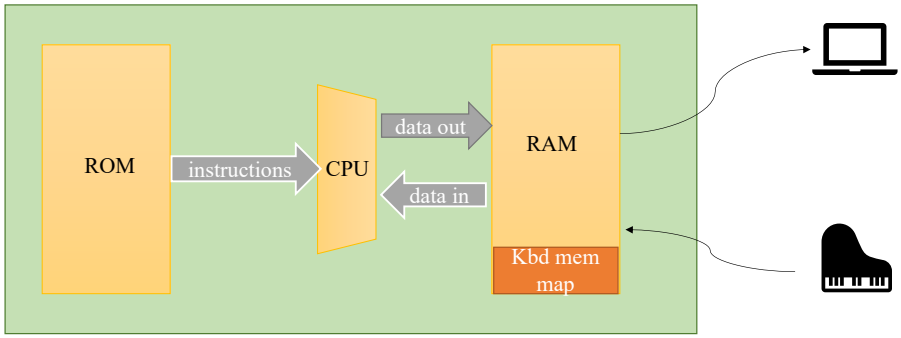
1. word=Screen[32*row+col/16]

   word=RAM[16384+32*row+col/16]
2. Set the (col % 16)th bit of word to 0 or 1
3. Commit word to the RAM

(1) and (3) are done using 16-bit RAM access operations

Hack computer platform: Input

The physical keyboard is associated with a keyboard memory map.



The Hack character set

When no key is pressed, the resulting code is 0

Key	Code
0	48
1	49
...	...
9	57

Key	Code
A	65
B	66
...	...
Z	90

Key	Code
(space)	32
!	33
“	34

Key	Code
#	35
$	36
%	37
&	38
‘	39
(	40
)	41
*	42
+	43
,	44
-	45
.	46
/	47

Key	Code
:	58
;	59
<	60
=	61
>	62
?	63
@	64

Key	Code
[	91
/	92
]	93
^	94
_	95

Key	Code

Key	Code
newline	128
backspace	129
left arrow	130
up arrow	131
right arrow	132
down arrow	133
home	134
end	135
Page up	136
Page down	137
insert	138
delete	139
esc	140
f1	141
...	...
f12	152

Keyboard memory map

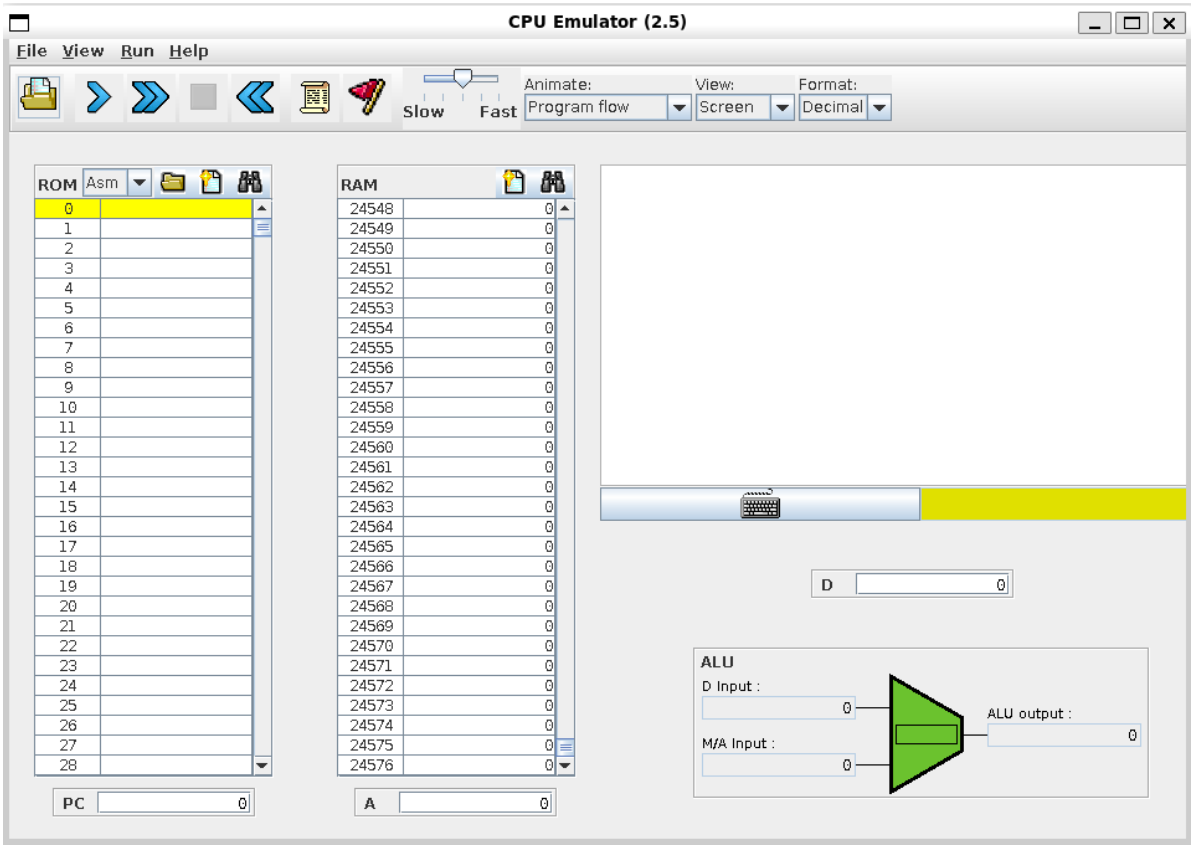
To check which key is currently pressed:

- Probe the contents of the Keyboard chip
- In the Hack computer: probe the contents of RAM[24576]

If the register contains 0, no key is pressed

Unit 4.6: Hack Programming, Part1

CPU Emulator



Hack programming

- part I
  - Working with registers and memory
- part II
  - Branching
  - Variables
  - Iteration
- unit 4.8
  - Pointers
  - Input/Output

Working with register and memory

D: data register

A: address/ data register

M: the currently selected memory register, M=RAM[A]

Typical operations:

```
// D=10
@10
D=A
```

```
// D++
D=D+1

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

//RAM[17]=0
@17
M=0

//RAM[17]=0
@10
D=A
@17
M=D

// RAM[5]=RAM[3]
@3
D=M
@5
M=D
```

Hack program example: add two numbers

Hack assembly code

```
// Program: Add2.asm
// Computes: RAM[2] = RAM[0]+RAM[1]
// Usage: put values in RAM[0],RAM[1]

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

@1
D=D+M // D=D+RAM[1]

@2
M=D // RAM[2]=D

@6
0;JMP
```

Best practice:

To terminate a program safely, end it with an infinite loop

Built-in symbols

The Hack assembly language features *built-in symbols:*

symbol	value

---

symbol	value
R0	0
R1	1
R2	2
...	...
R15	15
SCREEN	16384
KBD	24576
SP	0
LCL	1
ARG	2
THIS	3
THAT	4

- R0, R1,...,R15 : "virtual registers"
- SCREEN and KBD : base address of I/O memory maps
- Remaining symbols: used in the implementation of the Hack *virtual machine,*disscussed in *Nand to Teris,Part II*

whenever you want to address one of the sixteen registers in the memory, use the label convention

attention:Hack is case-sensitive! R5 and r5 are different symbols

These symbols can be used to denote "virtual registers":

instead of:

```
// RAM[5]=15
@15
D=A

@5
M=D
```

better style:

```
// RAM[5]=15
@15
D=A

@R5
M=D
```

Unit 4.7: Hack Programming, Part2

Branching

Example:

```
// Program: Signum.asm
// Computes:    if R0>0
//                 R1=1
//              else
//                 R1=0
    @R0
    D=M     // D=RAM[0]

    @8
    D;JGT   // If R0>0 goto8

    @R1
    M=0     // RAM[1]=0
    @10
    0;JMP   // end of profram

    @R1
    M=1     // R1=1

    @10
    0;JMP
```

> Instead of imagining that our mian task as programmers is to instruct a computer what to do, let us concentrate rather on expalining to human beings what we want a computer to do.
>
> --Donald Knuth

```
// Program: Signum.asm
// Computes:    if R0>0
//                 R1=1
//              else
//                 R1=0
    @R0
    D=M     // D=RAM[0]

    @POSITIVE
    D;JGT   // If R0>0 goto8

    @R1
    M=0     // RAM[1]=0
    @END
    0;JMP   // end of profram

(POSITIVE)
    @R1
    M=1     // R1=1
(END)
```

```
    @10
    0;JMP
```

@LABEL translates to @n, where n is the instruction number following the (LABEL) declaration

↓Contract

- Label declarations are not translated
- Each reference to a label is replaced with a reference to the instruction number following that label's declaration

Memory(ROM)

```
@0
D=M
@8
D;JGT
@1
M=0
@10
0;JMP
@1
M=1
@10
0;JMP
```

Variables

Variable usage example:

```
// Program: Flip.asm
// flips the values of
//  RAM[0] and RAM[1]

// temp = R1
// R1=R0
// R0=temp
    @R1
    D=M
    @temp
    M=D     // temp=R1

    @R0
    D=M
    @R1
    M=D     // R1=R0

    @temp
    D=M
    @R0
    M=D     // R0=temp
(END)
    @END
    0;JMP
```

@temp: find some available memory register (say register n), and use it to represent the variable temp.So,from now on, each occurance of @temp in the program will be translated into @n

↓Contract

- A reference to a symbol that has no corresponding label declaration is treated as a reference to a variable
- Variables are allocated to the RAM from address 16 onward

Memory(ROM)

```
@1
D=M
@16
M=D
@0
D=M
@1
M=D
@16
D=M
@0
M=D
@12
0;JMP
```

Iterative processing

Example:

Compute $1+2+...+n$

Preudo code:

```
// computes RAM[1] = 1+2+...+RAM[0]

    n = R0
    i = 1
    sim = 0
LOOP:
    if i > n goto STOP
    sum = sum + i
    i= i + 1
    goto LOOP
STOP:
    R1 = sum
```

Hack assembly code

```
// Program: Sum1toN.asm
// Computes RAM[1] = 1+2+...+n
// Usage:put a number(n) in RAM[0]
    @R0
    D=M
    @n
    M=D //n=R0
```

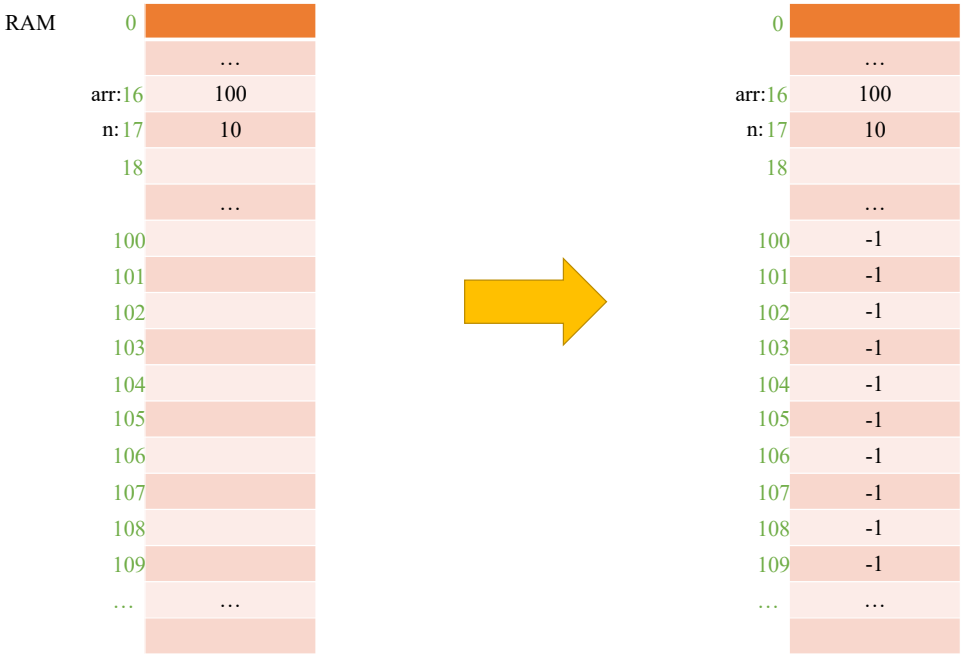
```
    @i
    M=1 //i=1
    @sum
    M=0 //sum=0
(LOOP)
    @i
    D=M
    @n
    D=D-M
    @STOP
    D;JGT   //if i>n goto STOP

    @sum
    D=M
    @i
    D=D+M
    @sum
    M=D     //sum=sum+i
    @i
    M=M+1   //i=i+1
    @LOOP
    0;JMP
(STOP)
    @sum
    D=M
    @R1
    M=D     //RAM[1]=sum
(END)
    @END
    0;JMP
```

Best practice:

