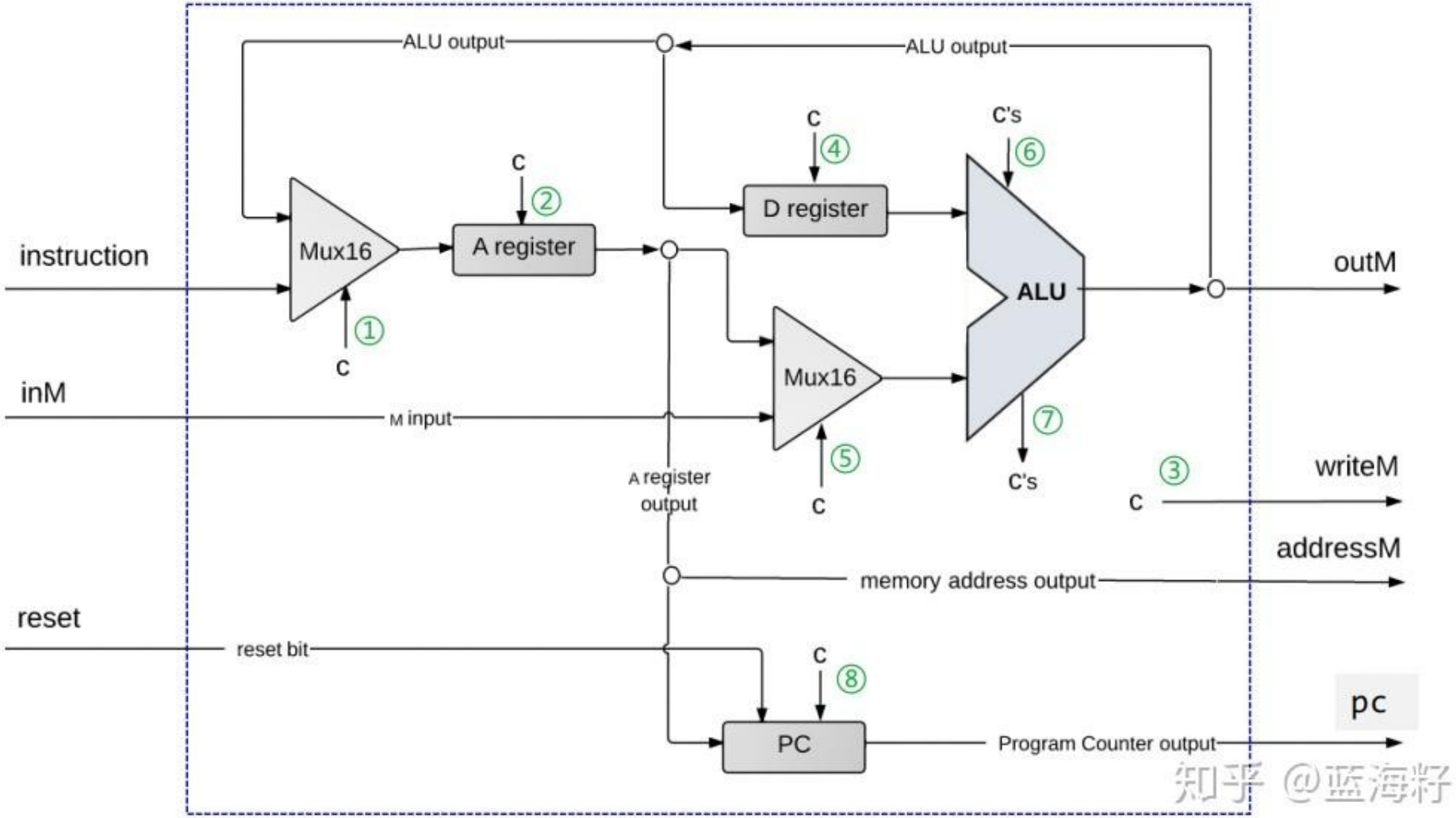
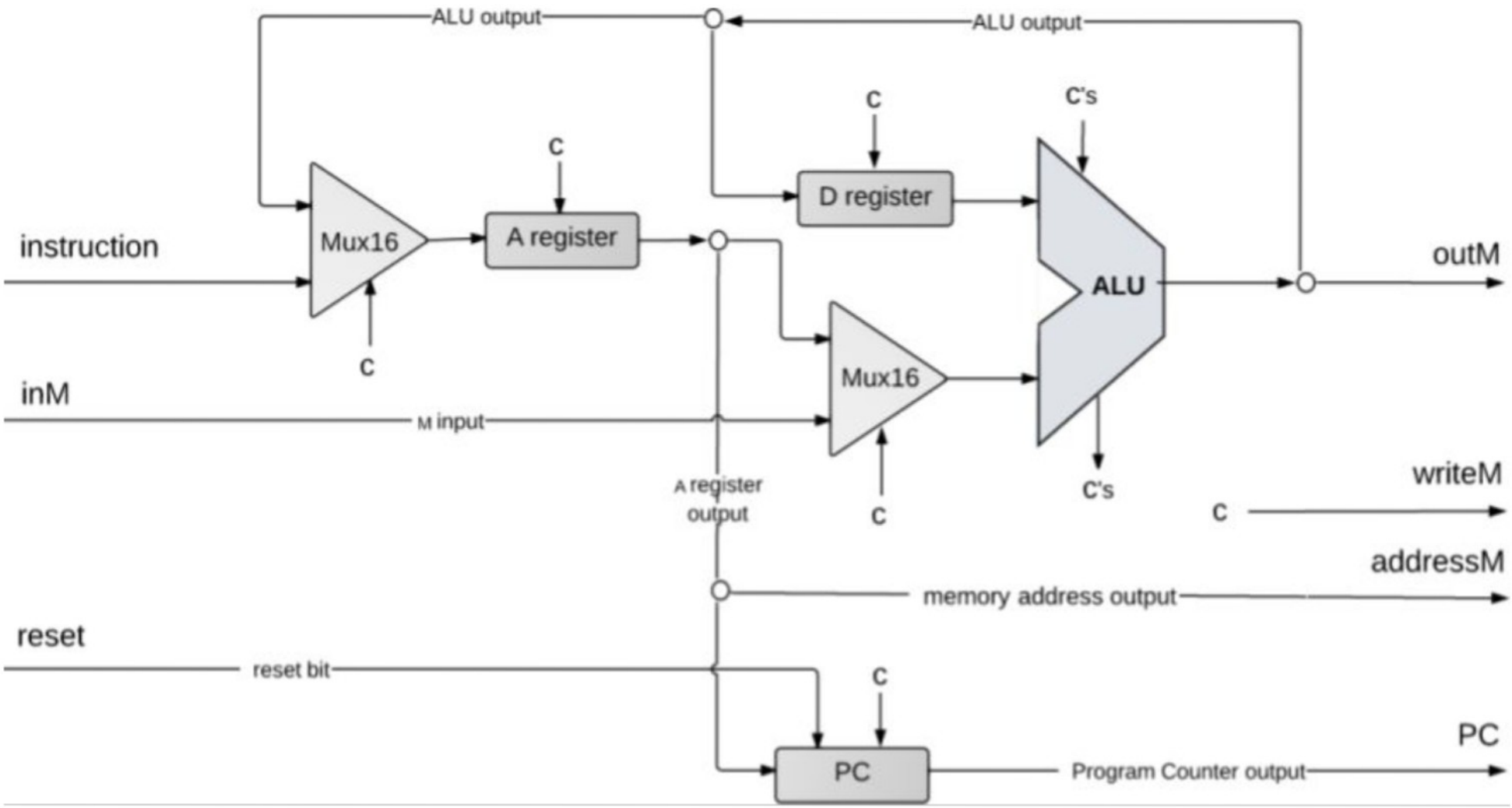
Kontrollbit-Designpunkte

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
A register
output
C
C's
writeM
C
addressM
memory address output
reset
reset bit
C
PC
PC
Program Counter output

