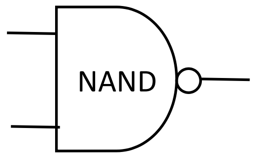
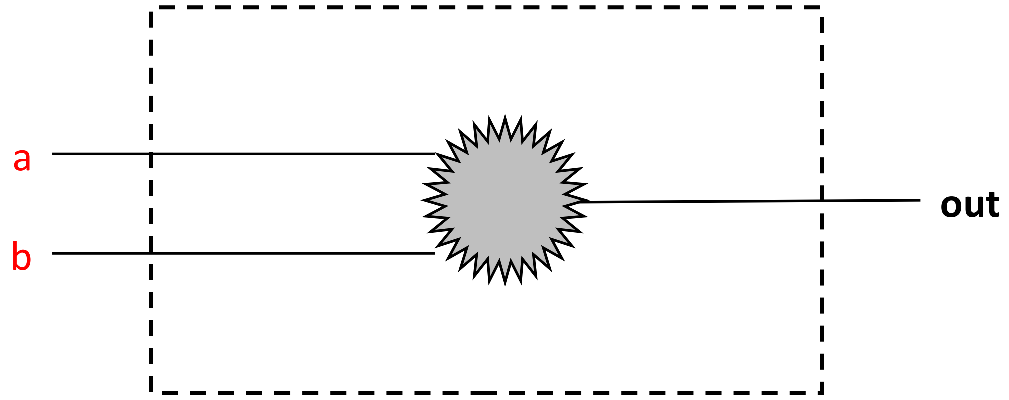
NAND



2 transistors

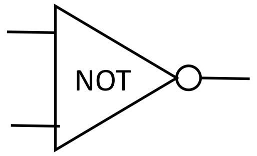


(circuit élémentaire)

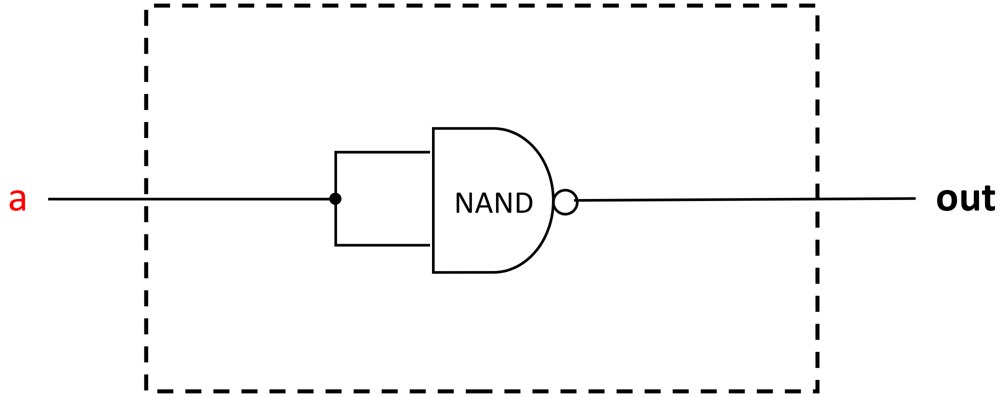
a	b	out
0	0	**1**
0	1	**1**
1	0	**1**
1	1	**0**

```
def op_nand(a,b):
    return ((1,1),(1,0))[a][b]
```

NOT



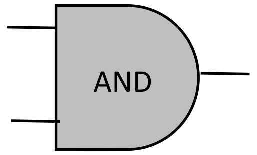
2 transistors



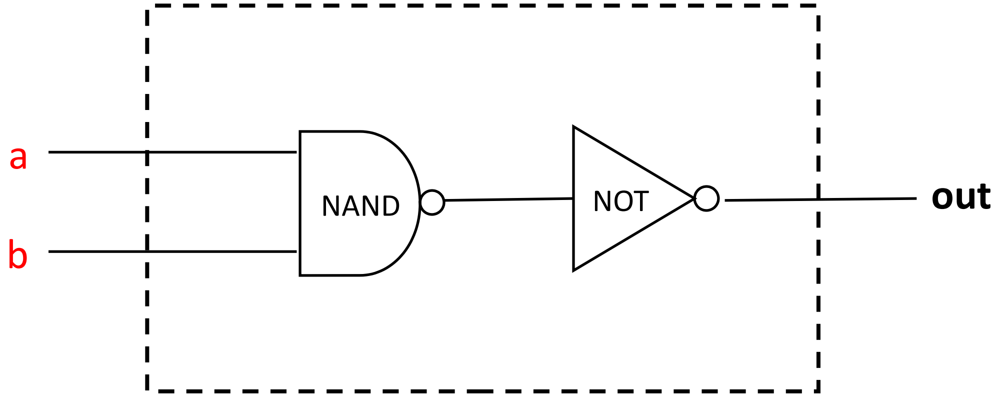
a	**out**
0	**1**
1	**0**

```
def op_not(a):
    return op_nand(a,a)
```

AND



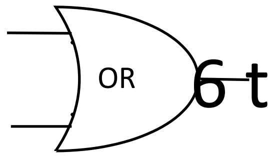
4 transistors



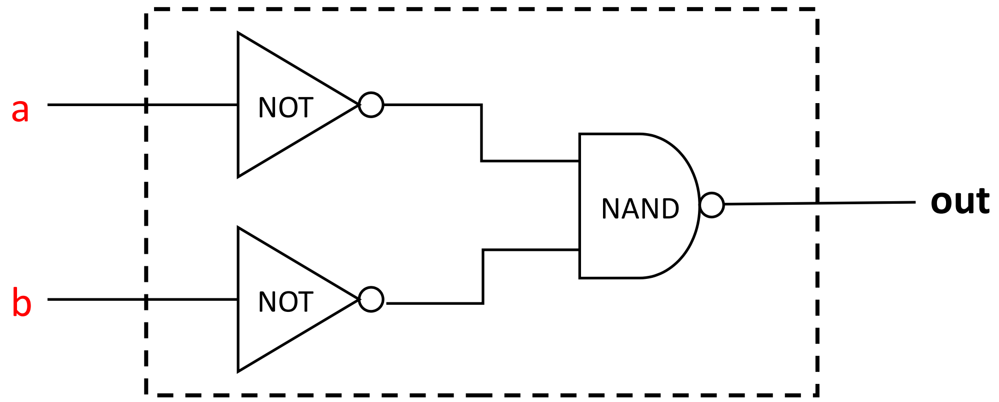
a	b	out
0	0	**0**
0	1	**0**
1	0	**0**
1	1	**1**

```
def op_and(a,b):
    return op_not(op_nand(a,b))
```

