

ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH  
TRƯỜNG ĐẠI HỌC BÁCH KHOA  
KHOA KHOA HỌC VÀ KỸ THUẬT MÁY TÍNH



## KIẾN TRÚC MÁY TÍNH (CO2011)

---

### NAND2TETRIS

---

GVHD: Trần Thanh Bình  
SVTH: Nguyễn Đắc Hoàng Phú - 2010514

Hồ Chí Minh, tháng 12/2021

## Contents

<b>1</b>	<b>GIỚI THIỆU TỔNG QUAN:</b>	<b>3</b>
1.1	MỤC ĐÍCH NAND2TETRIS . . . . .	3
1.2	NỘI DUNG TỔNG QUÁT . . . . .	3
1.3	PROJECT TRONG KHÓA HỌC . . . . .	4
1.4	NHỮNG TOOL HỖ TRỢ ĐI KÈM TRONG KHÓA THỨ NHẤT: . . . . .	5
1.4.1	HardwareSimulator: . . . . .	5
1.4.2	Assembler: . . . . .	6
1.4.3	CPUEmulator: . . . . .	6
1.4.4	Biểu tượng của các tool ở local . . . . .	7
1.5	CHIP INTERFACE: . . . . .	7
<b>2</b>	<b>WEEK1: BOOLEAN FUNCTIONS AND GATE LOGIC</b>	<b>8</b>
2.1	ĐẠI SỐ BOOL (BOOLEAN ALGEBRA) . . . . .	8
2.2	CỔNG LOGIC(LOGIC GATE) . . . . .	9
2.3	HARDWARE DESCRIPTION LANGUAGE (HDL): . . . . .	10
2.4	PROJECT . . . . .	12
2.4.1	Mux.hdl . . . . .	12
2.4.2	Mux16.hdl . . . . .	13
2.4.3	Mux-nWay16.hdl . . . . .	14
<b>3</b>	<b>BOOLEAN ARITHMETIC AND THE ALU</b>	<b>15</b>
3.1	PHÉP CỘNG (ADDITION): . . . . .	15
3.2	BIỂU DIỄN SỐ ÂM (SIGN INTEGER): . . . . .	15
3.3	VON-NEUMANN ARCHITECTURE . . . . .	17
3.4	ALU & HACK ALU . . . . .	17
3.5	Project . . . . .	18
3.5.1	Adder: . . . . .	19
3.5.2	ALU.hdl . . . . .	21
<b>4</b>	<b>WEEK 3 - SEQUENTIAL LOGIC</b>	<b>23</b>
4.1	Clock . . . . .	23
4.2	Flip-Flops . . . . .	23
4.3	Register . . . . .	24
4.4	MEMORY (RAM) . . . . .	24
4.5	PROGRAM COUNTER (PC): . . . . .	25
4.6	Project . . . . .	26
4.6.1	register: . . . . .	26
4.6.2	Ram: . . . . .	27
4.6.3	PC: . . . . .	28
<b>5</b>	<b>WEEK4 - MACHINE LANGUAGE</b>	<b>29</b>
5.1	ĐỊNH NGHĨA: . . . . .	29
5.2	CÁC LOẠI CÂU LỆNH: . . . . .	29
5.3	MÁY TÍNH HACK . . . . .	30
5.3.1	Không gian địa chỉ bộ nhớ: . . . . .	30
5.3.2	Thanh ghi: . . . . .	30
5.4	NGÔN NGỮ MÁY HACK . . . . .	31
5.4.1	Câu lệnh A - (A instruction) . . . . .	31
5.4.2	Câu lệnh C - (C instruction) . . . . .	32
5.4.2.a	comp - field . . . . .	32
5.4.2.b	dest - field . . . . .	33
5.4.2.c	jump-field . . . . .	33
5.4.3	Symbol . . . . .	34
5.5	PROJECT . . . . .	34

5.5.1	Mult . . . . .	34
5.5.2	Fill . . . . .	35
<b>6</b>	<b>WEEK 5 - COMPUTER ARCHITECTURE</b>	<b>37</b>
6.1	CPU . . . . .	37
6.2	INSTRUCTION DECODING - Giải mã lệnh . . . . .	37
6.3	INSTRUCTION EXECUTION - Thực thi lệnh . . . . .	37
6.4	INSTRUCTION FETCHING - NẠP LỆNH . . . . .	38
6.5	INPUT/OUTPUT DEVICE . . . . .	39
6.5.1	SCREEN . . . . .	39
6.5.2	KEYBOARD . . . . .	39
6.6	Data Memory . . . . .	40
6.7	COMPUTER . . . . .	40
6.8	PROJECT . . . . .	41
6.8.1	CPU.hdl . . . . .	41
6.8.2	Memory.hdl . . . . .	43
6.8.3	Computer.hdl . . . . .	43
<b>7</b>	<b>WEEK 6 - ASSEMBLER</b>	<b>44</b>
7.1	ĐỊNH NGHĨA: . . . . .	44
7.2	PREPROCESS: . . . . .	44
7.3	LEXER AND PARSER MODULE: . . . . .	45
7.4	SYMBOL TABLE . . . . .	45
7.5	PROJECT . . . . .	46
<b>8</b>	<b>REFERENCES</b>	<b>46</b>

# 1 GIỚI THIỆU TỔNG QUAN:

## 1.1 MỤC ĐÍCH NAND2TETRIS

Đây là một khóa học được cung cấp trên nền tảng Coursera với nội dung về xây dựng từng tầng của một chiếc máy tính từ những cổng cơ bản nhất như NOT, NAND đến cả các bộ xử lý như CPU, MEMORY cũng như các trình biên dịch cơ bản như ASSEMBLER, COMPILER,... Rồi từ đó sau khi hoàn thành xong khóa học chúng ta sẽ xây dựng được một máy tính 16bit có thể làm bất cứ việc gì như các máy tính hiện nay vậy. Đặc biệt, chúng ta sẽ không cần vật liệu vật lý, vì sẽ xây dựng máy tính trực tiếp trên PC của riêng mình, sử dụng các trình mô phỏng phần cứng được cung cấp sẵn dựa trên phần mềm, giống như máy tính thực được thiết kế bởi các kỹ sư máy tính trong lĩnh vực này. Bên cạnh việc xây dựng chúng ta sẽ học được thêm cách mà chiếc máy tính hoạt động, hiểu được quá trình biên dịch từ những dòng lệnh hết sức trừu tượng đến những dãy số nhị phân 0, 1 liên tiếp nhau, hiểu được quá trình cấp phát bộ nhớ và cả cách làm cách nào mà hình ảnh lại hiển thị được trên màn hình máy tính dù đơn vị cơ bản nhất của nó chỉ là một pixel nhỏ bé .Bấy nhiêu đó thôi, cũng là cơ hội để ta có thể nâng cao thêm kiến thức trong ngành khoa học máy tính cũng như nâng cao kĩ năng làm việc với các phần mềm, ngôn ngữ lập trình.



(a) giáo sư Shimon Schoken



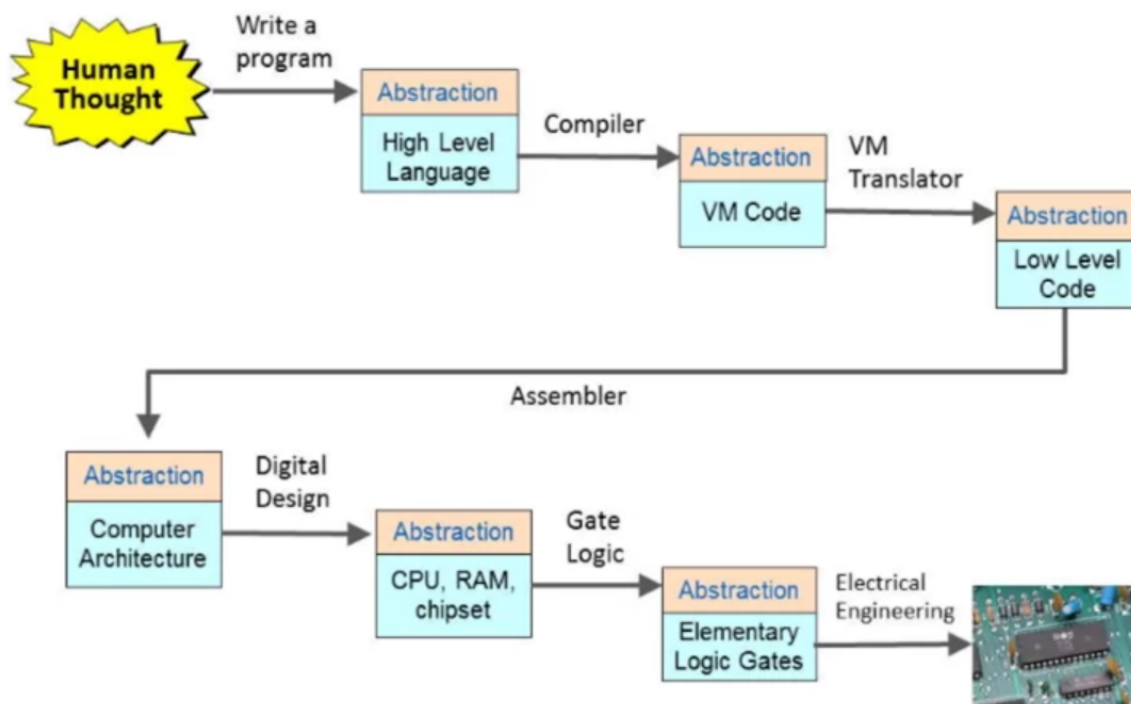
(b) giáo sư noam-nisan

**Figure 1:** các tác giả chính khóa học Nand2Teris đến từ Hebrew University of Jerusalem

## 1.2 NỘI DUNG TỔNG QUÁT

Nand2Teris được chia làm hai khóa chính được thể hiện trên 2 phần khác nhau.

- + **Khóa thứ nhất sẽ nói về phần cứng**, về luồng thực thi các con đường từ đại số Boolean và các cổng logic cơ bản đến xây dựng bộ xử lý trung tâm (CPU), hệ thống bộ nhớ (Memory) và nền tảng phần cứng, dẫn đến một máy tính đa năng có thể chạy bất kỳ chương trình nào mà bạn yêu thích. Trong quá trình xây dựng chiếc máy tính này, bạn sẽ làm quen với nhiều phần cứng trừu tượng quan trọng, và bạn sẽ thực hiện chúng một cách thực hành, được tự tay xây dựng nó từ những phần cơ bản nhất đến những phần phức tạp nhất. Chúng ta sẽ được làm quen với một hợp ngữ Hack, được hiểu cách dịch một mã máy và cách xây dựng một Assembler đơn giản để có thể "thực hiện" giao tiếp với máy tính.
- + **Khóa thứ hai sẽ nói về phần mềm**, về việc xây dựng hệ thống phân cấp phần mềm hiện đại, được thiết kế để cho phép dịch và thực thi các ngôn ngữ cấp cao hướng đối tượng trên nền tảng phần cứng máy tính đơn giản. Cụ thể, chúng ta sẽ triển khai một máy ảo và một trình biên dịch cho một ngôn ngữ lập trình đơn giản, giống như Java, và bạn sẽ phát triển một hệ điều hành cơ bản giúp thu hẹp khoảng cách giữa ngôn ngữ cấp cao và nền tảng phần cứng bên dưới. Bên cạnh đó chúng ta còn tìm hiểu được stack-processing, syntax analysis, thuật toán và cấu trúc dữ liệu để quản lý bộ nhớ, đồ họa vector, xử lý đầu vào đầu ra và những chủ đề cốt lõi nhất của Khoa học máy tính.



**Figure 2:** Tổng quan về các phần trong Nand2Teris

### 1.3 PROJECT TRONG KHÓA HỌC

**Khóa thứ nhất sẽ bao gồm 6 tuần để hoàn thành:** mỗi một tuần đều có các project liên quan đến chủ đề của tuần học đó. Các chủ đề và project liên quan được liệt kê cụ thể theo thứ tự như sau:

1. Xây dựng các cổng logic cơ bản như And, Or, Not, Multiplexor,...
2. Xây dựng các chip, đơn vị logic số học (ALU).
3. Xây dựng thanh ghi và đơn vị bộ nhớ, đặc biệt là xây dựng Bộ nhớ truy cập ngẫu nhiên (RAM).
4. Học ngôn ngữ máy HACK và sử dụng nó để viết một số chương trình cấp thấp.
5. Sử dụng chipset được tích hợp trong các dự án 1-3 để xây dựng Bộ xử lý trung tâm (CPU) và nền tảng phần cứng có khả năng thực thi các chương trình được viết bằng ngôn ngữ máy Hack.
6. Xây dựng và phát triển Assembler, tức là khả năng dịch các chương trình được viết bằng ngôn ngữ máy tương đương thành mã nhị phân, có thể thực thi được.

**Tương tự khóa thứ hai cũng gồm 6 tuần để hoàn thành** với các chủ đề cụ thể trong mỗi project như sau:

1. Hiểu về virtual-machine và xây dựng VM-translator sử dụng stack để dịch các phép tính số học, các câu lệnh nhảy bước, ..
2. Tiếp tục xây dựng VM-translator để dịch các bước lặp, các câu lệnh if else, các thủ tục gọi hàm
3. Tìm hiểu về các ngôn ngữ bậc cao, lấy Jack-language làm ví dụ và sử dụng để viết một số chương trình cơ bản.
4. Bắt đầu xây dựng compiler, tìm hiểu về syntax analysis bao gồm hai giai đoạn phụ: phân tích từ vựng lexical analysis (tokenizing trên từ ngữ) và parsing (tokenizing trên một câu). Tạo ra một chương trình tiết lộ cú pháp của các chương trình Jack mà không tạo ra mã thực thi.

5. Xây dựng tiếp phần thứ hai của compiler - code generation, tìm cách mô tả cách tạo mã VM để dịch các chương trình thủ tục thành các chương trình VM và cách tạo mã VM để xây dựng và thao tác các mảng và đối tượng. => tạo được một trình biên dịch đầy đủ cho ngôn ngữ lập trình Jack.
6. Tìm hiểu và xây dựng hệ điều hành và hoàn thiện máy tính Hack.

## 1.4 NHỮNG TOOL HỖ TRỢ ĐI KÈM TRONG KHÓA THỨ NHẤT:

### 1.4.1 HardwareSimulator:

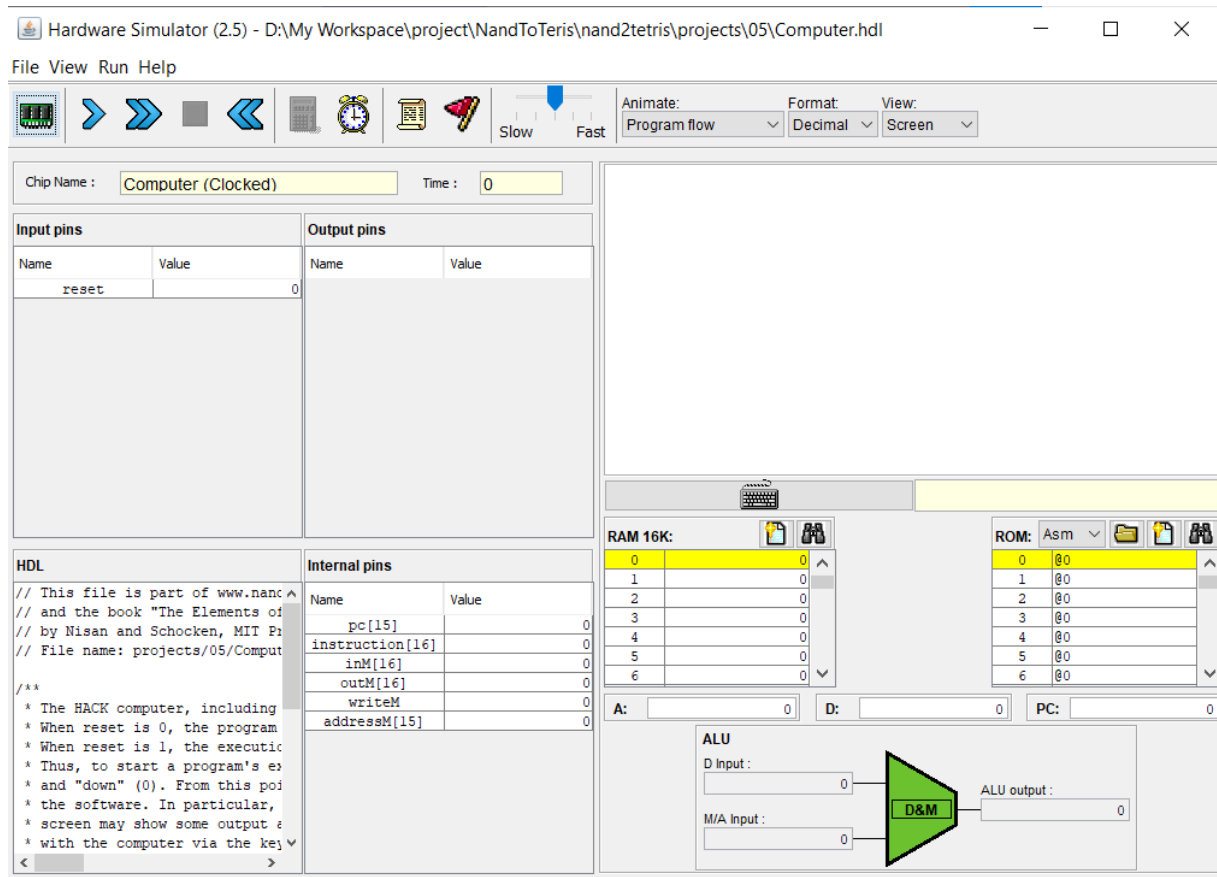
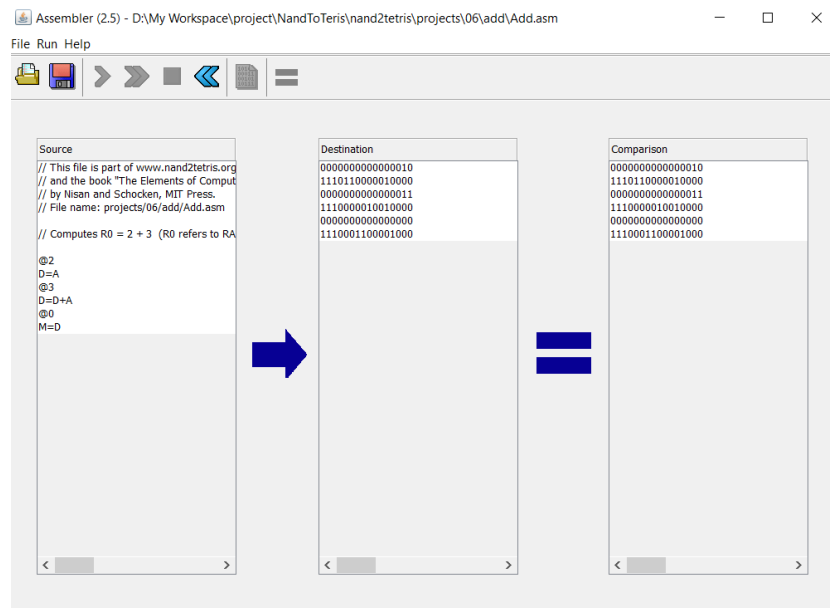


Figure 3: Trình mô phỏng phần cứng

- + **Thanh Taskbar:** Dùng để test các file có đuôi .hdl (là file text mô phỏng chức năng của các bộ phận phần cứng) với testcase được chứa trong file cùng tên có đuôi .tst với out put là kết quả của các bộ test đã được cung cấp đó. Ngoài ra có thể so sánh trực tiếp kết quả với file kết quả test(.cmp)
- + **Màn hình chính bên trái:** Bảng input pins giúp ta có thể test trực tiếp phần cứng bằng cách nhập testcase từ bàn phím. Khi đó output sẽ có 2 dạng: kết quả cuối cùng được thể hiện trong input pins, còn các kết quả thành phần (đầu ra các luồng thực thi) được hiển thị trong internal pins.
- + **Màn chính bên phải:** đối với mỗi bộ phận phần cứng để test sẽ hiển thị những thành phần bên trong của nó như với hình ảnh là thanh ram sẽ gồm những ô nhớ và check xem các ô nhớ đó đang chứa gì còn với PC-counter sẽ hiển thị những câu lệnh nào được ghi vào trong đó. Còn những thành phần cơ bản như các cổng sẽ không hiển thị.