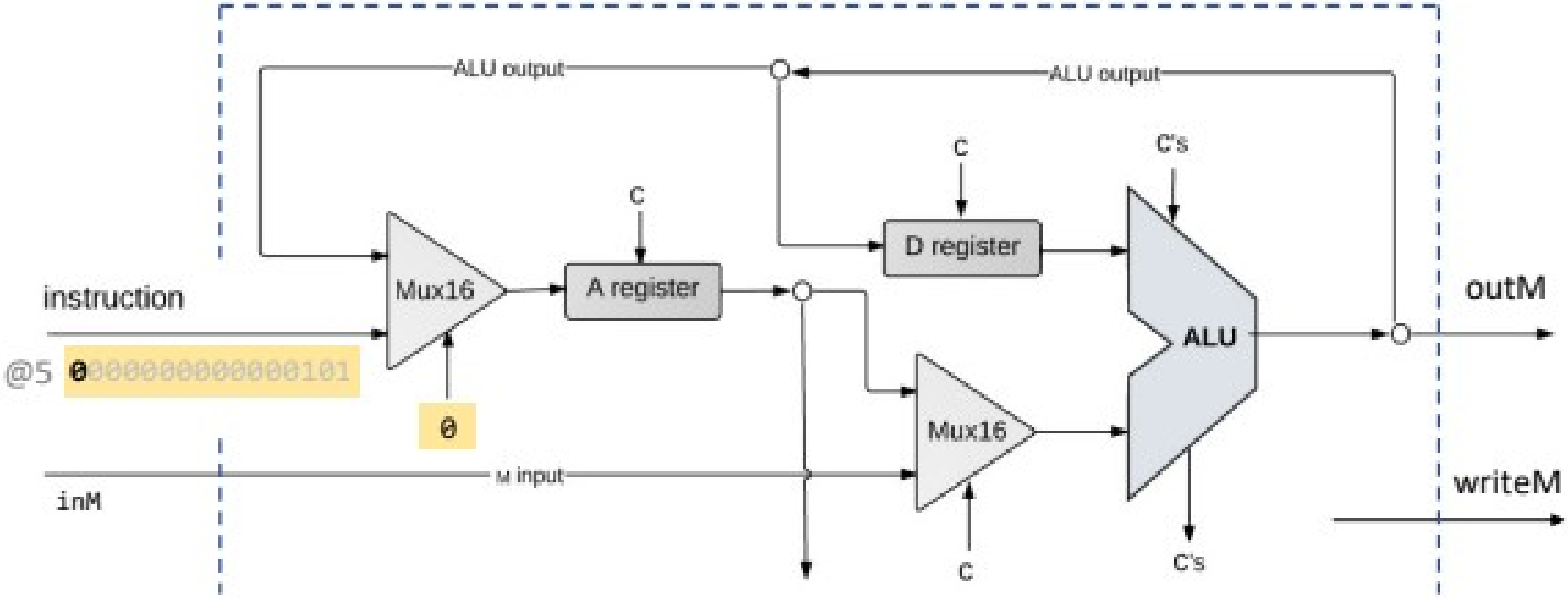
C-instruction specification

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

Binary syntax:

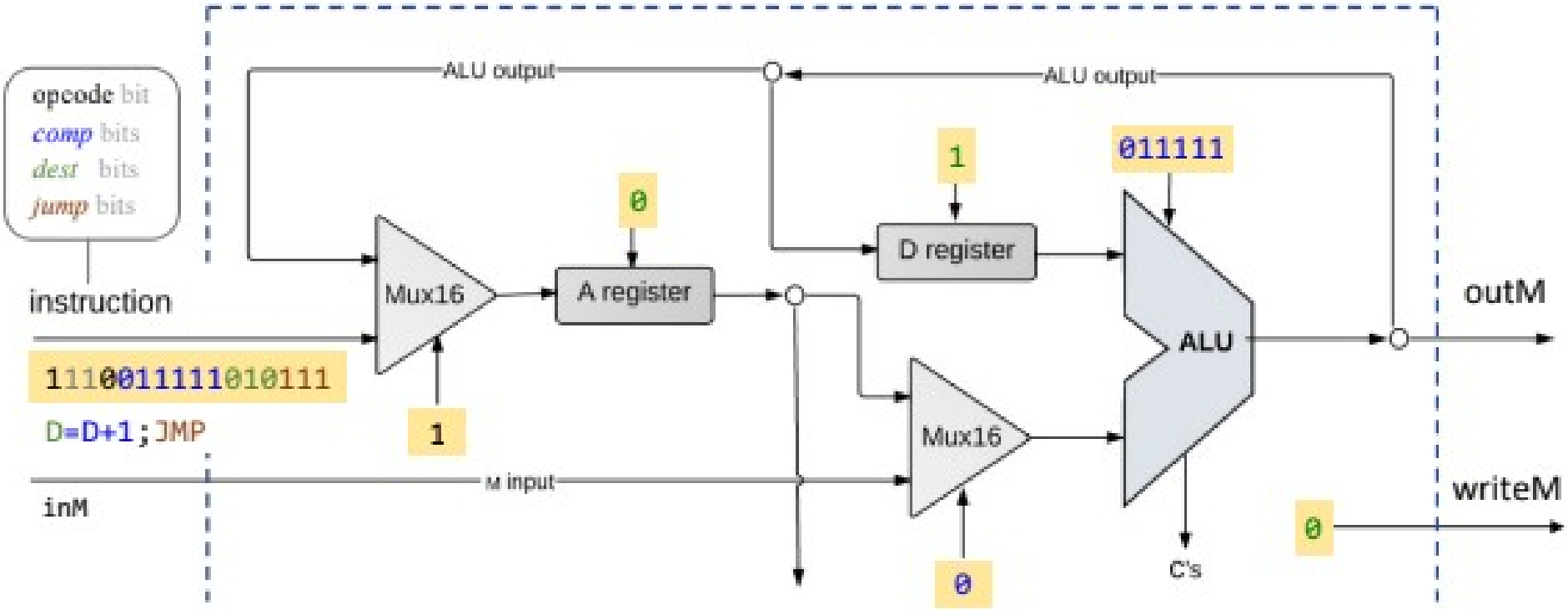
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	a	c1	c2	c3	c4	c5	c6	d1	d2	d3	j1	j2	j3

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
1110001111010111
D=D+1;JMP
inM
ALU output
ALU output
0
1
011111
Mux16
A register
D register
Mux16
ALU
outM
writeM
M input
1
0
0
C's

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*

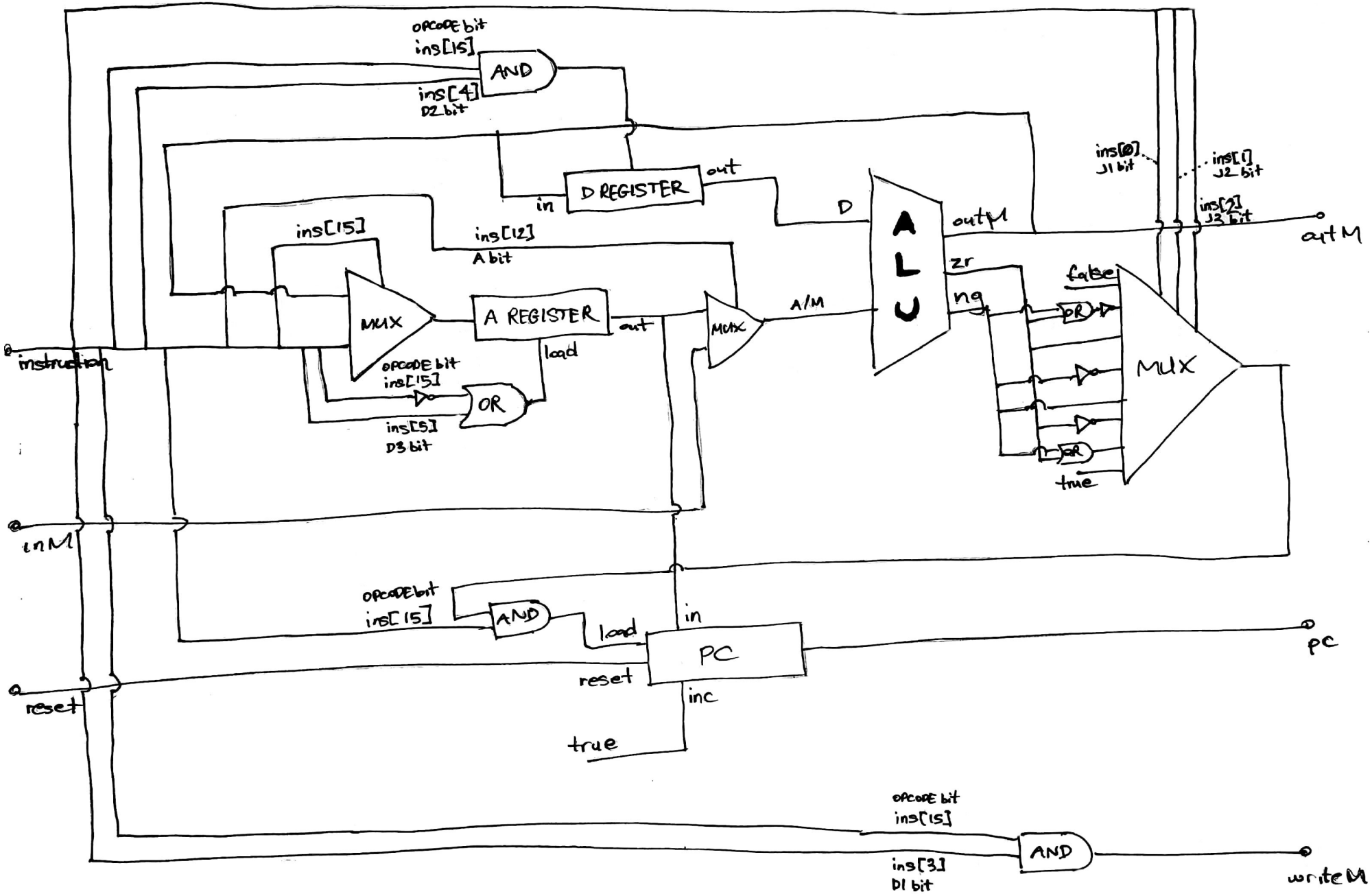
Predefined symbols:

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

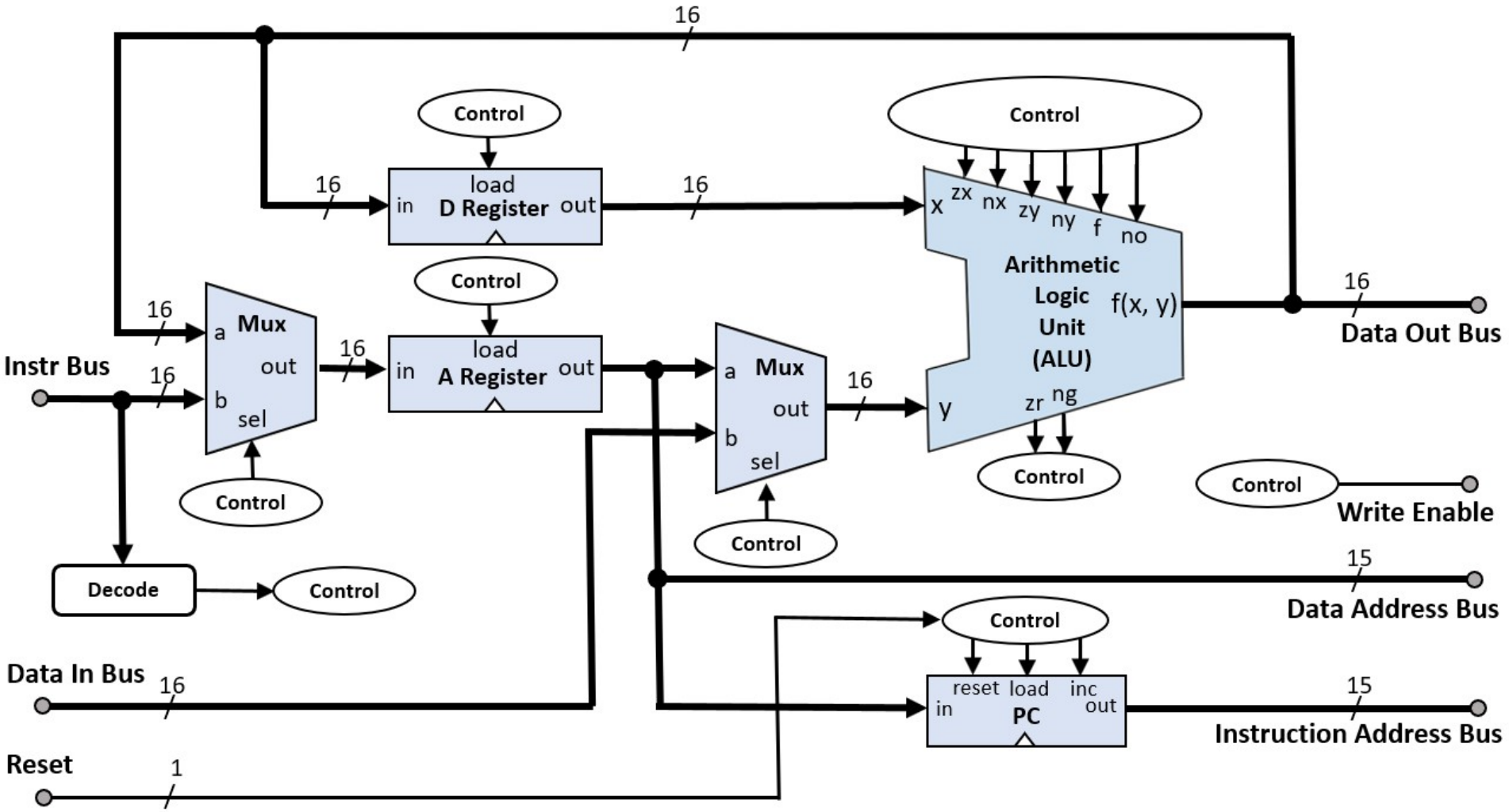
comp		*c*	*c*	*c*	*c*	*c*	*c*
0		1	0	1	0	1	0
1		1	1	1	1	1	1
-1		1	1	1	0	1	0
D		0	0	1	1	0	0
A	M	1	1	0	0	0	0
!D		0	0	1	1	0	1
!A	!M	1	1	0	0	0	1
-D		0	0	1	1	1	1
-A	-M	1	1	0	0	1	1
D+1		0	1	1	1	1	1
A+1	M+1	1	1	0	1	1	1
D-1		0	0	1	1	1	0
A-1	M-1	1	1	0	0	1	0
D+A	D+M	0	0	0	0	1	0
D-A	D-M	0	1	0	0	1	1
A-D	M-D	0	0	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]

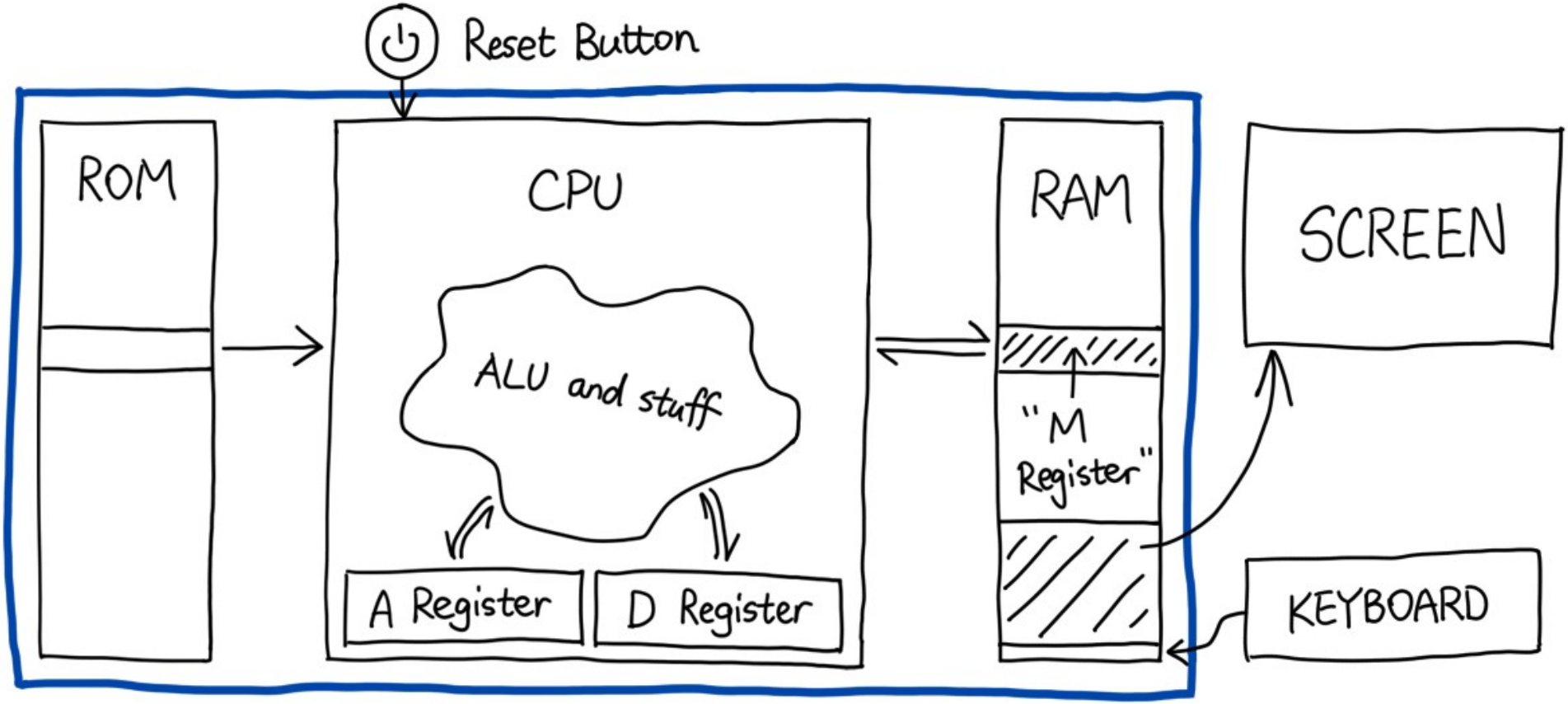
jump	*j*	*j*	*j*	Effect:
null	0	0	0	no jump
JGT	0	0	1	if *comp* > 0 jump
JEQ	0	1	0	if *comp* = 0 jump
JGE	0	1	1	if *comp* ≥ 0 jump
JLT	1	0	0	if *comp* < 0 jump
JNE	1	0	1	if *comp* ≠ 0 jump
JLE	1	1	0	if *comp* ≤ 0 jump
JMP	1	1	1	unconditional jump


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
OR
MUX
true
instruction
opcode bit
ins[15]
OR
ins[5]
D3 bit
inM
opcodebit
ins[15]
AND
load
in
PC
reset
inc
true
pc
reset
opcode bit
ins[15]
AND
ins[3]
D1 bit
writeM

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



In Projekt 04 gibt es kein HDL, stattdessen habe ich gelernt, Assembler zu schreiben. Nisan und Shocken haben wieder einmal eine Assemblersprache für genau diese Architektur namens HACK erfunden. Die Sprache versteht zwei Arten von Anweisungen. Aber zuerst müssen wir uns vorstellen, dass wir einen Computer haben.



Hier sehen Sie drei Geräte innerhalb des blauen Rechtecks: ROM, CPU und RAM.

- ROM: Nur-Lese-Speicher. Unser Computer holt sich daraus Anweisungen und führt sie aus. Wir haben kein HDL dafür geschrieben, da es sich nur um eine schreibgeschützte Version des RAM handelt, die bereits eingebaut ist.
- CPU: Eine Verkapselung der ALU, die bis Projekt 05 mit schwarzer Magie arbeitet.
- RAM: Random Access Memory. Unser Computer kann darin Daten lesen und schreiben.