1. **Design** the program using pseudo code
2. **Write** the program in assembly language
3. **Test** the program (on paper) using a variable-value trace table

Unit 4.8: Hack Programming, Part3

Pointers



Example:

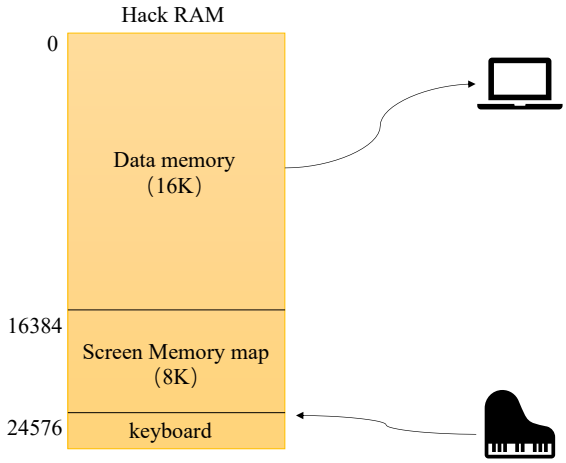
```
// for(i=0;i<n;i++){
//   arr[i]=-1
// }
    // Suppose that arr=100 and n=10

    // arr=100
    @100
    D=A
    @arr
    M=D

    // n=10
    @10
    D=A
    @n
    M=D

    //initialize i=0
    @i
    M=0
(LOOP)
    //if (i==n) goto END
    @i
    D=M
    @n
    D=D-M
    @END
    D;JEQ

    //RAM[arr+i]=-1
    @arr
    D=M
    @i
```

```
    A=D+M
    M=-1

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

    @LOOP
    0;JMP

(END)
    @END
    0;JMP
```

- Variables that store memory address, like arr and i, are called *pointers*
- Hack pointer logic : whenever we have to access memory using a pointer,we need an instruction like A=M
- Typical pointer semantics:

  "set the address register to the contents of some memory register"

Input/output



Hack language convention:

- SCREEN: base address of the screen memory map
- KBD: address of the keyboard memory map

I/O programming example: drawing a rectangle

The task: draw a filled rectangle at the upper left corner of the screen, 16 pixels wide and RAM[0] pixels long

pseudo code

```
// for(i=0;i<n;i++){
```

```
//      draw 16 black pixels at the beginning of row i
//}
addr = SCREEN
n = RAM[0]
i = 0

LOOP:
   if i > n goto END
   RAM[addr] = -1 //1111111111111111
   // advances to next row
   addr = addr + 32
   i = i + 1
   goto LOOP

END:
   goto END
```

assembly code

```
// Program: Rectangle.asm
// Draws a filled rectangle at the
// screen's top left corner, with
// width of 16 pixels and height of
// RAM[0] pixels
// Usage: put a non-negative number
// (rectangle's height) in RAM[0]

    @SCREEN
    D=A
    @addr
    M=D //addr=16384
        // (screen's base address)
    @0
    D=M
    @n
    M=D // n=RAM[0]

    @i
    M=0 // i=0
(LOOP)
    @i
    D=M
    @n
    D=D-M
    @END
    D;JGT   // if i>n goto END

    @addr
    A=M
    M=-1    //RAM[addr]=1111111111111111

    @i
    M=M+1   //i=i+1
    @32
    D=A
```

```
    @addr
    M=D+M   //addr=addr+32
    @LOOP
    0;JMP   //goto LOOP
(END)
    @END //program's end
    0;JMP   //infinite loop
```

```
@0
D=M
@16
M=D
@17
M=0
@16384
D=A
@18
M=D
@17
D=M
@16
D=D-M
@27
D;JGT
@18
A=M
M=-1
@17
M=M+1
@32
D=A
@18
M=D+M
@10
0;JMP
@27
0;JMP
```

Handling the keyboard

To check which key is currently pressed:

- Read the content of RAM[24576] (address KBD)
- If the register contains 0,no key is pressed
- Otherwise, the register contains the scan code of the currently pressed key

Low level programming is ...

- Low level
- Profound
- Subtle

- Intellectually challenging

Unit 4.9: Project4 Overview

In a Nutshell

Project objectives

Have a taste of:

- low-level programming
- Hack assembly language
- Hack hardware

Tasks

- Write a simple algebraic program
  - Mult: a program performing R2 = R0 * R1
- Write a simple interactive program
  - Fill: a simple interactive program

Fill: a program that does some basic I/O effects

Implementation strategy

- Listen to the keyboard
- To blacken / clear the screen, write code that fills the entire screen memory map with either "white" or black pixels
- Addressing the memory requires working with pointers

Testing

- Select "no animation"
- Mannual testing

Program development process

---

1. write/edit the program using a text editor
2. load the program into the CPU Emulator, and run it
3. Find and fix the errors

Best practice

Well-written low-level code is

- Short
- Efficient
- Elegant
- Self-describing

Technical tips

- Use symbolic variables and lables
- Use sensible variable and label names

- Variables: low-case
- Lables: upper-case
- Use indentation（缩进）
- Start with pseudo code

Q&A

how does the Hack machine language differ from the machine languages of typical computers like personal computers?

Well the Hack machine language is very simple because it is designed to operate on top of a very simple hardware platform. And we built this computer in purpose as a very simple architecture. Because we want to, to be able to actually build it in, you know, in the a space of a six weeks course and therefore the computer is very simple, but it sufficiently powerful to to offer almost everything that you need for reasons that I will explain in just a minute.

So, typical machine languages are much more rich than the Hack instruction set. They offer more commands more instruction types more data types, like floating point and more operations, like for example, multiplication and division. And yet, as I said before, there's no need to worry because all these fancy things that other languages offer can be delivered at the software level at a higher level of obstruction and that's exactly what we'll do in the second part of this course. In Nand2Tetris part two, we build an operating system, and we introduce all sorts of operations that are currently not supported by the machine language.

It looks like in the Hack language, whenever you want to do something meaningful, you need two machine language instructions. One to address the memory, and one to operate on the selected memory register. Is this the standard way to do things?

Well, most machine languages are more sophisticated, more powerful than ours and usually do allow you to specify both operator and operand in a single machine language command. For example, you can say I want to add and I want to use this memory location and put if all on a single machine language instruction. Now in our computer, in the Hack computer we only have 16-bit instructions and in 16 bits it's very difficult to put both both a lot of information about the operation and a lot of information about the operand. In particular, you can't put a whole memory address as well as more information about what kind of operation you want to do. So in our case, you have to split the point where you specify the address, which requires 15 bits of its own, and the place where you actually specify the operation, which requires a few more bits.

In other machine languages, some, usually, they have wider machine instructions, or sometimes even variable with machine instruction, and then they could fit more information into a single machine instruction. That said, the basic idea of taking part of the address on which the machine operation is supposed to work on, taking part of that address from a previous command is not an unusual kind of thing, and many machine languages do have that.

the Hack machine language indeed has some strange or peculiar syntax rules. Can you say a few words about the the choice of this syntax and how it differs from the syntax of regular machine languages?

Well, unlike high-level languages like Java machine languages are not designed to make people happy. They are designed to operate on harder platforms. So the commands of machine language must deal directly with the ALU, with the memory, and the registers. And they must be extremely efficient. And that's one reason why machine languages are so simple, as we discussed before. So as we said before the Hack machine language is is very simple. And and therefore, we also decided to make the syntax somewhat simpler than what you normally find in in machine languages.

For example, if you want to compute addition and store the result in some register, in the Hack language, you say something like D = D + M. Whereas in a normal machine language typically commands like this start with the, the specification of the operator, normally you will have something like, ADD along with some address： ADD D,M. And you can specify both the address and the register of the destination and the operator in one instruction, because normally, in machine languages, you have 32 bits or 64 bits. So there's enough space to pack all this information into a single command. Whereas in Hack, we do only this. Likewise, if you want to put the contents of one register in another register, in Hack you say something like D = M, for example. Whereas in a normal machine language typically you would say something like LOAD D, M or something like this. And once again, instead of M you can write an address and pack all this information into a single instruction.

do people actually have to go through the trouble of writing programs in machine language?

Well, the answer is that people don't really write programs in machine language. Instead some developers sometimes write programs that generate machine code, and these programs are called compilers. And a compiler is a program that takes another program written in some high-level language like Java and translates this program into binary code. So in order to write a compiler, obviously you have to understand the machine language that the compiler has to generate. And that's exactly what we are going to do in the second part of the NAND to Tetris course, where we actually write a compiler for a high-level language. And this compiler will generate a Hack code that runs in the machine that we build in this course. Now, there's one exception to what I said before, and that is that in some applications, in particular, in real-time systems or in applications where performance is incredibly important programmers sometimes have to look at the gory details of the machine language code if they want to optimize their programs. So they don't try the programs in machine language. They write them normally in, in, in a language like the C programming language but once they translate the program into machine language, they look at the code and if the code looks terribly tangled or unnecessarily long, they can go back and and rewrite their high-level code. But in the most part most most programmers writing high-level language, and that's it.

Solution

- Mult.asm

```
// for(i=0;i<n;i++)
// {
//   R2 = R2 + R0
//}
// n = R1
// i = 0
// R2 = 0
// LOOP:
//     if i >= n goto END
//      R2 = R2 + R0
//      i = i + 1
//      goto LOOP
// END

    @R1
    D=M
    @n
    M=D   // n=R1
```

```
    @i
    M=0    // i=0

    @R2
    M=0     // R2=0

(LOOP)
    @i
    D=M
    @n
    D=D-M
    @END
    D;JGE  // if i >= n goto END

    @R0
    D=M
    @R2
    M=D+M   //  R2 = R2 + R0

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

    @LOOP
    0;JMP

(END)
    @END
    0;JMP
```

- Fill.asm

```
//pseudo code:
//    while(true)){
//        if(KBD=0):  white
//        else:       black
//    }
//
//addr = SCREEN
//n = 8192
//i = 0
//LOOP:
//    if KBD=0: goto WHITE
//    goto BLACK
//WHITE:
//    if i >= n goto LOOP
//    RAM[addr] = 0
//    addr = addr + 1
//    i = i + 1
//    goto WHITE
//BLACK:
//    if i >= n goto LOOP
//    RAM[addr] = -1
//    addr = addr + 1
//    i = i + 1
```

```
//    goto BLACK
//

(LOOP)
    @SCREEN
    D=A
    @addr
    M=D // addr=SCREEN

    @8192
    D=A
    @n
    M=D // n=8192

    @i
    M=0 // i=0

    @KBD
    D=M
    @WHITE
    D;JEQ   //  if KBD=0: goto WHITE
    @BLACK
    0;JMP   // goto BLACK

(WHITE)
    @i
    D=M
    @n
    D=D-M
    @LOOP
    D;JGE   // if i >= n goto LOOP

    @addr
    A=M
    M=0     //  RAM[addr] = 0

    @addr
    M=M+1   //  addr = addr + 1

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

    @WHITE
    0;JMP   // goto WHITE

(BLACK)
    @i
    D=M
    @n
    D=D-M
    @LOOP
    D;JGE   // if i >= n goto LOOP

    @addr
    A=M
```

```
    M=-1     //  RAM[addr] = -1

    @addr
    M=M+1    //  addr = addr + 1

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

    @BLACK
    0;JMP    // goto WHITE
```

Computer Architecture

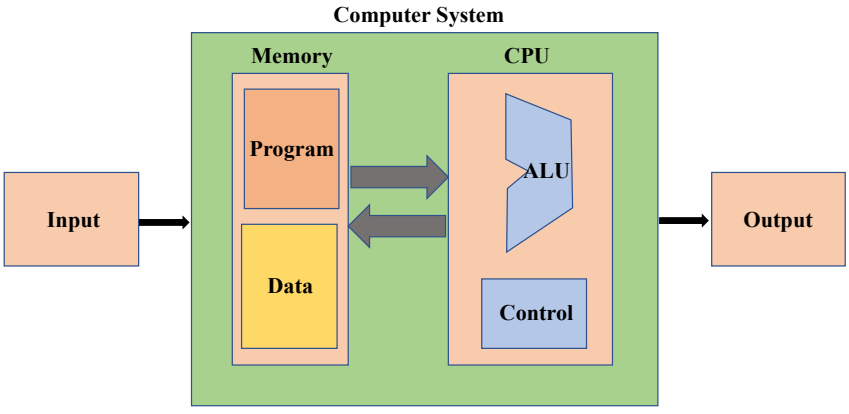
Let's recap the last four modules: we've built some elementary logic gates (module 1), and then used them to build an ALU (module 2) and a RAM (module 3). We then played with low-level programming (module 4), assuming that the overall computer is actually available. In this module we assemble all these building blocks into a general-purpose 16-bit computer called *Hack*. We will start by building the Hack Central Processing Unit (CPU), and we will then integrate the CPU with the RAM, creating a full-blown computer system capable of executing programs written in the Hack machine language.

