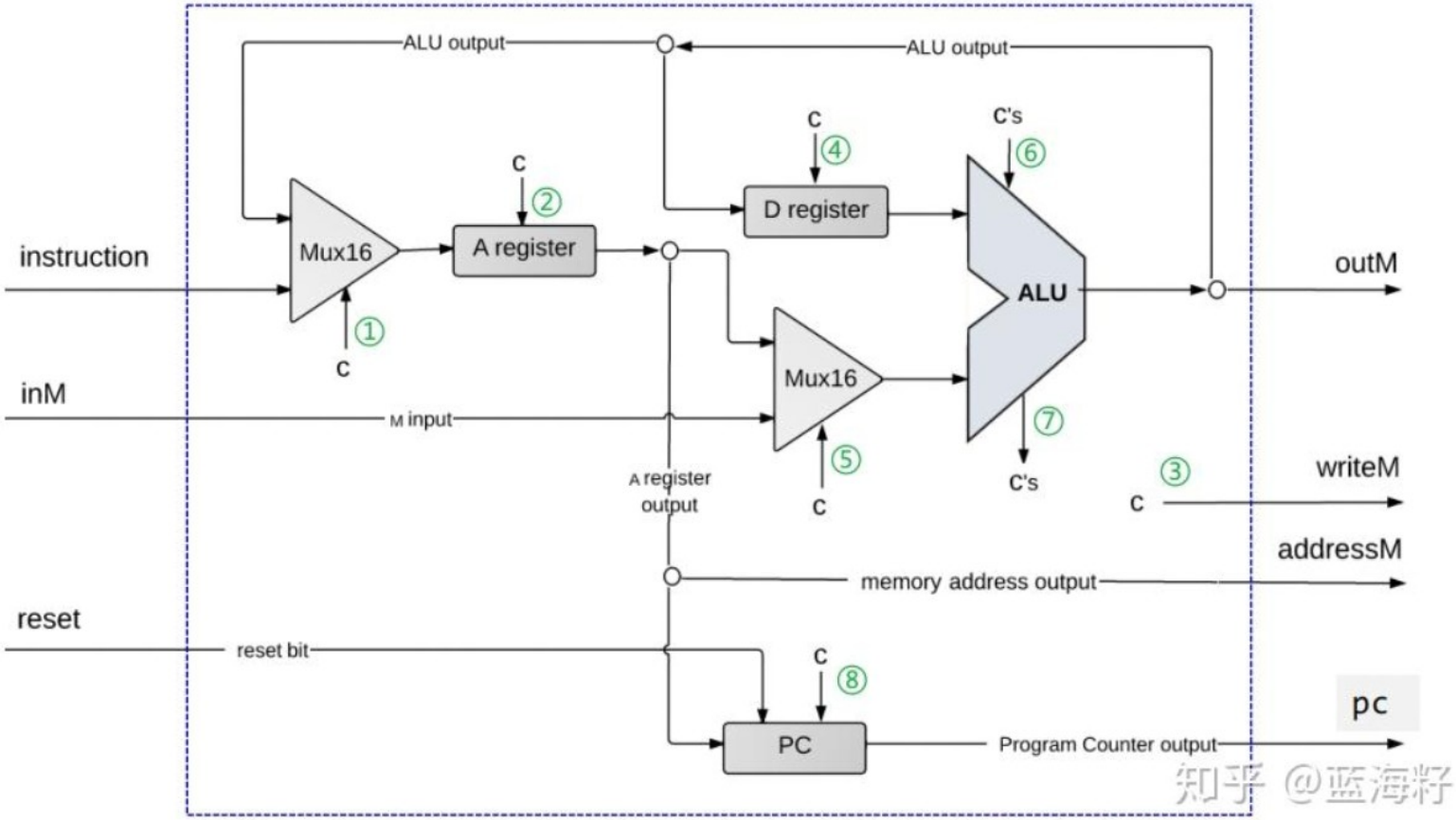
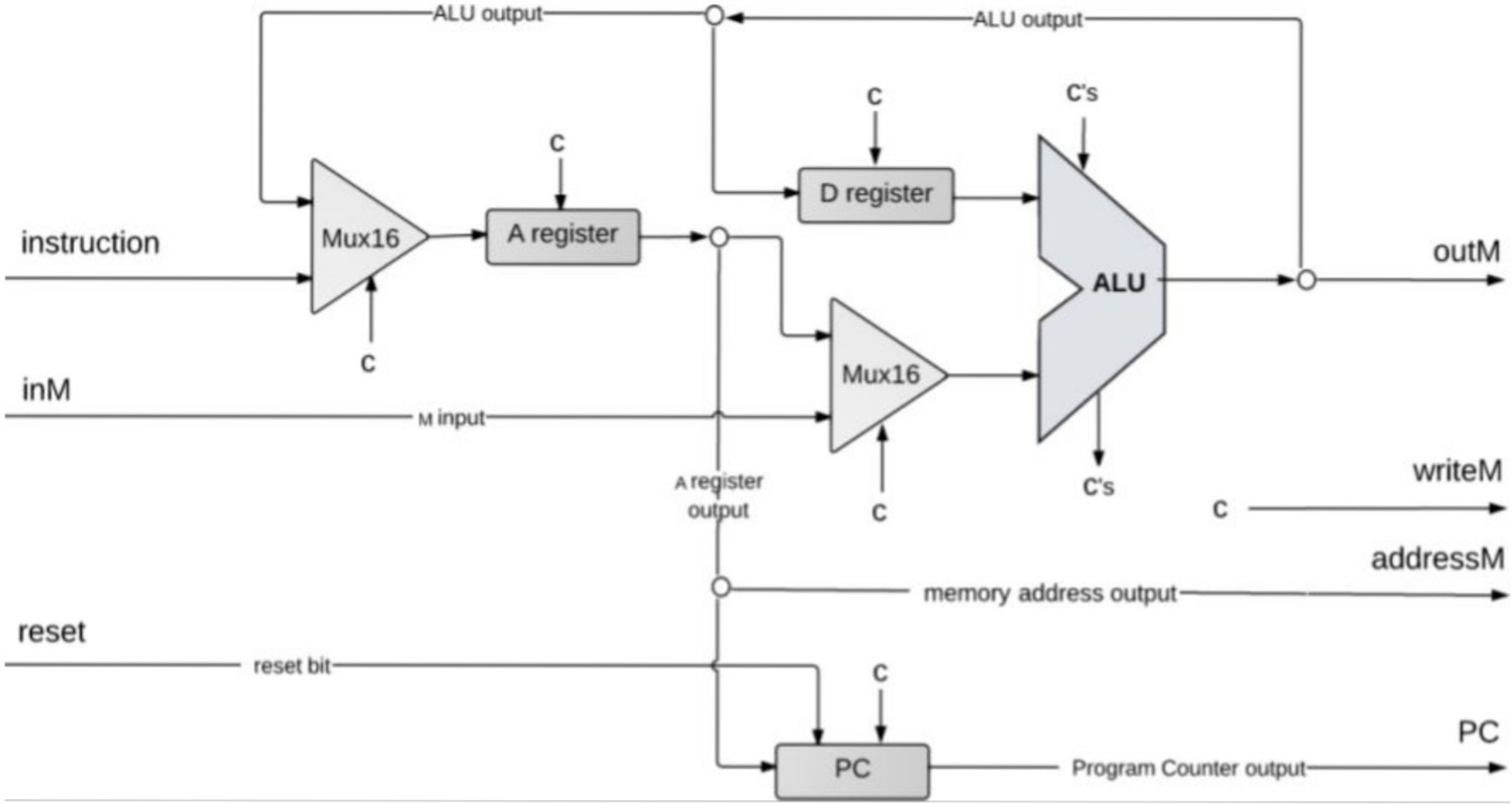
Kontrollbit-Designpunkte