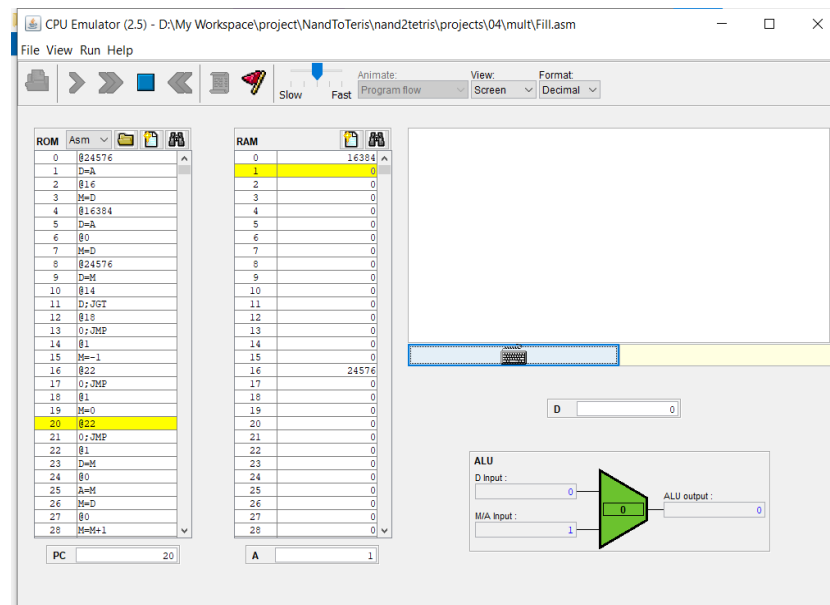
### 1.4.2 Assembler:



**Figure 4:** Trình test chương trình dịch từ hợp ngữ sang ngôn ngữ máy

Dùng để test các file có đuôi .asm (hợp ngữ) sang ngôn ngữ máy. Như các bạn có thể thấy phần source là file hợp ngữ, phần destination sẽ là kết quả được dịch và có thể so sánh kết quả đó với file comparison có trong .comp. Bất cứ khi nào chương trình cho ra kết quả dòng lệnh dịch sai, phần mềm ngay tức khắc sẽ dừng lại để ta biết lỗi nằm ở phần nào













### 1.4.3 CPUEmulator:



**Figure 5:** Trình để test CPU

Console với ROM là các câu lệnh trong file hợp ngữ sau khi đã bỏ đi các dòng trống và các comment. RAM dùng để thể hiện quá trình lưu trữ, ghi và đọc bộ nhớ khi thực hiện các câu lệnh trên và phần ALU bên dưới dùng để hiển thị trực tiếp các quá trình với những câu lệnh tính toán.

#### 1.4.4 Biểu tượng của các tool ở local

	VMEulator.sh	3/12/2016 1:29 PM	Shell Script	2 KB
	VMEulator.bat	3/12/2016 1:29 PM	Windows Batch File	1 KB
	TextComparer.sh	3/12/2016 1:29 PM	Shell Script	1 KB
	TextComparer.bat	3/12/2016 1:29 PM	Windows Batch File	1 KB
	JackCompiler.sh	3/12/2016 1:29 PM	Shell Script	2 KB
	JackCompiler.bat	3/12/2016 1:29 PM	Windows Batch File	1 KB
	HardwareSimulator.sh	3/12/2016 1:29 PM	Shell Script	2 KB
	HardwareSimulator.bat	3/12/2016 1:29 PM	Windows Batch File	2 KB
	CPUEmulator.sh	3/12/2016 1:29 PM	Shell Script	2 KB
	CPUEmulator.bat	3/12/2016 1:29 PM	Windows Batch File	1 KB
	Assembler.sh	3/12/2016 1:29 PM	Shell Script	2 KB
	Assembler.bat	3/12/2016 1:29 PM	Windows Batch File	1 KB

**Figure 6:** Các tool được cung cấp bởi Nand2Teris, đều được viết bằng Java

(\*) **NOTE: giải thích các file .sh:** Shell là chương trình người dùng đặc biệt, cung cấp giao diện cho người dùng sử dụng các dịch vụ hệ điều hành. Shell chấp nhận các lệnh có thể đọc được từ người dùng và chuyển đổi chúng thành thứ mà kernel có thể hiểu được. Vì shell cũng có thể nhận các lệnh làm đầu vào từ file, chúng ta có thể viết các lệnh trong một file và có thể thực thi chúng trong shell, tránh các công việc lặp đi lặp lại. Các file này được gọi là Shell Script hoặc Shell Programs. Các Shell script tương tự như batch file trong MS-DOS. Mỗi shell script được lưu với phần mở rộng tệp .sh.

### 1.5 CHIP INTERFACE:

Là một file chứa tất cả prototype của các chip, giúp dễ dàng vận dụng tránh những lỗi sai syntax. [Hack-chip set API](#) Một số API thông dụng:

```
Add16(a= ,b= ,out= ) /* Adds up two 16-bit two's complement values */
ALU(x= ,y= ,zx= ,nx= ,zy= ,ny= ,f= ,no= ,out= ,zr= ,ng= ) /* Hack ALU */
And(a= ,b= ,out= ) /* And gate */
And16(a= ,b= ,out= ) /* 16-bit And */
ARegister(in= ,load= ,out= ) /* Address register (built-in) */
Bit(in= ,load= ,out= ) /* 1-bit register */
CPU(inM= ,instruction= ,reset= ,outM= ,writeM= ,addressM= ,pc= ) /* Hack CPU */
DFF(in= ,out= ) /* Data flip-flop gate (built-in) */
DMux(in= ,sel= ,a= ,b= ) /* Channels the input to one out of two outputs */
DMux4Way(in= ,sel= ,a= ,b= ,c= ,d= ) /* Channels the input to one out of four outputs */
DRegister(in= ,load= ,out= ) /* Data register (built-in) */
HalfAdder(a= ,b= ,sum= ,carry= ) /* Adds up 2 bits */
FullAdder(a= ,b= ,c= ,sum= ,carry= ) /* Adds up 3 bits */
Inc16(in= ,out= ) /* Sets out to in + 1 */
Keyboard(out= ) /* Keyboard memory map (built-in) */
Memory(in= ,load= ,address= ,out= ) /* Data memory of the Hack platform (RAM) */
Mux(a= ,b= ,sel= ,out= ) /* Selects between two inputs */
Mux16(a= ,b= ,sel= ,out= ) /* Selects between two 16-bit inputs */
Mux4Way16(a= ,b= ,c= ,d= ,sel= ,out= ) /* Selects between four 16-bit inputs */
Nand(a= ,b= ,out= ) /* Nand gate (built-in) */
Not16(in= ,out= ) /* 16-bit Not */
Not(in= ,out= ) /* Not gate */
```



## 2 WEEK1: BOOLEAN FUNCTIONS AND GATE LOGIC

Vì cổng Boolean là triển khai vật lý của các hàm Boolean, chúng ta sẽ bắt đầu làm quen xử lý ngắn gọn của đại số Boolean. Sau đó là tìm ra cách các cổng Boolean thực hiện các chức năng Boolean đơn giản có thể được kết nối với nhau để cung cấp chức năng của các chip phức tạp hơn.

### 2.1 ĐẠI SỐ BOOL (BOOLEAN ALGEBRA)

**Đại số boolean:** xử lý các giá trị Boolean (còn được gọi là giá trị nhị phân) thường là được gắn nhãn true / false, 1/0, yes / no, on / off,.. Nhưng thường được biểu diễn ở dạng 0 và 1. Một boolean function là tổ hợp các biến boolean với đầu vào là giá trị nhị phân và đầu ra cũng giữ giá trị nhị phân. Vì phần cứng máy tính dựa trên sự biểu diễn và thao tác của hệ nhị phân nên các giá trị của các hàm Boolean đóng một vai trò trung tâm trong đặc điểm kỹ thuật, cấu trúc và tối ưu hóa các kiến trúc phần cứng. Do đó, khả năng hình thành và phân tích các hàm boolean là bước đầu để xây dựng các kiến trúc máy tính.

**Bảng chân trị (Truth Table Representation):** Cách đơn giản nhất để chỉ định một hàm Boolean là liệt kê tất cả các giá trị có thể có của các biến đầu vào của hàm, cùng với đầu ra của chức năng cho từng tập hợp đầu vào. Đây được gọi là biểu diễn bảng chân trị của boolean function.

$x$	$y$	$z$	$f(x, y, z)$
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	0

Figure 7: Bảng chân trị

Ba cột đầu tiên của bảng đếm tất cả các trường hợp có thể có của đầu vào. Với  $2^n$  trường hợp nếu đầu vào là  $n$  biến. Mỗi một đầu vào sẽ cho ra một đầu ra nhất định được thể hiện ở cột cuối cùng

#### Biểu thức Boolean (Boolean Expression)

Một hàm Boolean cũng có thể được chỉ định bằng cách sử dụng các phép toán Boolean trên các biến đầu vào. Các hàm boolean cơ bản nhất với 2 biến đầu vào  $x, y$ :

- + **"AND"**:  $x$  And  $y$  bằng 1 khi chỉ khi cả hai biến  $x$  và  $y$  bằng 1 các trường hợp còn lại bằng 0. Biểu diễn đại số là:  $x * y$
- + **"OR"**:  $x$  Or  $y$  bằng 0 khi chỉ khi cả hai biến  $x$  và  $y$  bằng 0 các trường hợp còn lại bằng 1. Biểu diễn đại số là:  $x + y$
- + **"NOT"**: not  $x$  bằng 0 khi chỉ khi  $x = 1$  và ngược lại bằng 1 khi chỉ khi  $x = 0$ . Biểu diễn đại số là:  $\neg x$

Hàm boolean biểu diễn bằng bảng chân trị ở hình 7 có dạng  $f(x, y, z) = (x \text{ or } y) \text{ and } (\text{not } z)$ . Ta sẽ đi kiểm chứng điều này với một vài trường hợp cụ thể:

- + Với  $x = 0, y = 1, z = 0$  thì  $x \text{ or } y$  là 1 và  $\text{not } z = 1$ . Suy ra:  $f(x, y, z) = 1$
- + Với  $x = 1, y = 1, z = 1$  thì  $x \text{ or } y$  là 1 và  $\text{not } z = 0$ . Suy ra:  $f(x, y, z) = 0$

Thử từng trường hợp cụ thể ta đều xác minh được tính đúng đắn của biểu thức. Dựa vào bảng có một số hàm boolean thông dụng khác như sau:

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

Figure 8: Hàm boolean thông dụng 2 biến

- + Hàm Nor: là tên viết tắt của "not or" : lấy or của x và y, sau đó phủ định kết quả.
- + Hàm Nand: là tên viết tắt của "not and" : lấy and của x và y, sau đó phủ định kết quả.
- + Hàm Xor: là viết tắt của "exclusive or" trả về 1 nếu x và y khác nhau. Ngược lại trả về 0.
- + Hàm Equivalence: trả về 1 nếu hai biến có chân trị như nhau ngược lại trả về 0.

**Note:** Đặc biệt Hàm Nand và Nor còn gọi là Universal Unit Gates vì các hàm chứa toán tử And, Not, Or đều có thể được xây dựng từ nó. Điều đó suy ra được rằng: mọi hàm Boolean đều có thể xây dựng chỉ từ hoạt động của cổng. Bằng cách chỉ xài cổng Nand, ta có thể sử dụng nhiều bản sao của thiết bị này để triển khai một phần cứng được biểu diễn bởi hàm Boolean.

## 2.2 CỔNG LOGIC(LOGIC GATE)

Một cổng là một thiết bị vật lý đại diện cho một hàm boolean. Nếu một hàm boolean nhận vào n biến và trả ra m giá trị thì cổng cũng có n chân đầu vào và m chân đầu ra. Và giống như các hàm Boolean phức tạp là tổ hợp các hàm Boolean đơn giản hơn thì các cổng phức tạp được cấu tạo từ các cổng sơ cấp hơn. Những chiếc cổng đơn giản nhất như not, and, or nand được cấu tạo. Những chiếc cổng đơn giản nhất được làm từ những chiếc thiết bị chuyển mạch được gọi là bóng bán dẫn. Đại số Boolean có thể được sử dụng để tóm tắt hành vi của bất kỳ các mặt công nghệ.

- + **Cổng tiêu chuẩn (Primitive-gate):** là các cổng cơ bản nhất mà không cấu tạo bằng cách kết hợp nhiều cổng lại với nhau. Ví dụ: cổng Not, And Or. Đặc biệt do tính thông dụng của các cổng Nand và Nor mặc dù có biểu thức luận lý kết hợp từ các hàm And, Not, Or nhưng nó vẫn được thiết kế là một cổng tiêu chuẩn.

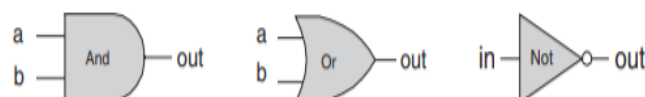
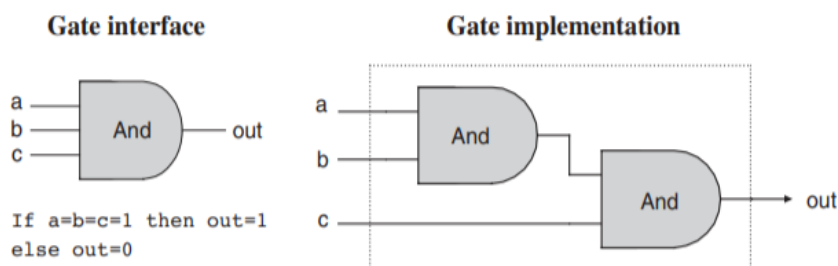
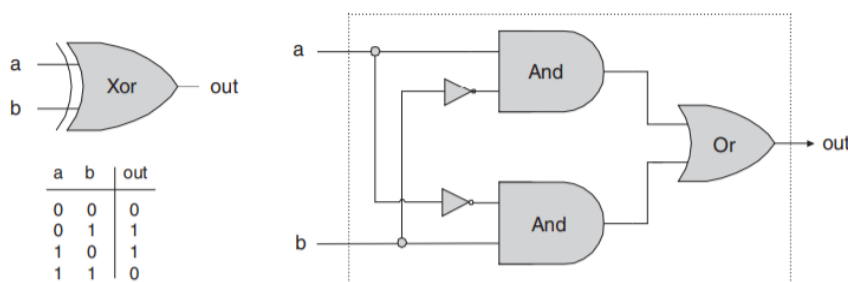


Figure 9: 3 ký hiệu tiêu chuẩn của một số cổng logic cơ bản

- + **Cổng kết hợp (Composit-gate)** là các cổng kết hợp từ 2 hay nhiều cổng lại với nhau. Diễn hình như thiết kế các cổng như n-way gate, Xor, Xnor như hình bên trên.



**Figure 10:** Cổng kết hợp And three-way. Hình chữ nhật bên phải là ranh giới giao diện cổng



**Figure 11:** Thiết kế cổng Xor

## 2.3 HARDWARE DESCRIPTION LANGUAGE (HDL):

**Vấn đề:** Nếu ta bắt tay vào làm các cổng hay mạch vật lý bằng tay không trước thì dễ xảy ra sai sót dẫn đến tốn kém về mặt thời gian sửa chữa, lắp ráp kèm theo là tốn kém chi phí vì phần cứng rất khó để tháo lắp sau khi sử dụng các biện pháp hàn, hồ và đương nhiên việc tối ưu công đoạn rất khó để đạt được.

Do đó để khắc phục, ta lập kế hoạch và tối ưu hóa kiến trúc chip trên các phần mềm máy tính trước bằng các hình thức mô hình hóa có cấu trúc như Ngôn ngữ mô tả phần cứng hay còn được gọi là HDL (cũng được gọi là VHDL, trong đó V là viết tắt của Virtual). Nhà thiết kế chỉ định cấu trúc chip bằng cách viết một chương trình HDL. Xây dựng hình ảnh của chip được mô hình hóa trong bộ nhớ. Tiếp theo, nhà thiết kế có thể debug bằng trình mô phỏng để kiểm tra chip ảo trên các bộ đầu vào khác nhau, tạo ra chip mô phỏng kết quả đầu ra. Sau đó, kết quả đầu ra có thể được so sánh với kết quả mong muốn, theo yêu cầu của khách hàng đã đặt hàng xây dựng chip.