OR



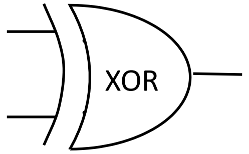
6 transistors



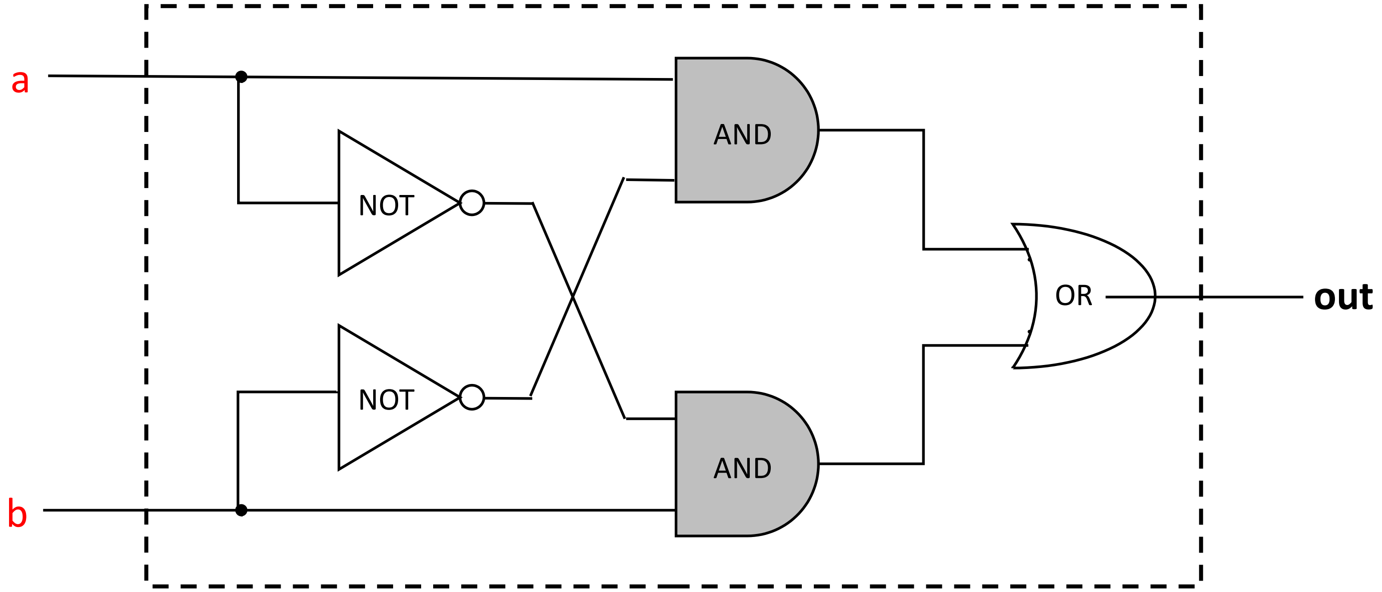
a	b	**out**
0	0	**0**
0	1	**1**
1	0	**1**
1	1	**1**

```
def op_or(a,b):
    return op_nand(op_not(a),op_not(b))
```

XOR



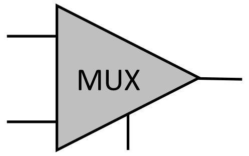
18 transistors



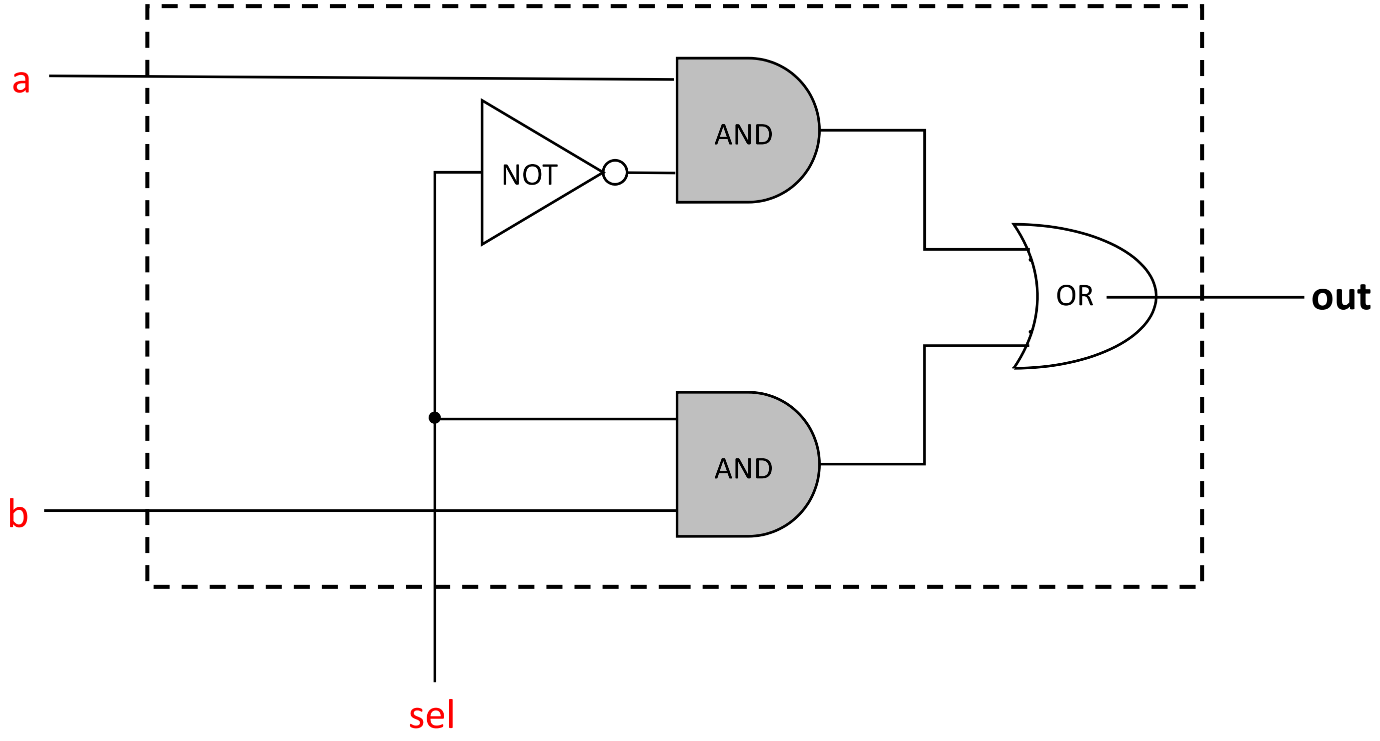
a	b	out
0	0	**0**
0	1	**1**
1	0	**1**
1	1	**0**

```
def op_xor(a,b):
    return op_or(op_and(a,op_not(b)),op_and(op_not(a),b))
```

MUX



16 transistors



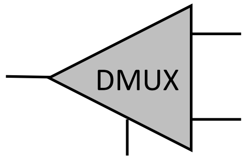
a	b	sel	**out**
0	0	0	**0**
0	1	0	**0**
1	0	0	**1**
1	1	0	**1**
0	0	1	**0**
0	1	1	**1**
1	0	1	**0**
1	1	1	**1**

$\approx$

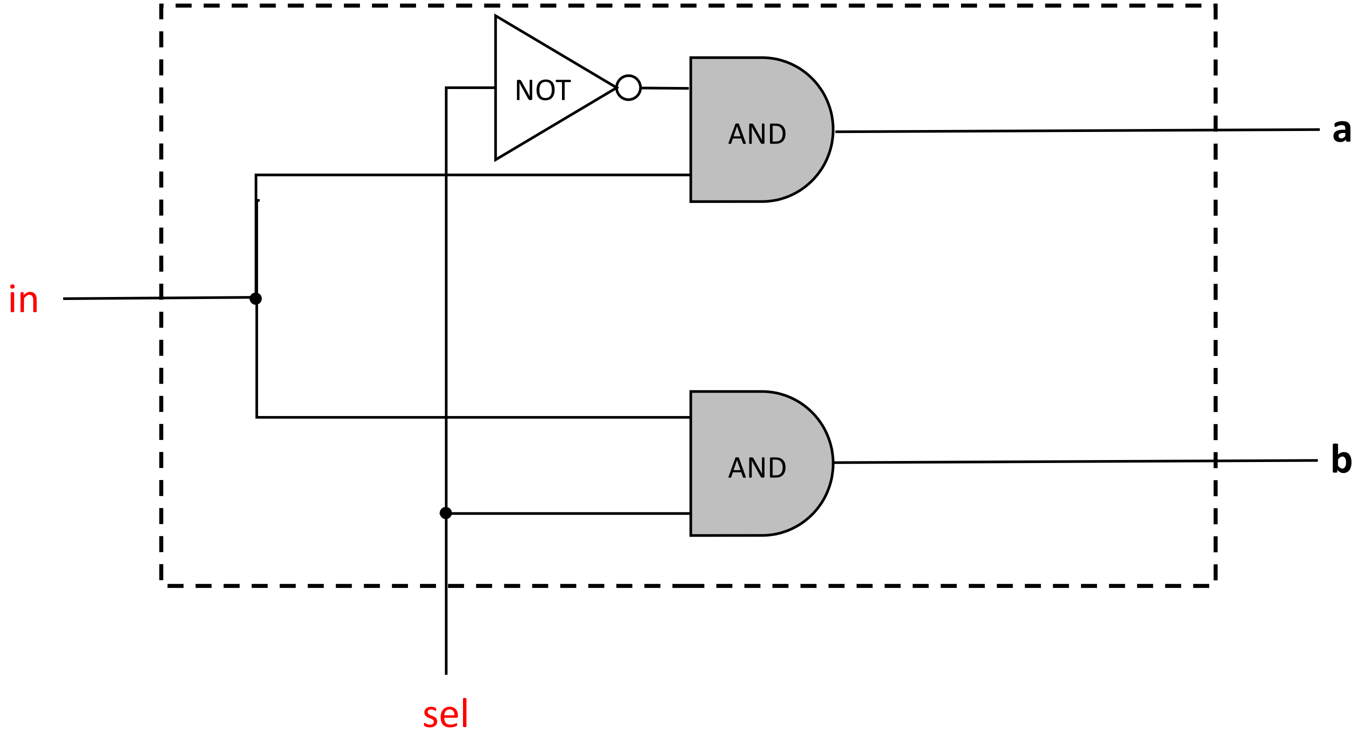
sel	**out**
0	**a**
1	**b**

```
def op_mux(sel,a,b):
    return op_or(op_and(a,op_not(sel)),op_and(b,sel))
```