Lassen Sie mich die Hauptpunkte des CPU-Designs analysieren

1.Wie wählt der erste Mux den Eingang aus? Offensichtlich hofft man für den A-Befehl, den Befehl als Adresse in das A-Register einzulesen. Wenn der Befehl ein C-Befehl ist, wird die ALU ausgewählt `aluoutput`.
2.Der A-Befehl muss auf das A-Register zugreifen, und der C-Befehl beurteilt anhand des 5. Bits des Befehls, ob auf das A-Register zugegriffen werden soll.
3.Wird die Ausgabe des A-Registers in den Speicher zurückgeschrieben `writeM`? Der A-Befehl schreibt niemals zurück, und der C-Befehl beurteilt anhand des dritten Bits.
4.Der Sprungzugriff auf das D-Register `D register`, das M-Register `inM`und den PC `load`hat nichts mit dem A-Befehl zu tun. Wenn es sich also um einen A-Befehl handelt, ist das Steuerbit ④⑤⑧ falsch; die Steuerinformationen bei ④, der C-Befehl wird entsprechend beurteilt 4. Bit;
5.⑤ Steuerinformationen: Der C-Befehl beurteilt anhand des 12. Bits;
6.⑥ Für Steuerinformationen beurteilt der C-Befehl anhand des 6. bis 11. Bits;
7.⑦ ist die Symbolausgabebitsumme der ALU , `ng`und `zr`die Untertabelle gibt das negative Bit und das Nullbit an, und das positive Bit kann indirekt berechnet werden.
8.Wenn das negative Bit, das Nullbit und das positive Bit jeweils `jmp`mit der Phase des C-Befehls kombiniert werden und keines davon Null ist (es liegt ein Sprung vor), wird das Sprungsteuerbit bei ⑧ erhalten `true`.