Đây là một ngôn ngữ lập trình hình thức được viết bằng file tex, không có trình biên dịch đi kèm như C++ hay Java. Để kiểm tra tính đúng đắn của nó, Nand2Teris đã cung cấp tool HardwareSimulator cùng testcase có sẵn trong file .tst. Ngôn ngữ HDL gồm hai phần: phần tiêu đề (mô tả) và phần thân (phần lập trình).

- + **Phần tiêu đề:** chỉ định giao diện chip, cụ thể là tên chip và tên của các chân đầu vào và đầu ra của nó. Phần comment tuy không cần thiết nhưng nó giúp cho việc đọc hiểu giải thuật, những thành phần chính và đầu ra mong muốn.
- + **Phần thân:** mô tả tên và cấu trúc liên kết của tất cả các bộ phận cấp thấp hơn (các chip khác) mà từ đó chip này được cấu tạo. Mỗi một phần được thể hiện bằng một câu lệnh chỉ định tên bộ phận và cách nó được kết nối với các bộ phận khác trong thiết kế.

```
// This file is part of www.nand2tetris.org
// and the book "The Elements of Computing Systems"
// by Nisan and Schocken, MIT Press.
// File name: projects/01/DMux.hdl
```

```
/**
 * Demultiplexor:
 * {a, b} = {in, 0} if sel == 0
 *          {0, in} if sel == 1
 */
```

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

(a) Phần tiêu đề của HDL

PARTS:

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

```
Not(in = sel, out = selNot);
```

```
And(a = in, b = selNot, out = a);
```

```
And(a = in, b = sel, out = b);
```

```
}
```

(b) Phần thân file HDL

Figure 12: Các thành phần của file HDL

**Lưu ý:** để viết các câu lệnh như vậy một cách dễ hiểu, người lập trình HDL phải có một tài liệu đầy đủ về giao diện các bộ phận.

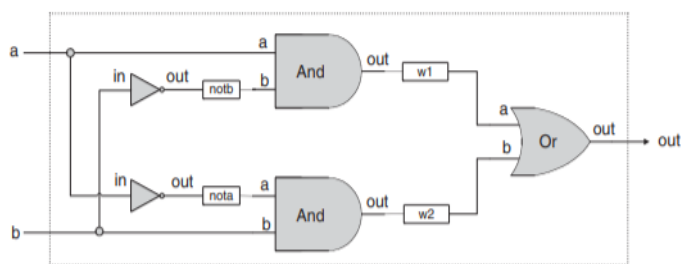


Figure 13: Giao diện bộ phận của thiết bị Xor

Còn về các file chứa testcase và debug được đính kèm trong folder của tuần 1.

<i>Test script (Xor.tst)</i>	<i>Output file (Xor.out)</i>
load Xor.hdl,	a   b   out
output-list a, b, out;	0   0   0
set a 0, set b 0,	0   1   1
eval, output;	1   0   1
set a 0, set b 1,	1   1   0
eval, output;	
set a 1, set b 0,	
eval, output;	
set a 1, set b 1,	
eval, output;	

Figure 14: Hình ảnh file test và file output

## 2.4 PROJECT

Trong tuần thứ nhất, ta sẽ lần lượt thực thi các mẫu thiết kế thiết bị phần cứng đơn giản như cổng Not, And, DMux, Mux,... từ duy nhất cổng Nand được cung cấp sẵn. Cụ thể các file ta cần làm đó là: Ta sẽ



(a) các file .hdl cần hiện thực

(b) các file liên quan để so sánh kết quả

Figure 15: Project 1

lấy thiết kế chip Mux làm ví dụ để hình dung cần phải làm như thế nào trong file hdl cũng như xây dựng các chip khác phức tạp hơn từ chip Mux.

### 2.4.1 Mux.hdl

Bộ ghép kênh (Multiplexer) là một cổng ba đầu vào sử dụng một trong các đầu vào, được gọi là "bit lựa chọn" (Trong file kí hiệu là sel), để chọn và xuất một trong hai đầu vào khác, được gọi là "Dữ liệu bit" (Trong file kí hiệu là a,b).

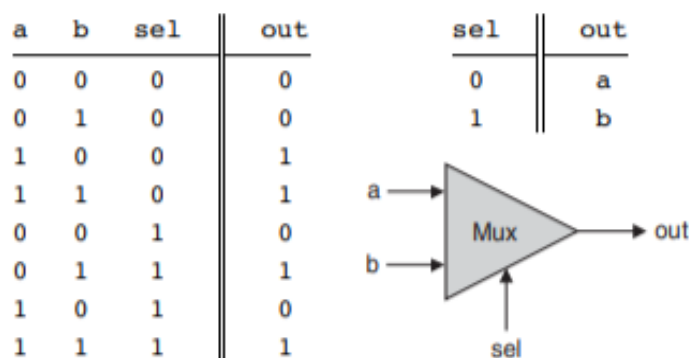


Figure 16: Thiết kế và bảng chân trị của chip Mux

```

1 /**
2  * Multiplexor:
3  * out = a if sel == 0
4  *     b otherwise
5  */
6
7 CHIP Mux {

```

```
8 //Header file
9     IN a, b, sel;
10    OUT out;
11 //Section file
12 PARTS:
13     // Put your code here:
14     // Function: Mux(a,b) = a * not_sel + b * sel
15     Not(in = sel,out = selNot);
16     And(a = a,b = selNot,out = muxA);
17     And(a = b,b = sel,out = muxB);
18     Or(a = muxA,b = muxB,out = out);
19 }
```

**Note:** Phần cứng máy tính thường được thiết kế để hoạt động trên mảng nhiều bit được gọi là "bus", Khi đề cập đến các bit riêng lẻ trong một bus, người ta thường sử dụng cú pháp mảng. Ví dụ với máy tính Hack 16bit ta đang xây dựng: để tham chiếu đến các bit riêng lẻ trong bus có 16 bit. Ta sử dụng ký hiệu: data[0], data[1], ..., data[15],...

#### 2.4.2 Mux16.hdl

Bộ ghép kênh n-bit hoàn toàn giống với bộ ghép kênh nhị phân, ngoại trừ hai đầu vào đều có độ rộng n-bit; bộ chọn là một bit duy nhất (sel)

```
1 /**
2  * 16-bit multiplexor:
3  * for i = 0..15 out[i] = a[i] if sel == 0
4  *                      b[i] if sel == 1
5  */
6 CHIP Mux16 {
7     //Header file
8     IN a[16], b[16], sel;
9     OUT out[16];
10
11     //Section file
12     PARTS:
13         // Put your code here:
14         Mux(a=a[0], b=b[0], sel=sel, out=out[0]);
15         Mux(a=a[1], b=b[1], sel=sel, out=out[1]);
16         Mux(a=a[2], b=b[2], sel=sel, out=out[2]);
17         Mux(a=a[3], b=b[3], sel=sel, out=out[3]);
18         Mux(a=a[4], b=b[4], sel=sel, out=out[4]);
19         Mux(a=a[5], b=b[5], sel=sel, out=out[5]);
20         Mux(a=a[6], b=b[6], sel=sel, out=out[6]);
21         Mux(a=a[7], b=b[7], sel=sel, out=out[7]);
22         Mux(a=a[8], b=b[8], sel=sel, out=out[8]);
23         Mux(a=a[9], b=b[9], sel=sel, out=out[9]);
24         Mux(a=a[10], b=b[10], sel=sel, out=out[10]);
25         Mux(a=a[11], b=b[11], sel=sel, out=out[11]);
26         Mux(a=a[12], b=b[12], sel=sel, out=out[12]);
27         Mux(a=a[13], b=b[13], sel=sel, out=out[13]);
28         Mux(a=a[14], b=b[14], sel=sel, out=out[14]);
29         Mux(a=a[15], b=b[15], sel=sel, out=out[15]);
30 }
```

**Note:** Điểm hạn chế của cổng logic nhị phân là nó chỉ nhận vô 2 đầu vào. Tuy nhiên ta có thể kết hợp lại các chip cũng như mạch chọn kênh để tạo ra các chip chấp nhận vào một số lượng đầu vào tùy ý (Thường là mũ của 2). Nhờ vậy ta có thể dễ dàng thiết kế các thiết bị phức tạp hơn như CPU, RAM, ALU,...

### 2.4.3 Mux-nWay16.hdl

Một bộ ghép kênh  $n$  bit  $m$  chiều chọn một trong số  $m$  bus đầu vào  $n$ bit và xuất nó tới một bus đầu ra  $n$  bit duy nhất. Việc lựa chọn được xác định bởi tập  $k$  bit điều khiển, trong đó  $k = \log_2(m)$ .

**Mux4Way16:** Ta kết hợp Từ các bộ chọn kênh 16 bit 2 cổng để tạo nên bộ chọn kênh 16 bit 4 cổng.

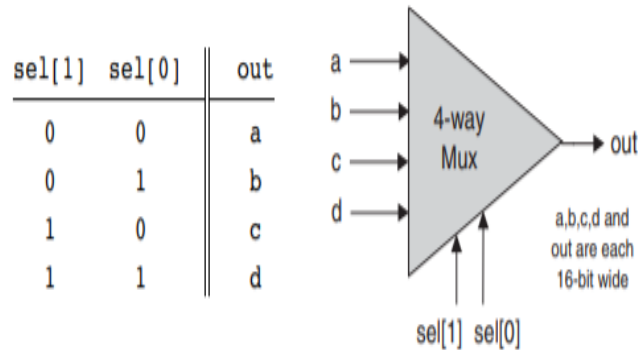


Figure 17: Thiết kế và bảng chân trị của mux4way

```

1 /**
2  * 4-way 16-bit multiplexor:
3  * out = a if sel == 00
4  *   b if sel == 01
5  *   c if sel == 10
6  *   d if sel == 11
7  */
8 CHIP Mux4Way16 {
9     IN a[16], b[16], c[16], d[16], sel[2];
10     OUT out[16];
11
12     PARTS:
13     // Put your code here:
14     Mux16(a = a, b = b, sel = sel[0], out = ab);
15     Mux16(a = c, b = d, sel = sel[0], out = cd);
16     Mux16(a = ab, b = cd, sel = sel[1], out = out);}

```

**Mux8Way16:** Từ chip mux4way đã xây dựng, ta sẽ chia 8 đầu vào thành 2 kênh mỗi kênh gồm 4 đầu vào rồi lấy mạch chọn kênh 16 bit để chọn kênh nào trong các kênh đó.

```

1 /**
2  * 8-way 16-bit multiplexor:
3  * out = a if sel == 000
4  *   b if sel == 001
5  *   etc.
6  *   h if sel == 111
7  */
8 CHIP Mux8Way16 {
9     IN a[16], b[16], c[16], d[16],
10     e[16], f[16], g[16], h[16],
11     sel[3];
12     OUT out[16];
13
14     PARTS:
15     // Put your code here:
16     Mux4Way16(a = a, b = b, c = c, d = d, sel[0] = sel[0], sel[1] = sel[1], out = abcd);
17     Mux4Way16(a = e, b = f, c = g, d = h, sel[0] = sel[0], sel[1] = sel[1], out = efgh);
18     Mux16(a = abcd, b = efgh, sel = sel[2], out = out);
19 }

```

### 3 BOOLEAN ARITHMETIC AND THE ALU

#### 3.1 PHÉP CỘNG (ADDITION):

Đối với máy tính việc cộng 2 số thực chất là biểu diễn 2 số đó ở dạng cơ số 2 rồi cộng nó theo thuật toán đã học từ những năm cấp 1.

$$\begin{array}{r}
 1010 \\
 + \quad \quad 11 \\
 \hline
 1101
 \end{array}$$

(a) Phép cộng 2 số nhị phân

$$\begin{array}{r}
 7875 \\
 + \quad \quad 562 \\
 \hline
 8437
 \end{array}$$

(b) Phép cộng 2 số thập phân

**Figure 18:** Hiện thực phép cộng

Máy tính biểu diễn số nguyên bằng cách sử dụng một số bit cố định, đôi khi được gọi là "word-size". Điều này dẫn đến một vấn đề liệu có thể có tràn số xảy ra hay không (hay đối với máy tính còn gọi là tràn bit). Với  $n = 4$ , ta có ví dụ như sau: Để xử lý tràn số là ta sẽ bỏ qua nó. Điều này có 2 lợi, hại. Nói

$$\begin{array}{r}
 \begin{array}{|c|c|c|c|} \hline 0 & 0 & 0 & 1 \\ \hline \end{array} \\
 + \begin{array}{|c|c|c|c|} \hline 0 & 1 & 0 & 1 \\ \hline \end{array} \\
 \hline
 \begin{array}{|c|c|c|c|} \hline 0 & 1 & 1 & 0 \\ \hline \end{array}
 \end{array}$$

(a) Không xảy ra tràn số

$$\begin{array}{r}
 \begin{array}{|c|c|c|c|} \hline 0 & 1 & 1 & 1 \\ \hline \end{array} \\
 + \begin{array}{|c|c|c|c|} \hline 1 & 1 & 1 & 0 \\ \hline \end{array} \\
 \hline
 \begin{array}{|c|c|c|c|} \hline 1 & 0 & 1 & 0 & 1 \\ \hline \end{array}
 \end{array}$$

(b) Xảy ra tràn số (Overflow)

**Figure 19:** word-size:  $n = 4$

về mặt lợi: giúp chúng ta cố định thống nhất được một cách biểu diễn duy nhất. Còn về mặt hại: có thể thay đổi kết quả dẫn đến kết quả không mong muốn (Điều này có thể hạn chế nếu word-size đủ lớn).

#### 3.2 BIỂU DIỄN SỐ ÂM (SIGN INTEGER):

Trong hầu hết các ngôn ngữ lập trình, kiểu dữ liệu short, int và long long lần lượt sử dụng 16, 32, và 64 bit để biểu diễn số nguyên có dấu. Trước đây  $n$  bit cho phép biểu diễn tất cả các số nguyên từ  $0 \dots 2^n - 1$ .

2-bin	10-dec
0000	0
0001	1
0010	2
0011	3
0100	4



0101	5
0110	6
0111	7
1000	8
1001	9
1010	10
1011	11
1100	12
1101	13
1110	14
1111	15

Vậy để biểu diễn số âm, chúng ta có thể sử dụng một nửa không gian dành cho số dương, còn lại là cho số âm tức là nếu word-size là n-bit thì phạm vi của số dương là từ 0 đến  $2^{n/2} - 1$ . Còn số âm là từ  $-2^{n/2}$  đến -1.

**Bù 2 (Two's complement):** Đây là cách biểu diễn số nguyên âm mà máy tính sử dụng. Giả sử word-size = n. Chi tiết để biểu diễn số nguyên âm -x thì ta sẽ thực hiện phép trừ  $2^n - x$  để lấy kết quả với quy ước bit đầu là bit đầu tiên từ trái sang phải trong dãy n bit.

2-bin	10-dec
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	-8
1001	-7
1010	-6
1011	-5
1100	-4
1101	-3
1110	-2
1111	-1

Để thực hiện phép tính cộng cho số có dấu. Trước mắt ta đưa số có dấu âm về dạng bù 2 rồi đổi ra dec theo đúng quy tắc biểu diễn số không dấu, thực hiện phép cộng bình thường rồi lấy modulo  $2^n$  với word-size = n. Tuy nhiên vẫn có thể xảy ra tràn số. Cách để phát hiện tràn số là khi ta cộng 2 số cùng

$$\begin{array}{rcl}
 \begin{array}{r} +4 \\ -7 \end{array} & = & \begin{array}{r} +4 \\ 9 \end{array} \\
 & & \hline
 13 \% 16 = 13 \text{ codes } -3
 \end{array}
 \qquad
 \begin{array}{rcl}
 \begin{array}{r} -2 \\ -4 \end{array} & = & \begin{array}{r} 14 \\ 12 \end{array} \\
 & & \hline
 26 \% 16 = 10 \text{ codes } -6
 \end{array}$$

**Figure 20:** word-size: n = 4

dấu ra kết quả có dấu ngược lại. Để giải thích ra cách biến đổi rõ hơn số có dấu: ta sẽ đi giải thích cách biến đổi -x từ x.

Giả sử:  $n = 4$

$$\begin{aligned} \text{code}(-x) &= 2^n - x = 1 + (2^n - 1) - x \\ &= 1 + (1111) - x \\ &= 1 + \text{FlipBit}(x) \end{aligned}$$

Trong đó code là cách biểu diễn số âm  $-x$  mang phần nguyên  $x$ . FlipBit là phép đảo bit toàn bộ

### 3.3 VON-NEUMANN ARCHITECTURE

Kiến trúc máy tính Vonneumann: - còn được gọi là mô hình von Neumann hoặc kiến trúc Princeton - là kiến trúc máy tính dựa trên mô tả năm 1945 của nhà toán học và vật lý John von Neumann. Ngoài ra còn các kiến trúc máy tính khác như Harvard, RISC, CISC,... Chúng ta sẽ xây dựng máy tính Hack theo kiến trúc Vonneumann. Đặc điểm của kiến trúc:

- + **Một đơn vị xử lý trung tâm (Central Processing Unit):** có chứa đơn vị logic số học và thanh ghi bộ xử lý.
- + **Control Unit:** Một đơn vị điều khiển có chứa thanh ghi lệnh và bộ đếm chương trình.
- + **Memory:** Bộ nhớ lưu trữ dữ liệu và lệnh xử lý.
- + **External Memory:** Lưu trữ khối ngoài.
- + **Input & Output device:** Cơ chế đầu vào và đầu ra.

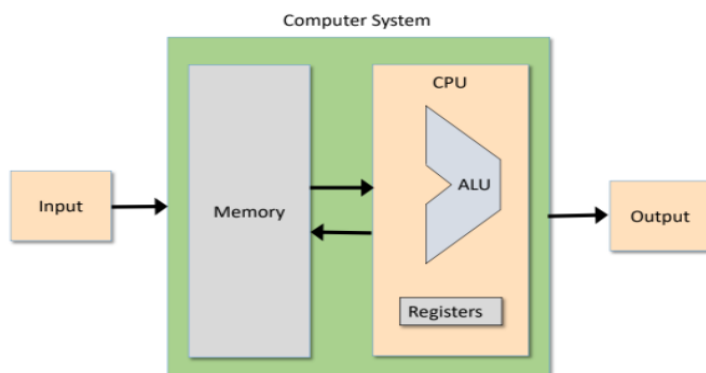


Figure 21: Kiến trúc Von-neumann

### 3.4 ALU & HACK ALU

ALU tính toán một chức năng trên hai dữ liệu nhất định từ đầu vào của nó và cho ra một kết quả duy nhất. Có một tín hiệu nữa để xác định phép tính của nó cần thực hiện. Các hàm đó có thể là một trong các hàm đại số như cộng, trừ, nhân, chia hay các cổng logic như And, Or, Xor,...