DMUX



10 transistors



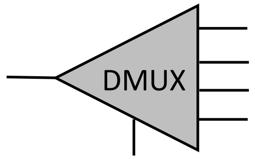
in	sel	**a**	**b**
0	0	**0**	**0**
0	1	**0**	**0**
1	0	**1**	**0**
1	1	**0**	**1**

≈

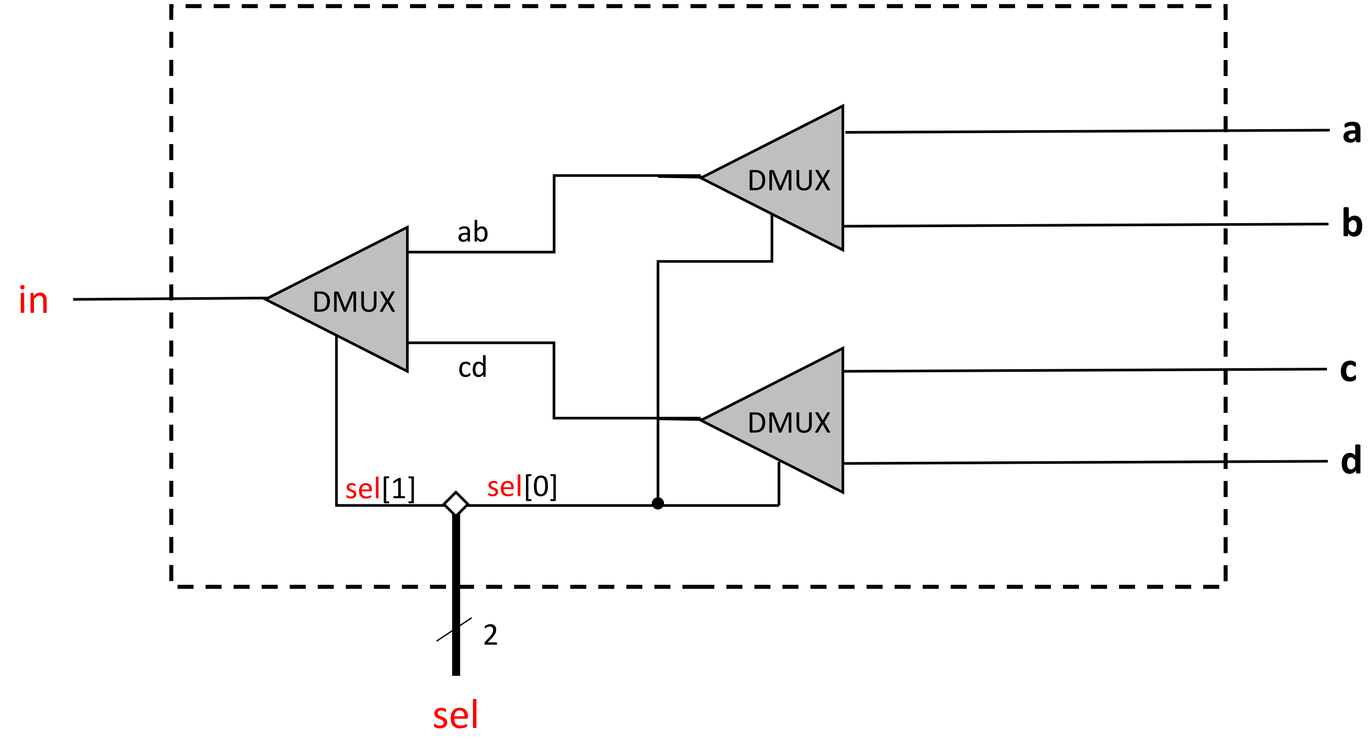
sel	**a**	**b**
0	**in**	**0**
1	**0**	**in**

```
def op_dmux(sel,inp):
    return op_and(op_not(sel),inp),op_and(sel,inp)
```

DMUX4WAY



30 transistors



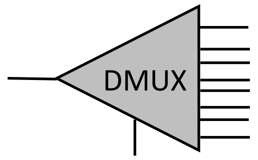
```
def op_dmux4way(sel,inp):
    ab,cd = op_dmux(sel[1],inp)
    return op_dmux(sel[0],ab)+op_dmux(sel[0],cd)
```

in	sel	a	b	c	d
0	00	**0**	**0**	**0**	**0**
0	01	**0**	**0**	**0**	**0**
0	10	**0**	**0**	**0**	**0**
0	11	**0**	**0**	**0**	**0**
1	00	**1**	**0**	**0**	**0**
1	01	**0**	**1**	**0**	**0**
1	10	**0**	**0**	**1**	**0**
1	11	**0**	**0**	**0**	**1**

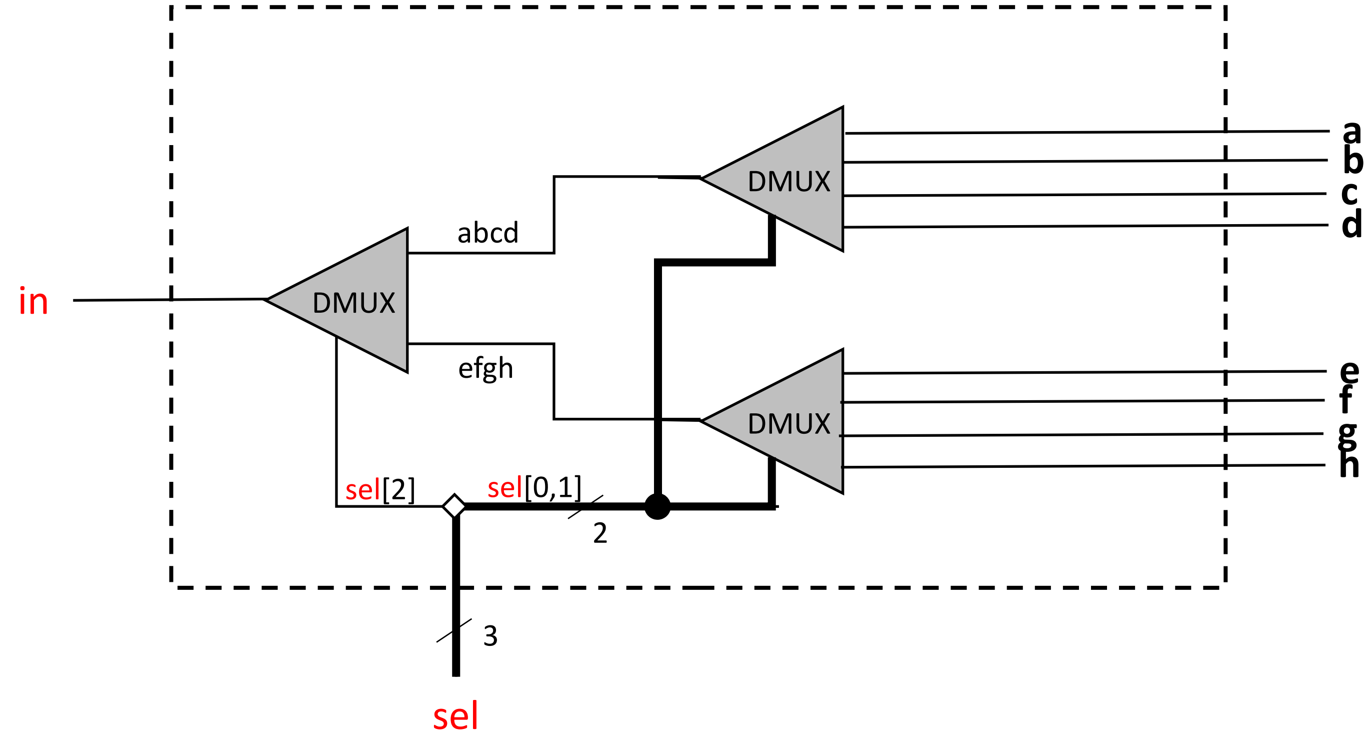
≈

sel	a	b	c	d
00	**in**	**0**	**0**	**0**
01	**0**	**in**	**0**	**0**
10	**0**	**0**	**in**	**0**
11	**0**	**0**	**0**	**in**