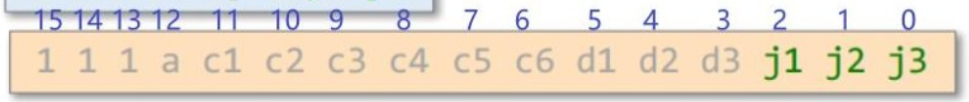
CPU operation



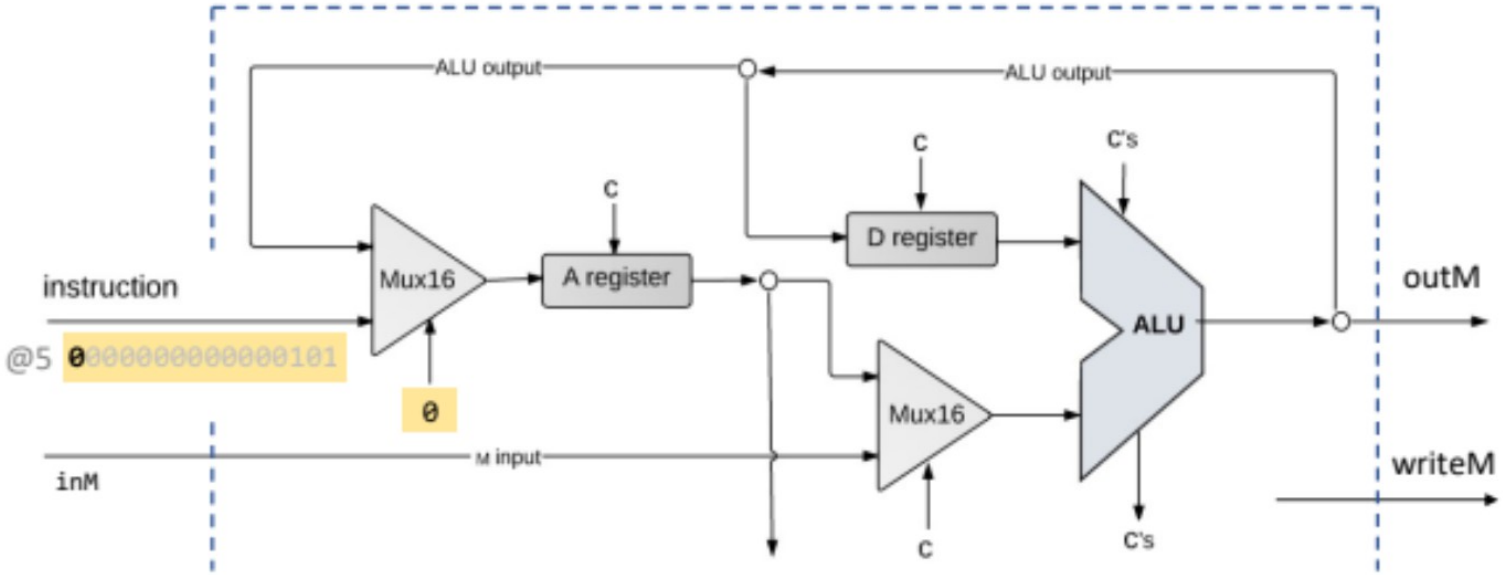

ALU output
ALU output
C
C's
C
D register
instruction
Mux16
A register
ALU
outM
C
Mux16
inM
M input
C's
writeM
A register output
C
C
addressM
memory address output
reset
reset bit
C
PC
PC
Program Counter output

C-instruction specification

Symbolic syntax: *dest* = *comp* ; *jump*

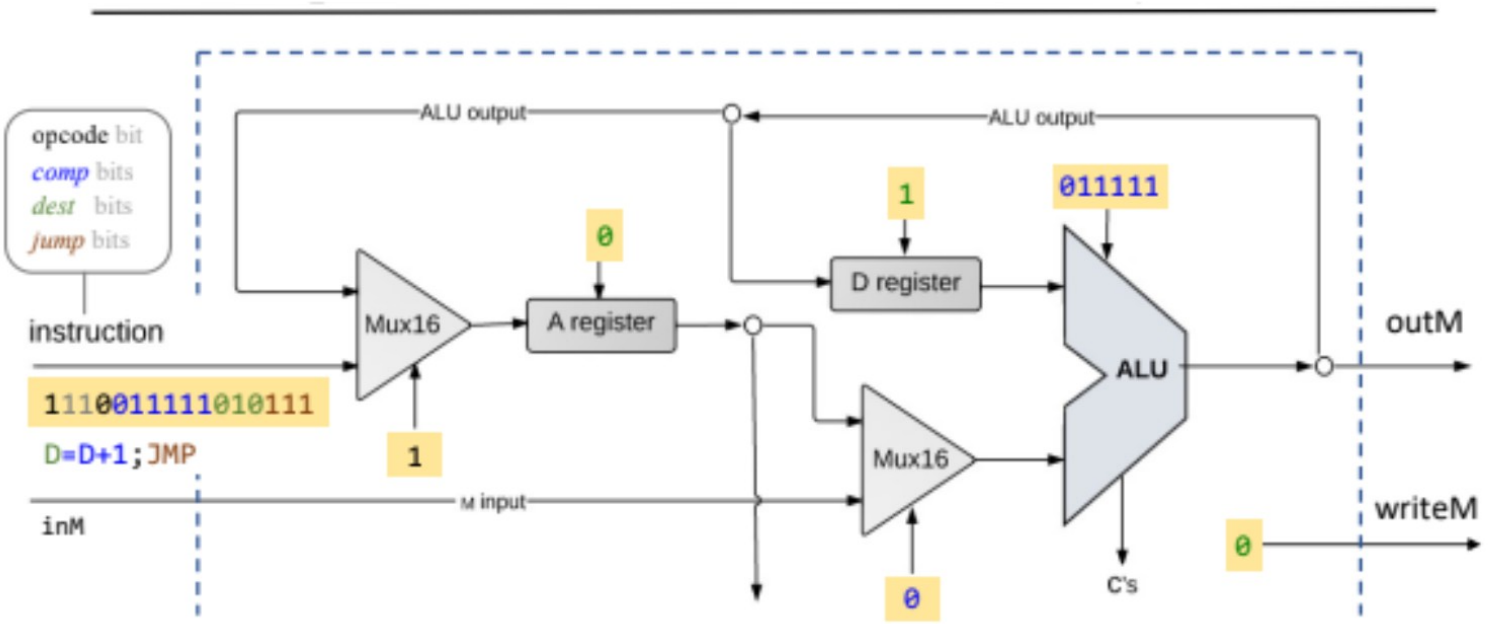


jump	j1	j2	J3	effect
null	0	0	0	no jump
JGT	0	0	1	if out>0 jump
JEQ	0	1	0	if out=0 jump
JGE	0	1	1	if out≥0 jump
JLT	1	0	0	if out<0 jump
JNE	1	0	1	if out≠0 jump
JLE	1	1	0	if out≤0 jump
JMP	1	1	1	unconditional jump