HACK-Assembly bietet eine äußerst rudimentäre Möglichkeit, die CPU zu steuern. Der Schlüssel zum Verständnis von HACK Assembly sind die drei Register: D, A und M. D steht für "Data" (wahrscheinlich). Es enthält einen 16-Bit-Wert, genau wie die beiden anderen. In unserer Assemblersprache können Sie zwei Dinge damit tun:

- ihn als ALU-Eingang lesen
- ALU-Ergebnis in ihn schreiben

A steht für "Adresse". Es funktioniert ziemlich genau so wie D, mit einer zusätzlichen Funktion:

- Es wird auf eine Assembler-Zeit-Konstante gesetzt

die mit dieser Assemblerzeile (oder Anweisung) realisiert werden kann:

```
@42
```

Erweitern Sie, um mehr über den Unterschied zwischen A und D zu erfahren

M steht für "Speicher". Beachten Sie, dass es sich nicht um ein Register im wörtlichen Sinne handelt, sondern um einen Alias für RAM[A]. Zum Beispiel,

```
@42
M=-1
```

setzt das 42. Wort (wenn man von Null aus zählt) im RAM auf -1, d. h. alle Bits werden auf Eins gesetzt.

Wir nennen eine Anweisung eine "A-Anweisung", wenn sie mit "@" beginnt, und andernfalls eine "C-Anweisung". Bei der Ausführung eines C-Befehls berechnet die ALU immer etwas - auch Null. Man kann ihr ein Ziel voranstellen, um zu entscheiden, wo das Ergebnis gespeichert werden soll. Sie können entweder eines der drei Register A, D und M auswählen oder nicht, so dass sich 8 Möglichkeiten ergeben:

```
D+M; // Berechne D+M, aber speichere es nirgendwo
AD=1; // A und D gleichzeitig auf 1 setzen
AMD=1 // setzt A, M und D gleichzeitig auf 1
```

C-Anweisungen können auch etwas wirklich Cooles tun, nämlich springen. Die Syntax unterscheidet sich von der in einer industriellen Assemblersprache.

```
@42
D;JEQ // dies springt zu ROM[42], wenn D=0
```

Das JEQ hier ist ein Sprungbefehl. Eine Sprunganweisung vergleicht die ALU-Ausgabe mit Null und springt, wenn sie größer/gleich/nicht gleich/etc. ist.

Was ist, wenn ich zu 42 springen will, wenn RAM[69] Null ist?

Der Einfachheit halber können wir einer Anweisung in der Assemblerdatei ein Etikett anhängen und es später wiederverwenden:

```
(END)
@END
0;JMP
```

In diesem Code zeigt (END) auf @END, und JMP ist ein unbedingter Sprung, so dass eine Endlosschleife entsteht. Es ist eine gute Praxis, ein Programm auf diese Weise zu beenden, damit die ROM-Adresse nicht überläuft und der Computer das Programm erneut von der Anweisung 0 aus startet.

Sie können auch benutzerdefinierte Symbole nach einem "@" verwenden, um eine statische Adresse ab 16 zuzuweisen oder wiederzuverwenden. Das ist so, als würde man eine Variable deklarieren, deren Wert in RAM[n] gespeichert wird, wobei $n \geq 16$ ist. Warum nicht 0-15, fragen Sie sich? Nun, sie sind für die Symbole R0-R15 reserviert.

Diese @Symbol-Notation geht noch weiter, indem sie einen monochromen Bildschirm auf einen Speicherbereich @SCREEN, nämlich 16384-24575, und eine Tastatur auf @KBD, also 24576, abbildet. Wenn Sie das höchste Bit in RAM[16384] auf Eins setzen, malen Sie das obere linke Pixel schwarz. Wenn Sie die Leertaste drücken, wird RAM[24576] auf 32 (0x20) gesetzt. Sie können im CPU-Emulator mit ihnen interagieren.

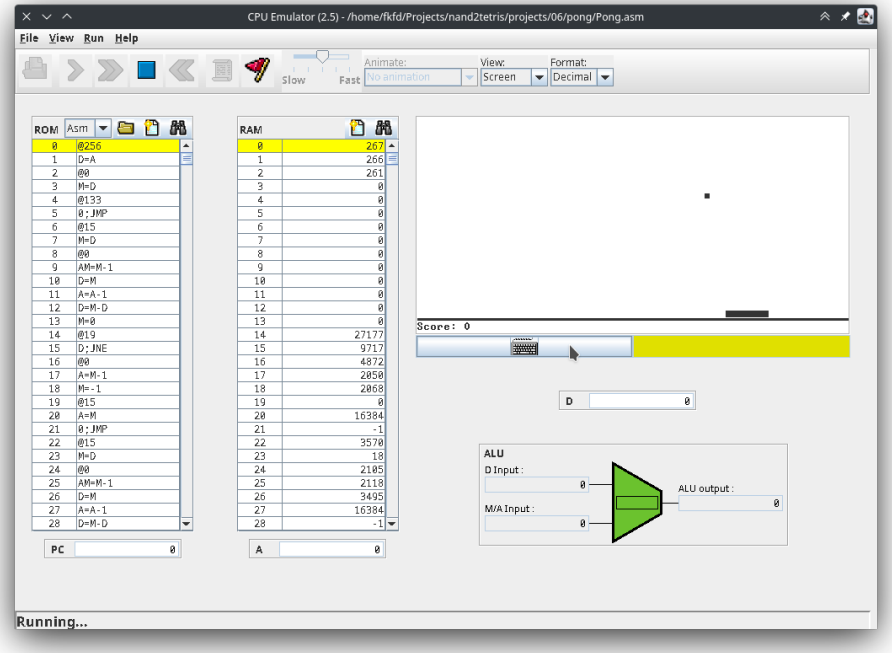


Schaubild 1: CPU-Emulator mit Pong

Hier ist ein Schnipsel von HACK-Assembler, den ich geschrieben habe und der RAM[16] von 0 inkrementiert, bis er gleich dem Wert ist, den Sie vor dem Programmstart in RAM[0] eingegeben haben:

```
@x
M=0     // x = RAM[16] = 0

(LOOP)
@R0
D=M     // D = RAM[0]
@x
D=D-M   // D -= x
@END
D;JEQ   // if D == 0 goto END
@x
M=M+1   // otherwise x += 1
@LOOP
0;JMP   // goto LOOP

(END)
@END
0;JMP
```

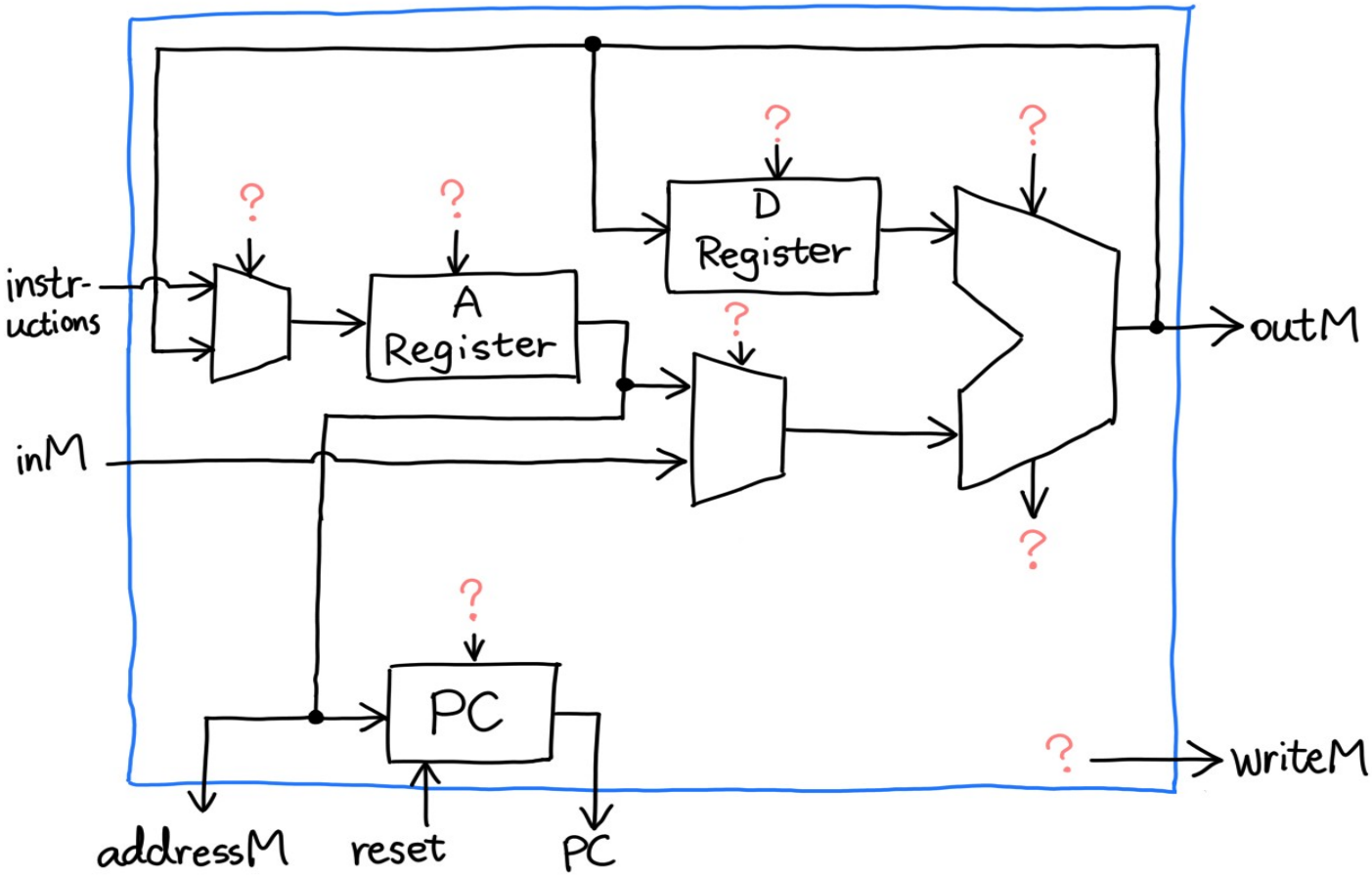
Projekt 05: Computer

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:



Igitt! Was hat es mit den Fragezeichen auf sich? Offenbar ist das die eigene Version der Autoren von Spoilerwarnungen. Ich musste selbst herausfinden, was das ist, und das ist auch gut so. Ich mag Herausforderungen.