DMUX8WAY



70 transistors



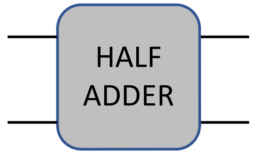
```
def op_dmux8way(sel,inp):
    abcd,efgh = op_dmux(sel[2],inp)
    return op_dmux4way(sel[:2],abcd)+\
           op_dmux4way(sel[:2],efgh)
```

in	sel	a	b	c	d	e	f	g	h
0	000	0	0	0	0	0	0	0	0
0	001	0	0	0	0	0	0	0	0
0	010	0	0	0	0	0	0	0	0
0	011	0	0	0	0	0	0	0	0
0	100	0	0	0	0	0	0	0	0
0	101	0	0	0	0	0	0	0	0
0	110	0	0	0	0	0	0	0	0
0	111	0	0	0	0	0	0	0	0
1	000	1	0	0	0	0	0	0	0
1	001	0	1	0	0	0	0	0	0
1	010	0	0	1	0	0	0	0	0
1	011	0	0	0	1	0	0	0	0
1	100	0	0	0	0	1	0	0	0
1	101	0	0	0	0	0	1	0	0
1	110	0	0	0	0	0	0	1	0
1	111	0	0	0	0	0	0	0	1

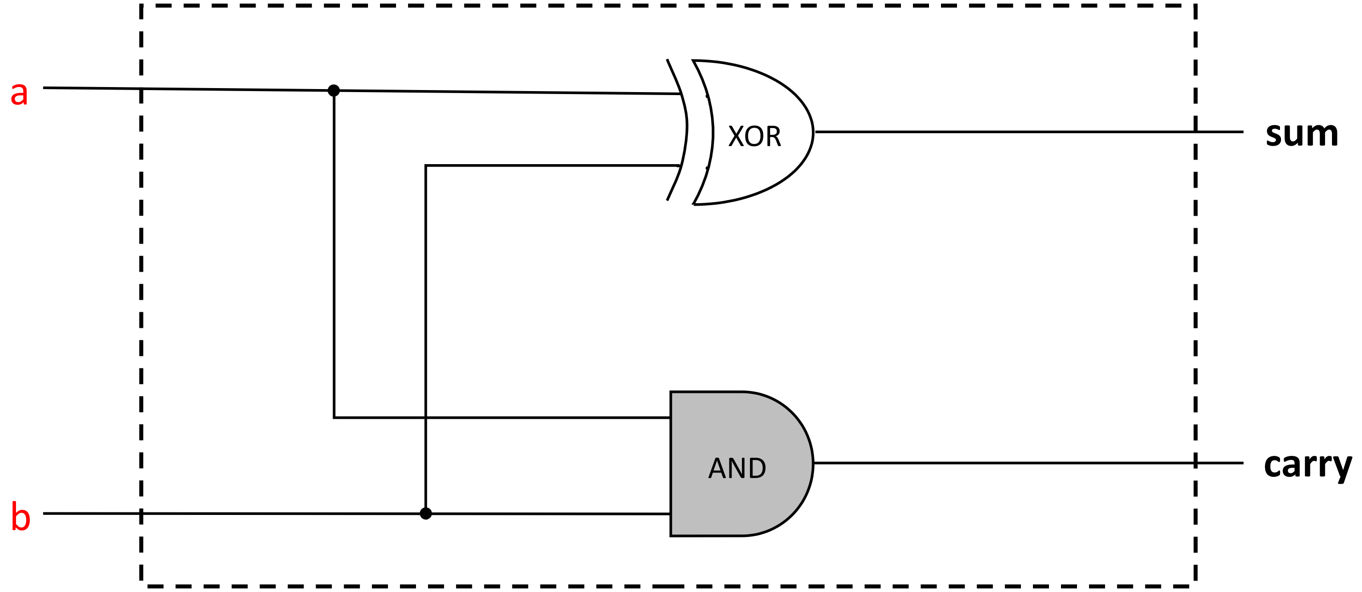
≈

sel	a	b	c	d	e	f	g	h
000	in	0	0	0	0	0	0	0
001	0	in	0	0	0	0	0	0
010	0	0	in	0	0	0	0	0
011	0	0	0	in	0	0	0	0
100	0	0	0	0	in	0	0	0
101	0	0	0	0	0	in	0	0
110	0	0	0	0	0	0	in	0
111	0	0	0	0	0	0	0	in

HALF ADDER



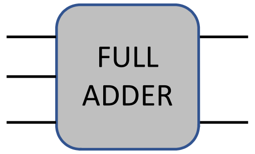
22 transistors



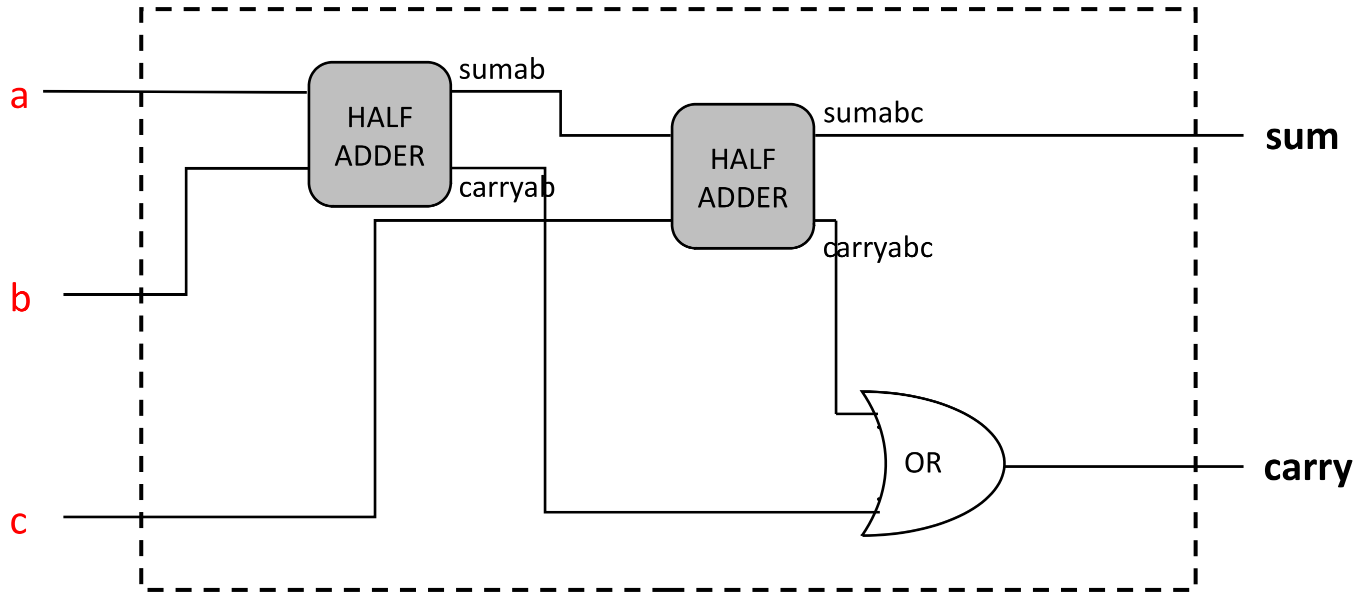
a	b	sum	carry
0	0	**0**	**0**
0	1	**1**	**0**
1	0	**1**	**0**
1	1	**0**	**1**

```
def op_halfadder(a,b):
    return op_xor(a,b),op_and(a,b)
```