Unit 5.1: Von Neumann Architechture

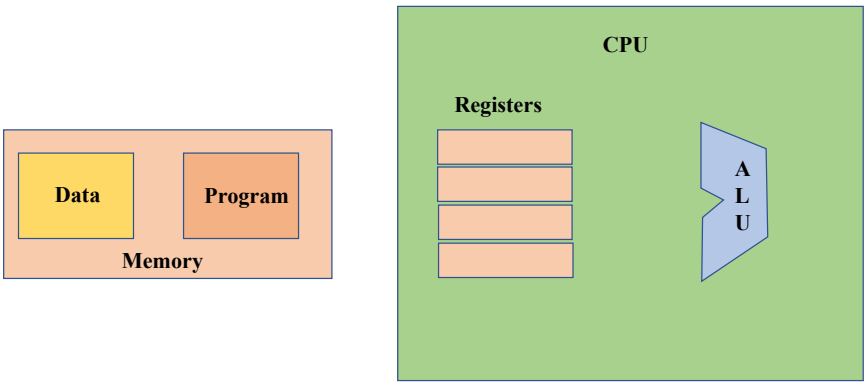
Universality

- Some **Hardware** can run many different **Software** programs
    - Theory: Universal Turing Machine
    - Practice: von Neumann Architecture

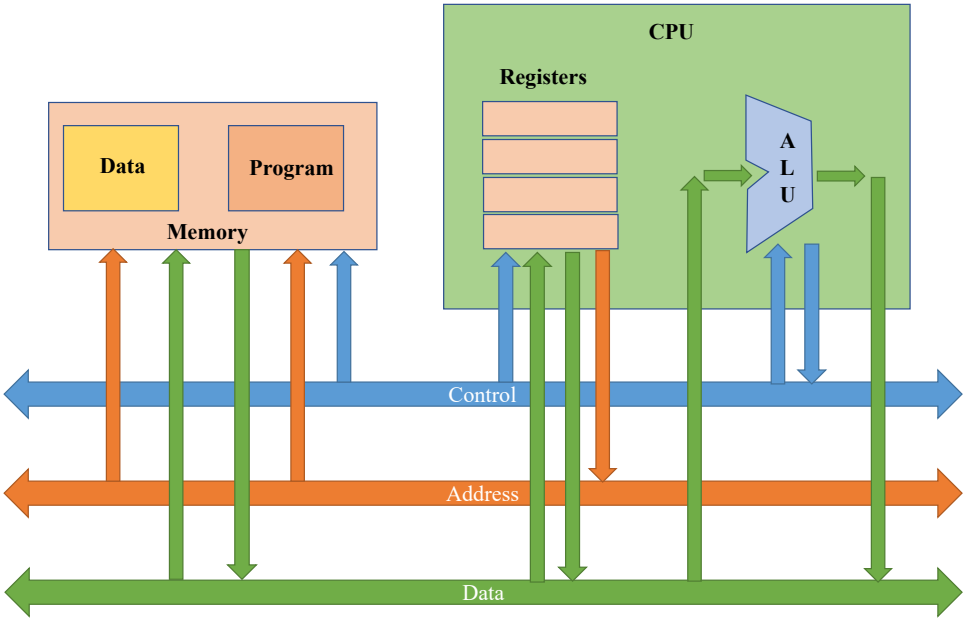
Stored Program Computer



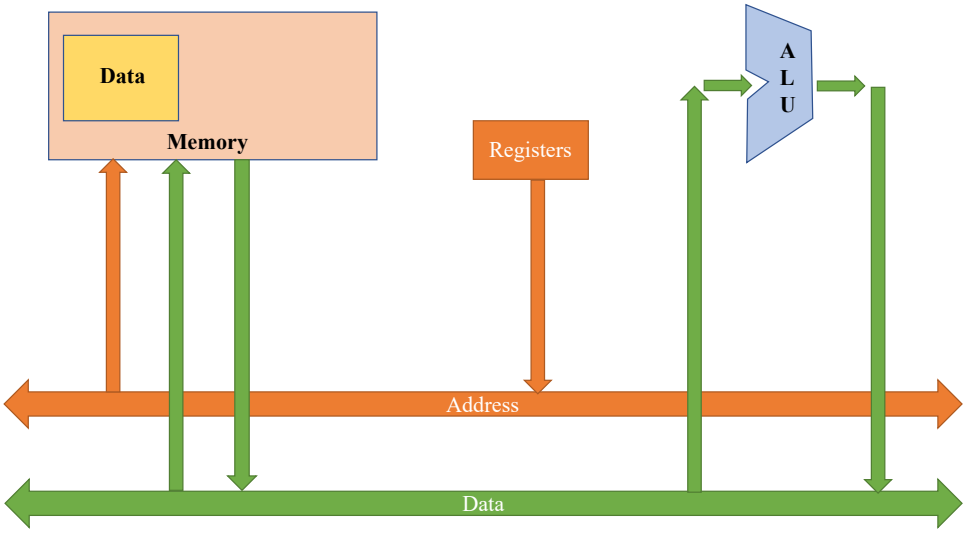
Elements



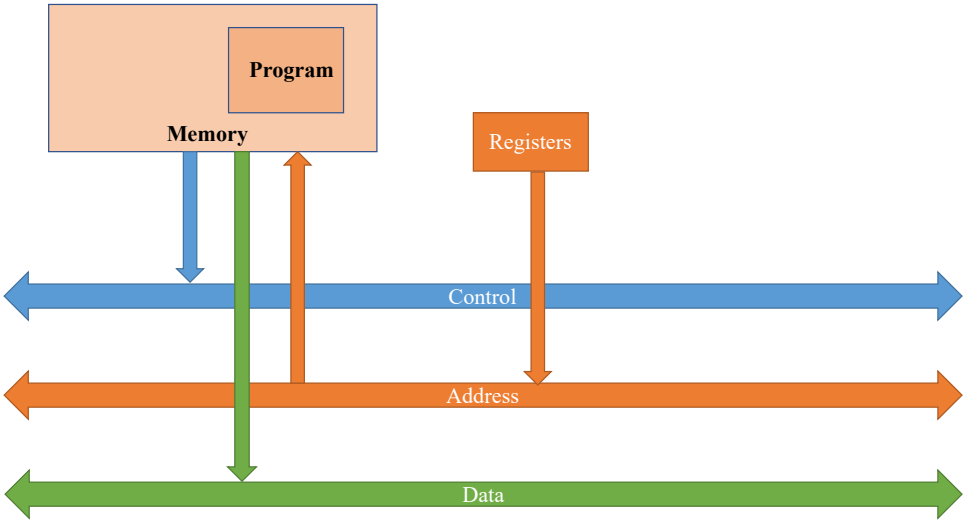
information flows



Data memory



Program Memory



Unit 5.2: The Fetch-Execute Cycle

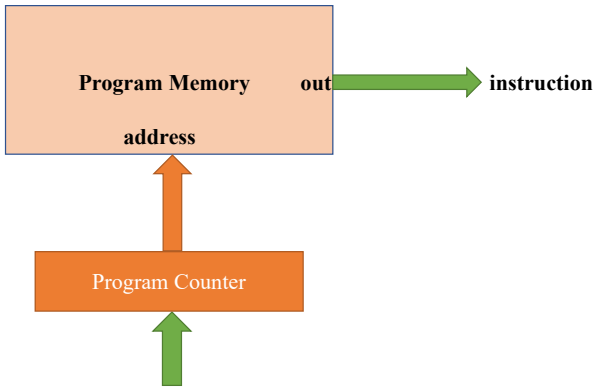
The basic CPU loop

- **Fetch** an Instruction from the program memory
- **Execute** it

Fetching

- Put the location of the next instruction into the address of "address" of the program memory
- Get the instruction code itself by reading the memory contents at that location

The program Counter

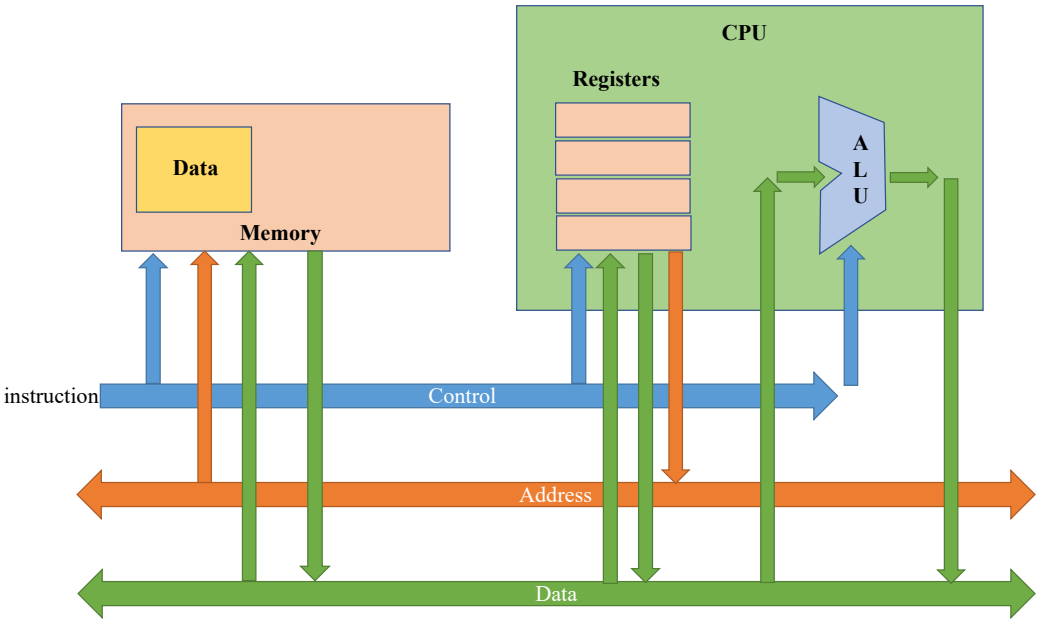


- Normally, old_value+1
- Jump achieved by loading another value

Executing

- The instruction code specifies "what to do"
  - Which arithmetic or logical instruction
  - What memory to access (read/write)
  - If/where to jump
  - ...
- Often, different subsets of the bits control different aspects of the operation
- Executing the operation involves also accessing registers and/or data memory

Executing an Instruction

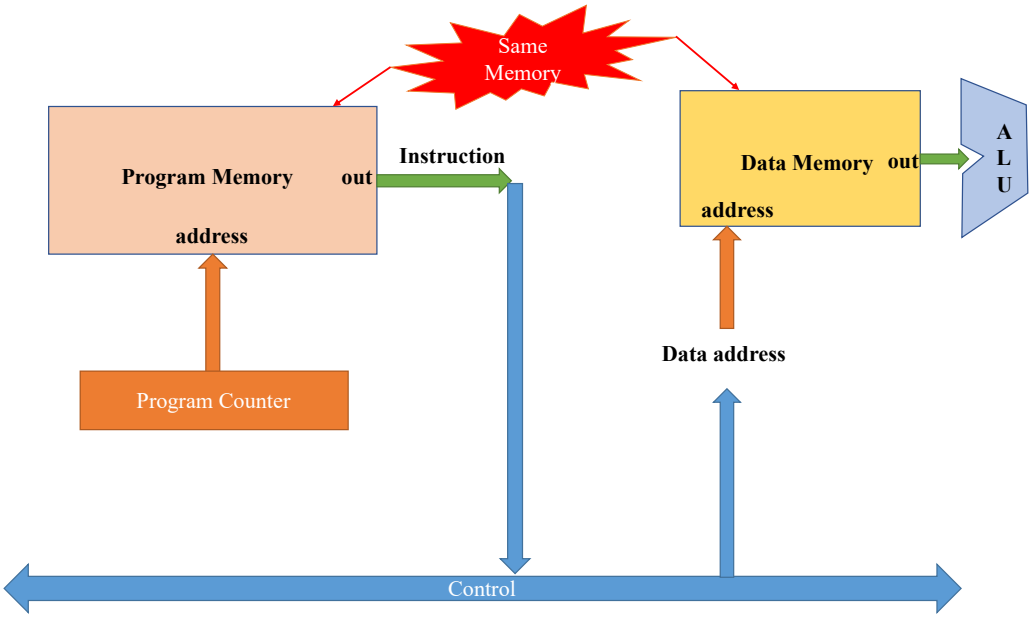


Fetch-Execute Clash

In the fetch cycle, basically we need to get from the program memory, the next instruction. So we need to put into the address of the memory, the address of the next instruction and get the instructions output.