Handling A-instructions

Routes the instruction's MSB (op-code) to the Mux16 control bit


opcode bit
comp bits
dest bits
jump bits
instruction
1110011111010111
D=D+1;JMP
inM
ALU output
ALU output
Mux16
A register
0
1
D register
1
011111
Mux16
M input
ALU
outM
writeM
0
C's
0

The Hack language specification

A instruction

Symbolic: `@`*xxx* (*xxx* is a decimal value ranging from 0 to 32767, or a symbol bound to such a decimal value)

Binary: `0` *vvvvvvvvvvvvvvv* (*vv ... v* = 15-bit value of *xxx*)

C instruction

Symbolic: *dest* `=` *comp* `;` *jump* (*comp* is mandatory. If *dest* is empty, the `=` is omitted; If *jump* is empty, the `;` is omitted)

Binary: `111`*accccccdddjjj*

Predefined symbols:

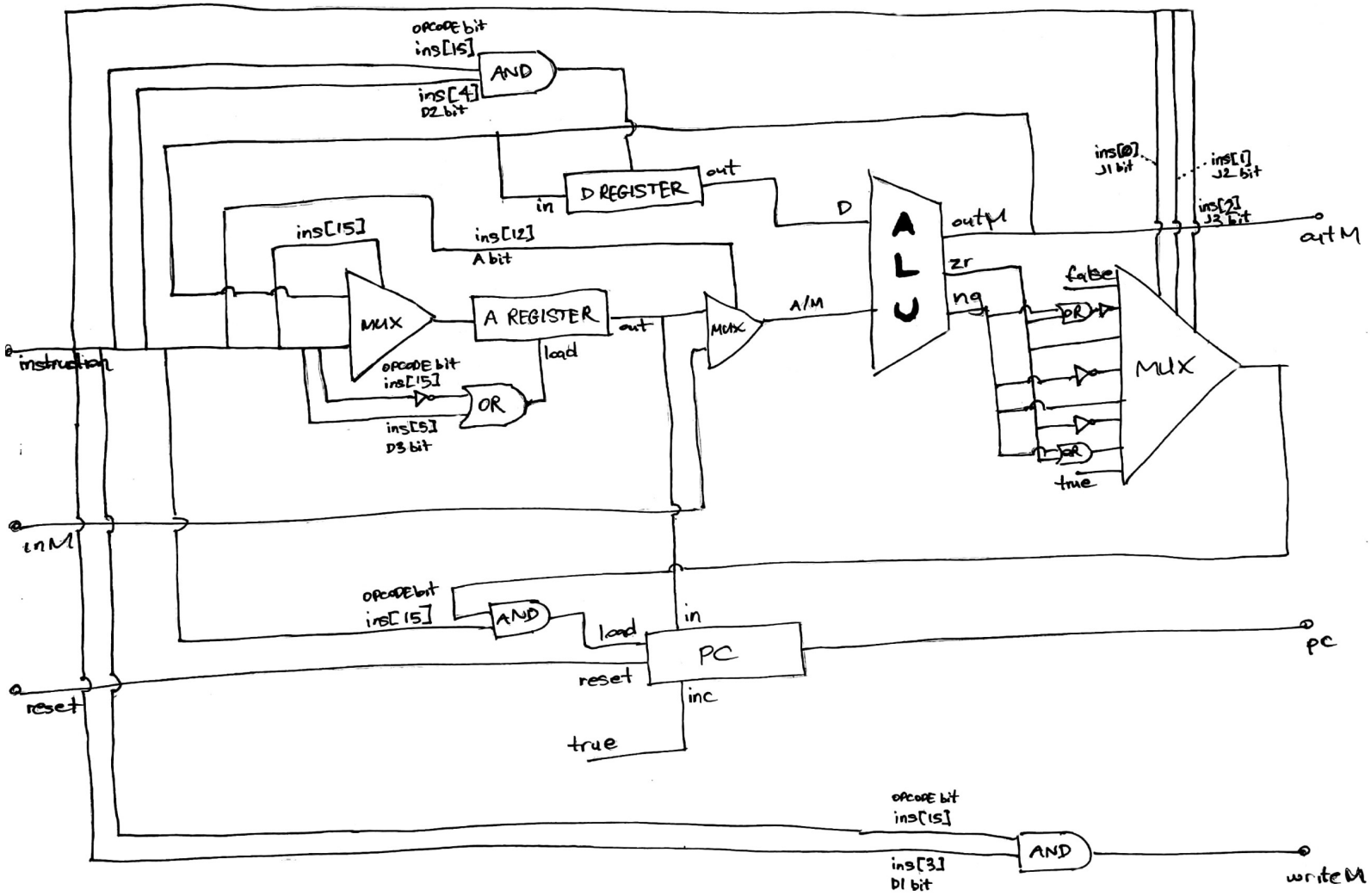
symbol	value
R0	0
R1	1
R2	2
...	...
R15	15
SP	0
LCL	1
ARG	2
THIS	3
THAT	4
SCREEN	16384
KBD	24576