FULL ADDER



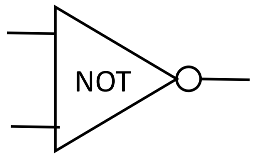
50 transistors



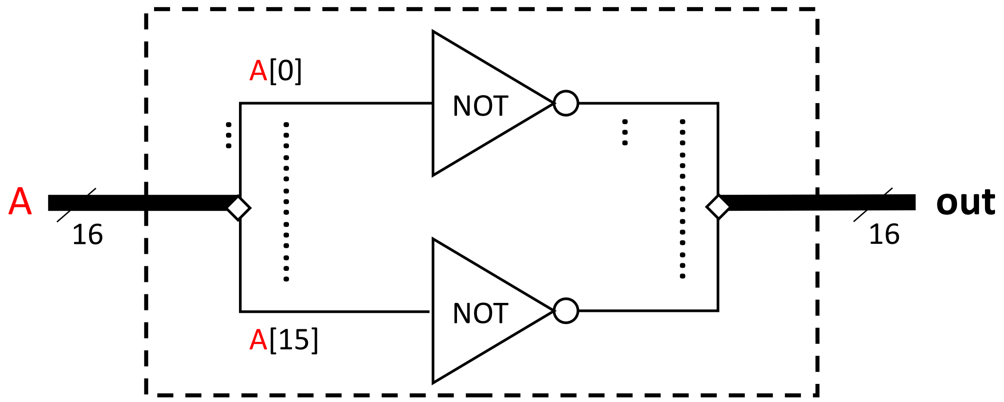
a	b	c	sum	carry
0	0	0	**0**	**0**
0	0	1	**1**	**0**
0	1	0	**1**	**0**
0	1	1	**0**	**1**
1	0	0	**1**	**0**
1	0	1	**0**	**1**
1	1	0	**0**	**1**
1	1	1	**1**	**1**

```
def op_fulladder(a,b,c):
    sumab,carryab = op_halfadder(a,b)
    sumabc,carryabc = op_halfaddr(sumab,c)
    return sumabc,op_or(carryab,carryabc)
```

NOT16

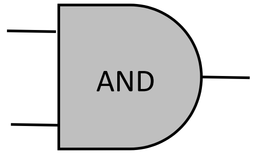


32 transistors

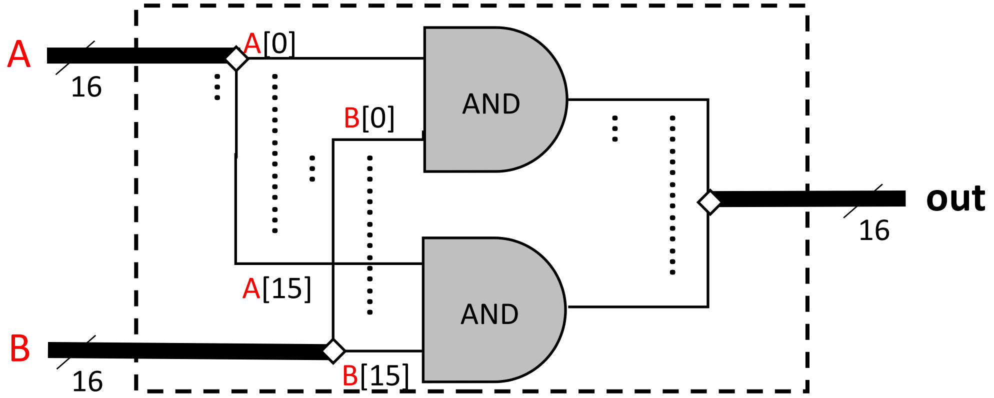


```
def op_not16(A):
    return tuple(op_not(a) for a in A)
```

AND16

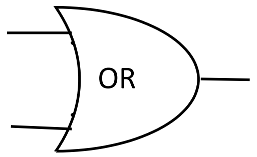


64 transistors

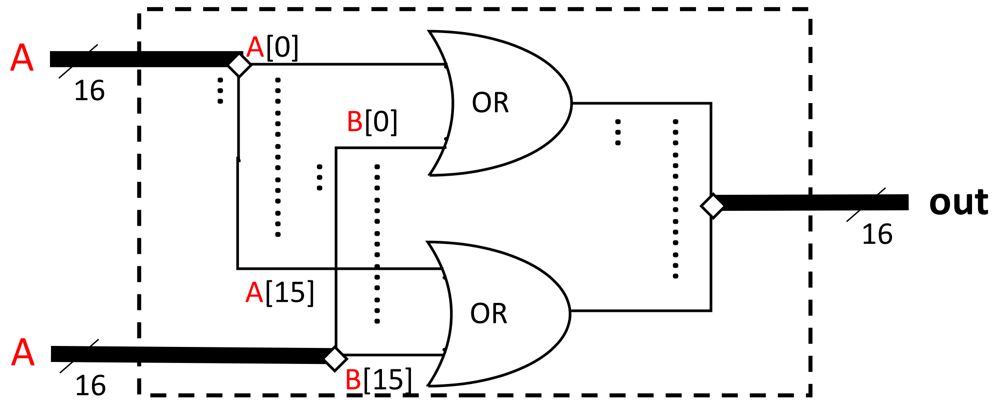


```
def op_and16(A,B):
    return tuple(op_and(a,b) for a,b in zip(A,B))
```

OR16

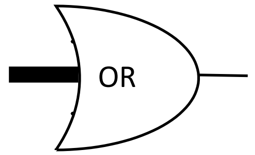


96 transistors

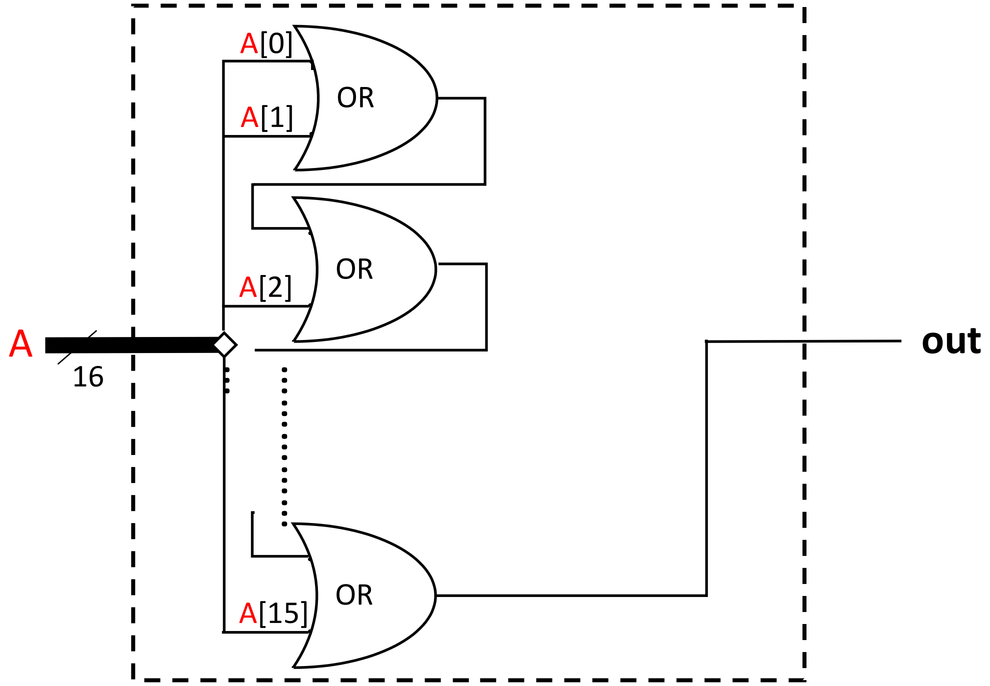


```
def op_or16(A,B):
    return tuple(op_or(a,b) for a,b in zip(A,B))
```