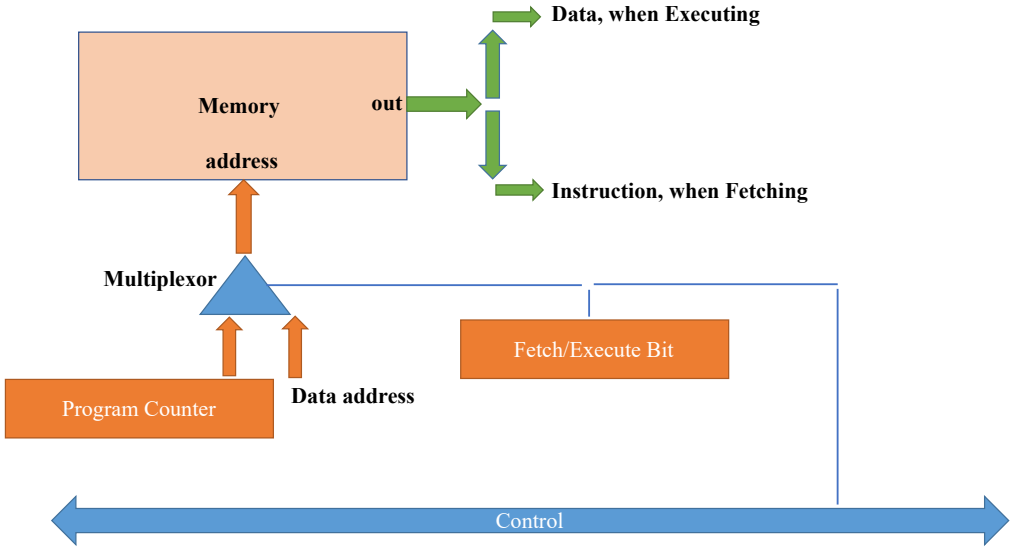
On the other hand, in the execute cycle, we need to access data that also resides in memory. So we need to put into the address of the memory, the address of the data piece that we want to operate on, which has nothing to do with the program piece that gave us the instruction.

And because we have a single memory, that is a clash, because what are we going to put into the address, are we going to put there, the address of the instruction, or is address of the data piece? We need to do both and that's a problem

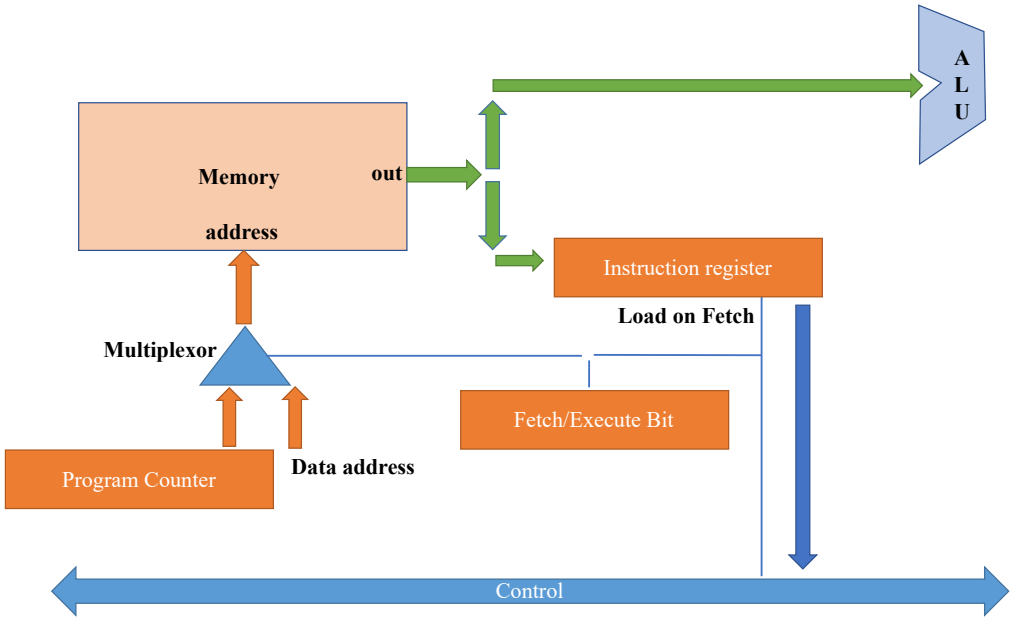


Solution: Multiplex

When we actually are in the fetch cycle, we put the address of the next instruction, we get the next instruction, and then we need to remember it inside an instruction register. And that instruction register is exactly what is remains holding the value of the instruction that we're now executing in the execute cycle. And then of course, in the execute cycle now we have the instruction already stored in this register. And we can work with all the information that we need a, for the execute cycle using the data memory that is being addressed in the second part of the cycle. So, this is the usual way it's done:



Instruction register



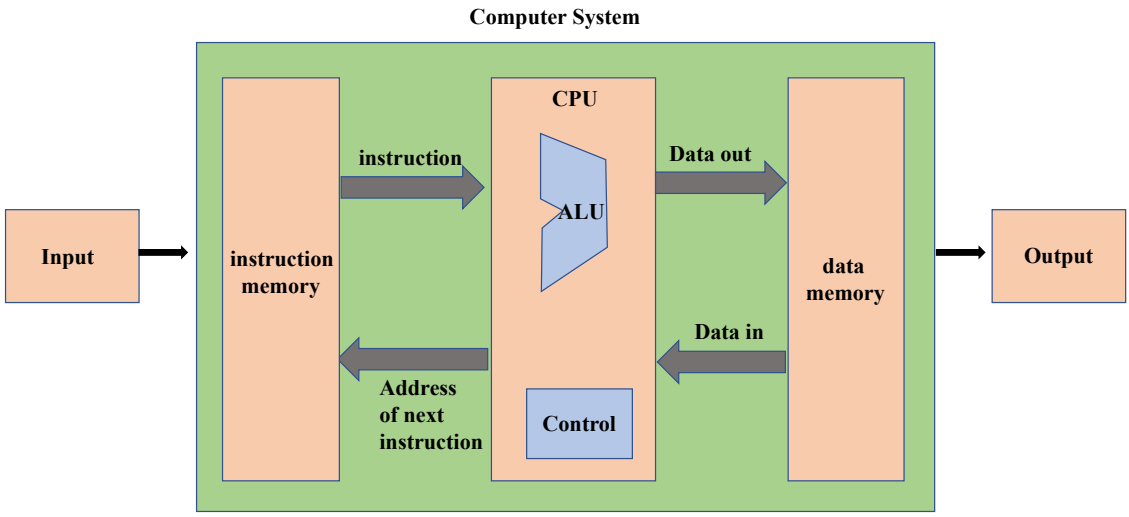
Simpler Solution: Harvard Architecture

- Variant of von Neumann Architecture
- Keep Program and Data in two separate memory modules
- Complication avoided

代价是程序不能被动态改变

Unit 5.3: Central Processing Unit

The overall computer architecture

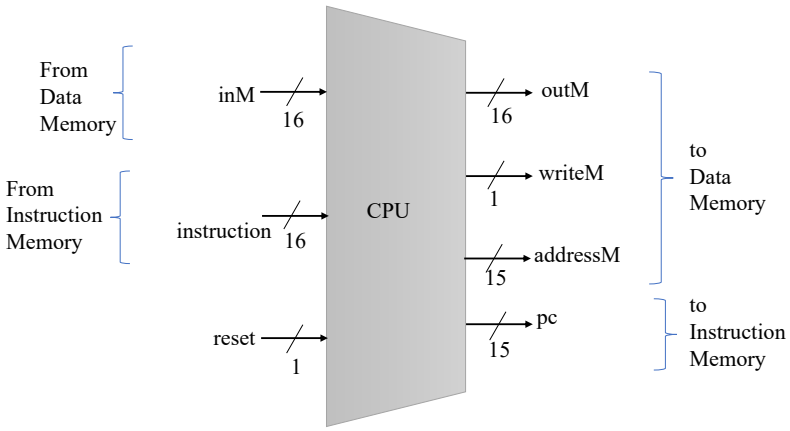


The Hack CPU: Abstraction

A 16-bit processor, designed to:

- Execute the current instruction
- Figure out which instruction to execute next (instructions written in the Hack language)

Hack CPU Interface



Inputs:

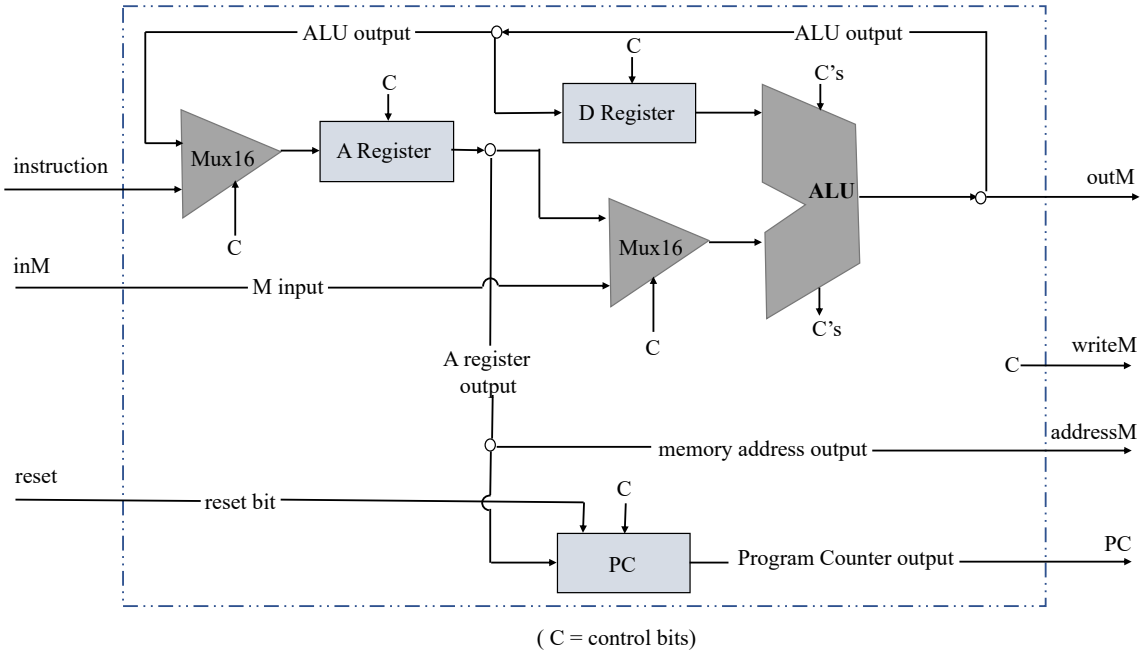
- Data value
- Instruction
- Reset bit

Outputs:

- Data value

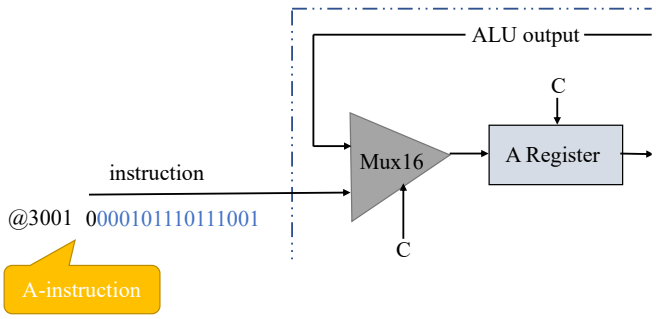
- Write to memory? (Yes/no)
- Memory Address
- Address of next instruction

Hack CPU Implementation



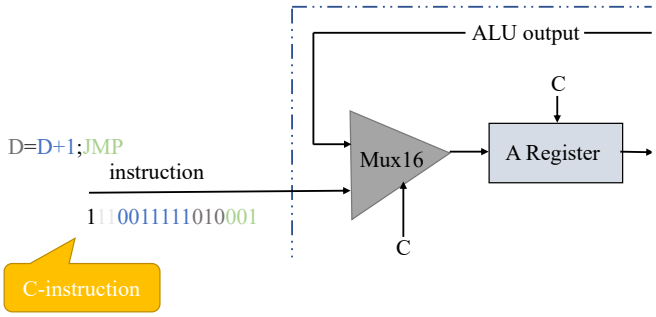
(C = control bits)

Instruction handling



CPU handling of A-instruction:

- Decodes the instruction into op-code + 15-bit value
- Stores the value in the A-register
- Outputs the value (not shown in this diagram)