comp		*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

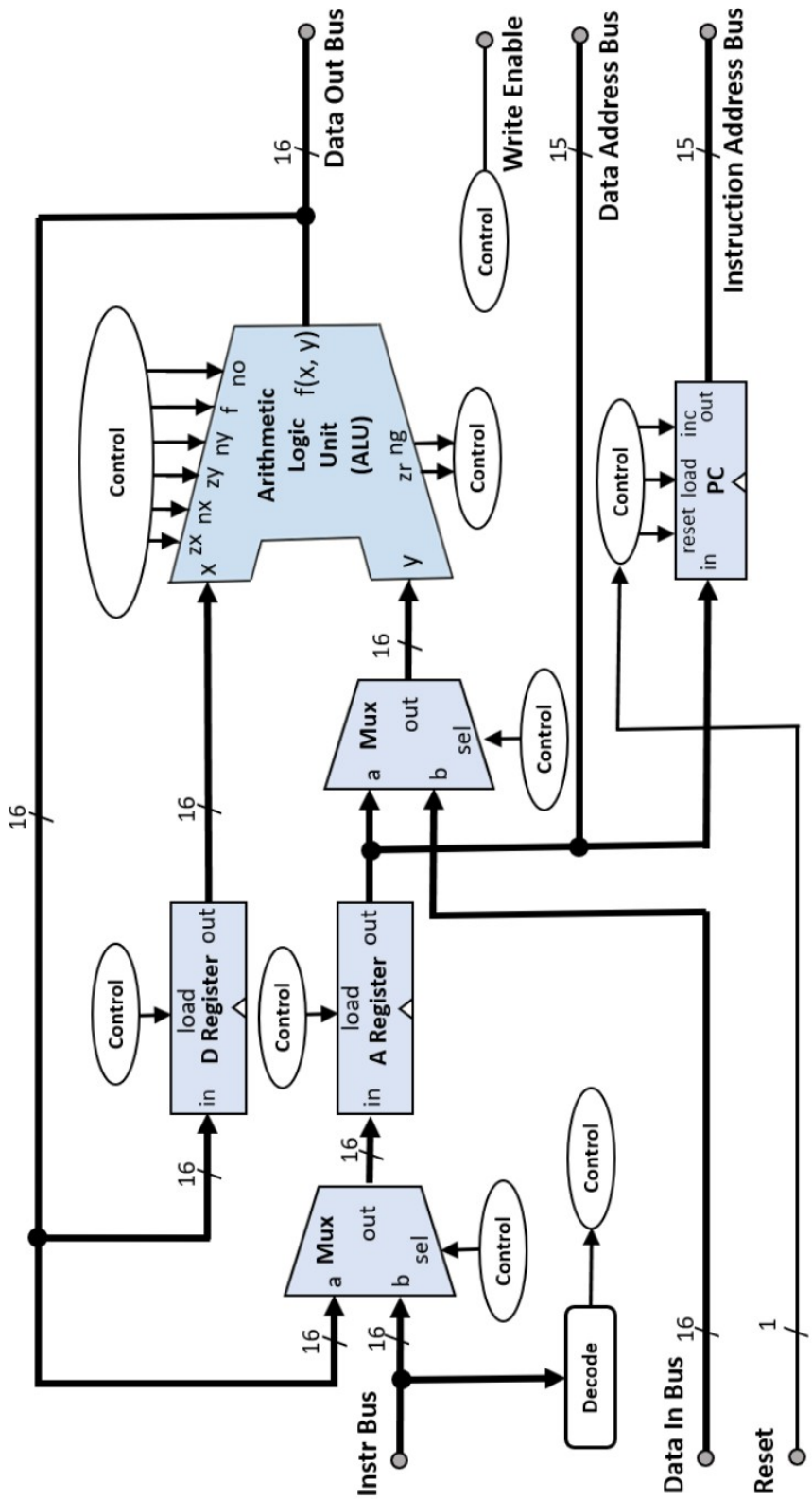
a == 0 *a* == 1

dest	*d* *d* *d*	Effect: store *comp* in:
null	0 0 0	the value is not stored
M	0 0 1	RAM[A]
D	0 1 0	D register (reg)
DM	0 1 1	RAM[A] and D reg
A	1 0 0	A reg
AM	1 0 1	A reg and RAM[A]
AD	1 1 0	A reg and D reg
ADM	1 1 1	A reg, D reg, and RAM[A]

jump	*j* *j* *j*	Effect:
null	0 0 0	no jump
JGT	0 0 1	if *comp* > 0 jump
JEQ	0 1 0	if *comp* = 0 jump
JGE	0 1 1	if *comp* ≥ 0 jump
JLT	1 0 0	if *comp* < 0 jump
JNE	1 0 1	if *comp* ≠ 0 jump
JLE	1 1 0	if *comp* ≤ 0 jump
JMP	1 1 1	unconditional jump


opcode bit
ins[15]
AND
ins[4]
D2 bit
D REGISTER
in
out
D
ALU
outM
zr
ng
ins[0]
J1 bit
ins[1]
J2 bit
ins[2]
J3 bit
outM
ins[15]
ins[12]
A bit
MUX
A REGISTER
out
load
MUX
A/M
false
DR
MUX
OR
opcode bit
ins[15]
ins[5]
D3 bit
true
instruction
inM
opcode bit
ins[15]
AND
load
in
PC
reset
inc
pc
reset
true
opcode bit
ins[15]
ins[3]
D1 bit
AND
writeM