OR16WAY

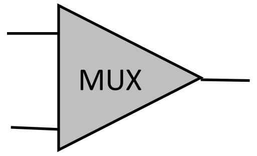


96 transistors

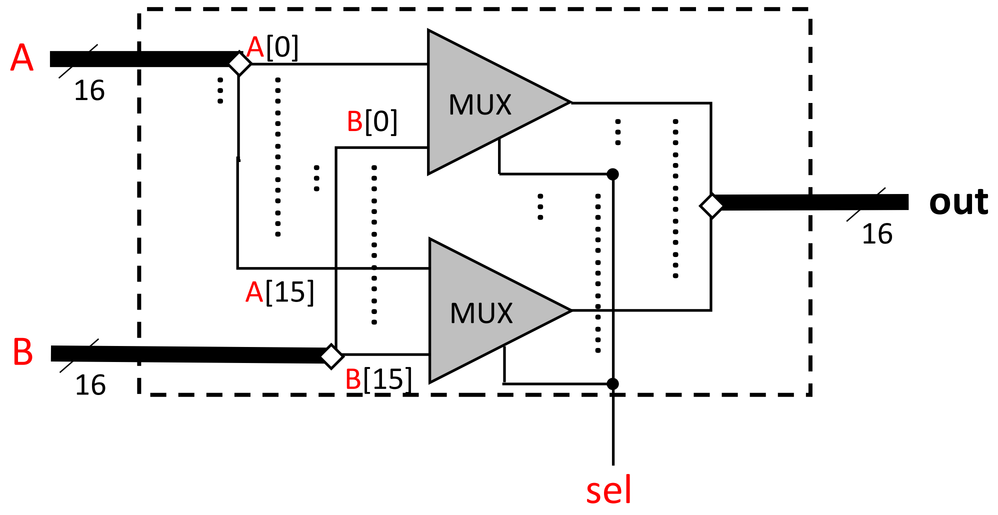


```
def op_or16way(A):
    return reduce(op_or,A)
```

MUX16

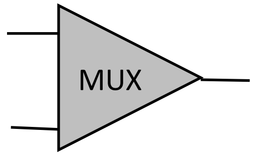


256 transistors

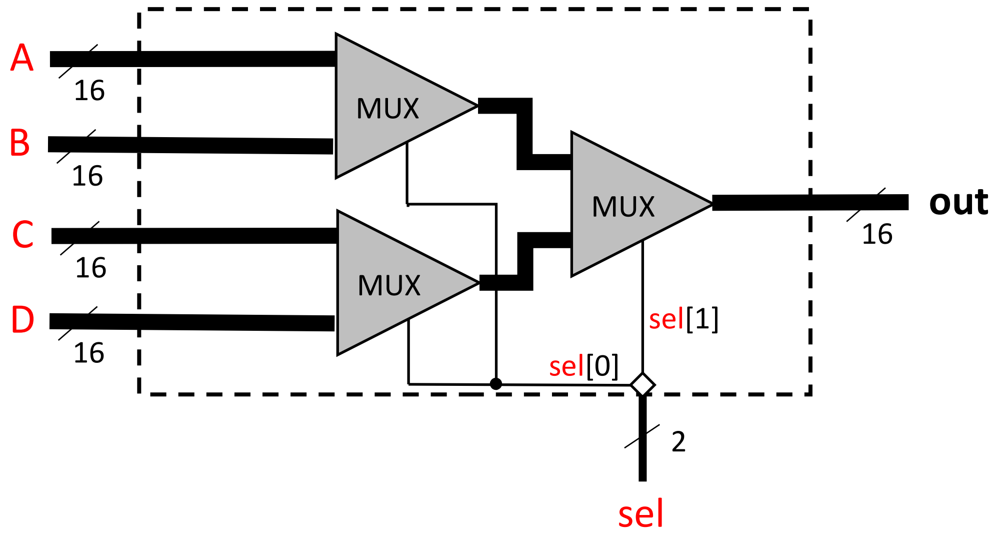


```
def op_mux16(sel,A,B):
    return tuple(op_mux(sel,a,b) for a,b in zip(A,B))
```

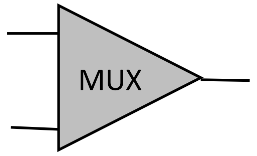
MUX4WAY16

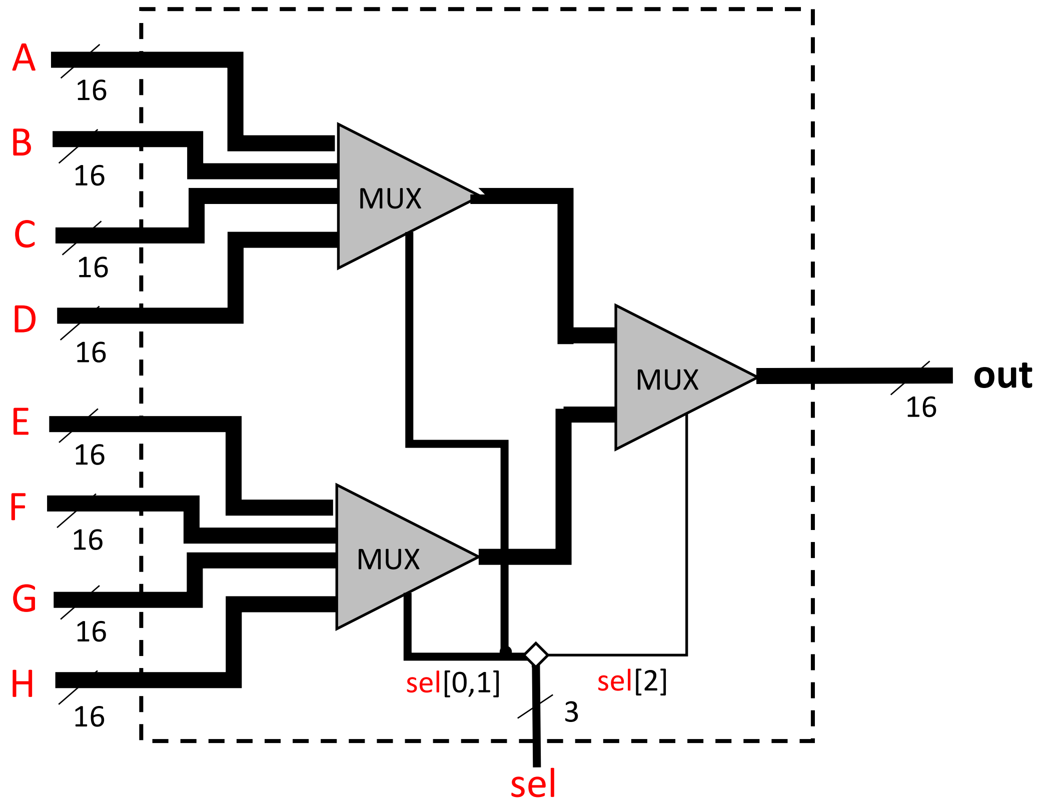


768 transistors



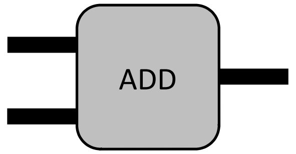
```
def op_mux4way16(sel,A,B,C,D):
    return op_mux16(sel[1],op_mux16(sel[0],A,B),op_mux16(sel[0],C,D))
```