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

- Befehle im ROM kommen vom Pin instruction
- Daten aus dem RAM kommen vom Pin inM

- Sowohl RAM- als auch ROM-Adressen werden über die addressM ausgewählt.
- Die ALU verarbeitet zwei Register und gibt einen Ausgang aus
- Die Register A und D nehmen Eingaben von der ALU entgegen
- Die ALU-Ausgabe geht über outM auch an den RAM
- writeM weist den RAM an, Daten zu laden
- Der PC inkrementiert, setzt zurück oder springt zu einer Anweisung

Was ist eigentlich eine Anweisung? Es ist ein 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 nehmen wir an, dass er aus dem Nichts kam.
Was die 16 Bits darstellen, hängt von der Art des Befehls ab, den Sie schreiben. Sie wird durch das höchste Bit, den sogenannten Opcode, angegeben. Bei einem A-Befehl ist der Opcode 0, gefolgt von 15 Bit Adresse. Wenn man bedenkt, dass die Größe unserer größeren Speichereinheit, des ROM, 32768 Wörter beträgt, sind 15 sinnvoll. In diesem Fall speichert die CPU den Wert im A-Register.

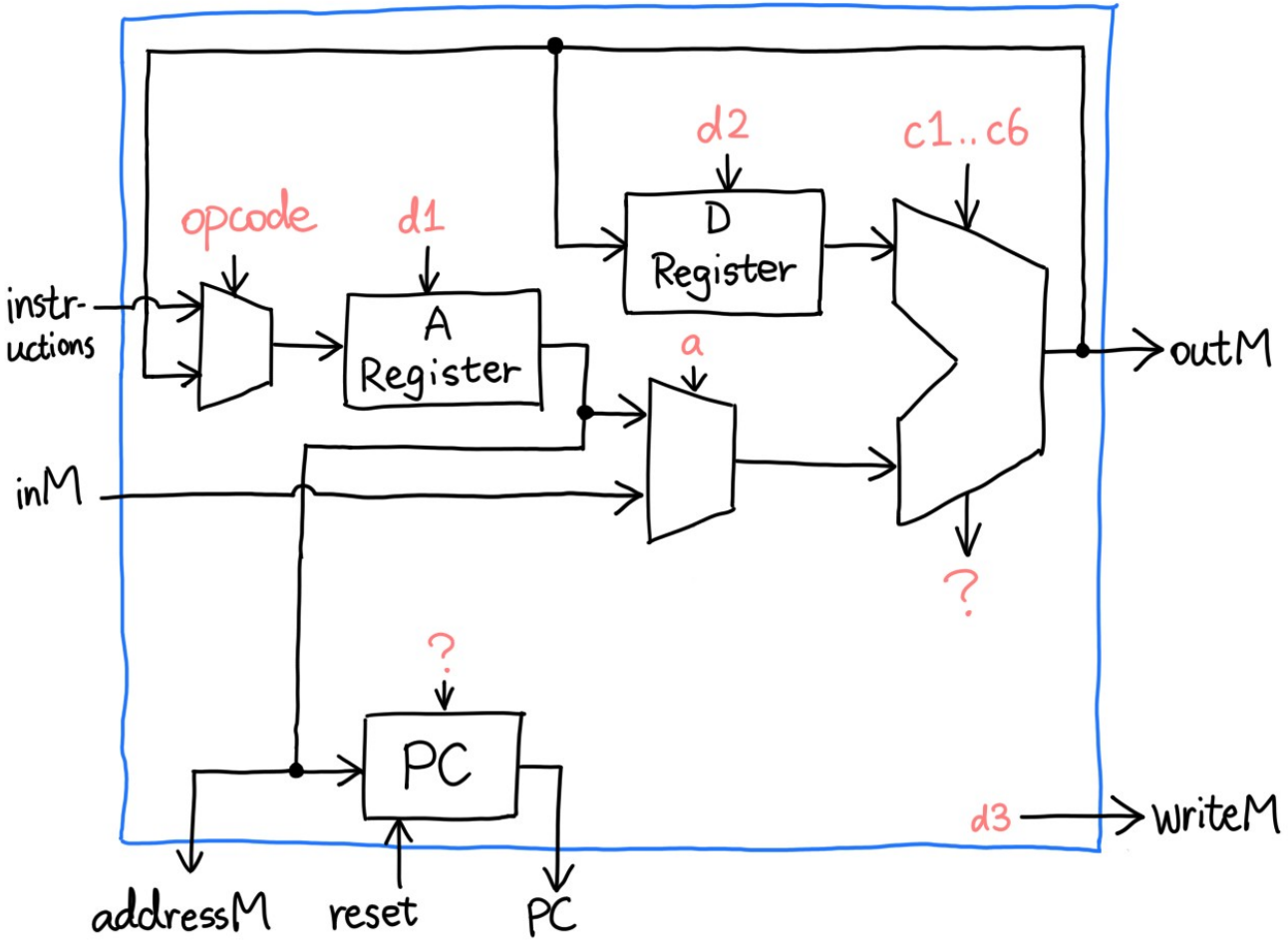
Die C-Anweisung mit Opcode 1 ist komplizierter, besteht aber aus vier Gruppen von Steuerbits:

```
 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 den Steuerbits der ALU.

- d1..d3 weisen die CPU an, die ALU-Ausgabe in A, D bzw. M zu speichern.
- j1..j3 sagen der CPU, dass sie zu ROM[A] springen soll, wenn die ALU-Ausgabe <0, =0 bzw. >0 ist

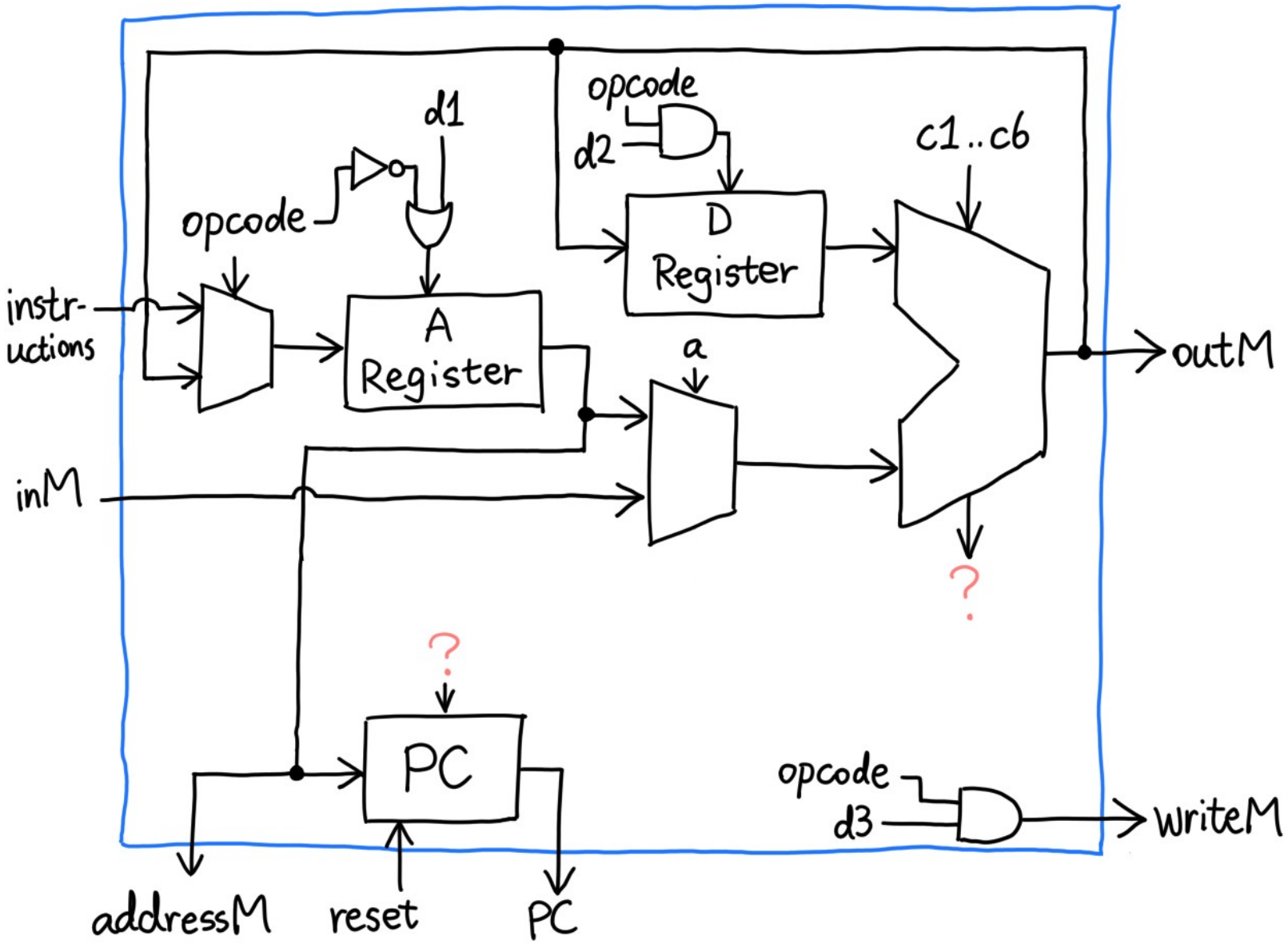
Nach einer Vermutung scheinen wir die Antwort auf die meisten Fragezeichen zu haben.