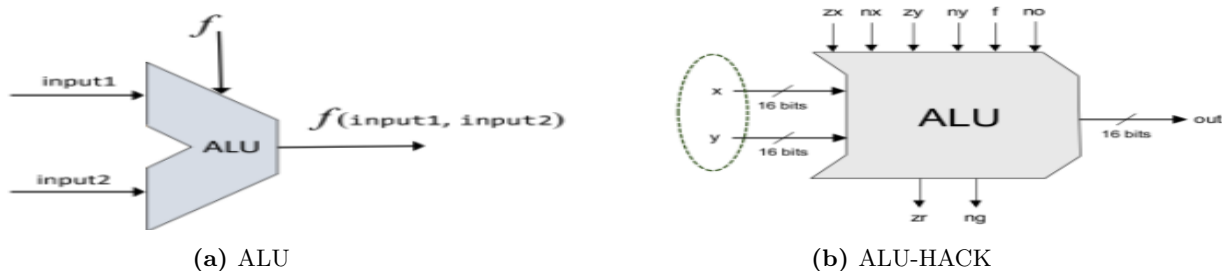


Figure 22: Biểu diễn ALU

**ALU-HACK:** hai đầu vào 16 bit và 1 đầu ra out: 16 bit cũng như hai tín hiệu: zr (để xác định có phải phép so sánh hai số) và neg (để xác định xem kết quả có phải là số âm hay không). Các bit đầu ra zr và ng sẽ phát huy tác dụng khi chúng ta xây dựng CPU trong project sau. Hoàn thiện kiến trúc CPU, sau này trong khóa học. Việc xác định hàm để tính toán được đặt bởi sáu đầu vào 1 bit là zx, nx, zy, ny, f, no. Để một tín hiệu hoạt động ta đặt nó ở giá trị 1.

pre-setting the x input		pre-setting the y input		selecting between computing + or &	post-setting the output	Resulting ALU output
zx	nx	zy	ny	f	no	out
if zx then x=0	if nx then x=!x	if zy then y=0	if ny then y=!y	if f then out=x+y else out=x&y	if no then out=!out	out(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

**Figure 23:** Bảng tín hiệu điều khiển

- + **zx và nx:** dùng để xác định x giữ giá trị 0 (zx) và x có giữ giá trị âm hay không (nx). Tương tự zy và ny cũng như vậy.
- + **f:** để xác định hàm được lựa chọn là add hay and. Vì tất cả các phép tính toán đều có thể biểu diễn từ hai hàm này.
- + **no:** để xác định dấu của đầu ra output.
- + **out:** Kết quả ứng với phép tính được chọn.

### 3.5 Project

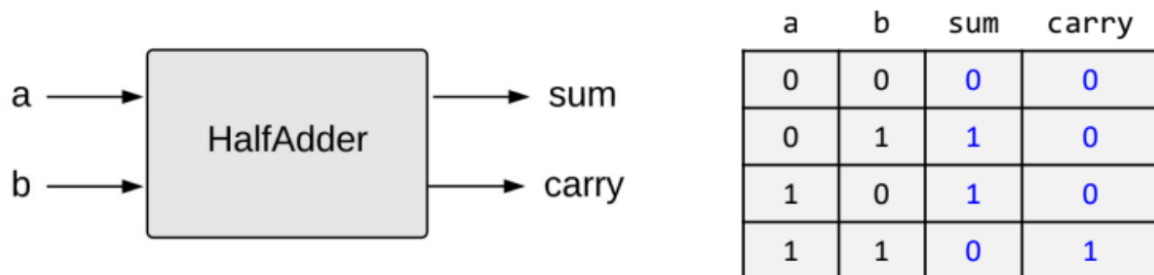
Trong tuần thứ hai, ta sẽ lần lượt thực thi các mẫu thiết kế thiết bị phần cứng đó là bộ cộng ADDer, Bộ tăng lệnh và quan trọng nhất là ALU từ các chip đã xây dựng sẵn ở tuần thứ nhất. Cụ thể các file ta cần làm đó là:

<b>ALU.hdl</b> D:\My Workspace\project\NandToTeris\nand2tetris\... Type: HDL File	Date modified: 10/19/2021 11:05 PM Size: 2.39 KB
<b>Inc16.hdl</b> D:\My Workspace\project\NandToTeris\nand2tetris\... Type: HDL File	Date modified: 10/19/2021 10:29 PM Size: 380 bytes
<b>Add16.hdl</b> D:\My Workspace\project\NandToTeris\nand2tetris\... Type: HDL File	Date modified: 10/19/2021 10:26 PM Size: 1.29 KB
<b>FullAdder.hdl</b> D:\My Workspace\project\NandToTeris\nand2tetris\... Type: HDL File	Date modified: 10/19/2021 10:22 PM Size: 557 bytes
<b>HalfAdder.hdl</b> D:\My Workspace\project\NandToTeris\nand2tetris\... Type: HDL File	Date modified: 10/19/2021 10:16 PM Size: 467 bytes

**Figure 24:** Các file .hdl cần hiện thực

### 3.5.1 Adder:

**Mạch cộng bán phần (HalfAdder.hdl):** là mạch tổ hợp kỹ thuật số được sử dụng để cộng hai số. Mạch cộng gồm một bit tổng (ký hiệu là sum) và một bit nhớ (ký hiệu là carry) làm đầu ra. Nếu đầu vào là a và b thì  $sum = a \oplus b$  và  $carry = a \text{ and } b$ .



**Figure 25:** Thiết kế mạch cộng bán phần

```

1 CHIP HalfAdder {
2     IN a, b;    // 1-bit inputs
3     OUT sum,    // Right bit of a + b
4         carry;  // Left bit of a + b
5
6     PARTS:
7         // Put you code here:
8         Xor(a = a, b = b, out = sum);
9         And(a = a, b = b, out = carry);
10 }

```

**Mạch cộng toàn phần: (FullAdder.hdl)** Đây là loại mạch khó thực hiện hơn một chút so với mạch cộng bán phần. Sự khác biệt chính giữa bộ cộng bán phần và bộ cộng toàn phần là bộ cộng toàn phần có ba ngõ vào và hai ngõ ra. Hai đầu vào đầu tiên là a và b và đầu vào thứ ba là c chính là số dư của phép tính trước. Mạch này tận dụng HalfAdder để thực hiện phép tính tổng của  $res = a + b$  và tổng res với c từ phép tính trước.

```

1 CHIP FullAdder {
2     IN a, b, c; // 1-bit inputs
3     OUT sum,    // Right bit of a + b + c
4         carry;  // Left bit of a + b + c
5
6     PARTS:
7         // Put you code here:
8         HalfAdder(a = a, b = b, sum = sab, carry = cab);

```

```

9 HalfAdder(a = sab,b = c,sum = sum,carry = cc);
10 Or(a = cab,b = cc,out = carry);
11 }

```

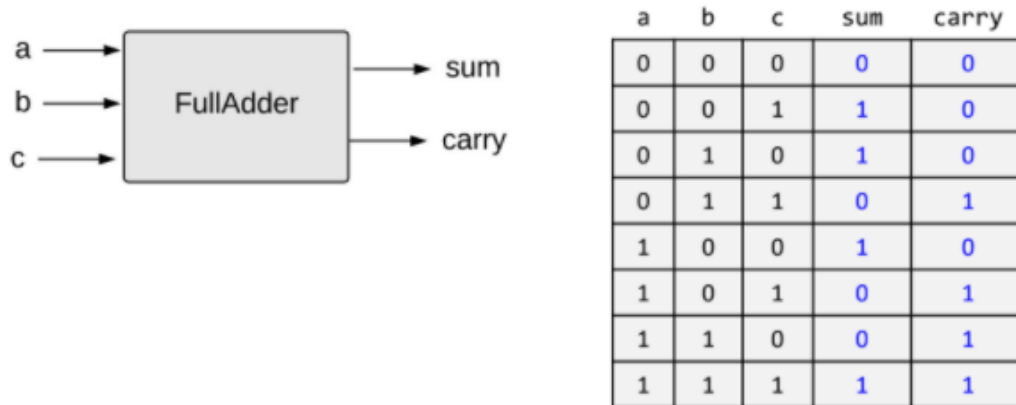


Figure 26: Thiết kế mạch cộng toàn phần

**Mạch cộng 16bit: (Add16.hdl)** Ta mở rộng thay vì cộng 2 giá trị 1 bit chuyển sang cộng 2 số được biểu diễn bởi 16 bit. Việc bổ sung cộng dư được thực hiện song song. Sự lan truyền này xảy ra theo tuần tự.

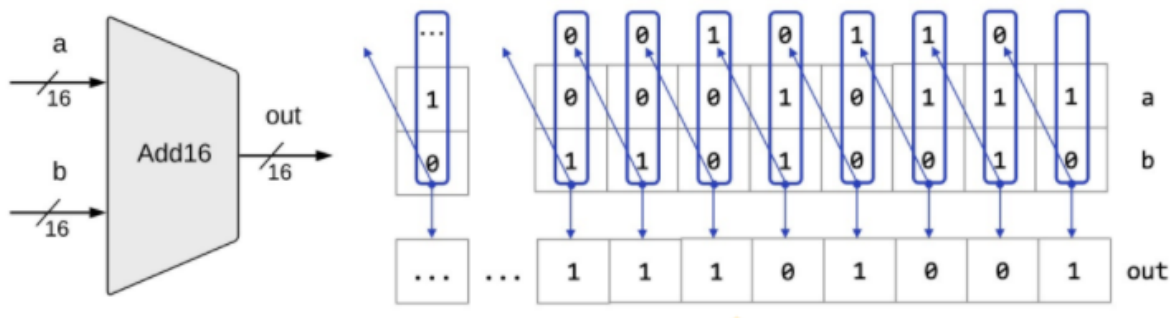


Figure 27: Thiết kế mạch 16bit

```

1 CHIP Add16 {
2   IN a[16], b[16];
3   OUT out[16];
4
5   PARTS:
6   // Put you code here:
7   // Sequentially add each bit together
8   HalfAdder(a=a[0], b=b[0], sum=out[0], carry=c1);
9   FullAdder(a=a[1], b=b[1], c=c1, sum=out[1], carry=c2);
10  FullAdder(a=a[2], b=b[2], c=c2, sum=out[2], carry=c3);
11  FullAdder(a=a[3], b=b[3], c=c3, sum=out[3], carry=c4);
12  FullAdder(a=a[4], b=b[4], c=c4, sum=out[4], carry=c5);
13  FullAdder(a=a[5], b=b[5], c=c5, sum=out[5], carry=c6);
14  FullAdder(a=a[6], b=b[6], c=c6, sum=out[6], carry=c7);
15  FullAdder(a=a[7], b=b[7], c=c7, sum=out[7], carry=c8);
16  FullAdder(a=a[8], b=b[8], c=c8, sum=out[8], carry=c9);
17  FullAdder(a=a[9], b=b[9], c=c9, sum=out[9], carry=c10);
18  FullAdder(a=a[10], b=b[10], c=c10, sum=out[10], carry=c11);
19  FullAdder(a=a[11], b=b[11], c=c11, sum=out[11], carry=c12);

```

```

20 FullAdder(a=a[12], b=b[12], c=c12, sum=out[12], carry=c13);
21 FullAdder(a=a[13], b=b[13], c=c13, sum=out[13], carry=c14);
22 FullAdder(a=a[14], b=b[14], c=c14, sum=out[14], carry=c15);
23 FullAdder(a=a[15], b=b[15], c=c15, sum=out[15], carry=c16);
24 }

```

### 3.5.2 ALU.hdl

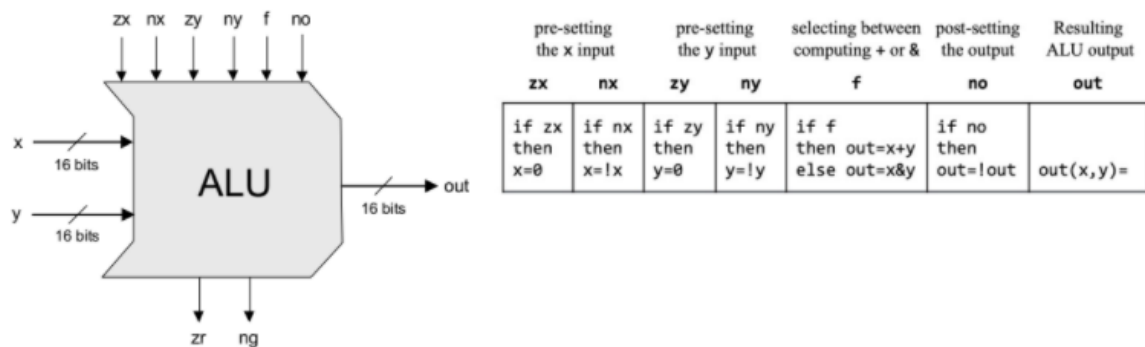


Figure 28: ALU-Hack Abstraction

#### Chiến lược thiết kế ALU:

- + Hiện thực câu lệnh điều kiện "if bit == 0/1"
- + Hiện thực đặt giá trị 16-bit là 0000000000000000. (Nếu tín hiệu zx = 0)
- + Hiện thực đặt giá trị 16-bit là 1111111111111111. (Cần cho phép tính bù 2).
- + Cần áp dụng hàm Add và Or cho 2 số 16 bit đã áp dụng trước đó.
- + Xây dựng tín hiệu out trước các tín hiệu zr và ng.

```

1 CHIP ALU {
2     IN
3         x[16], y[16], // 16-bit inputs
4         zx, // zero the x input?
5         nx, // negate the x input?
6         zy, // zero the y input?
7         ny, // negate the y input?
8         f, // compute out = x + y (if 1) or x & y (if 0)
9         no; // negate the out output?
10
11     OUT
12         out[16], // 16-bit output
13         zr, // 1 if (out == 0), 0 otherwise
14         ng; // 1 if (out < 0), 0 otherwise
15
16     PARTS:
17         // Put you code here:
18         //zx
19         Mux16(a = x, b = false, sel = zx, out = x0);
20         //nx
21         Not16(in = x0, out = nX0);
22         Mux16(a = x0, b = nX0, sel = nx, out = x1);
23         //zy
24         Mux16(a = y, b = false, sel = zy, out = y0);
25         //ny

```

```
26 Not16(in = y0,out = nY0);
27 Mux16(a = y0,b = nY0,sel = ny,out = y1);
28 //f
29 Add16(a = x1,b = y1,out = addXY);
30 And16(a = x1,b = y1,out = andXY);
31 Mux16(a = andXY,b = addXY,sel = f,out = out0);
32 //no
33 Not16(in = out0,out = nOut);
34 Mux16(a = out0,b = nOut,sel = no,out = out,out[15] = signBit,out[0..7] = left,out[8..15] =
    right);
35 //zr
36 Or8Way(in = left,out = L);
37 Or8Way(in = right,out = R);
38 Or(a = L,b = R,out = isZ);
39 Not(in= isZ, out=zr);
40 //if (out < 0) set ng = 1
41 Mux(a = false,b = true,sel = signBit,out = ng);
42 }
```

## 4 WEEK 3 - SEQUENTIAL LOGIC

### 4.1 Clock

Là một chuỗi tín hiệu xen kẽ liên tục. Việc triển khai chính xác thường dựa trên một bộ dao động luân phiên liên tục giữa hai giai đoạn có nhãn 0–1, thấp-cao, tích tắc. Thời gian trôi qua từ khi bắt đầu "tik" đến khi kết thúc giai đoạn tiếp theo "tock" được gọi là chu kỳ và mỗi chu kỳ được đưa vào mô hình một đơn vị thời gian rời rạc. Sử dụng mạch của phần cứng, tín hiệu này được phát đồng thời tới mọi chip tuần tự trên toàn bộ nền tảng máy tính.

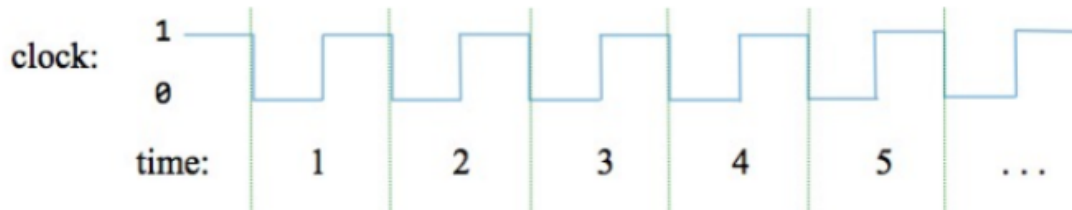


Figure 29: tín hiệu clock

### 4.2 Flip-Flops

Phần tử tuần tự cơ bản nhất trong máy tính được gọi là flip-flop. Trong Hack-Computer, ta chỉ sử dụng đến DFF. DFF có giao diện bao gồm đầu vào dữ liệu một bit và đầu ra dữ liệu một bit. Ngoài ra, DFF còn có một đầu vào dành cho tín hiệu clock, nó sẽ thay đổi phụ thuộc vào sự thay đổi của tín hiệu clock. DFF sẽ xuất giá trị đầu vào từ đơn vị thời gian trước đó. Tức là  $out(t) = in(t - 1)$ . Chính hành vi cơ bản này là nền tảng cho thiết bị phần cứng với mục đích lưu trữ.

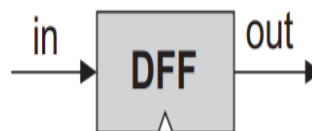


Figure 30: Abstract of DFF (Data Flip Flop)

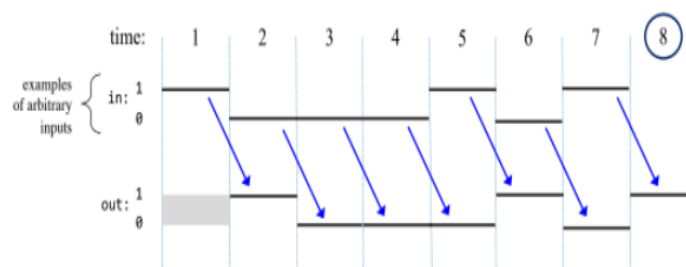


Figure 31: Biểu diễn hành vi của DFF



### 4.3 Register

Register là một thiết bị lưu trữ có thể "lưu trữ" hoặc "ghi nhớ" một giá trị theo thời gian, triển khai hành vi:  $out(t) = out(t - 1)$ . Nhờ vào hành vi của DFF, mà Register có thể xây dựng từ các DFF. Do đó, đầu ra của thiết bị này tại bất kỳ thời điểm nào  $t$  sẽ lặp lại đầu ra của nó tại thời gian  $t - 1$ .