MUX8WAY16  1792 transistors

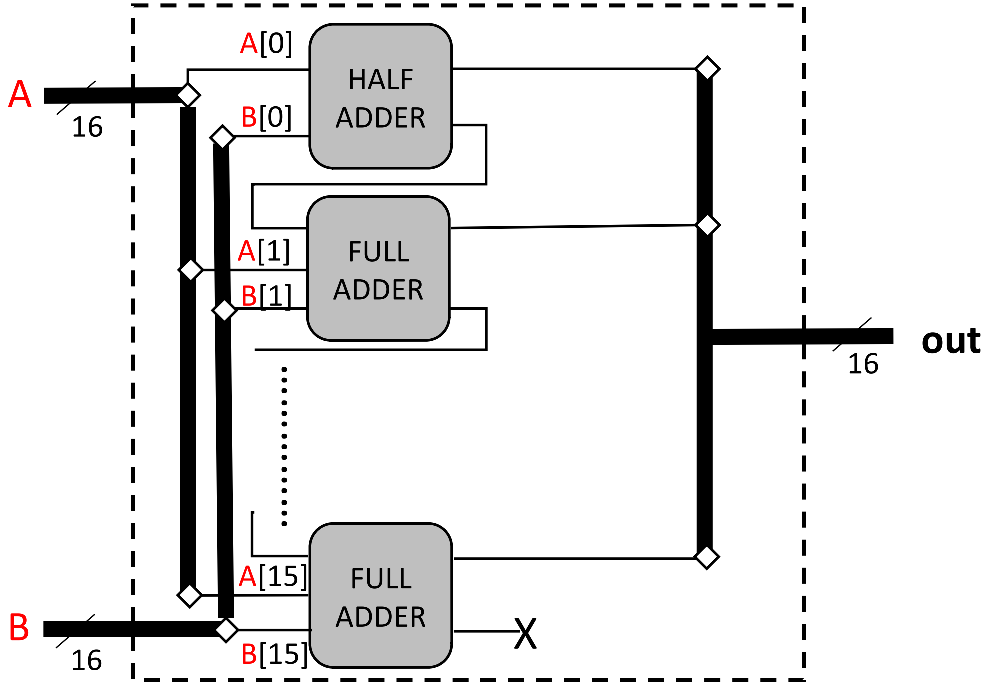


```
def op_mux8way16(sel,A,B,C,D,E,F,G,H):
    return op_mux16(sel[2],op_mux4way16(sel[:2],A,B,C,D),op_mux4way16(sel[:2],E,F,G,H))
```

ADD16

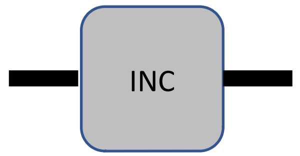


800 transistors

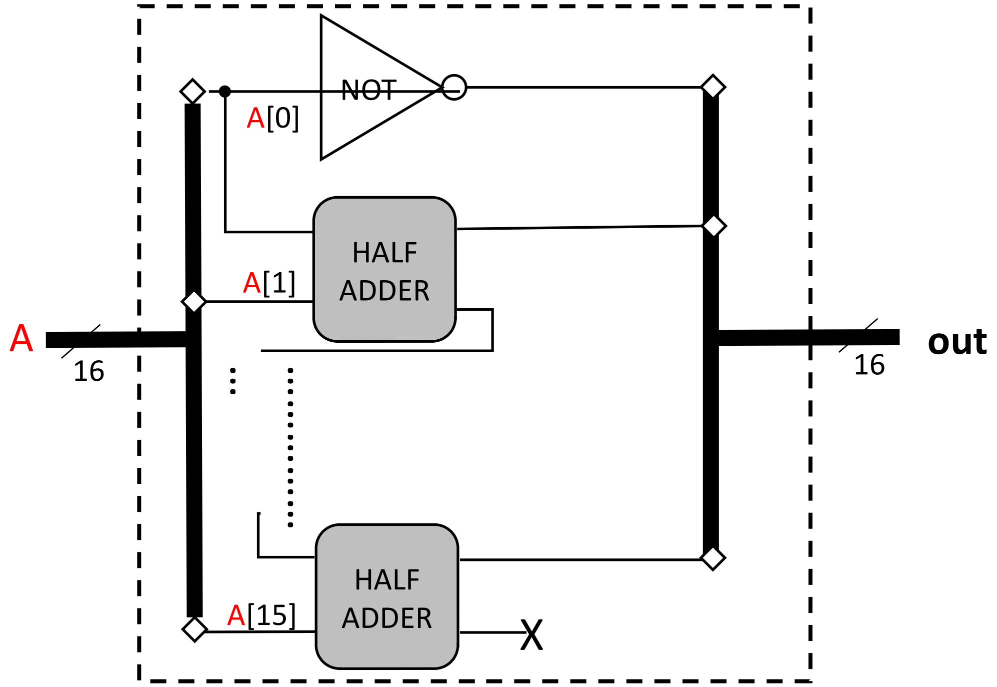


```
def op_add16(A,B):
    r = 0
    return tuple(outs[0] for a,b in zip(A,B) if (outs:=op_fulladder(a,b,r), r:=outs[1]))
```

INC16

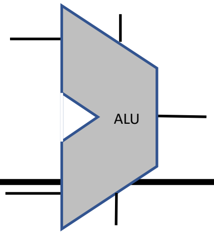


352 transistors

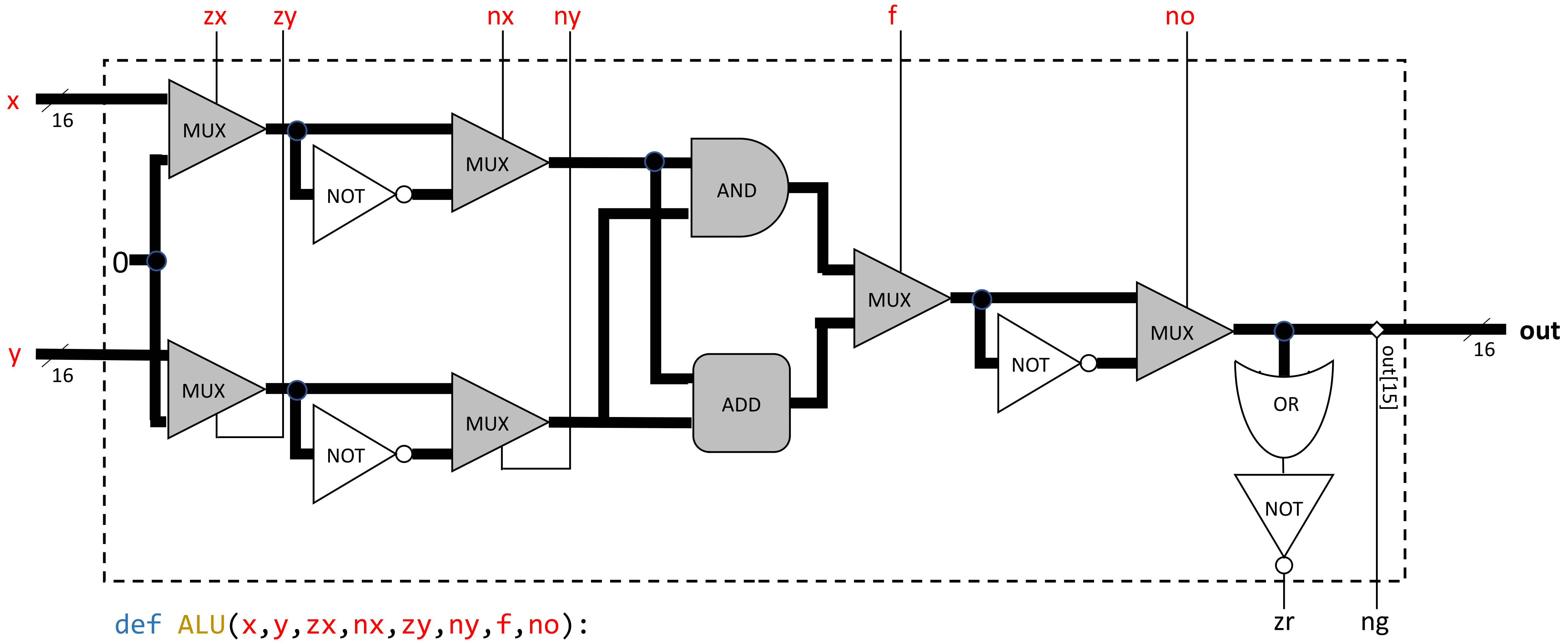


```
def op_inc16(A):
    r = 1
    return tuple(outs[0] for a in A if (outs:=op_halfadder(a,r), r:=outs[1]))
```