Das ist einfach, aber falsch. Schauen wir uns den mit d1 bezeichneten Lade-Pin des A-Registers genauer an. Was passiert, wenn wir versuchen, die Adresse 1 mit dem A-Befehl @1 zu laden? Fügen wir es zusammen:

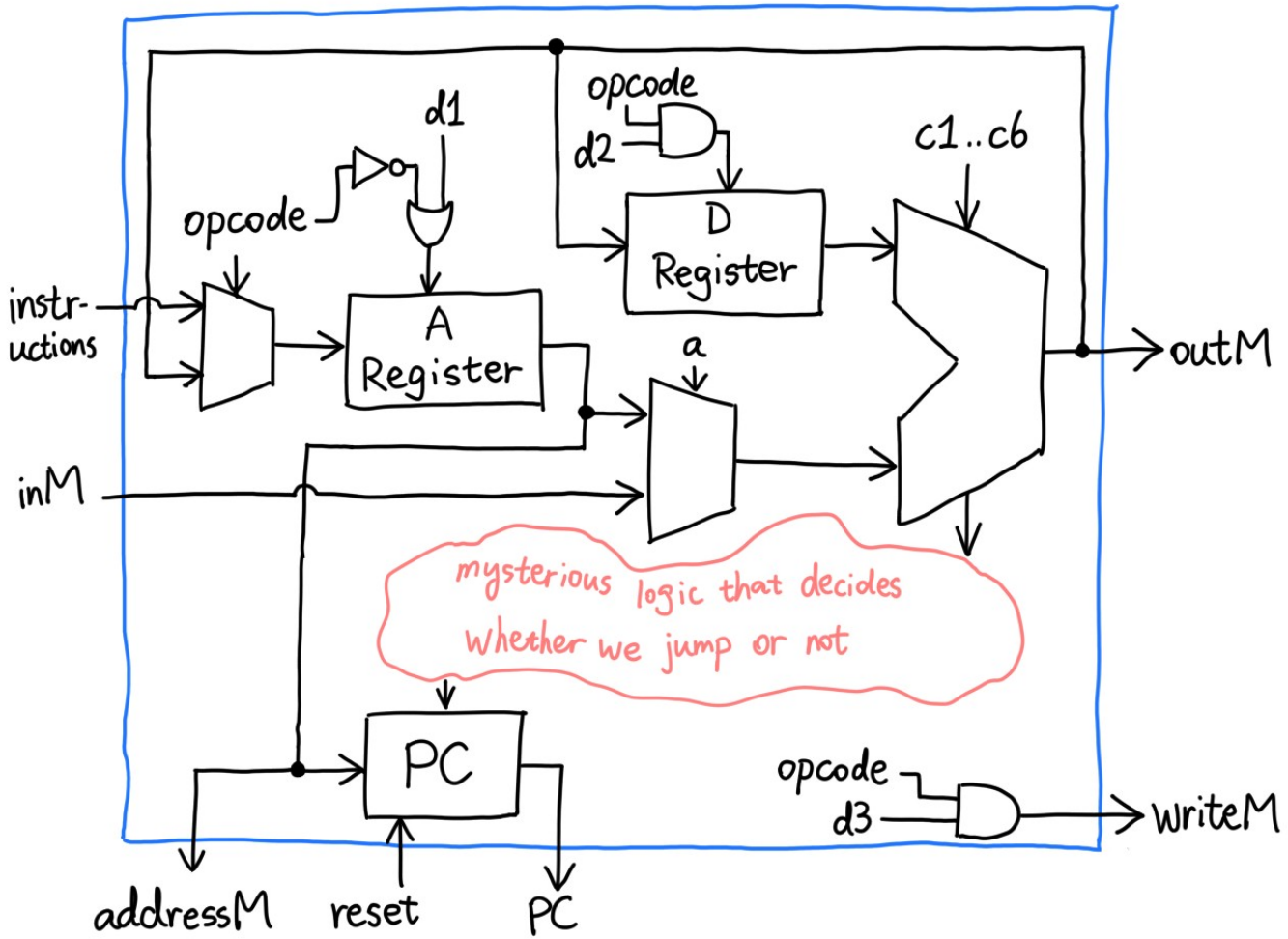
```
0000 0000 0000 0001
^\_______________/
|        ^
|        |
opcode   integer value 1
```

Der Mux auf der linken Seite wird den Befehl durchlassen, weil der Opcode 0 ist, aber denken Sie daran, dass HDL d1 nicht erkennt, sondern nur Befehl[5].Was passieren wird, ist, dass das A-Register sich weigern wird, zu laden, weil Befehl[5] Null ist.Etwas Ähnliches wird auch mit d2 und d3 passieren, also fügen wir ein wenig Logik hinzu:


d1
opcode
opcode
d2
c1..c6
D
Register
instr-
uctions
A
Register
a
outM
inM
?
?
PC
opcode
d3
writeM
addressM
reset
PC

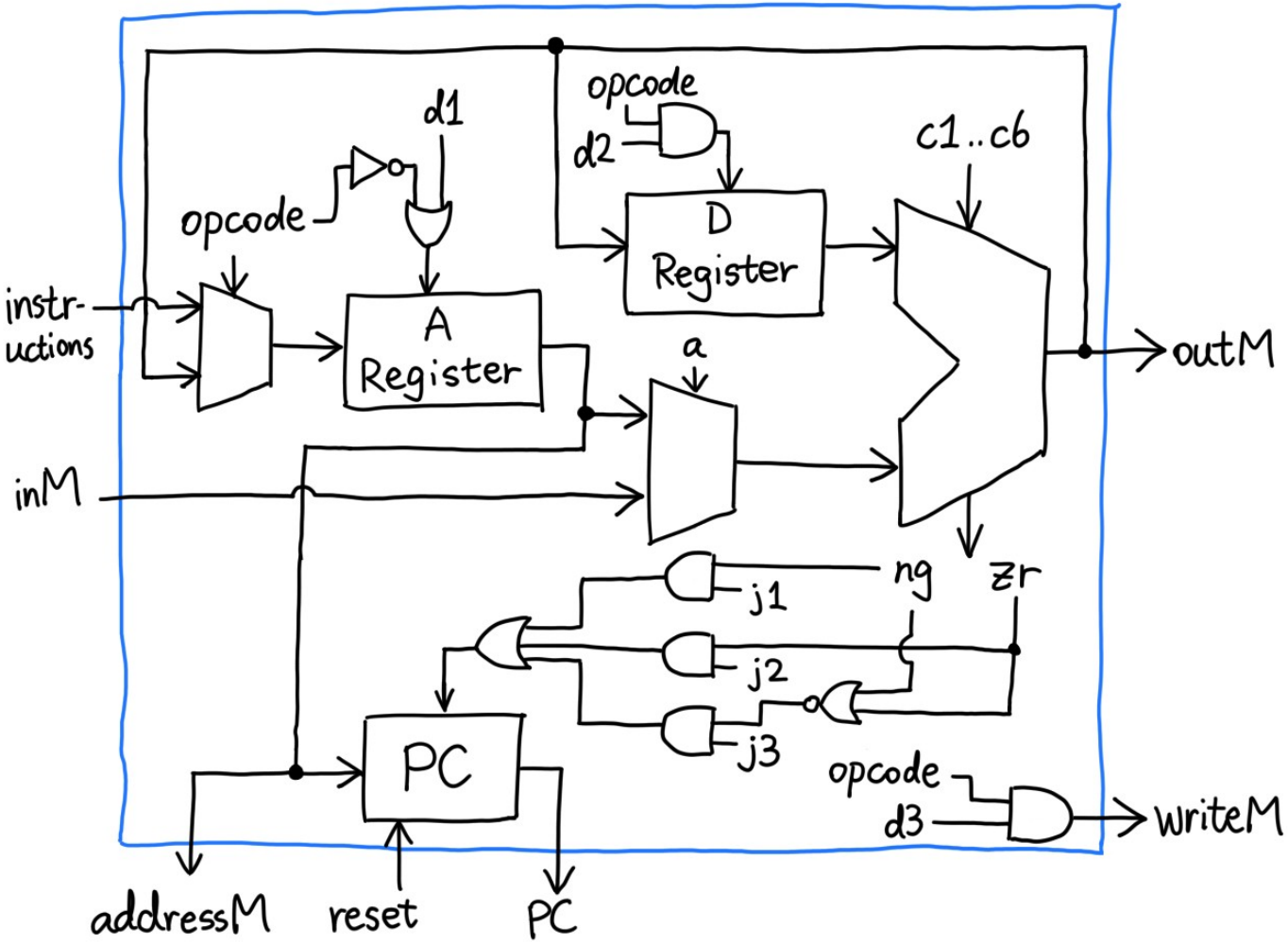
Damit ist sichergestellt, dass A, D und M nur dann Daten laden, wenn wir es ihnen ausdrücklich befehlen.Jetzt richten wir unsere Aufmerksamkeit auf die beiden einzigen Fragezeichen, die noch übrig sind: zr und ng, die von der ALU kommen, und load, das in den PC geht.

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



Es ist eigentlich sehr einfach!

Offenbar haben die Autoren bei der Spezifikation der ALU an alles gedacht.



(Tatsächliche Gates können abweichen)

Und das war's, wir haben eine CPU gebaut! Wir sind dem Computer schon sehr nahe; wir müssen nur noch alle Drähte anschließen. Ich bin zu müde, um ein weiteres Diagramm zu entwerfen, also hier ist das HDL:

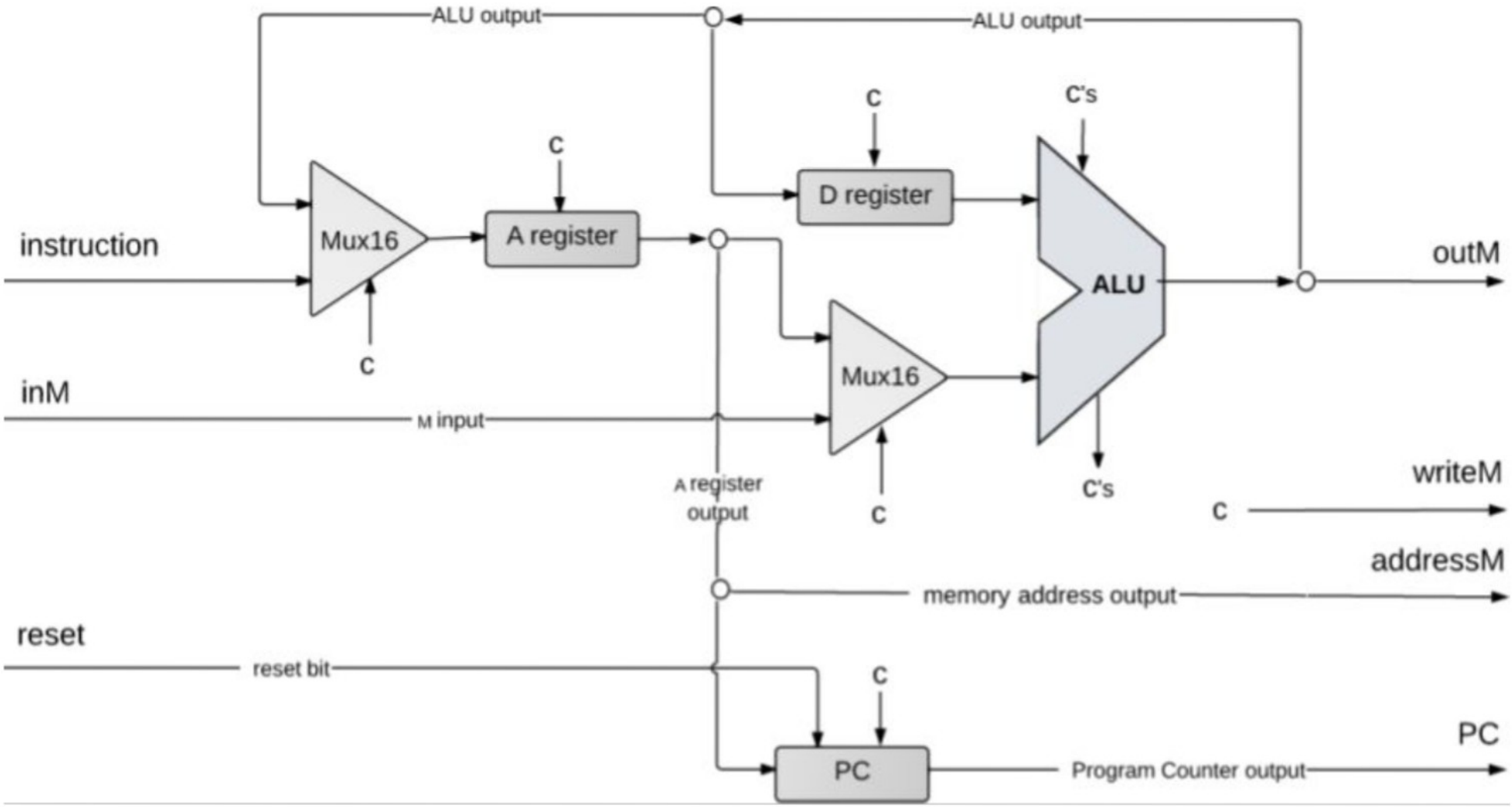
```
CHIP Computer {
    IN reset;