https://www.bencode.net/posts/2017-04-17-diy-computer-architecture/



Projekt 05: Computer

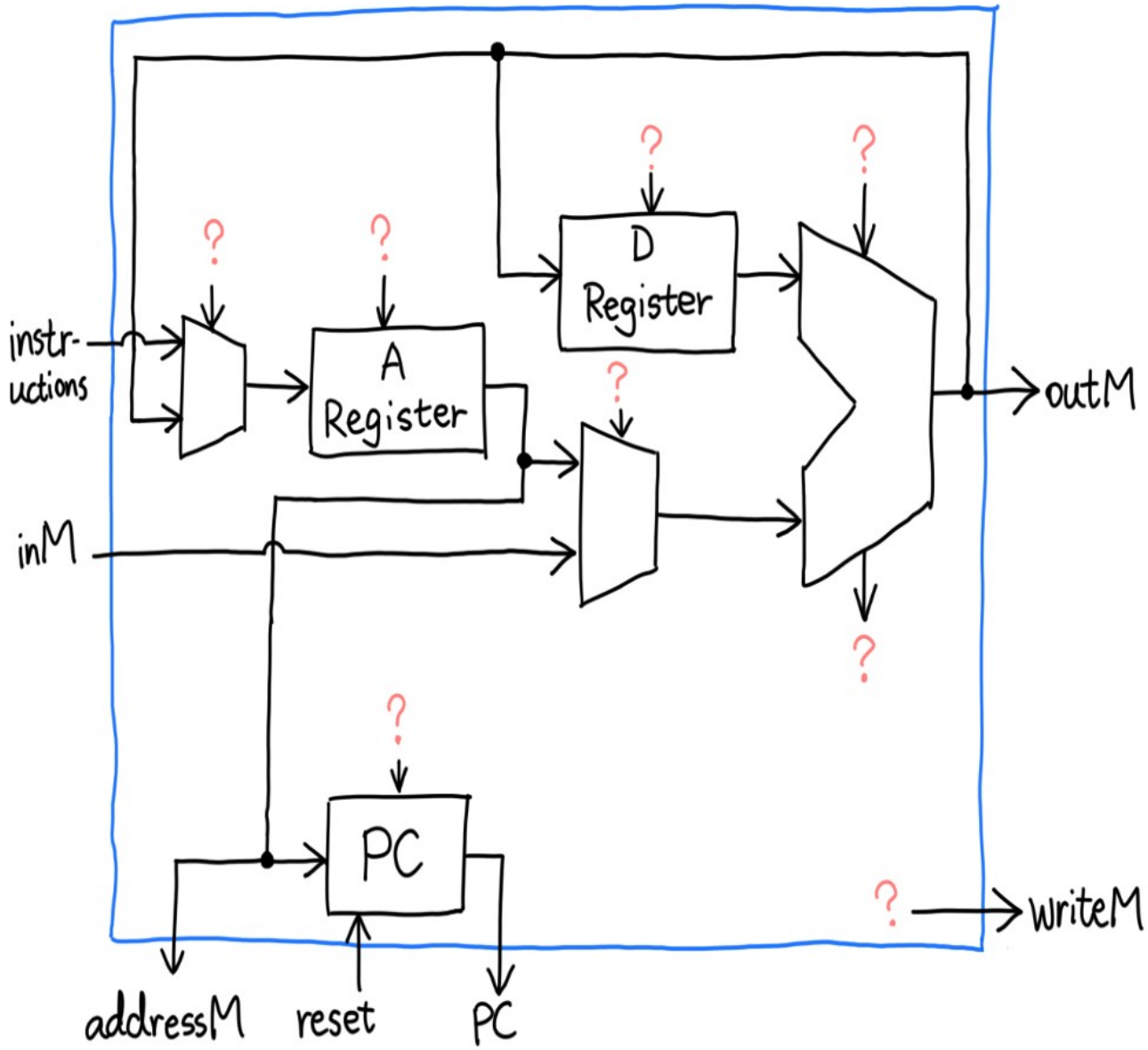
Nach einem Projekt, das scheinbar aus dem Nichts kam, klingt es beruhigend, dass wir wieder mit dem Bau der Computerhardware beginnen. Zuerst müssen wir uns für die Architektur entscheiden. Es ist üblich, einen Computer auf der Von-Neumann-Architektur aufzubauen. Aber Canon Von Neumann speichert sowohl Anweisungen als auch Daten in einer einzigen Speichereinheit, üblicherweise RAM, sodass die CPU beides ändern kann. Dennoch wird ein winziges ROM benötigt, um die CPU beim Bootvorgang zu unterstützen. In unserer Anwendung werden wir jedoch einfach zwei Einheiten ähnlicher Größe verwenden. Das bedeutet, dass unser Computer mit einem einzigen ROM nur ein bestimmtes Programm ausführen kann. Dies wird als Harvard-Architektur bezeichnet, technisch gesehen eine Teilmenge von Von Neumann. Auch AVR-Mikrocontroller nutzen diese Architektur.

Das bedeutet also, dass sich im Computer drei Dinge befinden: CPU, ROM und RAM. Da wir in Projekt 03 RAM erstellt haben und ROM integriert ist, bleibt nur noch die CPU übrig.

Die CPU muss:

- •Lesen Sie die Anweisungen aus dem ROM
- •Daten aus dem RAM lesen
- •Berechnen Sie etwas
- •Schreiben Sie Daten auf A, D und RAM
- •Führen Sie die Anweisungen einzeln aus
- •Springen Sie zu einer anderen Anweisung, wenn der Programmierer dies verlangt

Als ich die Anforderungen las, wusste ich sofort, dass es eine Menge interner Kabel geben wird. Glücklicherweise haben die Autoren ein Blockdiagramm bereitgestellt, das diesem ähnelt:


instr-
uctions
A
Register
D
Register
inM
outM
PC
addressM
reset
PC
writeM

Huch! Was ist mit den Fragezeichen? Anscheinend handelt es sich um die eigene Version des Spoiler-Alarms der Autoren. Ich musste selbst herausfinden, was sie sind, und das ist gut so. Ich mag Herausforderungen.

Stellen wir zunächst sicher, dass wir wissen, was jeder Pin/Chip tut.