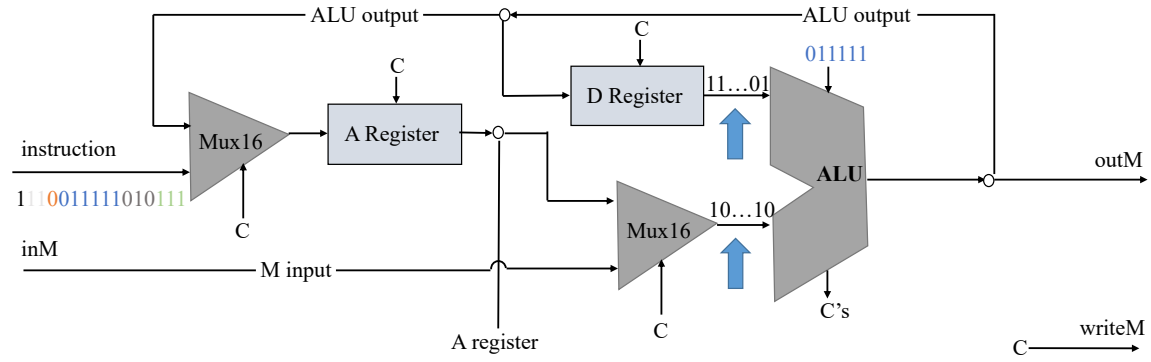


CPU handling of c-instruction:

The instruction bits are decoded into:

- Op-code
- ALU control bits
- Destination load bits
- Jump bits

ALU operations: inputs



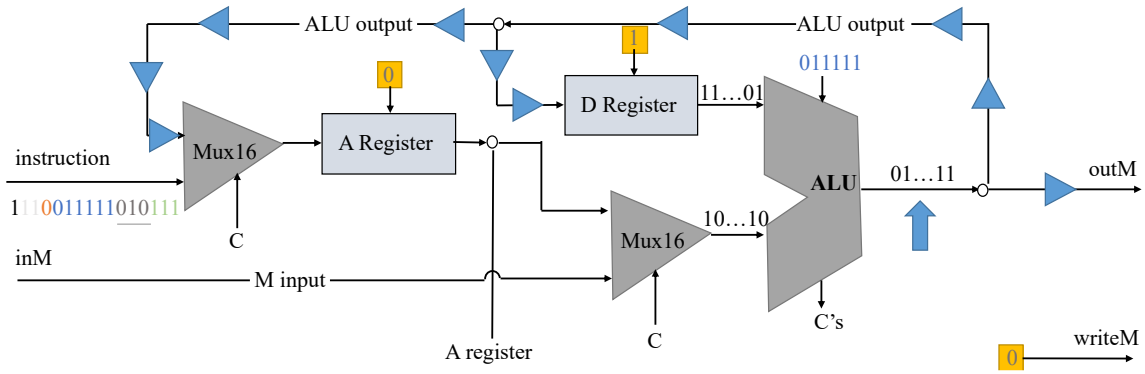
ALU data inputs:

- From the D-register
- From the A-register/M-register

ALU control inputs:

- Control bits (from the instruction)

ALU operation: outputs



ALU data output:

- Result of ALU calculation, fed simultaneously to: D-register, A-register, M-register
- Which register *actually* received the incoming value is determined by the instruction's destination bits

ALU control outputs:

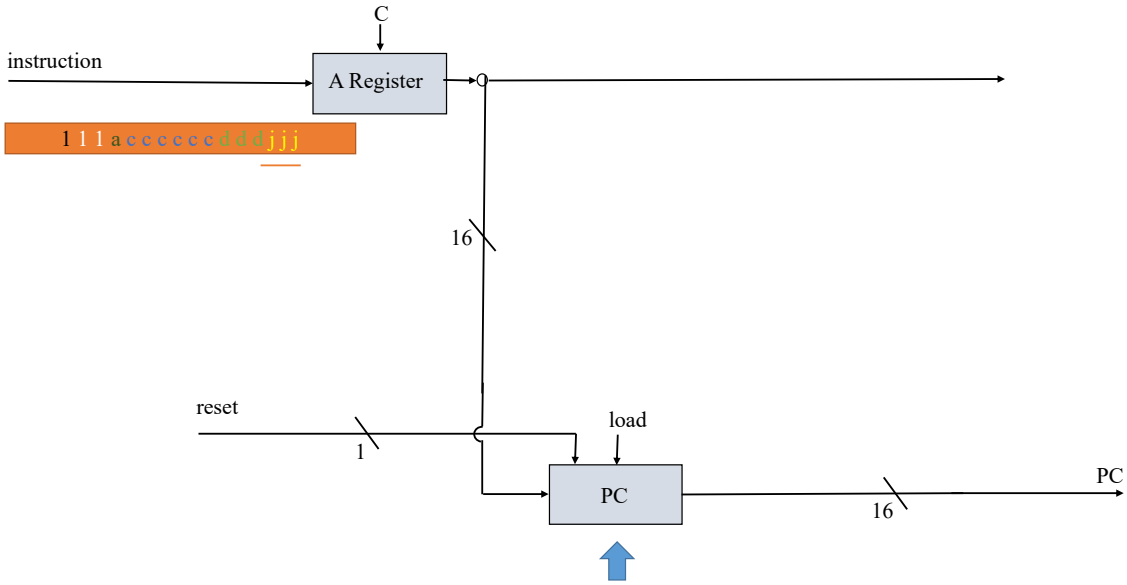
- Negative outputs?
- Zero output?

These two outputs play a key role in what will be described next, which is the control logic of the CPU

Possible outside view of the Hack computer

- The computer is loaded with some program
- When you push **reset**, the program starts running

Control (abstraction)



Program Counter abstraction

Emits the address of the next instruction:

To start/restart the program's execution: PC=0

- no jump: PC++
- goto: PC=A
- conditional goto: if the condition is true PC=A else PC++

Control (implementation)

PC logic

if (reset==1) PC=0

else

// current instruction

load = f(jump bits,ALU control outputs)

if (load==1) PC=A

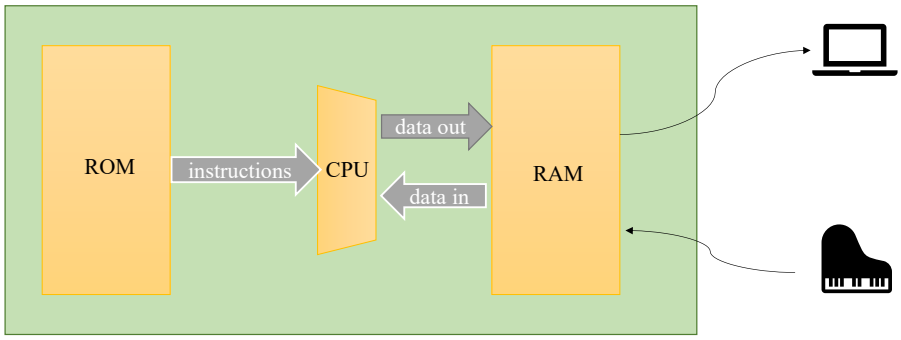
else PC++

PC output

the address of the next instruction

Unit 5.4: The Hack Computer

The Hack Computer



Abstraction:

A computer capable of running programs written in the Hack machine language

Implementation:

Bulit from the Hack chip-set

Hack CPU Operation

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

- If the instruction includes D and A, the respective values are read from,and/or written to, the CPU-resident D-register and A-register
- If the instruction is @x, then x is stored in the A-register; this value is emitted by addressM
- If the instruction's RHS includes M,this value is read from inM
- If the instruction's LHS includes M,then the ALU output is emitted by outM, and the writeM is asserted
- If (reset==0)

  The CPU logic uses the instruction's jump bits and the ALU's output to decide if there should be a jump

  If there is a jump: PC is set to the value of the A-register

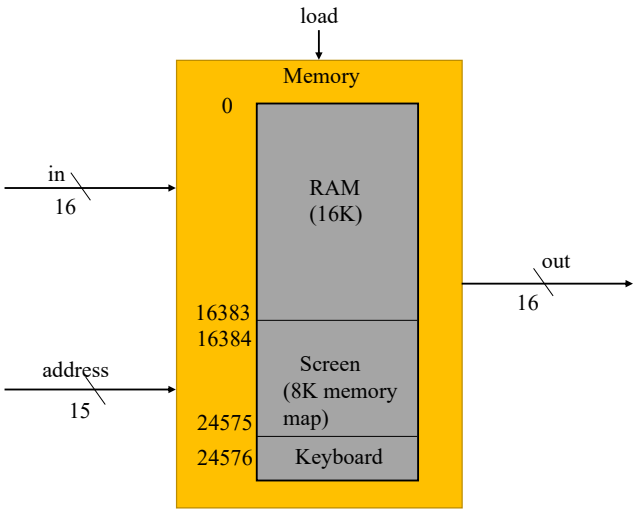
  Else(no jump): PC++

  The updated PC value is emitted by pc

if (reset==1)

PC is set to 0,pc emits 0 (causing a program restart)

Memory



Abstraction:

- Address 0 to 16383: data memory
- Address 16284 to 24575: screen memory map
- Address 24576: keyboard memory map

Implementation

- RAM: 16-bit/16K RAM chip
- Screen: 16-bit/8K memory chip with a raster display side-effect
- Keyboard: 16-bit register with a keyboard effect

RAM

Implemented by a 16-bit, 16K RAM chip

Screen implementation

Implemented by a built-in 8K Screen chip featuring a raster dispaly refresh side-effect

Keyboard implementation

Implemented by a built-in Keyboard chip featuring a keyboard capture side-effect

Instruction Memory(ROM32K)

To run a program on the Hack computer

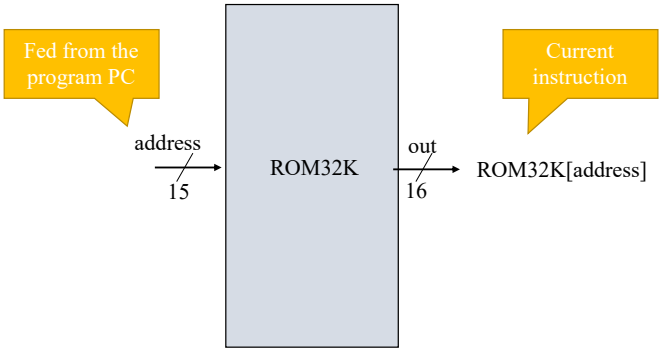
- Load the program into the ROM32K
- Press "reset"
- The program starts running

Loading a program

- Hardware implementation: plug-and-play(即插即用) ROM chips

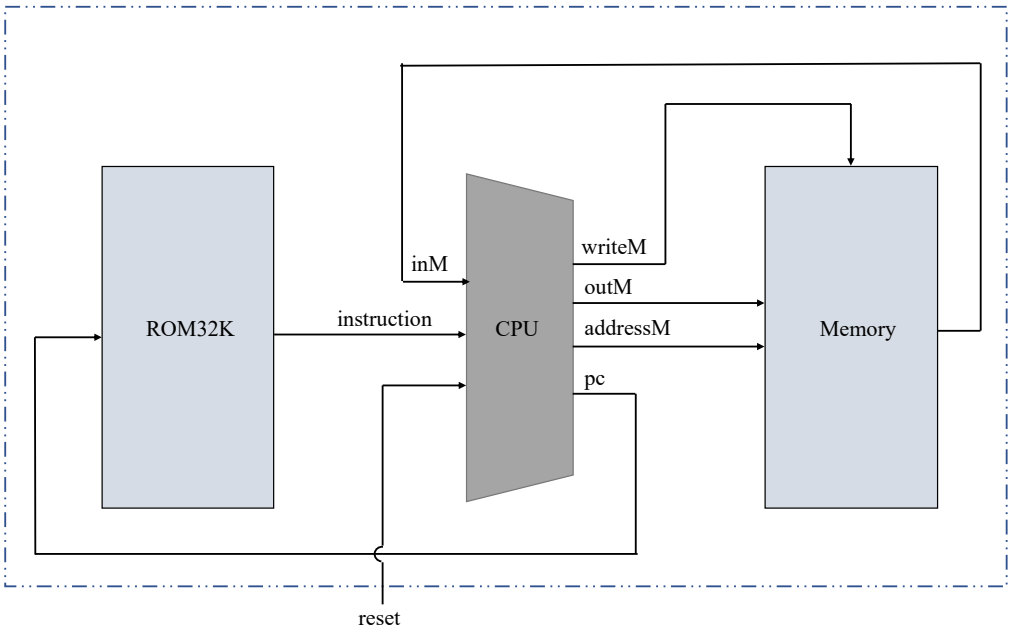
- Hardware simulation: programs are stored in text files; program loading is emulated by the built-in ROM chip

ROM interface



- Implemented as a built-in ROM32 chip
- Contains a sequence of Hack instructions(program)

Hack Computer implementation

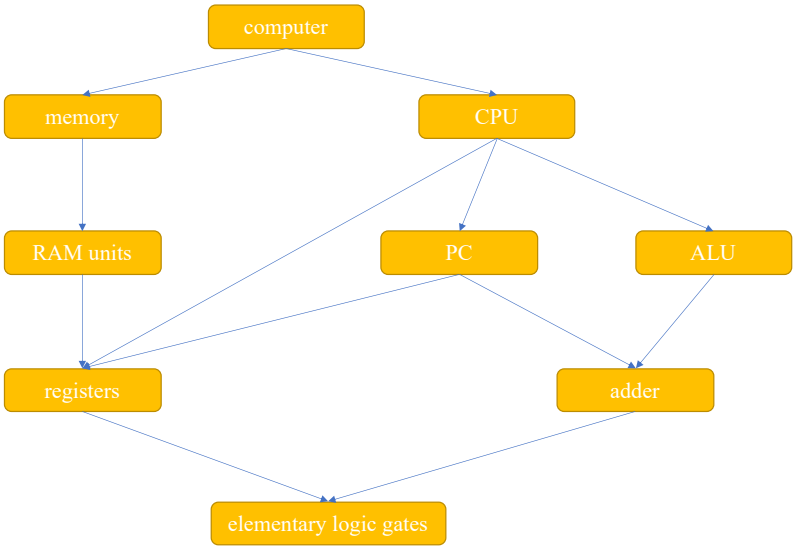


"We ascribe beauty to that which is simple; which has no superfluous parts; which exactly answers its end; which stands related to all things; which is the mean of many extremes."