    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/

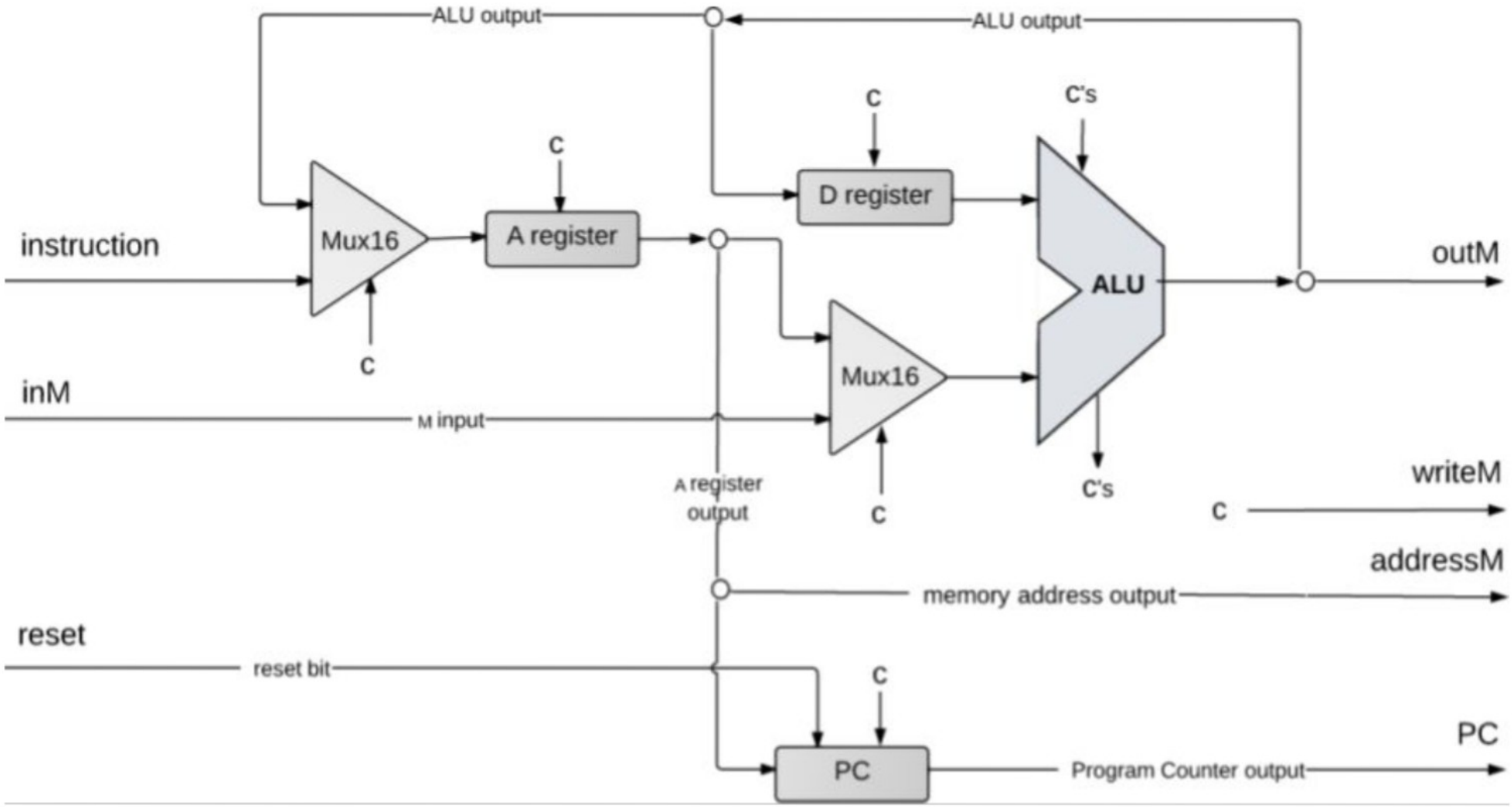
https://zhangruochi.com/Computer-Architecture/2019/06/03/