**1-bit register:** Tín hiệu Load để nhận ra hành vi "tải" và "lưu trữ" và có thể chọn giữa hai trạng thái này cho đầu ra. Nếu  $load == 1$  thì register sẽ nhận giá trị mới từ đầu vào  $in$ , ngược lại sẽ giữ nguyên giá trị hiện tại. Như vậy chúng ta có thể hiện thực được việc lưu trữ 1 bit dữ liệu

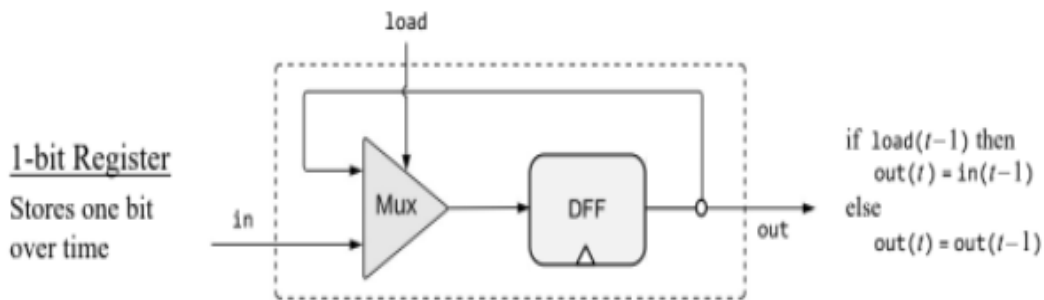


Figure 32: register 1-bit

**n-bit register:** Còn để hiện thực các register lưu trữ nhiều bit hơn ta chỉ cần đặt các 1-bit register kề cạnh nhau. Tham số thiết kế cơ bản của một thanh ghi như là chiều rộng — số lượng bit mà nó giữ — ví dụ: 16, 32 hoặc 64. Nội dung nhiều bit của các thanh ghi như vậy thường được gọi là "word".

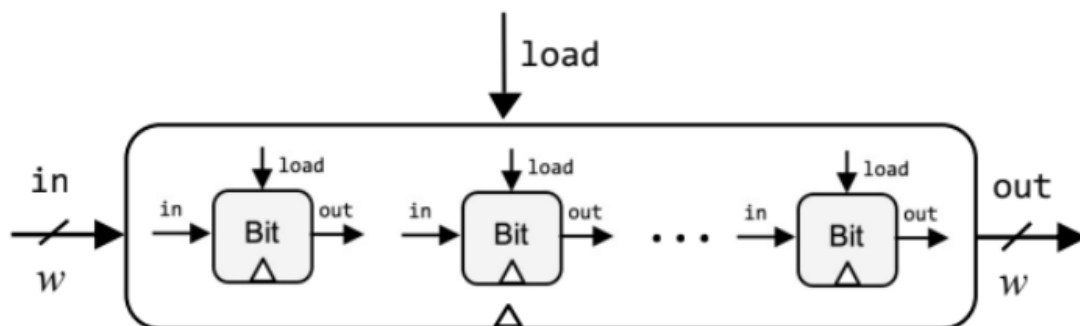


Figure 33: register nn-bit

### 4.4 MEMORY (RAM)

Khi chúng ta có khả năng cơ bản để biểu diễn các "word", chúng ta có thể tiếp tục xây dựng các ngân hàng bộ nhớ có độ dài tùy ý. Bộ nhớ RAM là Random Access Memory, trên RAM sẽ có thể truy cập các "word" được chọn ngẫu nhiên, không có ảnh hưởng thứ tự mà chúng được truy cập. Có nghĩ là khi truy cập bất cứ "word" nào, thời gian truy xuất và tốc độ xử lý đều ngang nhau. Một thiết bị RAM sẽ chấp nhận 3 đầu vào: dữ liệu, đầu vào địa chỉ, đầu vào bit tải. Để xây dựng RAM ta bắt đầu từ 2g bước sau

1. Gán cho mỗi từ trong nregister RAM một địa chỉ duy nhất (một số nguyên từ 0 đến  $n - 1$ ), theo đó nó sẽ được truy cập.
2. ngoài việc xây dựng một mảng  $n$  thanh ghi, ta xây dựng một thiết kế logic cổng, cho một địa chỉ  $j$ , có khả năng chọn thanh ghi riêng lẻ có địa chỉ là  $j$ . Khái niệm "địa chỉ" ở đây là ảo (Do nó không liên kết với bất kỳ bộ phận vật lý nào) => tạo cho RAM khả năng truy cập trực tiếp.

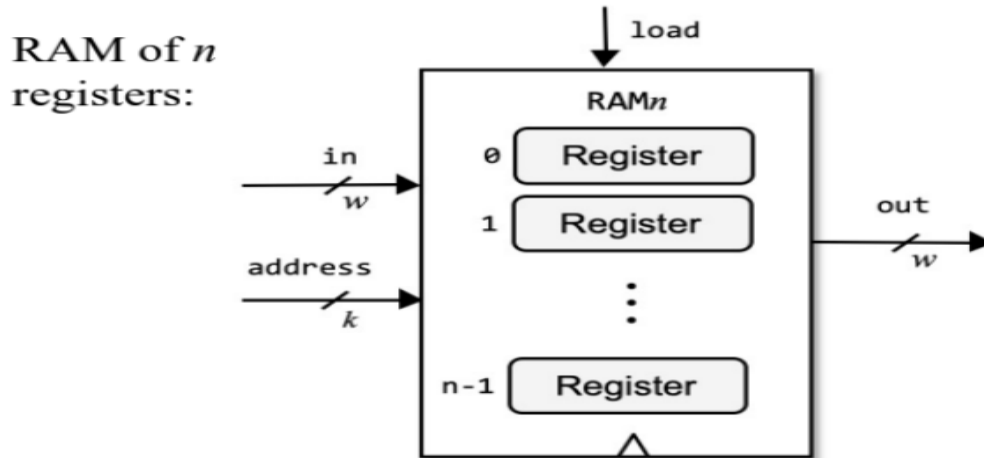


Figure 34: RAM

Có hai hành động được thực hiện với RAM:

- + Đọc thanh ghi  $i$ : set address thanh ghi cần lựa chọn là  $i$ . Output ra kết quả thanh ghi đã chọn đó.
- + Ghi giá trị vào thanh ghi  $i$ : set address thanh ghi cần lựa chọn là  $i$ . set  $in = v$ , set  $load = 1$ .

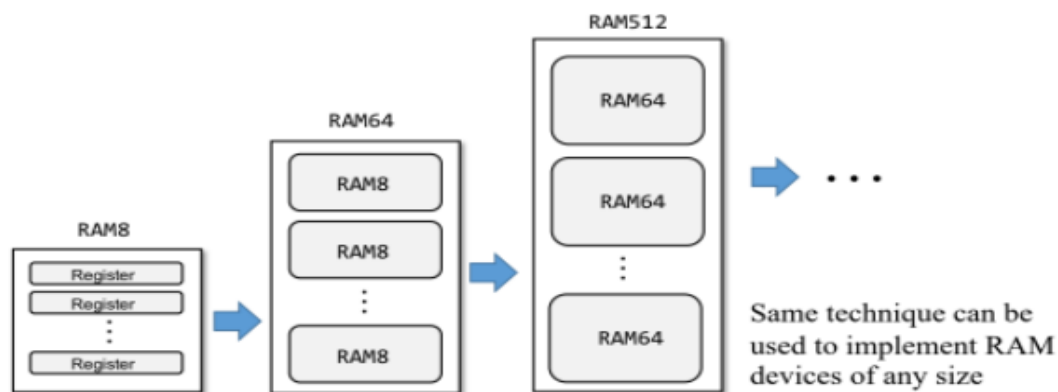


Figure 35: RAM HIERARCHY

chip name	n	k
RAM8	8 (8 register)	3
RAM64	64 (8 RAM8)	6
RAM512	512 (8 RAM64)	9
RAM4K	4096 (8 RAM512)	12
RAM16K	16834 (4 RAM4K)	14

## 4.5 PROGRAM COUNTER (PC):

Counter là một chip tuần tự có trạng thái là một số nguyên gia tăng dần theo mỗi đơn vị thời gian.  $out(t) = out(t - 1) + c$  với  $c$  điển hình là 1. CPU điển hình bao gồm một bộ đếm chương trình có đầu ra được hiểu là địa chỉ của lệnh sẽ được thực hiện tiếp theo trong chương trình hiện tại. Một chip bộ đếm có thể được thực hiện bằng cách kết hợp logic đầu vào / đầu ra của một thanh ghi chuẩn với logic tổ hợp để thêm một hằng số. Thông thường, bộ đếm sẽ phải được trang bị một số chức năng bổ sung, chẳng hạn như khả năng đặt lại số đếm về 0, tải cơ sở đếm mới hoặc giảm thay vì tăng dần.

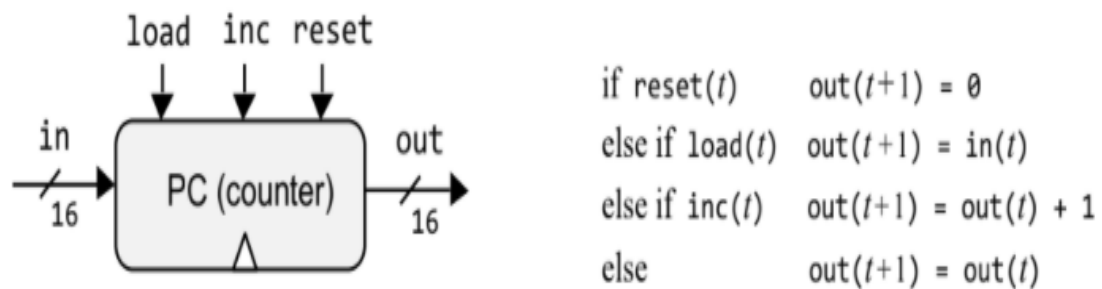


Figure 36: COUNTER

## 4.6 Project

Trong tuần thứ ba, ta sẽ lần lượt thực thi các mẫu thiết kế thiết bị phần cứng tuần tự có chức năng lưu trữ với DFF được cung cấp sẵn. Chẳng hạn như: Cụ thể các file .hdl ta cần làm đó là: PC, BIT, Register, RAM.

	RAM512.hdl	D:\My Workspace\project\NandToTeris\nand2tetris\...	Type: HDL File
	RAM16K.hdl	D:\My Workspace\project\NandToTeris\nand2tetris\...	Type: HDL File
	RAM4K.hdl	D:\My Workspace\project\NandToTeris\nand2tetris\...	Type: HDL File
	RAM8.hdl	D:\My Workspace\project\NandToTeris\nand2tetris\...	Type: HDL File
	RAM64.hdl	D:\My Workspace\project\NandToTeris\nand2tetris\...	Type: HDL File
	PC.hdl	D:\My Workspace\project\NandToTeris\nand2tetris\...	Type: HDL File
	Register.hdl	D:\My Workspace\project\NandToTeris\nand2tetris\...	Type: HDL File
	Bit.hdl	D:\My Workspace\project\NandToTeris\nand2tetris\...	Type: HDL File

Figure 37: Các file .hdl cần hiện thực

### 4.6.1 register:

**1-bit register:** Giao diện chip bao gồm một chân đầu vào mang dữ liệu bit, một chân tải cho phép ghi và một chân đầu ra phát ra trạng thái hiện tại của ô.

```

1 CHIP Bit {
2     IN in, load;
3     OUT out;
  
```

```

4
5 PARTS:
6 // Put your code here:
7 Mux(a = fed,b = in,sel = load,out = out0);
8 DFF(in = out0,out = out,out = fed);
9 }

```

**register (16bit):** Xây dựng mảng các thanh ghi và cấp dữ liệu đầu vào cho mảng đó.

```

1 CHIP Register {
2     IN in[16], load;
3     OUT out[16];
4
5     PARTS:
6     // Put your code here:
7     Bit(in = in[0],load = load,out = out[0]);
8     Bit(in= in[1], load = load,out = out[1]);
9     Bit(in= in[2], load = load,out = out[2]);
10    Bit(in= in[3], load = load,out = out[3]);
11    Bit(in= in[4], load = load,out = out[4]);
12    Bit(in= in[5], load = load,out = out[5]);
13    Bit(in= in[6], load = load,out = out[6]);
14    Bit(in= in[7], load = load,out = out[7]);
15    Bit(in= in[8], load = load,out = out[8]);
16    Bit(in= in[9], load = load,out = out[9]);
17    Bit(in= in[10], load = load,out = out[10]);
18    Bit(in= in[11], load = load,out = out[11]);
19    Bit(in= in[12], load = load,out = out[12]);
20    Bit(in= in[13], load = load,out = out[13]);
21    Bit(in= in[14], load = load,out = out[14]);
22    Bit(in= in[15], load = load,out = out[15]);
23 }

```

#### 4.6.2 Ram:

Lấy đại diện là Ram8k. Bộ nhớ 8 thanh ghi, mỗi thanh ghi rộng 16 bit. Out giữ giá trị được lưu trữ tại vị trí bộ nhớ được chỉ định theo địa chỉ. Nếu tải == 1, thì giá trị in được tải vào vị trí bộ nhớ được chỉ định bởi địa chỉ (giá trị đã nạp sẽ được phát ra từ bước thời gian tiếp theo trở đi).

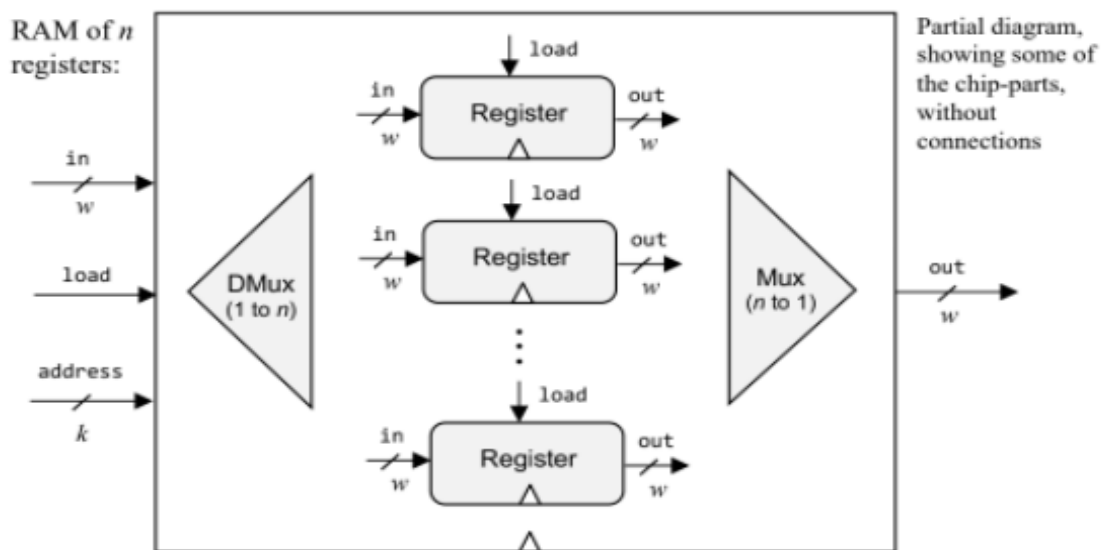


Figure 38: RAM design

```
1 CHIP RAM8 {
2     IN in[16], load, address[3];
3     OUT out[16];
4
5     PARTS:
6     // Put your code here:
7     DMux8Way(in = load, sel = address, a = load0, b = load1, c = load2, d = load3, e = load4, f =
8         load5, g = load6, h = load7);
9
10    Register(in = in, load = load0, out = out0);
11    Register(in = in, load = load1, out = out1);
12    Register(in = in, load = load2, out = out2);
13    Register(in = in, load = load3, out = out3);
14    Register(in = in, load = load4, out = out4);
15    Register(in = in, load = load5, out = out5);
16    Register(in = in, load = load6, out = out6);
17    Register(in = in, load = load7, out = out7);
18
19    Mux8Way16(a = out0, b = out1, c = out2, d = out3, e = out4, f = out5, g = out6, h = out7, sel =
    address, out = out);
20 }
```

#### 4.6.3 PC:

Bộ đếm 16 bit với các bit nhập (load), đặt lại (reset) và bit chuyển đến giá trị tiếp theo (inc)

- + Nếu (reset[t] == 1) out[t+1] = 0
- + Nếu (load[t] == 1) out[t+1] = in[t]
- + Nếu (inc[t] == 1) out[t+1] = out[t] + 1
- + Còn lại ta trả ra out[t+1] = out[t]

```
1 CHIP PC {
2     IN in[16], load, inc, reset;
3     OUT out[16];
4
5     PARTS:
6     // Put your code here:
7
8     //reset
9     Inc16(in = prev, out = outInc);
10    Mux16(a = prev, b = outInc, sel = inc, out = out1);
11    Mux16(a = out1, b = in, sel = load, out = out2);
12    Mux16(a = out2, b = false, sel = reset, out = out3);
13    Register(in = out3, load = true, out = out, out = prev);
14 }
```

## 5 WEEK4 - MACHINE LANGUAGE

### 5.1 ĐỊNH NGHĨA:

- + Ngôn ngữ máy được thiết kế để viết mã cấp thấp các chương trình như một loạt các lệnh máy. Sử dụng các hướng dẫn này, lập trình viên có thể ra lệnh cho bộ xử lý thực hiện các phép toán số học và logic, tìm nạp và lưu trữ các giá trị từ và vào bộ nhớ, di chuyển các giá trị từ thanh ghi này sang thanh ghi khác, kiểm tra. Điều kiện luận lý,... Trái ngược với các ngôn ngữ cấp cao( những ngôn ngữ cơ bản mục tiêu thiết kế là tính tổng quát và sức mạnh diễn đạt), mục tiêu của ngôn ngữ máy là thiết kế là thực thi trực tiếp cũng như kiểm soát toàn bộ nền tảng phần cứng. Chúng hỗ trợ yêu cầu cơ bản của việc thực thi trực tiếp trong phần cứng. Ngôn ngữ máy là giao diện sâu nhất, là ranh giới giữa phần cứng và phần mềm. Đây là điểm mà những suy nghĩ trừu tượng có thể chuyển thành cơ chế hoạt động của phần cứng. Vì thế nó là cơ chế quan trọng trong việc giao tiếp vừa phần cứng và phần mềm trong máy tính.
- + Một chương trình ngôn ngữ máy là một loạt các lệnh được mã hóa. Ví dụ, một diễn hình hướng dẫn trong máy tính 16 bit có thể là 1010001100011001. Để tìm ra hướng dẫn này có nghĩa là gì, chúng ta phải biết các quy tắc của được quy định, cụ thể là, tập lệnh của nền tảng phần cứng bên dưới. Ví dụ: hướng dẫn trên có thể bao gồm bốn trường 4 bit: Trường ngoài cùng bên trái chứa mã hóa hoạt động của CPU và ba trường còn lại đại diện cho toán hạng của hoạt động.
- + Việc đọc các số 0, 1 trong một hướng dẫn khá khó hiểu nên mã nhị phân thường được tượng trưng bởi một nhãn đại diện cho mã đó. Ví dụ: 1010 với vị trí 4 bit đầu tiên của hướng dẫn trên đại diện cho một opcode có thể là phép cộng, di chuyển liên tục 4 bit tiếp theo ta có thể có R3, R1, R9 như vậy mã 1010001100011001 có thể viết lại là ADD R3, R1, R9. => Chính vì thế ta có thể viết một chương trình bằng các mã tượng trưng đó, mã này được gọi là hợp ngữ. Và chúng ta phải nhất thiết có một chương trình biên dịch từ hợp ngữ qua mã máy, và chương trình này được gọi là assembler.