--Ralph Waldo Emerson

Unit 5.5: Project 5 Overview

Hardware organization: a hierarchy of chip parts



CPU Implementation

Implementation tips

- Chip-parts: Mux16, ARegister, DRegister, PC, ALU, ...
- Control: use HDL subscripting to route instruction bits to the control bits of the relevant chip-parts

Q & A

The main architectural differences between Von Neumann and Harvard.

Well, in the, in our Hack computer, we had to separate, read-only memory for the program and the separate data memory that was read-write. Separating between these two different memories is sometimes called the Harvard architecture and which I tend to view as just a flavor, if you wish of Von Neumann architecture.

In a classic Von Neumann architecture, you have a single memory that contains both the data and the program. As we explained in the second unit of this week doing it the classical way has a tiny complication that you need to separately fetch an instruction, that is access a program instruction. And separately at a different time at the next cycle, execute the program which to, which to required for a, accessing the data memory.

In our Harvard architecture, we had the program in a separate memory unit. Basically, we could do both of them in a single cycle, which made it a bit simpler. Now this architecture, as is, is probably a very appropriate for sometimes what's called embedded computers, where for computer does a single thing that was pre-programmed to it and just put in to ROM.

A general purpose computer in which you want to keep on changing the program that you run, you would probably need the full, real Von Neumann architecture.

The differences are not that big between these two architectures, as we saw in a unit two of this week and we chose to take the simplest one. Again, keeping with the simplicity of our whole approach.

So normally, if I understood your explanation one implication from Harvard to Von Neumann architecture is that in Von Neumann, we expect the computer to do different things at different cycles or different points of time. So, is there some organized way in computer science to to model this this different behavior of computers?

Absolutely. The standard way of doing it is via the formalism of the finite machine(状态机).

A specification of the different things that the computer is supposed to do at different times steps and how we're supposed to move from one statement to another. Let me demonstrate. Okay. So the idea is that for every possible state that the machine can be in, you have a type of circle and you have arrows between the different, different circles or between the different states of the machine that tell you, from which state do you move to which other state and why.In each state, you can write what are the kind of thing that the machine needs to do in that situation. For example, maybe in this state, you want to add one to the program counter and maybe you want a certain address to get a certain value. For example, the value from the memory address, address by n. And so, on every different state, you can write this information in terms of the state.This gives you a very clear picture of what needs to be done in every possible state and where, what kind of state do you move for, do you move to after everything that happened? For example, you may move from here to here, if the clock if the current clock cycle changed or if the current memory location has value zero. Or various other situations, that are, can be found in the Harvard. The values of the different hardware logic. So, once you specify this and this kind of way formalism in this kind of finite state machine formalism, now we can translate this to the normal registers and combinatorial logics that we already know. The trick would be to simply add another register that is called state. A register that will encode in which one of these cycles are we in. In our situation, we'll need 2 bits to encode three possible states. So we'll have here 2 bits.Once we have this state register, now we can write combinatoty logics that basically affects everything that we need. So for example, if the 2 bits this is just a register is has 2 bits going in and 2 bits going out. But all the combinatorial logic that we have can also accept the state as part of its input. Like it accepts counter and so on. Different combinatorial parts of our computer, accepts various inputs. Now they also have this input, the state. And of course, they can do different things according to the different state, including deciding what the next state is going to be. Again, it's a function of the current state and all the other hardware signals in the computer. So the way you translate from this formalism from the finite state machine formalism to this actual implementation is pretty straight forward. And and oh, completely technical, if you wish. Once you actually coded everything, it's just like you organized hardware previously. And that's an organized way to design computers that do different things in different times and move between the different times and different states in an organized manner.

what does it take to connect a computer to more peripheral devices you know, in addition to a keyboard and a screen?

So indeed, real computers have many peripheral units which are the screen, the keyboard, a mouse, a microphone, a disk and so on so forth. And also this architecture is scalable. We can add more devices as we please and the question in there is how can you possibly do it? Well just like we did with the screen and the keyboard, we can allocate memory space to represent every one of these peripheral devices. And when we want to write something, for instance when we want to sound something on the microphone or write something to the disk, we can write into memory certain codes that are later later, will be translated into physical signals that actually operate

these peripheral devices. But at some point, when you add several such peripheral devices, the CPU becomes extremely overloaded. Because the poor CPU has to not only run your program, but also manage all these peripheral devices. So the typical approach is to offload the CPU from all this headache and use what is known as device controllers. So typically, when you add, for example, a disk. The disk will be equipped with device controller, which is a dedicated hardware, which knows how to manage the disk, which knows how to translate operations from what the CPU wants to do to actual movements of the disk an, and so on. And something like this happens with every particular I/O device. And for example, take the screen. In the, Hack platform, the screen is managed in a very simplistic way. When you want to turn on and off a pixel, you simply turn on and off a bit in memory. And you assume that at some point this this manipulation will be refreshed or will, will cause the screen to be refreshed. So the CPU is in charge for every thing. If you want to draw a line, the CPU has to actually write all these points all these bits into memory and, and the line will get drawn at a certain point of time. In a real computer the screen comes equipped with a graphics card or some graphics accelerator and this is a dedicated computer that can do all sorts of things internally. So if we want to draw a line from one coordinate to another, we can simply tell the controller go ahead and do it and the controller will, you know, will do everything, which is necessary to compute, you know, which pixels have to be turned on and off and so on. So, once again, we can add many I/O devices as we please. In principle, it will be very similar to what we did with the screen and the keyboard. But there are numerous details involved, which are specific to all these different devices.

Solution

- Memory.hdl

```
CHIP Memory {
    IN in[16], load, address[15];
    OUT out[16];

    PARTS:
    // Put your code here:
    DMux(in=true, sel=address[14], a=a, b=b);
    And(a=a, b=load, out=load0);
    RAM16K(in=in, load=load0, address=address[0..13], out=a1);
    DMux(in=b, sel=address[13], a=c, b=d);
    And(a=c, b=load, out=load1);
    Screen(in=in, load=load1, address=address[0..12], out=a2);
    Keyboard(out=a3);
    Mux16(a=a2, b=a3, sel=address[13], out=out2);
    Mux16(a=a1, b=out2, sel=address[14], out=out);
}
```

- CPU.hdl

```
CHIP CPU {

    IN  inM[16],         // M value input  (M = contents of RAM[A])
        instruction[16], // Instruction for execution
        reset;           // Signals whether to re-start the current
                         // program (reset==1) or continue executing
                         // the current program (reset==0).

    OUT outM[16],        // M value output
```

```
        writeM,          // Write to M?
        addressM[15],    // Address in data memory (of M)
        pc[15];          // address of next instruction

    PARTS:
    // Put your code here:
    Mux16(a=instruction, b=ALUoutput, sel=instruction[15],
out=Mux16toARegister);
    And(a=instruction[15], b=instruction[5], out=CinstructionAndd1);
    Not(in=instruction[15], out=not15);
    Or(a=not15, b=CinstructionAndd1, out=CtoARegister);
    ARegister(in=Mux16toARegister ,load=CtoARegister ,out=ARegisterOutput,
out[0..14]=addressM);
    Mux16(a=ARegisterOutput, b=inM, sel=instruction[12], out=Mux16toALU);
    ALU(x=DRegistertoALU, y=Mux16toALU, zx=instruction[11], nx=instruction[10],
zy=instruction[9], ny=instruction[8], f=instruction[7], no=instruction[6],
out=outM, out=ALUoutput, zr=outIsZero, ng=OutNegitive);
    And(a=instruction[15], b=instruction[4], out=i15andi4);
    DRegister(in=ALUoutput , load=i15andi4 , out=DRegistertoALU);
    And(a=instruction[15], b=instruction[3], out=writeM);
    And(a=outIsZero, b=instruction[1], out=w1);
    And(a=OutNegitive, b=instruction[2], out=w2);
    Not(in=outIsZero, out=notZR);
    Not(in=OutNegitive, out=notNG);
    And(a=notZR, b=notNG, out=h1);
    And(a=h1, b=instruction[0], out=w3);
    Or(a=w1, b=w2, out=w12);
    Or(a=w12, b=w3, out=loadofPC);
    And(a=loadofPC, b=instruction[15], out=CloadofPC);
    PC(in=ARegisterOutput, load=CloadofPC, inc=true, reset=reset, out=PCOutput,
out[0..14]=pc);
}
```

- Computer.hdl

```
CHIP Computer {

    IN reset;

    PARTS:
    // Put your code here:
    CPU(inM=inM, instruction=ins, reset=reset, outM=out1, writeM=outwriteIntoM,
addressM=outAddrM, pc=outPc);
    ROM32K(address=outPc, out=ins);
    Memory(in=out1, load=outwriteIntoM, address=outAddrM, out=inM);
}
```

Assembler（汇编器）

Every computer has a *binary machine language*, in which instructions are written as series of 0's and 1's, and a *symbolic machine language*, also known as *assembly language*, in which instructions are expressed using human-friendly mnemonics. Both languages do exactly the same thing, and are completely equivalent. But, writing programs in assembly is far easier and safer then writing in binary. In order to enjoy this luxury, someone has to translate our symbolic programs into

binary code that can execute as-is on the target computer. This translation service is done by an agent called *assembler*. The assembler can be either a person who carries out the translation manually, or a computer program that automates the process. In this module and final project in the course we learn how to build an assembler. In particular, we'll develop the capability of translating symbolic Hack programs into binary code that can be executed as-is on the Hack platform. Each one of you can choose to accomplish this feat in two different ways: you can either implement an assembler using a high-level language, or you can simulate the assembler's operation using paper and pencil. In both cases we give detailed guidelines about how to carry out your work.

Unit 6.1: Assembly Languages and Assemblers

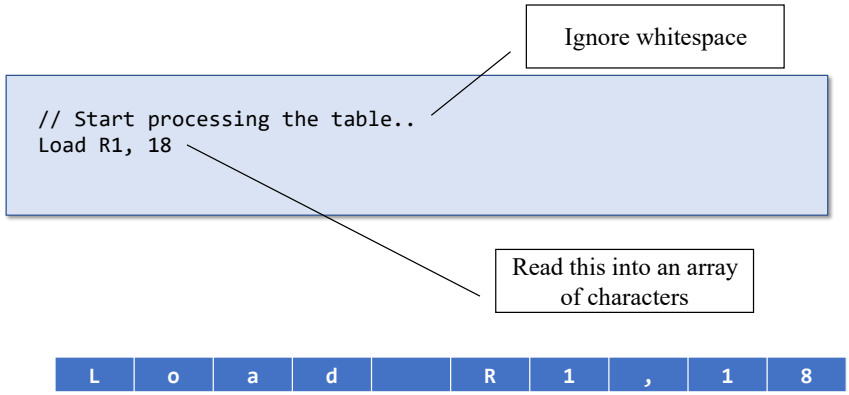
This is Software!

- The "Assembler" is software
- First software layer above the hardware

Basic Assembler Logic

Repeat:

- Read the next Assembly language command



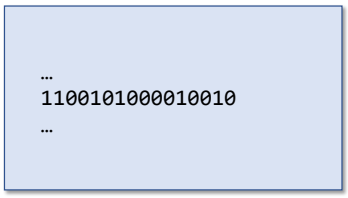
- Break it into the diffferent fields it is composed of

- Lookup the binary code for each field

Command	Opcode
Add	10010
Load	11001
...	...