https://qiita.com/dgkz/items/437426b6b50f41e718c7

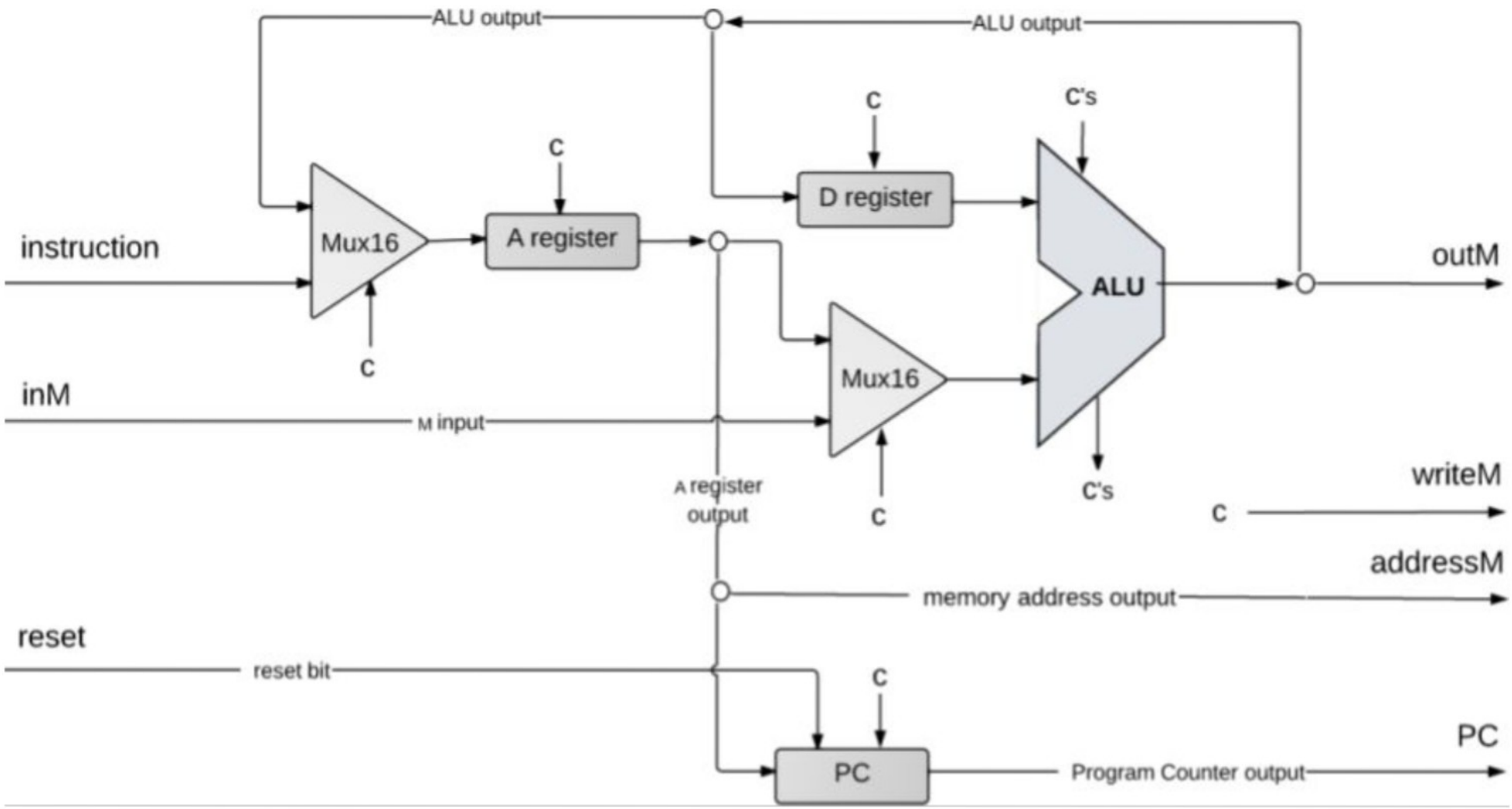
https://people.duke.edu/~nts9/logicgates/CPU.hdl



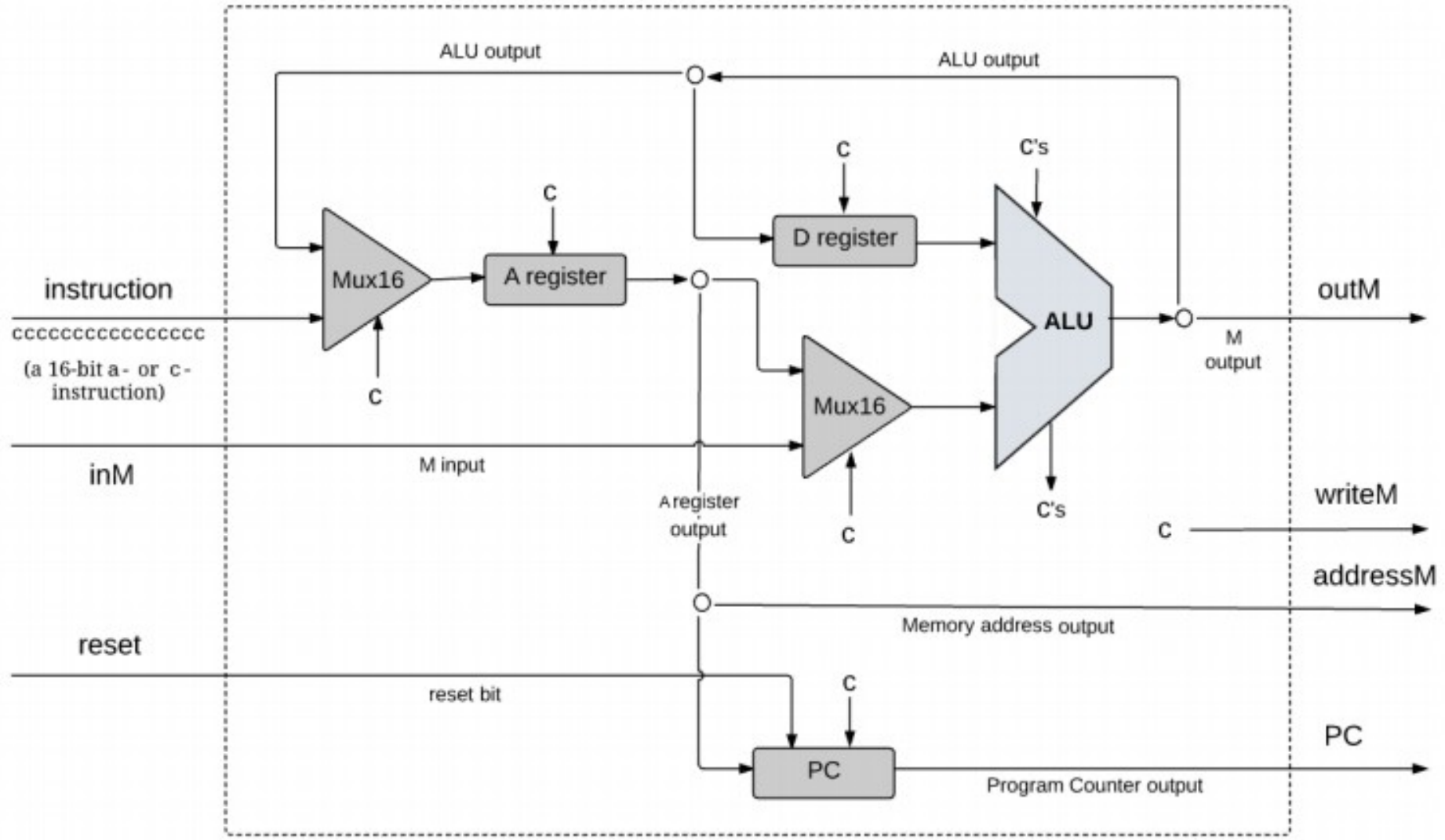
https://github.com/havivha/Nand2Tetris/blob/master/05/CPU.hdl



https://github.com/Olical/nand2tetris/blob/master/chips/architecture/CPU.hdl



https://aua-uff-co.de/prozessor/



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

    OUT outM[16],        // M value output
        writeM,          // Write to M?
        addressM[15],    // Address in data memory (of M)
        pc[15];          // address of next instruction

    PARTS:
    // a or c-type instruction?
    Not(in=instruction[15], out=aType);
    Or(a=instruction[12], b=false, out=cType);

    // where to store
    And(a=instruction[5], b=instruction[15], out=writeA);
    And(a=instruction[4], b=instruction[15], out=writeD);
    And(a=instruction[3], b=instruction[15], out=writeM);

    // logical conditions needed for the control logic
    Not(in=zr, out=jne);
    Not(in=jle, out=jgt);
    Or(a=zr, b=jgt, out=jge);
    Or(a=zr, b=ng, out=jle);

    // the actual control logic
    Mux8Way(a=false, b=jgt, c=zr, d=jge, e=ng, f=jne, g=jle, h=true,
sel=instruction[0..2],
            out=jumpIfaType);
    And(a=instruction[15], b=jumpIfaType, out=doJump);

    // the mux on the left of diagam 5.9
    Mux16(a=ALUout, b=instruction, sel=aType, out=Ain);

    // A register (could use plain register, but tests want it)
    Or(a=aType, b=writeA, out=loadA);
    ARegister(in=Ain, load=loadA, out=Aout, out[0..14]=addressM);

    // D register (could use plain register, but tests want it)
    DRegister(in=ALUout, load=writeD, out=registerD);

    // the mux on the right of diagam 5.9
    Mux16(a=Aout, b=inM, sel=cType, out=inputALU);
    // alu
    ALU(x=registerD, y=inputALU, zx=instruction[11], nx=instruction[10],
zy=instruction[9],
        ny=instruction[8], f=instruction[7], no=instruction[6], zr=zr, ng=ng,
out=ALUout);

    // needed for feeding back as outputs of the CPU cannot be fed back
    Or16(a=false, b=ALUout, out=outM);
    // program counter
    PC(in=Aout, load=doJump, inc=true, reset=reset, out[0..14]=pc);
```

```
CHIP Computer {
    IN reset;

    PARTS:
    CPU (reset=reset, instruction=i, inM=j, addressM=k, writeM=l, outM=m,
pc=n);
    Memory(in=m ,load=l ,address=k ,out=j);
    ROM32K (address=n, out=i);
}
```

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

    PARTS:
    DMux4Way(in=load, sel=address[13..14], a=a, b=b, c=c, d=d);
    Or(a=a, b=b, out=i);
    RAM16K(in=in, load=i, address=address[0..13], out=j);
    Screen(in=in, load=c, address=address[0..12], out=k);
    Keyboard(out=l);
    Mux4Way16(a=j, b=j, c=k, d=l, sel=address[13..14], out=out);
}
```