- Die Anweisung im ROM kommt vom Pin`instruction`
- Daten vom RAM kommen vom Pin`inM`
- Sowohl RAM- als auch ROM-Adressen werden mit ausgewählt`addressM`
- Die ALU nimmt zwei Register und gibt einen Ausgang aus
- Das A- und D-Register akzeptieren Eingaben von ALU
- Die ALU-Ausgabe geht auch über an den RAM`outM`
- `writeM`weist den RAM an, Daten zu laden
- Der PC erhöht, setzt zurück oder springt zur Anweisung

Was ist überhaupt eine Anweisung? Es handelt sich um einen 16-Bit-Wert, der beschreibt, was die CPU in diesem Taktzyklus tun soll. Ein Assembler, den wir in Projekt 06 schreiben werden, übersetzt Assembler in Binärcode, aber in diesem Projekt gehen wir davon aus, dass er aus dem Nichts kommt.

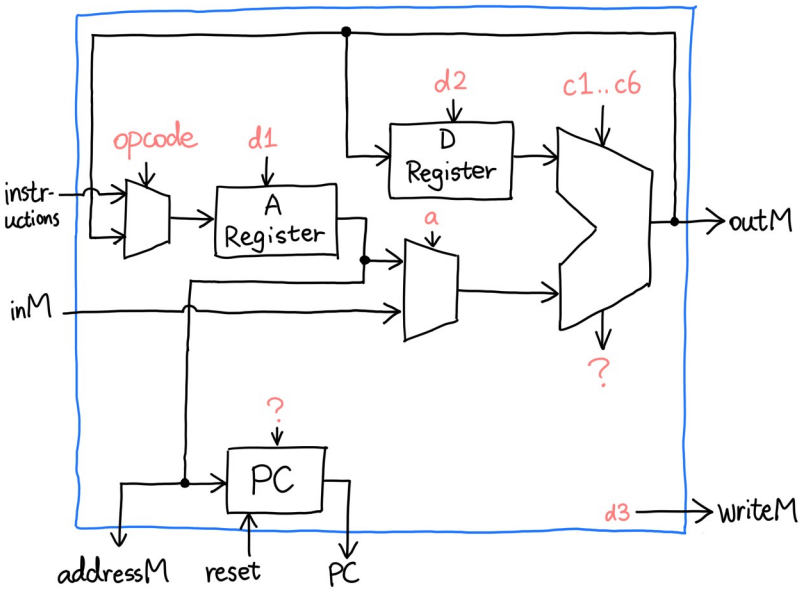
Was die 16 Bits darstellen, hängt von der Art der Anweisung ab, die Sie schreiben. Es wird durch das höchste Bit angegeben, das als Opcode bezeichnet wird . In einem A-Befehl ist der Opcode 0, gefolgt von 15 Adressbits. Wenn man bedenkt, dass unsere größere Speichereinheit, ROM, 32768 Wörter beträgt, sind 15 sinnvoll. In diesem Fall speichert die CPU den Wert im A-Register.

Der C-Befehl mit Opcode 1 ist komplizierter, läuft aber auf vier Gruppen von Steuerbits hinaus:

```
 fixed    ALU control    jump instruction

\     /   \        /        \        /

+-----+---+--------+--------+--------+

| 111 | a | c1..c6 | d1..d3 | j1..j3 |

+-----+---+--------+--------+--------+

      /   \        /        \

       A/M         destination
```

- `a`entscheidet, ob die ALU D und A oder D und M aufnimmt.
- `c1..c6`entsprechen Steuerbits auf der ALU.
- `d1..d3`Weisen Sie die CPU an, die ALU-Ausgabe jeweils auf A, D und M zu speichern.
- `j1..j3`Weisen Sie die CPU an, zu ROM[A] zu springen, wenn die ALU jeweils <0, =0 und >0 ausgibt.

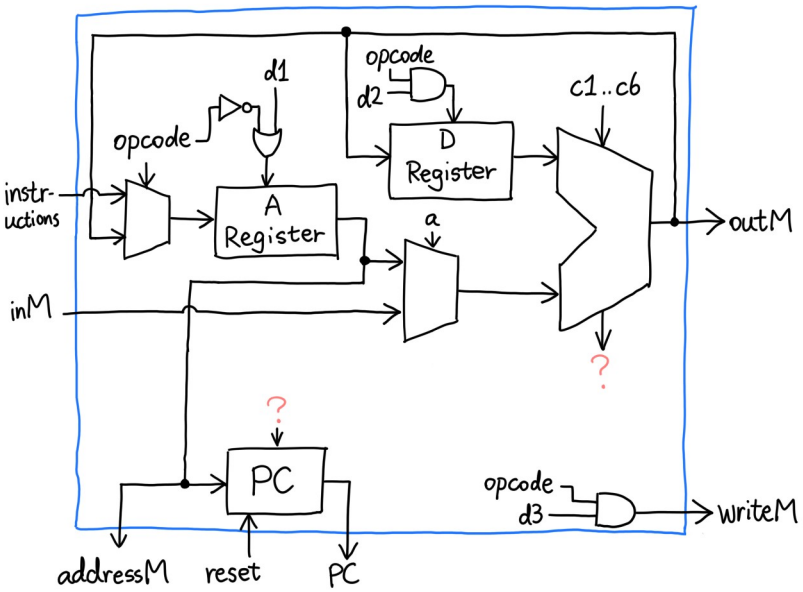
In einer Ahnung scheinen wir die Antwort auf die meisten Fragezeichen zu haben.