- Combine these codes into a single machine language command

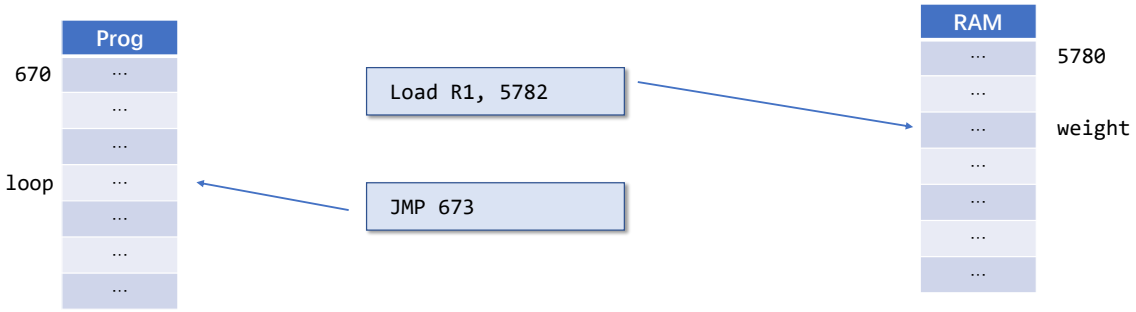
- Output this machine language command



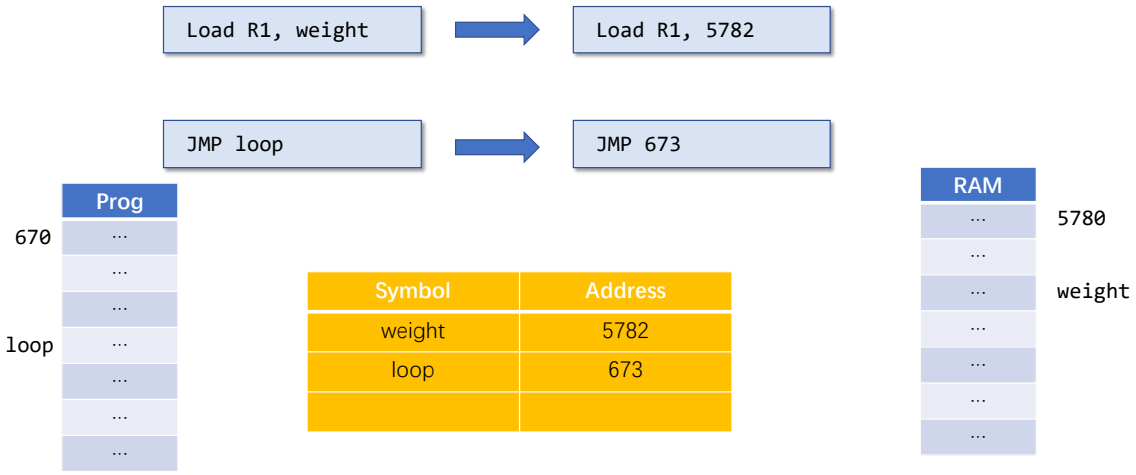
Until end-of-file reached

Symbols

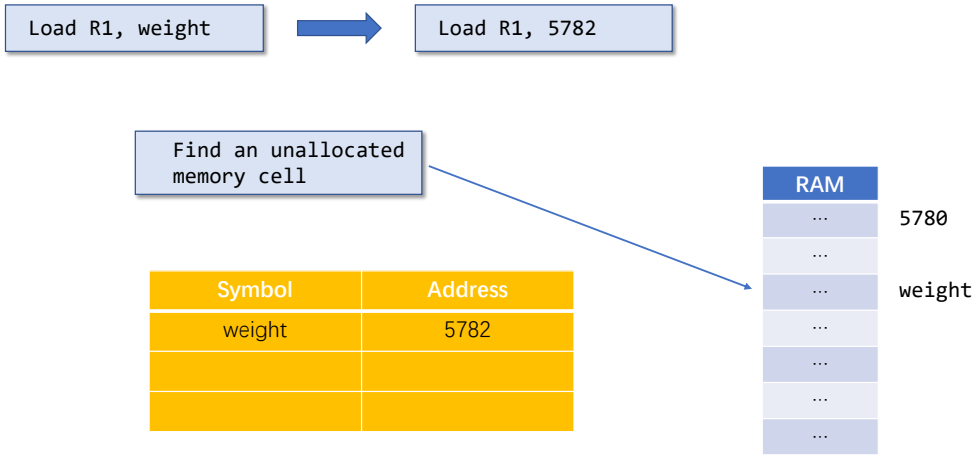
- Used for:
  - Labels `JMP loop`
  - Variables `Load R1, weight`
- Assembler must replace names with addresses



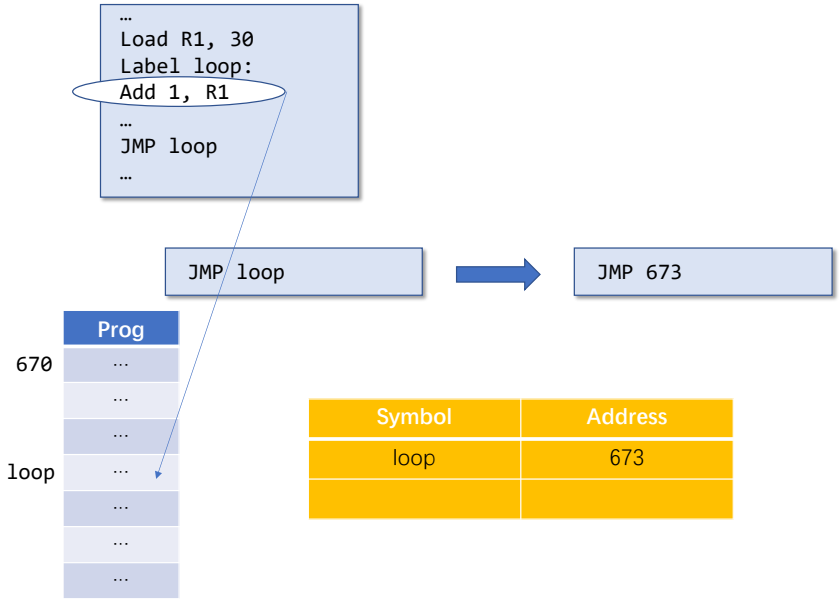
Symbol Table



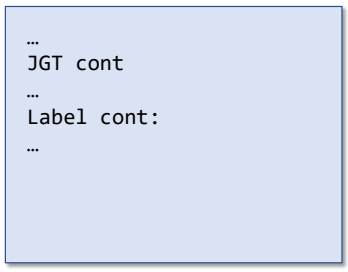
Allocation of variables



Labels



Forward references



Symbol	Address

Possible Solutions:

- Leave blank until label appears, then fix
- In first pass just figure out all address

> we can jump into a label before the label was actually defined. This is called a **forward reference**.
>
> So for example, it's very common in, in programs that I will need to jump, let's say, into a label called cont, before the label actually occurred in the program. If that is the case, when I reach the first jump, when I first, the, the jump instructions, it uses a label. That happens before I've already seen the place that defines the label in the label command, and how can I handle that because it's not in the table yet.
>
> Well, there are two ways to handle it. One way, which is usually a little bit more complicated, is to basically remember that I've seen the labels, and I don't know where it is yet. Keep it in a side table. And when I actually get into the definition of the correct address, fix it back.Another, another option, which turns out to be sometimes easier, is that I actually do everything in two passes. In the first pass, I read everything only paying attention to the labels and remembering where each label refers to. That, that's when i actually build the table for the labels. And only on the second pass do I actually go and put and convert each and every label into its correct code, into its correct address, that now is already in the table, because I put it there in the first pass. And that's a usual, usually slightly easier to do, but you may use each one of these two possibilities.

Unit 6.2: The Hack Assembly Language

The translator's challenge

Based on the syntax rules of:

- The source language
- The target language

Assembly program elements

- White space

    - Empty lines / indentation
    - Line comments
    - In-line comments
- Instructions
    - A-instructions
    - C-instructions
- Symbols
    - References
    - Label declarations

The plan ahead

1. Develop a basic assembler that translates symbol-less Hack programs (next unit)
2. Develop an ability to handle symbols (nexxt unit)
3. Morph the basic assembler into an assembler that translates any Hack program (nexxxt unit)

Unit 6.3: The Assembly Process - Handling Instructions

Translating A-instructions

Translation to binary:

- if value is a decimal constant, generate the equivalent 15-bit binary constant
- if value is a symbol, later

The overall assembly logic

For each instruction

- Parse the instruction:

  break it into its underlying fields
- A-instruction:

  translate the decimal value into a binary code
- 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

Unit 6.4: The Assembly Process - Handling Symbols

Handling Symbols

Symbols:

- variable symbols:

  represent memory locations where the programmer wants to maintain values
- label symbols:

  represent destinations of goto instructions

- pre-defined symbols:

  represent special memory locations

Handling pre-defined symbols

The Hack language specification describes 23 pre-defined symbols

Translating @preDefinedSymbol:

Replace preDefinedSymbol with its value

Handling symbols that denote labels

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 memory location holding the next instruction in the program

Translating @labelSymbol:

Replace labelSymbol with its value

Handling symbols that denote variables

Variable symbols

- Any symbol XXX appearing in an assembly program which is not pre-defined and is not defined elsewhere using the (XXX) directive is treated as a variable
- Each variable is assigned a unique memory address, starting at 16

Translating @variableSymbol:

- If you see it for the first time, assign a unique memory address
- Replace variableSymbol with its value

Symbol table

Initialization:

Add the pre-defined symbols

First pass:

Add the label symbols

Second pass:

Add the var.symbols

Usage:

To resolve a symbol, look up its value in the symbol table

The assembly process

Initialization:

- Construct an empty symbol table
- Add the pre-defined symbols to the symbol table

First pass:

Scan the entire program;

For each "instruction" of the form (xxx)

- Add the pair (xxx,address) to the symbol table, where address is the number of the instruction following (xxx)

Second pass:

Set n to 16

Scan the entire program again; for each instruction:

- If the instruction is @symbol, look up symbol in the symbol table
    - If (symbol,value) is found, use value to complete the instruction's translation
    - If not found:
        - Add (symbol,n) to the symbol table
        - Use n to complete the instruction's translation
        - n++
- If the instruction is a C-instruction, complete the instruction's translation
- Write the translated instruction to the output file

Unit 6.5: Developing a Hack Assembler

Sub-tasks that need to be done

- Reading and parsing commands
- Converting mnemonics -> code
- Handling symbols

Reading and Parsing Commands

No need to understand the meaning of anything

- Start reading a file with a given name
    - E.g. Constructor for a **Parser** object that accepts a string specifying a file name
    - Need to know how to read text files
- Move to the next command in the file
    - Are we finished? **boolean hasMoreCommands()**
    - Get the next command. **void advance()**
    - Need to read one line at a time
    - Need to skip whitespace including comments
- Get the fields of the current command

- Type of current command (A-Command, C-Command, or Label)
- Easy access to the fields:
  - `D = M+1; JGT`
D

    `String dest();`
M	+	1

    `String comp();`
J	G	T

    `String jump();`
  - `@sum`
s	u	m

    `String label()`

Translating Mnemonic to Code

No need to worry about how the mnemonic fields were obtained

Recap: Parsing + Translating

```
# Assume that current command is
# D=M+1; JGT
# Parser class breaks the assembly language command into its components
String c = parser.comp() #"M+1"
String d = parser.dest() #"D"
String j = parser.jump() #"JGT"
# Code class translates each component to its binary format
string cc = code.comp(c) #"1110111"
String dd = code.dest(d) #"010"
String jj = code.jump(j) #"001"
String out = "111" + cc + dd + jj
```

The Symbol Table

No need to worry about what these symbols mean

- Create an new empty table
- Add a (symbol,address) pair to the table
- Does the table contain a given symbol?
- What is the address associated with a given symbol?