### 5.2 CÁC LOẠI CÂU LỆNH:

Mọi máy tính đều được yêu cầu để thực hiện các phép toán, truy xuất vùng nhớ và cả các câu lệnh thể hiện luồng thực thi chương trình:

1. **phép toán đại số và logic:** chẳng hạn như AND R2, R1, R3 hoặc ADDI R1, R2, 100,...
2. **truy xuất vùng nhớ:** Các lệnh truy cập bộ nhớ được chia thành hai loại. Đầu tiên là việc các phép toán có thể thay vì chọn các thanh ghi tham gia tính toán mà trực tiếp chọn vùng nhớ để tính toán. Thứ hai có thể là hoạt động đọc và ghi vùng nhớ từ các thanh ghi. Chẳng hạn như: LOAD R1, 67. LOAD R1, bar (bar là nhãn đại diện cho địa chỉ), STORE R2,712.
3. **Các câu lệnh điều khiển:** Trong khi các chương trình thường thực thi theo kiểu tuyến tính, tức là hết lệnh này đến lệnh kia, chúng cũng bao gồm các nhánh không thường xuyên đến các vị trí khác lệnh tiếp theo. Việc phân nhánh phục vụ một số mục đích bao gồm cả sự lặp lại (bước nhảy lùi về đầu vòng lặp), thực thi có điều kiện (nếu điều kiện Boolean là false, chuyển tiếp đến vị trí sau mệnh đề "if-then") và gọi chương trình con (nhảy đến lệnh đầu tiên của một số đoạn mã khác). Và hầu hết các luồng chương trình như vậy đều bắt đầu từ các câu lệnh nhảy không điều kiện và có điều kiện. Ta có thể kể đến như JUMP BEGIN, JNG R2,endWhile (nếu R2 mang giá trị âm thì nhảy đến địa chỉ được tượng trưng bởi nhãn endWhile).

High-level	Low-level
<pre>// A while loop: while (R1&gt;=0) {     code segment 1 } code segment 2</pre>	<pre>// Typical translation: beginWhile:     JNG R1,endWhile // If R1&lt;0 goto endWhile     // Translation of code segment 1 comes here     JMP beginWhile // Goto beginWhile endWhile:     // Translation of code segment 2 comes here</pre>

Figure 39: Dịch từ ngôn ngữ cấp cao sang cấp thấp với một vòng lặp while

## 5.3 MÁY TÍNH HACK

Máy tính Hack là một kiến trúc von Neumann. Nó là một máy 16-bit, bao gồm của một CPU, hai mô-đun bộ nhớ riêng biệt phục vụ như bộ nhớ lệnh và dữ liệu bộ nhớ và hai thiết bị I / O được ánh xạ bộ nhớ: màn hình và bàn phím.

### 5.3.1 Không gian địa chỉ bộ nhớ:

Máy tính Hack chia không gian địa chỉ bộ nhớ ra làm hai bộ nhớ riêng biệt: bộ nhớ lệnh (instruction memory) và bộ nhớ dữ liệu (data memory). Mỗi vùng nhớ rộng 16 bit và có không gian địa chỉ rộng 15 bit tức kích thước tối đa vùng nhớ địa chỉ là 32K.

CPU chỉ có thể thực thi các chương trình nằm trong bộ nhớ lệnh. Các bộ nhớ lệnh là một thiết bị chỉ đọc và các chương trình được tải vào nó bằng cách sử dụng một số phương tiện ngoại sinh. Ví dụ, bộ nhớ lệnh có thể được triển khai trong Chip ROM được ghi sẵn chương trình cần thiết. Tải một chương trình mới được thực hiện bằng cách đọc tuần tự từ chip ROM rồi xuất nó ra mã máy cho đến khi đọc hết ROM mà thôi.

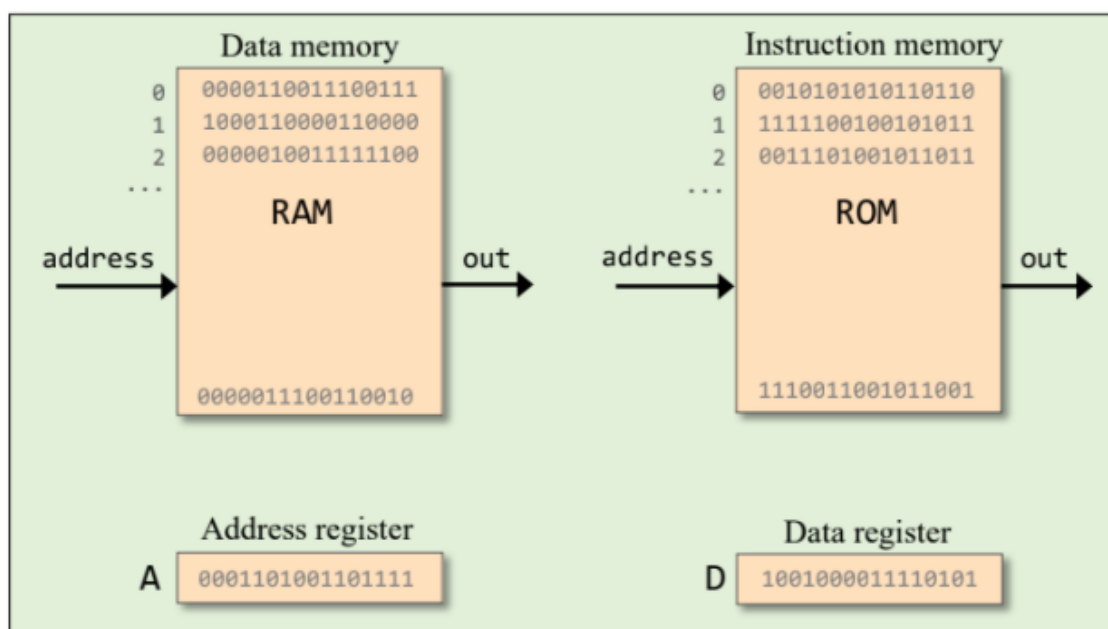


Figure 40: Bộ nhớ lệnh và bộ nhớ dữ liệu

### 5.3.2 Thanh ghi:

Máy tính Hack cho phép ta sử dụng hai thanh ghi 16 bit được gọi là D (Data register) và A (Address register). Các thanh ghi này có thể được thao tác một cách rõ ràng bằng các lệnh số học và logic như  $A = D - 1$  hoặc  $D \neq A$ . Trong khi D được sử dụng chỉ để lưu trữ các giá trị dữ liệu, A có cả hai vai trò là một thanh ghi dữ liệu và một thanh ghi địa chỉ tùy vào ngữ cảnh.

- Đối với RAM, nó là bộ nhớ đọc-ghi, nó chỉ được truy cập bởi thanh ghi A. Và để chọn thanh ghi truy cập, thì  $RAM[A]$  được thể hiện với kí hiệu M
- Đối với ROM, nó là bộ nhớ lệnh chỉ đọc. Chỉ được truy cập bởi thanh ghi A.  $ROM[A]$  chứa hướng dẫn lệnh hiện tại đang thực thi.

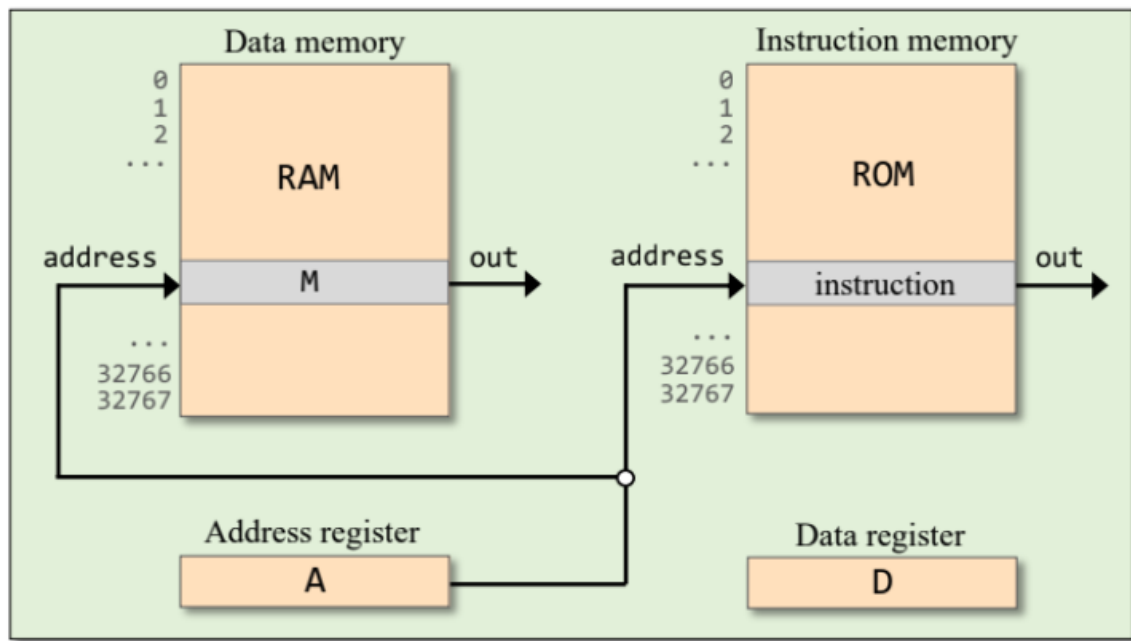


Figure 41: Thanh ghi của máy tính Hack

## 5.4 NGÔN NGỮ MÁY HACK

Mọi thao tác liên quan đến vị trí bộ nhớ yêu cầu hai lệnh Hack: Một để chọn địa chỉ mà ta muốn hoạt động và một để chỉ định hoạt động mong muốn. Ngôn ngữ Hack bao gồm 2 lệnh chung: lệnh địa chỉ, được gọi là lệnh A và lệnh tính toán (chỉ định hoạt động) được gọi là lệnh C). Mỗi một lệnh có một mã nhị phân biểu diễn, biểu diễn tượng trưng cho lệnh đó.

### 5.4.1 Câu lệnh A - (A instruction)

Lệnh này giúp máy tính lưu trữ giá trị được chỉ định trong thanh ghi A. Ví dụ với chỉ dẫn @5 nó sẽ biểu diễn giá trị 0000000000000101, rồi đưa vào máy tính để lưu trữ.

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

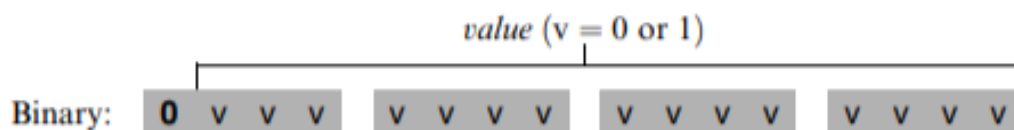


Figure 42: cú pháp câu lệnh AA

Câu lệnh A được sử dụng bởi 3 mục đích khác nhau: Đầu tiên, nó cung cấp cách nhập một hằng số vào máy tính dưới sự điều khiển của chương trình. Thứ hai, nó thiết lập giai đoạn cho một lệnh C tiếp theo được thiết kế để thao tác trên một bộ nhớ dữ liệu nhất định vị trí, bằng cách đầu tiên đặt A thành địa chỉ của vị trí đó. Thứ ba, nó tạo tiền đề cho một lệnh C tiếp theo chỉ định một bước nhảy, bằng cách tải trước tiên địa chỉ của nhảy đích đến thanh ghi A. Ví dụ về các mục đích đó:

1. Mục đích thứ nhất: @5, @19
2. Mục đích thứ hai:

@100



$M = 0$

### 3. Mục đích thứ ba:

@12  
D;JLT

#### 5.4.2 Câu lệnh C - (C instruction)

Câu lệnh C là quy trình lập trình của nền tảng Hack, đóng vai trò trong hầu hết các hướng dẫn. Hướng dẫn này có 3 vai trò chính: tính toán cái gì, nơi lưu trữ tài liệu đã tính toán và phải làm gì tiếp theo (có rẽ nhánh không). Cùng với hướng dẫn A, các thông số kĩ thuật này có thể xác định tất cả các hoạt động có thể có của máy tính.

**C-instruction:** *dest=comp;jump* // Either the *dest* or *jump* fields may be empty.  
// If *dest* is empty, the "=" is omitted;  
// If *jump* is empty, the ";" is omitted.

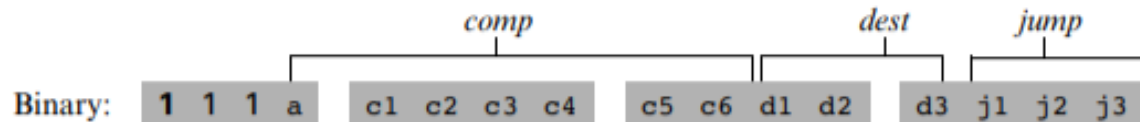


Figure 43: Cú pháp câu lệnh C

Bit ngoài cùng bên trái của mã lệnh C, là 1. Hai bit tiếp theo không được sử dụng. Các bit còn lại tạo thành ba trường tương ứng với ba phần của biểu diễn tượng trưng của hướng dẫn. Một hướng dẫn C tương đương với câu lệnh: *dest = comp; jump*. Trường "comp" hướng dẫn ALU những gì cần tính. Trường "dest" chỉ dẫn nơi lưu trữ giá trị đã tính (đầu ra ALU). Các trường "jump" chỉ định điều kiện nhảy, cụ thể là lệnh nào để tìm nạp và thực thi.

##### 5.4.2.a comp - field

Các toán hạng để tính toán bao gồm các thanh ghi, D A M (Trong đó M là Memory[A]). Trường comp này gồm 7 bit được chỉ định sau 3 bit 1 đầu tiên. Do có dung lượng 7 bit nên nó có thể biểu diễn 128 trường hợp khác nhau.

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

Figure 44: bảng chứa giá trị có thể có từ trường comp

#### 5.4.2.b dest - field

Giá trị được tính toán bởi phần comp của câu lệnh có thể được lưu trữ trong một số thanh ghi, ô nhớ, được chỉ định bởi hướng dẫn 3-bit. d1, chỉ định giá trị có lưu trữ trong thanh ghi A hay không, d2 chỉ định giá trị có lưu trữ trong thanh ghi D, tương tự d3 là thanh ghi M. Ví dụ:

```
0000 0000 0000 0111 // @7
1111 1101 1101 1000 // MD=M+1
```

d1	d2	d3	Mnemonic	Destination (where to store the computed value)
0	0	0	null	The value is not stored anywhere
0	0	1	M	Memory[A] (memory register addressed by A)
0	1	0	D	D register
0	1	1	MD	Memory[A] and D register
1	0	0	A	A register
1	0	1	AM	A register and Memory[A]
1	1	0	AD	A register and D register
1	1	1	AMD	A register, Memory[A], and D register

Figure 45: bảng chứa giá trị có thể có từ trường dest

#### 5.4.2.c jump-field

Trường jump của lệnh C cho máy tính biết điều gì để làm gì tiếp theo. Có hai khả năng: Máy tính sẽ tìm nạp và thực thi lệnh tiếp theo trong chương trình, là lệnh mặc định, hoặc nó sẽ tìm nạp và thực thi một lệnh nằm ở nơi khác trong chương trình (Trong lệnh thứ 2 giả sử đã có thanh ghi A được đặt trước địa chỉ mà chúng ta chuyển đến). J-bit đầu tiên chỉ định có nhảy trong trường hợp giá trị này là âm hay không, j-bit thứ hai trong trường hợp giá trị bằng 0 và bit j thứ ba trong trường hợp nó là số dương. Điều này cho tám điều kiện nhảy có thể có.

j1 (out < 0)	j2 (out = 0)	j3 (out > 0)	Mnemonic	Effect
0	0	0	null	No jump
0	0	1	JGT	If out > 0 jump
0	1	0	JEQ	If out = 0 jump
0	1	1	JGE	If out ≥ 0 jump
1	0	0	JLT	If out < 0 jump
1	0	1	JNE	If out ≠ 0 jump
1	1	0	JLE	If out ≤ 0 jump
1	1	1	JMP	Jump

Figure 46: bảng chứa giá trị có thể có từ trường jump, giá trị cuối là nhảy không điều kiện

### 5.4.3 Symbol

Các lệnh hợp ngữ có thể tham chiếu đến các vị trí bộ nhớ (địa chỉ) bằng cách sử dụng hằng số hoặc ký hiệu. Các ký hiệu được đưa vào chương trình hợp ngữ như sau theo ba cách: các ký hiệu được xác định trước, thanh ghi ảo: R0 -> R15 để xác định trước địa chỉ RAM từ 0 đến 15 tương ứng, các con trỏ được xác định trước: để thuận tiện cho việc truy cập vào ô nhớ.