Das ist einfach, aber falsch. Schauen wir uns den mit dem A-Register `load`beschrifteten Pin genauer an `d1`. `@1`Was passiert, wenn wir versuchen, Adresse 1 mithilfe einer A-Anweisung hineinzuladen ? Lass es uns zusammenbauen:

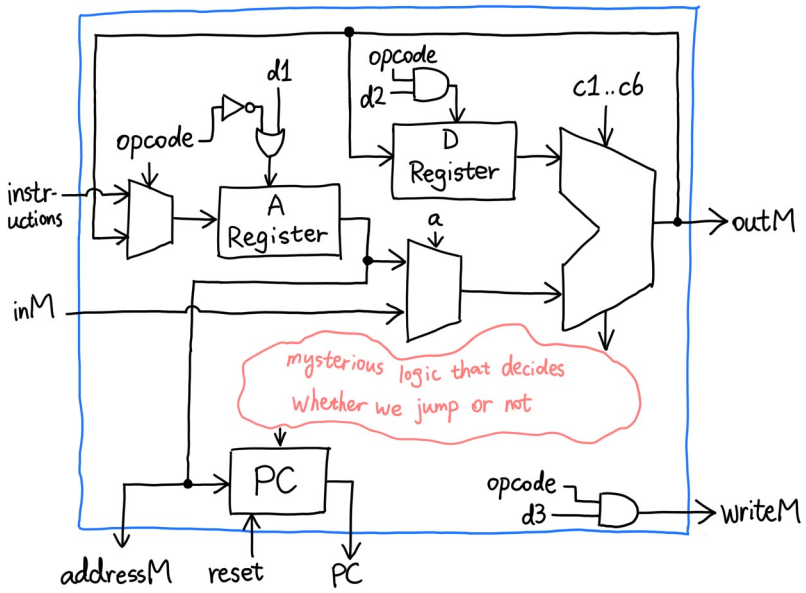
```
0000 0000 0000 0001

^\_________________/

|        ^

|        |

opcode   integer value 1
```

Der Multiplexer auf der linken Seite lässt den Befehl durch, da `opcode`er 0 ist, aber denken Sie daran, dass HDL nicht erkennt `d1`, sondern nur `instruction[5]`. Was passieren wird, ist, dass das A-Register das Laden verweigert, weil `instruction[5]`es Null ist. Etwas Ähnliches wird auch mit `d2`and passieren `d3`, daher fügen wir eine kleine Logik hinzu:

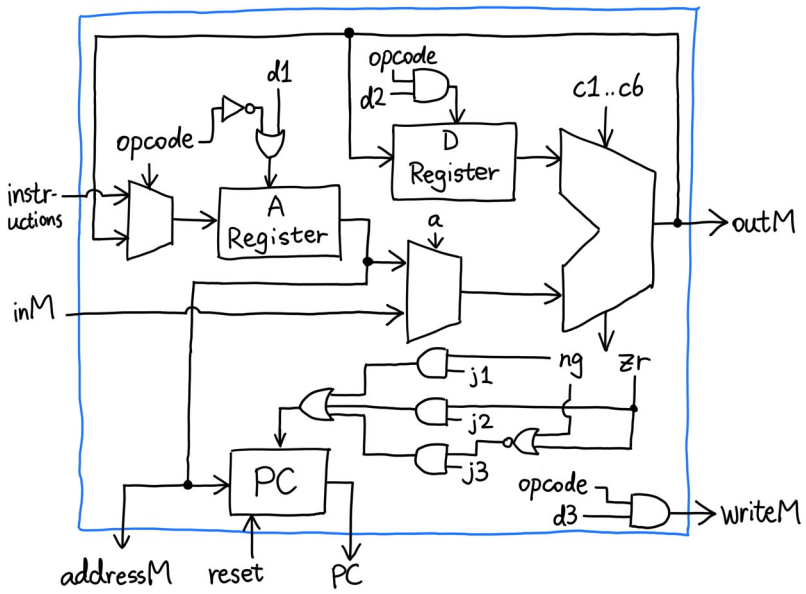


Dadurch wird sichergestellt, dass A, D und M genau dann Daten laden, wenn wir sie ausdrücklich dazu auffordern. Jetzt richten wir unsere Aufmerksamkeit auf die einzigen zwei Fragezeichen, die noch übrig sind: `zr`und `ng`von der ALU kommen und `load`in den PC gehen.

Fällt Ihnen etwas auf? `j1..j3`fehlen, also sind es definitiv sie. Denken Sie daran, dass wir den PC auf seinen Eingang setzen können, wenn wir ihn `load`auf hoch ziehen, und auf diese Weise zu ROM[A] springen können. Aber wie?



Es ist eigentlich ganz einfach! Anscheinend haben die Autoren bei der Angabe der ALU genau darüber nachgedacht.



(Die tatsächlichen Tore können abweichen)

Und das war's, wir haben eine CPU gemacht! Wir sind ganz nah am Computer; Alles, was wir tun müssen, ist, alle Drähte anzuschließen. Ich fühle mich zu müde, um ein weiteres Diagramm zu skizzieren. Hier ist das HDL:

```
CHIP Computer {

    IN reset;
```

```
    PARTS:
    ROM32K(address=pc, out=instruction);
    CPU(
        inM=inM, instruction=instruction, reset=reset,
        writeM=writeM, outM=outM, addressM=addressM, pc=pc
    );
    Memory(in=outM, address=addressM, load=writeM, out=inM);
}
```

https://fkfd.me/projects/nand2tetris_1/

https://zhangruochi.com/Computer-Architecture/2019/06/03/