Using the Symbol Table

- Create an new empty table

- Add all the pre-defined symbols to the table
- While reading the input, add labels and new variables to the table
- Whenever you see a "**@xxx**" command, where xxx is not a number, consult the table to replace the symbol xxx with its address
- While reading the input, add labels and new variables to the table
    - **Labels:** when you see a "**(xxx)**" command, add "xxx" and the address of the next machine language command
        - Comment 1 : this requires maintaining this running address
        - Comment 2 : this may need to be done in a first pass
    - **Variables:** when you see an "**@xxx**" command, where "xxx" is not a number and not already in the table, add "xxx" and the next free address for variable allocation

Overall logic

- Initialization
    - Of Parser
    - Of Symbol Table
- First Pass: Read all commands, only paying attention to labels and updating the symbol table
- Restart reading and translating commands
- Main loop:
    - Get the next Assembly Language Command and parse it
    - For A-commands: Translate symbols to binary addresses
    - For C-commands: get code for each part and put them together
    - Output the resulting machine language command

Unit 6.6: Project 6 Overview: Programming Option

Developing a Hack Assembler

Contract

- Develop a HackAssembler program that translates Hack assembly programs into executable Hack binary code
- The source program is supplied in a text file named Xxx.asm
- The generated code is written into a text file named Xxx.hack
- Assumption: Xxx.asm is error-free

Usage

```
prompt> java HackAssembler Xxx.asm
```

This command should create (or override) an Xxx.hack file that can be executed as-is on the Hack computer

Proposed design

The assembler can be implemented in any high-level language

Proposed software architexture

- Parser: unpacks each instruction into its underlying fields
- Code: translates each field into its corresponding binary value
- SymbolTable: manages the symbol table
- Main: initialzes the I/O files and drives the process

Proposed Implementation

Staged development

- Develop a basic assembler that translates assembly programs without symbols
- Develop an ability to handle symbols
- Morph the basic assembler into an assembler that can translate any assembly program

Supplied test programs

- `Add.asm`
- `Max.asm` `MaxL.asm`
- `Rectangle.asm` `ReactangleL.asm`
- `Pong.asm` `PongL.asm`

Test program: Add

Basic test of handling:

- White space
- Instructions

Test program:pong

Observations:

- Source code originally written in the Jack language
- The Hack code was generated by the Hack complier and the Hack assembler
- The resulting code is 28374 instructions long (includes the Jack OS)

Machine generated code:

- No white space
- "Strange" addresses
- "Strange" labels
- "Strange" pre-defined symbols

Testing options

Use your assembler to translate Xxx.asm

generating the executable file Xxx.hack

Hardware simulator:

load Xxx.hack into the Hack computer chip, then execute it

CPU Emulator:

load Xxx.hack into the supplied CPUEmulator, then execute it

Assembler:

use the supplied Assemblerto translate Xxx.asm

Compare the resulting code to the binary code generated by your assembler

Unit 6.6B: Project6 Overview: Without Programming

Project 6 for non-programers

- We give you a few very short assembly language programs, each in a **Xxx.asm** file
- For each one of these, you are supposed to write (using a simple text editor such as WordPad) its machine language translation into a corresponding file called **Xxx.hack** that can that be executed as it on the Hack computer
- How? Simulate by hand what the assembler program should do

Stages

---

1. Start by handling the easy case, where there happen not to be any symbols in the assembly program
2. Then, handle symbols:
    - Replace them with the correct numeric address
    - Now we are back to the easy case

Q&A

Can you possibly improve the symbolic Hack language without changing the binary code or the machine language which is underlying the symbolic level?

So indeed, we have two layers of of expression here. We have the symbolic level and we have the binary code that lies below it if you will. And the assembler bridges this difference. And, indeed, we can take this level, the symbolic level, and make it more user-friendly or more programmer-friendly. And there are various ways to do it. And in order to explain them, I will go to the board and and show you some examples. So how can we make the Hack symbolic language more programmer-friendly? Well what we can do is introduce the notions of macro assemblers and macro commands. So let me say a few words about this added layer of abstraction.Let's take a typical operation like loading the D register with a value of some memory register. It is quite natural to think about a command that looks like this. D equals M, let's say at 100, okay, which means go to register number 100, take its contents and put it in the D register. This will be a natural thing to write, and yet the standard, the Hack language does not feature such a, such a command. So what I can do is I can take a command like this and translate it into two valid commands in the Hack language, which will be at 100 and D equals M, okay? So this is an example of what is sometimes called a macro command. And I can decide that whenever I write this command, I actually mean that I want these two commands to get executed.Likewise, think about jump instructions. For example it would be very natural to say something like jump to this particular label. But once again in the Hack language such an instruction is not permitted. So instead, I can expect this instruction to be translated into two instructions, which would be @LOOP and followed by a standard jump command. So these are macro instructions. These are actual instructions. And in order to close this gap, I have to do something to my assembler. I have to extend the assembler. And and write it in such a way that whenever it encounter a command

like this, it translates it into two machine language commands rather than a single one, which is what we normally did in, in the standard assembler.

will I ever have to use an assembler outside school?

Well, the answer is exactly the same answer we gave to the question, will we ever need to use machine language that we gave in week four.

Very, very rarely. Most of the time, people write in high-level languages and do not worry about the machine language. When they do worry about the machine language of course they will do it in assembly language, and thus they will need to use an assembler. But this is very rare, only in the special cases where performance is of utmost criticality. Now in these cases, sometimes they will only want to focus on a very small part of the program and do it in assembly language. And they do rest in a high-level language. Eh, for example, C compilers allow you to do this kind of combination. To write most of the C lang, most of your program in the C language, and in it embed a few commands in assembly. So the assembler is sort of part of the C language compiler. And allows you to deal with very specific point of the program that you want to extremely optimize, and write them in assembly language. Beyond that, you will probably not worry that much, or at all, about assemblers or assembly language programming.

Now this course is about building computers. And we had the tremendous luxury of using other computers in order to build the Hack computer. And in particular, when we wrote the assembler we wrote it using a high-level language like Java, or Python, or whatever language students have chose have chosen to to use when they wrote the assembler. Those of you who actually programmed the assembler once again used high-level language to do it. So the question is historically what happened when the first computers were designed? When you didn't have this this tremendous luxury, how was the first assembler actually written?

So, this, this is a very mind bending question. Before I an, before I answer it I would like to re-emphasize again that conceptually, it only holds for the first time you write an assembler. In reality, most of the time there are already computers. And usually you use a computer that you already have with the high-level language that you already have to write code and assembler than basic fu, function and basically a functionality for the new computer. Now given that we are still talking about the first computer conceptually in history, how was that done? Well, in this course, you also saw that. Remember that the students that are not programmers in this course simply had to compile something by hand. They got an assembly language program and needed to comp, to translate it one, one command at a time into two machine language.That you can always do. So conceptually, what you can really do is you start to write the high-level assembler. You write an assembler in a high-level language and translate it by hand for the first time only into a machine language of your computer. Once you'll finish this translation, which is extremely time-consuming, extremely annoying to do, but only needs to be done once conceptually, then now you already have a machine language that runs your compiler or your assembler or any high-level help that you want. And then you can keep on using it for the rest of time.

Solution

- parser.py

```
import re


class Parser:
    def __init__(self, pos):
```

```
        with open(pos) as f:
            mylist = f.read().splitlines()
            alist = []
        for line in mylist:
            aline = line.split("//", 1)[0]
            alist.append(aline)
        self.fileList = [x.strip() for x in alist if x.strip() != '']
        self.currentLine = -1
        self.currentCommand = None

    def hasMoreCommands(self):
        if self.currentLine + 1 <= len(self.fileList) - 1:
            return True
        else:
            return False

    def advance(self):
        self.currentLine = self.currentLine + 1
        self.currentCommand = self.fileList[self.currentLine]

    def commandType(self):
        if re.match('@', self.currentCommand) is not None:
            return 'A_COMMAND'
        elif re.match('\(', self.currentCommand) is not None:
            return 'L_COMMAND'
        else:
            return 'C_COMMAND'

    def symbol(self):
        if self.commandType() == 'A_COMMAND':
            return self.currentCommand[1:]
        elif self.commandType() == 'L_COMMAND':
            return self.currentCommand[1:-1]
        else:
            return None

    def dest(self):
        if self.commandType() == 'C_COMMAND':
            positionOfEqual = self.currentCommand.find('=')
            if positionOfEqual != -1:
                return self.currentCommand[0:positionOfEqual].replace(" ", "")
            else:
                return 'null'

    def comp(self):
        if self.commandType() == 'C_COMMAND':
            positionOfEqual = self.currentCommand.find('=')
            positionOfsemicolons = self.currentCommand.find(';')
            if positionOfEqual == -1 and positionOfsemicolons == -1:
                return self.currentCommand.replace(" ", "")
            elif positionOfEqual == -1 and positionOfsemicolons != -1:
                return self.currentCommand[0:positionOfsemicolons].replace(" ",
""")
            elif positionOfEqual != -1 and positionOfsemicolons == -1:
```

```
                return self.currentCommand[positionOfEqual + 1:].replace(" ",
"")
            else:
                return self.currentCommand[positionOfEqual +
1:positionOfsemicolons].replace(" ", "")

    def jump(self):
        if self.commandType() == 'C_COMMAND':
            positionOfsemicolons = self.currentCommand.find(';')
            if positionOfsemicolons == -1:
                return 'null'
            else:
                return self.currentCommand[positionOfsemicolons + 1:].replace("
", "")
```

- code.py

```
class Code:
    def __init__(self):
        self.opcodeOfComp = {
            '0': '0101010',
            '1': '0111111',
            '-1': '0111010',
            'D': '0001100',
            'A': '0110000',
            'M': '1110000',
            '!D': '0001101',
            '!A': '0110001',
            '!M': '1110001',
            '-D': '0001111',
            '-A': '0110011',
            '-M': '1110011',
            'D+1': '0011111',
            'A+1': '0110111',
            'M+1': '1110111',
            'D-1': '0001110',
            'A-1': '0110010',
            'M-1': '1110010',
            'D+A': '0000010',
            'D+M': '1000010',
            'D-A': '0010011',
            'D-M': '1010011',
            'A-D': '0000111',
            'M-D': '1000111',
            'D&A': '0000000',
            'D&M': '1000000',
            'D|A': '0010101',
            'D|M': '1010101'
        }
        self.opcodeOfDest = {
            'null': '000',
            'M': '001',
            'D': '010',
            'MD': '011',
            'A': '100',
```

```
            'AM': '101',
            'AD': '110',
            'AMD': '111'
        }
        self.opcodeOfJump = {
            'null': '000',
            'JGT': '001',
            'JEQ': '010',
            'JGE': '011',
            'JLT': '100',
            'JNE': '101',
            'JLE': '110',
            'JMP': '111'
        }

    def dest(self, strDest):
        return self.opcodeOfDest.get(strDest)

    def comp(self, strComp):
        return self.opcodeOfComp.get(strComp)

    def jump(self, strJump):
        return self.opcodeOfJump.get(strJump)
```

- symboltable.py

```
class SymbolTable:
    def __init__(self):
        self.symbolTable = {
            'R0': '0',
            'R1': '1',
            'R2': '2',
            'R3': '3',
            'R4': '4',
            'R5': '5',
            'R6': '6',
            'R7': '7',
            'R8': '8',
            'R9': '9',
            'R10': '10',
            'R11': '11',
            'R12': '12',
            'R13': '13',
            'R14': '14',
            'R15': '15',
            'SCREEN': '16384',
            'KBD': '24576',
            'SP': '0',
            'LCL': '1',
            'ARG': '2',
            'THIS': '3',
            'THAT': '4'
        }

    def addEntry(self, strSymbol, strAddress):
```

```
        self.symbolTable[strSymbol] = strAddress

    def isContains(self, strSymbol):
        if self.symbolTable.get(strSymbol) is None:
            return False
        else:
            return True

    def getAddress(self, strSymbol):
        return self.symbolTable.get(strSymbol)
```

- main.py

```
from hackParser import Parser
from code import Code
from symboltable import SymbolTable
import os


def to_bin(value, num):  # 十进制数据，二进制位宽
    bin_chars = ""
    temp = value
    for i in range(num):
        bin_char = bin(temp % 2)[-1]
        temp = temp // 2
        bin_chars = bin_char + bin_chars
    return bin_chars.upper()  # 输出指定位宽的二进制字符串


def executeStepByStep(filePosition):
    toPosition = os.path.splitext(filePosition)[0] + '.hack'
    parser = Parser(filePosition)
    code = Code()
    symbolTable = SymbolTable()
    actualLine = 0
    while parser.hasMoreCommands():
        parser.advance()
        if parser.commandType() == 'L_COMMAND':
            strSymbol = parser.symbol()
            strAddress = str(actualLine)
            symbolTable.addEntry(strSymbol, strAddress)
        else:
            actualLine = actualLine + 1
    parser.currentLine = -1
    parser.currentCommand = None
    serialNumber = 16
    with open(toPosition, mode='a+') as f:
        f.seek(0)
        f.truncate()
        while parser.hasMoreCommands():
            parser.advance()
            if parser.commandType() == 'A_COMMAND':
                mystr = parser.symbol()
                if mystr.isdigit():
                    number = int(mystr)
```

```
                    out = '0' + to_bin(number, 15)
                    f.write(out + '\n')
                else:
                    if symbolTable.isContains(mystr):
                        address = symbolTable.getAddress(mystr)
                        number = int(address)
                        out = '0' + to_bin(number, 15)
                        f.write(out + '\n')
                    else:
                        symbolTable.addEntry(mystr, str(serialNumber))
                        out = '0' + to_bin(serialNumber, 15)
                        f.write(out + '\n')
                        serialNumber = serialNumber+1
            elif parser.commandType() == 'C_COMMAND':
                c = parser.comp()
                d = parser.dest()
                j = parser.jump()
                cc = code.comp(c)
                dd = code.dest(d)
                jj = code.jump(j)
                out = '111' + cc + dd + jj
                f.write(out + '\n')
            else:
                pass


if __name__ == '__main__':
    executeStepByStep('E:/HackAssembler/test/max/Max.asm')
```