1. **Con trỏ I / O:** Các ký hiệu SCREEN và KBD được xác định trước để chỉ RAM địa chỉ 16384 (0x4000) và 24576 (0x6000), tương ứng, là cơ sở địa chỉ của màn hình và bản đồ bộ nhớ bàn phím.
2. **Ký hiệu nhãn:** Các ký hiệu do người dùng xác định này, dùng để gắn nhãn các điểm đến của lệnh goto, được khai báo bằng lệnh giả "(abc)". Chỉ thị này xác định ký hiệu abc để chỉ vị trí bộ nhớ lệnh giữ tiếp theo lệnh trong chương trình. Một nhãn chỉ có thể được xác định một lần và có thể được sử dụng ở bất kỳ đâu trong chương trình hợp ngữ, ngay cả trước dòng mà nó được xác định.
3. **Ký hiệu biến:** Bất kỳ ký hiệu nào do người dùng xác định Xxx xuất hiện trong một chương trình hợp ngữ không được xác định trước và không được xác định ở nơi khác bằng cách sử dụng lệnh "(Xxx)" được coi là một biến và được gán một địa chỉ bộ nhớ duy nhất bởi trình hợp dịch, bắt đầu từ địa chỉ RAM 16 (0x0010). Chỉ sử dụng nếu khai báo trực tiếp ngay câu lệnh phía trên.

```
// sum = sum + x
@sum
D=M
@x
D=D+M
@sum
M=D
```

(a) variable

```
// if (RAM[100] < 0) goto 35
@100
D=M
@35
D;JLT
```

(b) label

Figure 47: Symbol

## 5.5 PROJECT

Trong tuần thứ tư, ta sẽ làm quen với hợp ngữ Hack, hiện thực hàm mult và fill (đuôi file .asm)

### 5.5.1 Mult

Multiply a,b

Peseudo Code:

```
int i = 1;
int mul = 0;
while (i <= b){
    mul = mul + a;
    i++;
}
```

```
1 //-----CODE-----
2
3 //Initialize
```

```
4      @i
5      M = 1
6      @mul
7      M = 0
8
9      (while)
10     //while (i <= b)
11     @R1
12     D = M
13     @i
14     D = M - D
15     @exit
16     D; JGT //i > b => out loop
17
18     //looping
19     @R0
20     D = M
21     @mul
22     M = D + M
23     @i
24     M = M + 1
25
26     @while //return loop
27     0; JMP
28
29     (exit)
30     @mul
31     D = M
32     @2
33     M = D
34     //end
35     @22
36     0; JMP
37     //-----
```

### 5.5.2 Fill

Giải thích hành vi:

- + Chạy một vòng lặp vô hạn để đợi bàn phím nhập.
- + Khi nhấn một phím (phím bất kỳ), chương trình sẽ bôi đen màn hình, tức là viết "đen" trong mỗi pixel. Màn hình sẽ vẫn hoàn toàn màu đen miễn là giữ phím được nhấn.
- + Khi không có phím nào được nhấn, chương trình sẽ xóa màn hình, tức là ghi "trắng" trong mọi pixel. Màn hình vẫn hoàn toàn rõ ràng miễn là không có phím nào được nhấn.

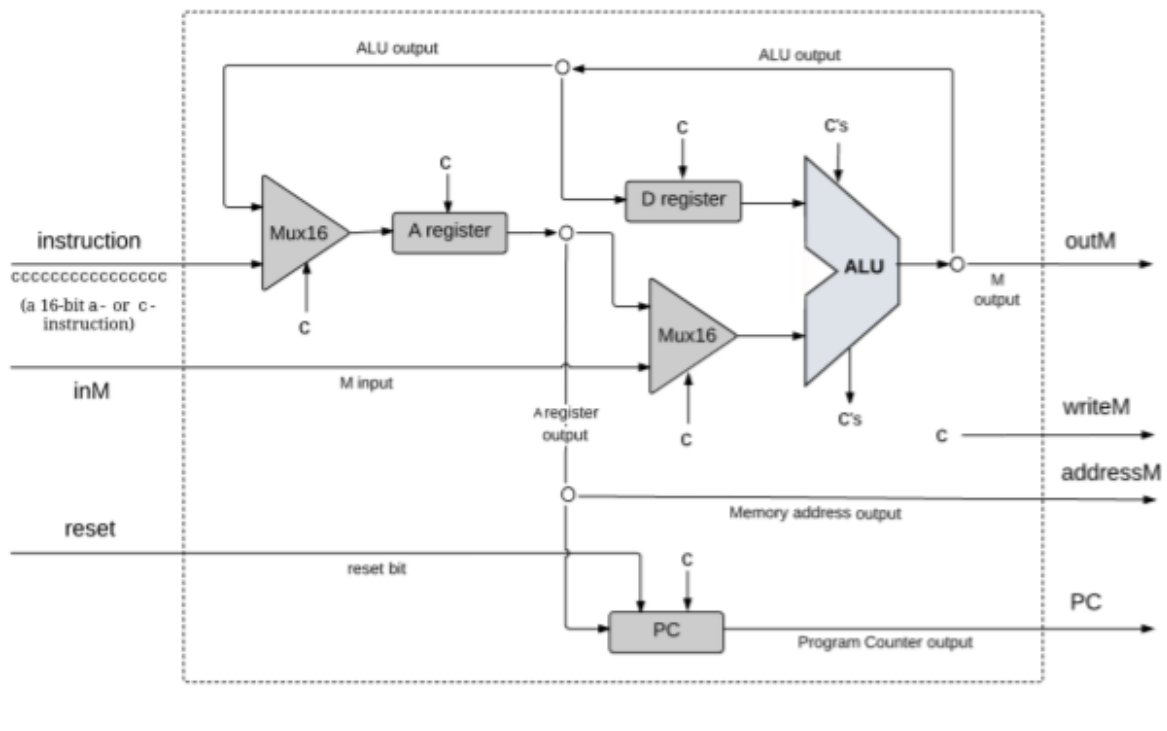
```
1      @KBD
2      D = A //check pointer index
3      @const
4      M = D //save value of Keyboard
5      (MAIN)
6      @SCREEN
7      D = A
8      @R0
9      M = D
10     (CHECK)
11     @KBD
12     D = M
13     @BLACK
14     D; JGT // if press then black => jump BLACK
```

```
15  @WHITE
16  0; JMP //else case
17  (BLACK)
18  @R1
19  M = -1 // (-1 = 1111 1111 1111 1111)
20  @CHANGE
21  0; JMP
22  (WHITE)
23  @R1
24  M = 0 // (0 = 0000 0000 0000 0000)
25  @CHANGE
26  0; JMP
27
28  (CHANGE)
29  @R1
30  D = M // set value is 0 or 1 depend on D
31
32  @R0
33  A = M
34  M = D // fill a space with color which is chosen
35
36  @R0
37  M = M + 1 //next line
38  D = M
39  @const
40  D = D - M
41
42  @CHANGE
43  D; JLT
44
45  @MAIN
46  0; JMP
```

## 6 WEEK 5 - COMPUTER ARCHITECTURE

## 6.1 CPU

Khi bắt đầu triển khai Hack CPU, mục tiêu của chúng ta là đưa ra một cổng logic kiến trúc có khả năng (i) thực hiện một lệnh Hack nhất định và (ii) xác định lệnh sẽ được tìm nạp và thực thi tiếp theo. Để làm như vậy, CPU được đề xuất triển khai bao gồm một chip ALU có khả năng tính toán các chức năng số học / logic, một bộ thanh ghi, bộ đếm chương trình và một số cổng bổ sung được thiết kế để giúp giải mã, thực thi và tìm nạp hướng dẫn. Vì tất cả các khối xây dựng này đã được xây dựng trong các tuần trước, câu hỏi chính mà chúng ta phải đối mặt bây giờ là làm thế nào để sắp xếp và kết nối chúng theo cách ảnh hưởng đến hoạt động mong muốn của CPU



**Figure 48:** KTMT của Hack-Computer với 3 hoạt động cơ bản: giải mã lệnh hiện tại, thực hiện lệnh hiện tại và quyết định lệnh nào để tìm nạp

## 6.2 INSTRUCTION DECODING - Giải mã lệnh

Giá trị 16 bit của đầu vào lệnh của CPU đại diện cho lệnh A hoặc lệnh C. Để tìm ra ngữ nghĩa của hướng dẫn này, chúng ta có thể phân tích cú pháp nó, vào các trường sau: `ixxacccccddjjj`. i-bit còn gọi là op-code để xác định xem nếu  $i = 0$  tức đây là lệnh A còn  $i = 1$  tức đây là lệnh C. Trong trường hợp là lệnh A, toàn bộ lệnh sẽ đại diện cho giá trị hằng số được ghi vào. Trong trường hợp là lệnh C, các bit a và c mã hóa phần comp, các bit mã hóa phần dest, và bit j để mã hóa phần jump. Các bit x không đóng vai trò trong lệnh C.

### 6.3 INSTRUCTION EXECUTION - Thực thi lệnh

Các trường được giải mã của lệnh (i-, a-, c-, d-, và j-bit) được định tuyến đồng thời tới các bộ phận khác nhau của kiến trúc CPU, nơi chúng khiến các bộ phận chip khác nhau thực hiện chức năng của chúng phải làm để thực hiện lệnh A hoặc C, theo sự ủy quyền của Hack để đặc tả ngôn ngữ máy. Trong trường hợp của một lệnh C, một bit đơn xác định ALU sẽ hoạt động trên đầu vào thanh ghi A hay đầu vào M

và sáu bit c xác định hàm ALU sẽ tính toán. Ba d-bit được sử dụng để xác định thanh ghi phải “chấp nhận” đầu ra kết quả ALU và ba bit j được sử dụng để phân nhánh kiểm soát. J có thể được tính bằng cách sử dụng logic, và sau đó giúp tính toán địa chỉ hướng dẫn tiếp theo

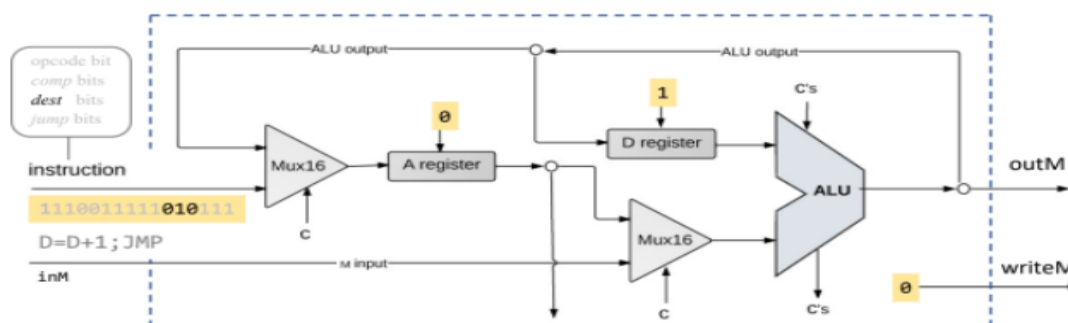


Figure 49: Định tuyến các bit d của lệnh tới các bit điều khiển (tải) của Thanh ghi A, thanh ghi D và bit writeM

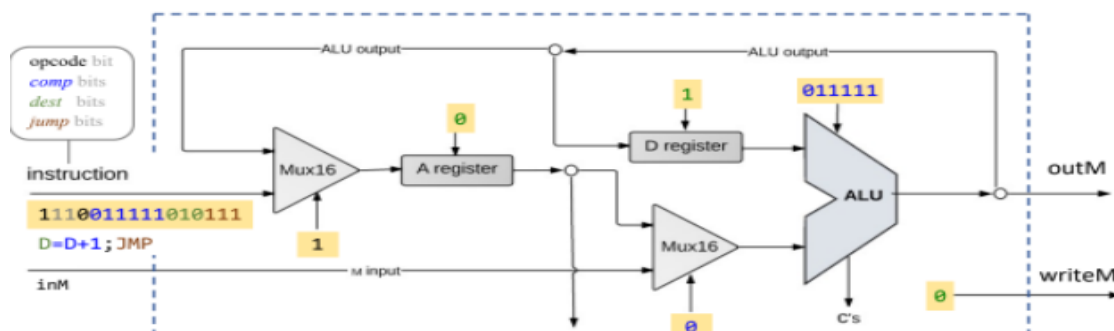


Figure 50: thực thi dest = comp

jump	j1	j2	j3	condition
null	0	0	0	no jump
JGT	0	0	1	if (ALU out > 0) jump
JEQ	0	1	0	if (ALU out = 0) jump
JGE	0	1	1	if (ALU out ≥ 0) jump
JLT	1	0	0	if (ALU out < 0) jump
JNE	1	0	1	if (ALU out ≠ 0) jump
JLE	1	1	0	if (ALU out ≤ 0) jump
JMP	1	1	1	Unconditional jump

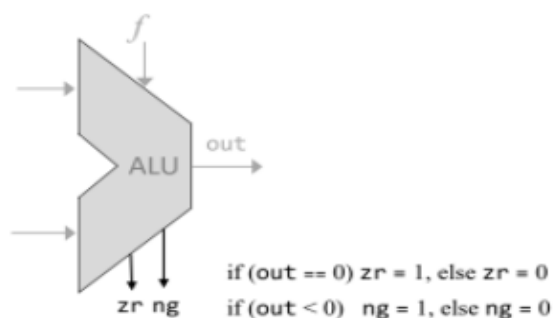


Figure 51: Jump decision

## 6.4 INSTRUCTION FETCHING - NẠP LỆNH

+ CPU phải xác định và phát ra địa chỉ của lệnh sẽ được tìm nạp và thực thi tiếp theo. Phần tử quan trọng trong nhiệm vụ con này là PC — một bộ phận chip CPU có vai trò là luôn lưu trữ địa chỉ của hướng dẫn tiếp theo. Theo đặc điểm kỹ thuật máy tính Hack, chương trình hiện tại được lưu trữ trong bộ nhớ lệnh, bắt đầu từ địa chỉ 0. Do đó, nếu chúng ta muốn bắt đầu (hoặc khởi động lại) một chương trình thực thi, chúng ta nên đặt lại PC thành 0. Việc đặt lại đầu vào của CPU được đưa trực tiếp vào đầu vào đặt lại của chip PC. Tiếp theo, nếu không có tín hiệu nào can thiệp hoạt động mặc định của PC sẽ là PC++. Còn về việc hiện thực câu lệnh "jump n" trong đó n là địa chỉ của hướng dẫn nằm ở bất kỳ đâu trong chương trình. Đầu tiên hiện thực lệnh A @n, lệnh này đặt thanh ghi A thành n; tiếp theo, thực thi lệnh C bao gồm một chỉ thị nhảy. Theo đặc tả ngôn ngữ, việc thực thi luôn luôn rẽ nhánh tới lệnh mà thanh ghi A trỏ tới.

+ Để xác định được tín hiệu làm cho  $PC = A$ . Nó được xác định bởi ba tín hiệu. Thứ nhất: j-bit của lệnh hiện tại, xác định bước nhảy. Các bit đầu ra ALU zr và ng, có thể được sử dụng để xác định xem điều kiện được thỏa mãn, hay không.

Abstraction:

Output of the address of the next instruction

+ reset:  $PC = 0$

+ no jump:  $PC++$

+ jump: if (condition)  $PC = AA$

## 6.5 INPUT/OUTPUT DEVICE

### 6.5.1 SCREEN

+ Máy tính Hack có thể tương tác với một màn hình vật lý bao gồm 256 hàng 512 mỗi pixel đen trắng. Máy tính giao tiếp với màn hình vật lý thông qua bộ nhớ bản đồ, được thực hiện bởi một chip RAM có tên là Screen. Con chip này hoạt động giống như bộ nhớ thông thường, nghĩa là nó có thể được đọc và ghi vào. Ngoài ra, nó có tác dụng phụ mà bất kỳ được ghi vào nó được phản ánh dưới dạng pixel trên màn hình vật lý (1 = đen, 0 = trắng).

+ Mỗi hàng trong màn hình vật lý, bắt đầu từ góc trên cùng bên trái, được thể hiện trong Bản đồ bộ nhớ màn hình bằng 32 từ 16 bit liên tiếp. Do đó pixel ở hàng r từ trên cùng và cột c từ bên trái ( $0 \leq r \leq 255$ ,  $0 \leq c \leq 511$ ) được ánh xạ trên c % 16 bit (tính từ LSB đến MSB) của từ 16 bit được lưu trữ trong  $Screen[r * 32 + c / 16]$

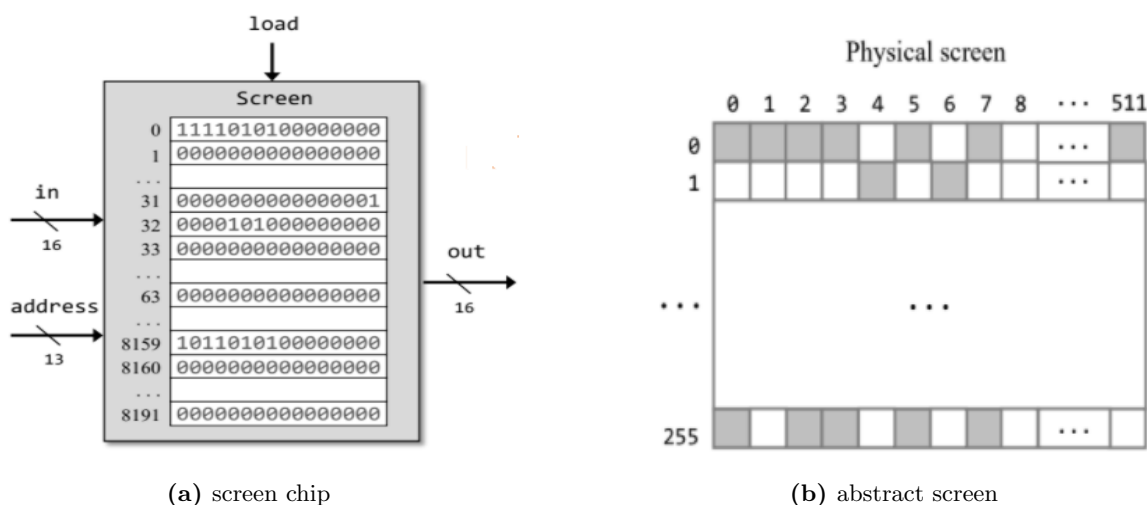


Figure 52: screen

### 6.5.2 KEYBOARD

Có thể tương tác với bàn phím vật lý, giống như bàn phím cá nhân máy vi tính. Máy tính giao tiếp với bàn phím vật lý thông qua một chip có tên là Keyboard. Khi nhấn một phím trên bàn phím vật lý, một mã quét 16 bit duy nhất sẽ được gửi tới đầu ra của chip Keyboard. Khi không có phím nào được nhấn, chip sẽ xuất ra 0.



Figure 53: Keyboard chip có độ dài 16 bit



key	code	key	code	key	code	key	code	key	code
(space)	32	0	48	A	65	a	97	newline	128
!	33	1	49	B	66	b	98	backspace	129
"	34	...	...	C	...	c	99	left arrow	130
#	35	9	57	...	...	...	...	up arrow	131
\$	36	:	58	Z	90	z	122	right arrow	132
%	37	;	59	[	91	{	123	down arrow	133
&	38	<	60	/	92		124	home	134
'	39	=	61	]	93	}	125	end	135
(	40	>	62	^	94	~	126	Page up	136
)	41	?	63	_	95			Page down	137
*	42	@	64	`	96			insert	138
+	43							delete	139
,	44							esc	140
-	45							f1	141
.	46							...	...
/	47							f12	152

(Subset of Unicode)

Figure 54: bảng kí tự tương ứng với code của nó khi giải mã

## 6.6 Data Memory

Không gian địa chỉ tổng thể được gọi là bộ nhớ dữ liệu Hack được thực hiện bởi một con chip có tên là Memory. Con chip này về cơ bản là một gói ba thiết bị lưu trữ 16 bit: một RAM (16K thanh ghi, cho lưu trữ dữ liệu thông thường), Màn hình (thanh ghi 8K, hoạt động như bản đồ bộ nhớ màn hình), và Bàn phím (1 thanh ghi, hoạt động như bản đồ bộ nhớ bàn phím).

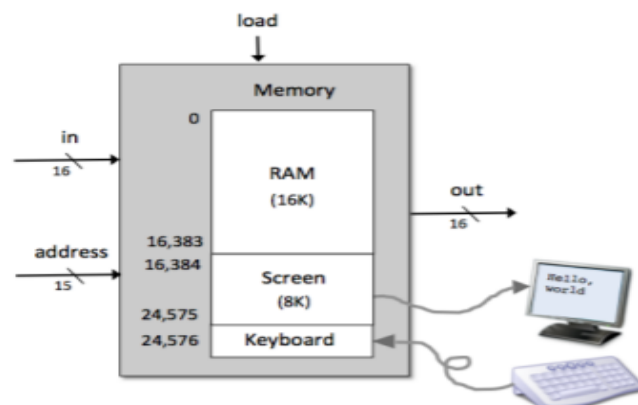


Figure 55: Memory Chip

## 6.7 COMPUTER

Chip cao nhất trong hệ thống phân cấp phần cứng Hack là chip Computer bao gồm một CPU, một bộ nhớ lệnh và bộ nhớ dữ liệu. Máy tính có thể tương tác với màn hình và bàn phím.

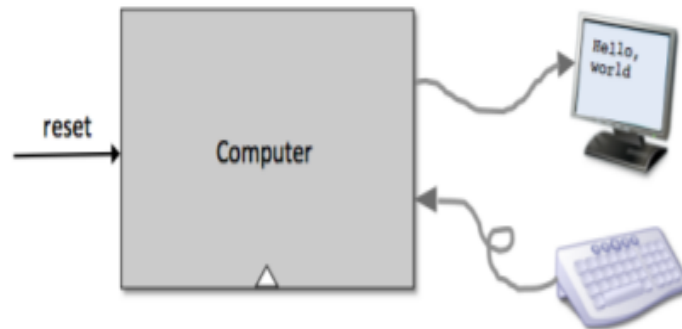


Figure 56: Computer Chip

## 6.8 PROJECT

Trong tuần thứ 5 ta sẽ hoàn thành các chip Memory, Computer nhưng phần cốt lõi quan trọng nhất là CPU.




	Computer.hdl	D:\My Workspace\project\NandToTeris\nand2tetris\...	Type: HDL File
	CPU.hdl	D:\My Workspace\project\NandToTeris\nand2tetris\...	Type: HDL File
	Memory.hdl	D:\My Workspace\project\NandToTeris\nand2tetris\...	Type: HDL File

Figure 57: project 5 hiện thực phần cứng quan trọng nhất của máy tính

### 6.8.1 CPU.hdl

Hack CPU (Đơn vị xử lý trung tâm), bao gồm một ALU, hai thanh ghi có tên A và D, và một bộ đếm chương trình có tên là PC. CPU được thiết kế để tìm nạp và thực thi các lệnh được viết bằng ngôn ngữ máy Hack. Cụ thể, các chức năng như sau:

- + Thực hiện lệnh đã nhập theo máy Hack đặc tả ngôn ngữ. D và A trong đặc tả ngôn ngữ đề cập đến các thanh ghi của CPU, trong khi M đề cập đến vị trí bộ nhớ ngoài được A định địa chỉ, tức là tới Memory[A].
- + Đầu vào inM giữ giá trị của vị trí này. Nếu lệnh hiện tại cần ghi giá trị vào M, giá trị được đặt trong outM, địa chỉ của vị trí đích được đặt trong đầu ra addressM và bit điều khiển writeM được xác nhận. (Khi writeM == 0, bất kỳ giá trị nào có thể xuất hiện trong outM).
- + Nếu reset == 1 thì CPU sẽ chuyển đến địa chỉ 0 (tức là máy tính được đặt thành 0 trong bước thời gian tiếp theo) chứ không phải đến địa chỉ do thực hiện lệnh hiện tại.

```

1 CHIP CPU {
2     IN inM[16],           // M value input (M = contents of RAM[A])
3     instruction[16],      // Instruction for execution
4     reset;                // Signals whether to re-start the current
5                           // program (reset==1) or continue executing
6                           // the current program (reset==0).
7
8     OUT outM[16],         // M value output
9     writeM,               // Write to M?
10    addressM[15],          // Address in data memory (of M)
11    pc[15];                // address of next instruction

```

```
12
13 PARTS:
14 // Put your code here:
15 //---Load A or Not ---//
16 //-----
17 //A-type if instruction[15] == 0
18 Not(in = instruction[15], out = Ains);
19 Not(in = Ains, out = Cins);
20
21 //instructionC with dest to A-reg
22 And(a = Cins, b = instruction[5], out = ALUtoA);
23 //Choose A-type or C-type with MUX
24 Mux16(a = instruction, b = ALUout, sel = ALUtoA, out = Aregin);
25
26 // Decide whether to load to Areg
27 Or(a = Ains, b = ALUtoA, out = loadA);
28 // load A if A-inst or C-inst&dest to A-reg
29 ARegister(in = Aregin, load = loadA, out = Aout);
30
31 //select A or M for ALU input
32 Mux16(a = Aout, b = inM, sel = instruction[12], out = AMout);
33 //-----
34
35 //---Load D or Not ---//
36 //-----
37 And(a=Cins, b=instruction[4], out=loadD);
38 DRegister(in=ALUout, load=loadD, out=Dout); // load the Dreg
39 //-----
40
41 //set-up ALU
42 ALU(x = Dout, y = AMout, zx = instruction[11], nx = instruction[10],
43     zy = instruction[9], ny = instruction[8], f = instruction[7],
44     no = instruction[6], out = ALUout, zr = ZRout, ng = NGout); //zr is zero-flag, ng is
45     negative-flag
46
47 //set outputs for writing memory
48 Or16(a = false, b = Aout, out[0..14] = addressM);
49 Or16(a = false, b = ALUout, out = outM);
50
51 And(a = Cins, b = instruction[3], out = writeM);
52
53 //calc PCload & PCinc - whether to load PC with A reg
54 And(a = ZRout, b = instruction[1], out = jeq); // == 0 ?
55 And(a = NGout, b = instruction[2], out = jlt); // Jump less than ?
56 Or(a = ZRout, b = NGout, out = zeroOrNeg);
57
58 Not(in = zeroOrNeg, out = positive); // > 0 ?
59 And(a = positive, b = instruction[0], out = jgt); // Jump greater than ?
60
61 Or(a = jeq, b = jlt, out = jle); // <= 0 ?
62 Or(a = jle, b = jgt, out = jumpToA); // load PC?
63
64 And(a = Cins, b = jumpToA, out = PCload); // Only jump if is instructionC
65
66 //if load is not found
67 Not(in = PCload, out = PCinc);
68
69 //output PC
70 PC(in = Aout, inc = PCinc, load = PCload, reset = reset, out[0..14] = pc);
71 }
```

### 6.8.2 Memory.hdl

1) Không gian địa chỉ hoàn chỉnh của bộ nhớ máy tính Hack, bao gồm RAM và I / O được ánh xạ bộ nhớ. Con chip này tạo điều kiện cho các hoạt động đọc và ghi, như sau:

+ Đọc:  $out(t) = \text{Bộ nhớ}[\text{địa chỉ}(t)](t)$

+ Ghi: if load  $(t-1)$  then Memory  $[\text{address}(t-1)](t) = in(t-1)$

2) Quy tắc không gian địa chỉ: Quyền truy cập vào địa chỉ  $> 0x6000$  không hợp lệ. Truy cập vào bất kỳ địa chỉ nào trong phạm vi  $0x4000-0x5FFF$  dẫn đến việc truy cập bộ nhớ screen map. Truy cập vào địa chỉ  $0x6000$  dẫn đến truy cập bàn phím.

```
1 CHIP Memory {
2     IN in[16], load, address[15];
3     OUT out[16];
4
5     PARTS:
6     // Put your code here:
7     DMux4Way(in = load, sel = address[13..14], a = ram1, b = ram2, c = screen, d = keyboard);
8
9     Or(a = ram1, b = ram2, out = ram);
10
11     RAM16K(in = in, load = ram, address = address[0..13], out = r);
12
13     Screen(in = in, load = screen, address = address[0..12], out = scrn);
14
15     Keyboard(out = k);
16
17     Mux4Way16(a = r, b = r, c = scrn, d = k, sel = address[13..14], out = out);
18 }
```

### 6.8.3 Computer.hdl

Máy tính HACK, bao gồm CPU, ROM và RAM. Khi reset bằng 0, chương trình được lưu trữ trong ROM của máy tính sẽ thực thi. Khi reset là 1, việc thực thi chương trình sẽ khởi động lại. Do đó, để bắt đầu thực thi một chương trình, thiết lập lại phải được đẩy "lên" (1) và "xuống" (0). Kể từ thời điểm này trở đi, người dùng sẽ sử dụng phần mềm. Đặc biệt, tùy thuộc vào mã của chương trình, màn hình có thể hiển thị một số đầu ra và người dùng có thể tương tác với máy tính thông qua bàn phím.

```
1 CHIP Computer {
2
3     IN reset;
4
5     PARTS:
6     // Put your code here:
7
8     //combine CPU and memory
9     ROM32K(address = pc, out = instruction);
10
11     CPU(inM = inM, instruction = instruction, reset = reset, outM = outM, writeM = writeM,
12         addressM = addressM, pc = pc);
13
14     Memory(in = outM, load = writeM, address = addressM, out = inM);
15 }
```

## 7 WEEK 6 - ASSEMBLER

### 7.1 ĐỊNH NGHĨA:

Trước khi chương trình hợp ngữ có thể được thực thi trên máy tính, chương trình đó phải được dịch sang ngôn ngữ máy nhị phân của máy tính. Nhiệm vụ dịch là của assembler, nó lấy đầu vào một dòng lệnh hợp ngữ và tạo ra dưới dạng đầu ra một dòng lệnh nhị phân tương đương. Mã kết quả có thể được tải vào bộ nhớ của máy tính và được thực thi bởi phần cứng. Ta thấy rằng assembler về cơ bản là một chương trình xử lý văn bản, được thiết kế để cung cấp dịch vụ dịch thuật. Các nhiệm vụ cơ bản của Assembler:

- + **Preprocessing:** Xử lý khoảng trắng, bỏ comment.
- + **Lexer and Parser:** Phân tích cú pháp lệnh tương trưng vào trường cơ bản của nó. Đối với mỗi trường, tạo các bit tương ứng trong ngôn ngữ máy.
- + **handleSymbol:** Thay thế tất cả các tham chiếu tượng trưng (nếu có) bằng các địa chỉ số của bộ nhớ.
- + **Assemble:** Tập hợp các mã nhị phân thành một lệnh máy hoàn chỉnh.

**Note:** Để tìm hiểu về ASSEMBLER, trong những phần tiếp theo ta sẽ hiểu nó theo prototype và chức năng nó cần làm là gì, các source code dưới đây được viết theo ngôn ngữ lập trình C++.

### 7.2 PREPROCESS:

Chức năng chính của module này để xóa không trắng, xóa comment và giữ lại các câu lệnh và tập hợp chúng lại trước khi đi qua các module tiếp theo.

```
1 //prototype:
2 bool delete_partial(string& cmd);
3 //Use to remove the comment, the white distance from each sentences.
4 vector <string> preprocessing();
5 //After processing each sentence, assemble them to send to another module for further
   processing
```

```
// Computes R1=1+...+R0
// i = 1
@16
M=1
// sum = 0
@17
M=0

// if i>R0 goto STOP
@16
D=M
@0
D=D-M
@18
D;JGT
// sum += i
@16
D=M
@17
M=D+M
// i++
@16
M=M+1
@4
0;JMP
@17
D=M
...
```

(a) text before preprocessing

```
@16
M=1
@17
M=0
@16
D=M
@0
D=D-M
@18
D;JGT
@16
D=M
@17
M=D+M
@16
M=M+1
@4
0;JMP
@17
D=M
...
```

(b) text after preprocessing

Figure 58: preprocessing process

### 7.3 LEXER AND PARSER MODULE:

LEXER là phân tích cú pháp với đơn vị là char còn Parser phân tích cú pháp token. Chức năng chính của trình phân tích cú pháp là ngắt mỗi lệnh hợp ngữ thành các thành phần cơ bản của nó (trường và ký hiệu). Chúng ta cần phải khởi tạo các bảng để ánh xạ trực tiếp các thành phần jump, comp, dest khi parser với mã nhị phân tương ứng của chúng.

<i>comp</i>		c	c	c	c	c	c
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
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

<i>dest</i>		d	d	d
null		0	0	0
M		0	0	1
D		0	1	0
DM		0	1	1
A		1	0	0
AM		1	0	1
AD		1	1	0
ADM		1	1	1

<i>jump</i>		j	j	j
null		0	0	0
JGT		0	0	1
JEQ		0	1	0
JGE		0	1	1
JLT		1	0	0
JNE		1	0	1
JLE		1	1	0
JMP		1	1	1

Figure 59: Các table cần init

```

1 #define mss map<string,string>
2
3 //-----prototype-----
4 //init
5 void convertBin(vector <string>&);
6
7 //make-paritalTable
8 void init_table(mss&, mss&, mss&);
9 //A-instruction
10 void processA(string&);
11
12 //C-instruction
13 //helper function:
14 //parser
15 vector <string> parser(const string&);
16 void processC(string&,mss&,mss&,mss&);
17 //-----

```

### 7.4 SYMBOL TABLE

+ Vì hướng dẫn Hack có thể chứa các ký hiệu, các ký hiệu phải được giải quyết thành địa chỉ thực tế như một phần của quá trình dịch. Trình lắp ráp giải quyết công việc này sử dụng bảng ký hiệu, được thiết kế để tạo và duy trì sự tương ứng giữa các ký hiệu và ý nghĩa của chúng (trong trường hợp của Hack, địa chỉ RAM và ROM). Một cách tự nhiên cấu trúc dữ liệu để biểu diễn mối quan hệ như vậy là bảng băm cổ điển. Ở hầu hết các ngôn ngữ lập trình, cấu trúc dữ liệu như vậy có sẵn như một phần của tiêu chuẩn thư viện, và do đó không cần phải phát triển nó từ đầu.

+ Khi mới vừa khởi tạo assembler: bảng symbol table sẽ định nghĩa sẵn 23 label ánh xạ trực tiếp giá trị đó.

symbol	value
R0	0

R1	1
R2	2
R3	3
...	...
R15	15
SCREEN	16384
KBD	24576
SP	0
LCL	1
ARG	2
THIS	3
THAT	4

+ **Label symbol:** Được sử dụng để gắn nhãn các điểm đến của hướng dẫn goto. Được khai báo bởi lệnh giả (label). Chỉ thị (label) xác định nhãn ký hiệu để chỉ vị trí bộ nhớ chứa lệnh tiếp theo trong chương trình, Tương ứng với số dòng của lệnh.

+ **Variable symbol:** Bất kỳ ký hiệu xxx nào không được xác định trước hoặc không được xác định ở nơi khác bằng cách sử dụng nhãn (xxx) khai báo, được coi như một biến. Mỗi biến được liên kết với một địa chỉ bộ nhớ đang chạy trong symbolTable, bắt đầu từ địa chỉ 16.

```
1 #define msi map<string,int>
2 //-----prototype-----
3 //detectTable
4 void detectSymbol(vector <string>& content);
5 //symbol_table_init
6 void symbol_table_init(msi&);
7 //detect (xxx) - label
8 //helper function:
9 string getInParenthesis(const string& s);
10 void detectLabel(vector <string>& line,msi& symTable);
11 //detect (@d+) - variable
12 //helper function:
13 bool isNumber(const string& s);
14 void detectVariable(vector <string>& line,msi& symTable);
15 //-----
```

## 7.5 PROJECT

Trong tuần cuối này, ta sẽ implement một assembler bằng một ngôn ngữ cấp cao bất kì. Nhiệm vụ là phải dịch từ hợp ngữ sang mã máy và gửi các file mã máy về nand2tetris để kiểm tra. Ngôn ngữ em chọn là C++. Source code đã implement trong đường link dưới đây và theo những prototype được khai báo ở trên: [github](#)

## 8 REFERENCES

- [NAND2TETRIS-ORG](#)
- [COURSERA](#)
- [NAND2TETRIS-YOUTUBE](#)
- [WIKIPEDIA](#)
- [THE ELEMENTS OF COMPUTING SYSTEMS](#)
- [COMPUTER ORGANIZATION AND DESIGN 4TH](#)
- [DIGITAL SYSTEM](#)