ALU

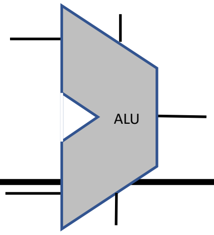


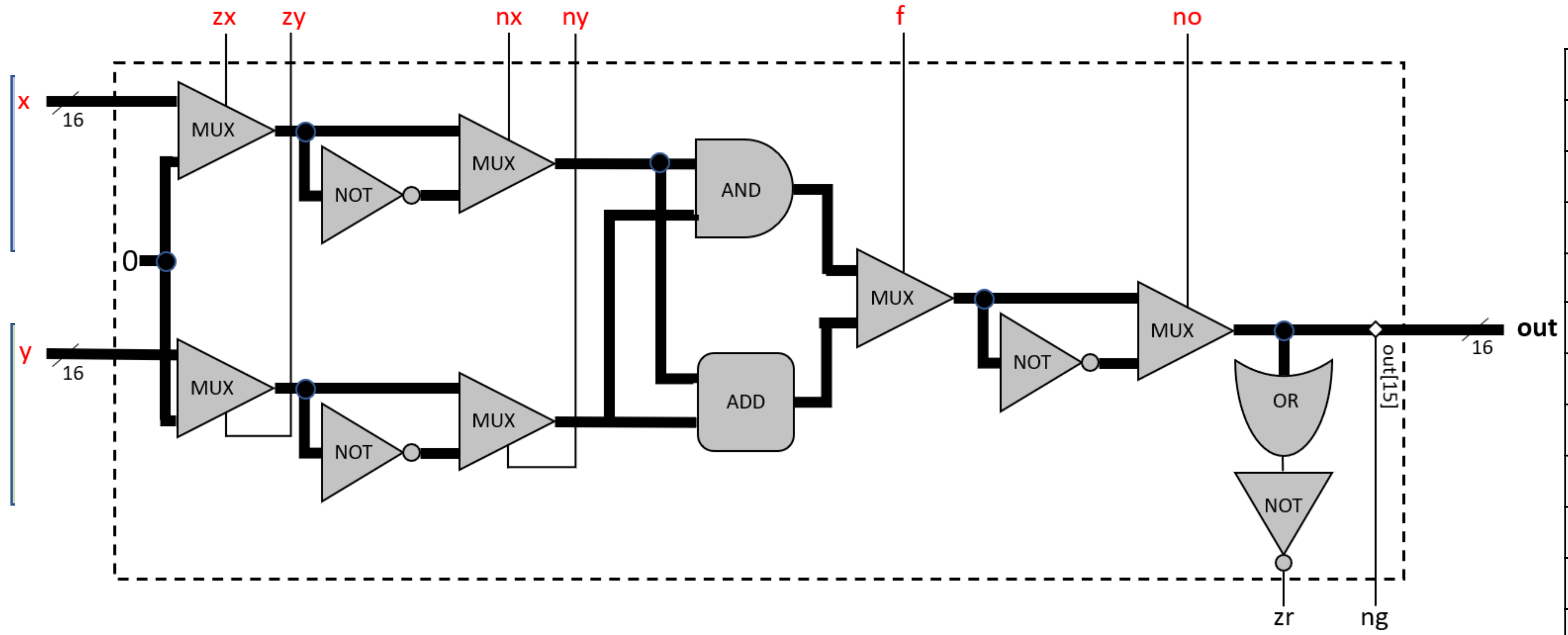
2594 transistors



```
def ALU(x,y,zx,nx,zy,ny,f,no):
    ...
    return out,zr,ng
```

ALU



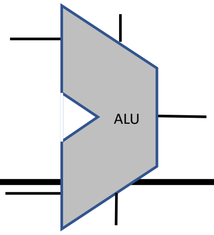


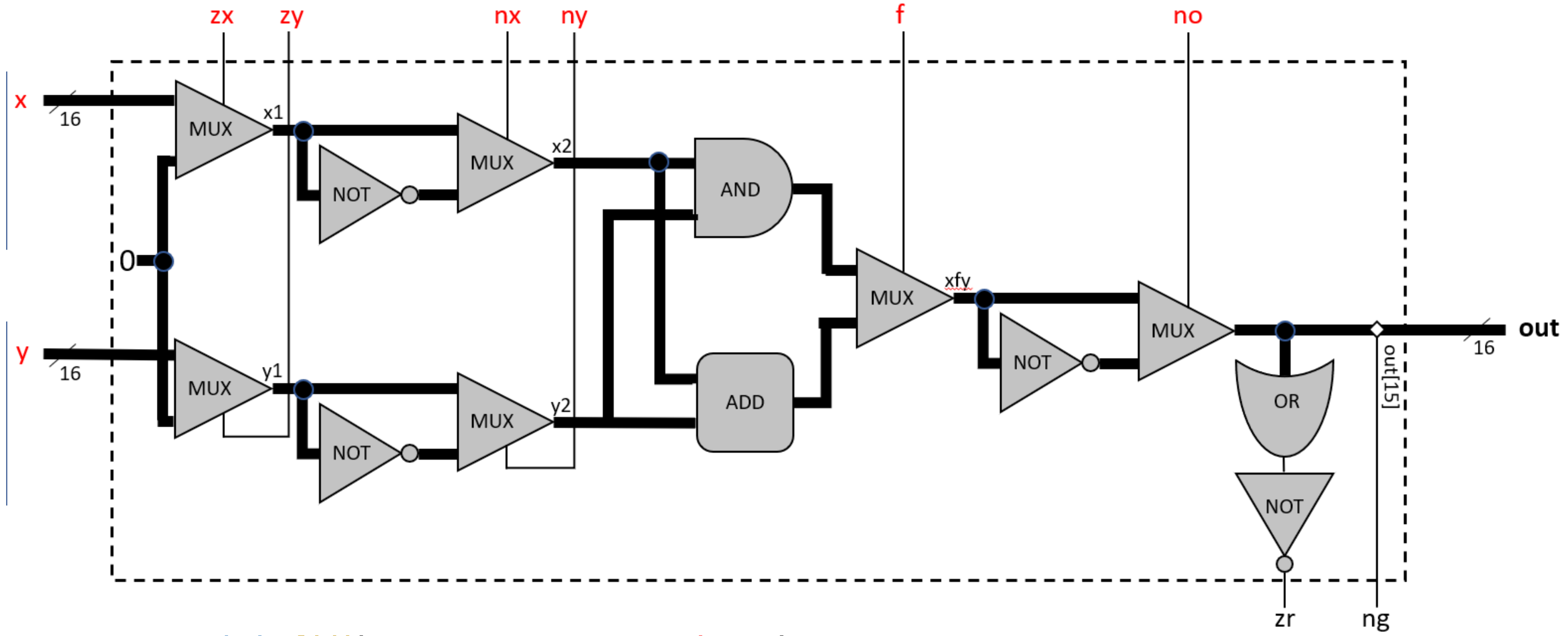
zx	nx	zy	ny	f	no	out
1	0	1	0	1	0	**0**
1	1	1	1	1	1	**1**
1	1	1	0	1	0	**-1**
0	0	1	1	0	0	**x**
1	1	0	0	0	0	**y**
0	0	1	1	0	1	**!x**
1	1	0	0	0	1	**!y**
0	0	1	1	1	1	**-x**
1	1	0	0	1	1	**-y**
0	1	1	1	1	1	**x+1**
1	1	0	1	1	1	**y+1**
0	0	1	1	1	0	**x-1**
1	1	0	0	1	0	**y-1**
0	0	0	0	1	0	**x+y**
0	1	0	0	1	1	**x-y**
0	0	0	1	1	1	**y-x**
0	0	0	0	0	0	**x&y**
0	1	0	1	0	1	**x|y**

$\approx$

Préparer x		Préparer y		Choisir opération	Préparer out			
zx	nx	zy	ny	f	no	out	zr	ng
`if zx:` `x=0`	`if nx:` `x=!x`	`if zy:` `y=0`	`if ny:` `y=!y`	`if f: out=x+y` `else: out=x&y`	`if no:` `out=!out`	`out`	`out==0`	`out<0`

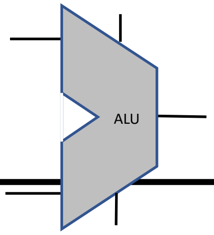
ALU

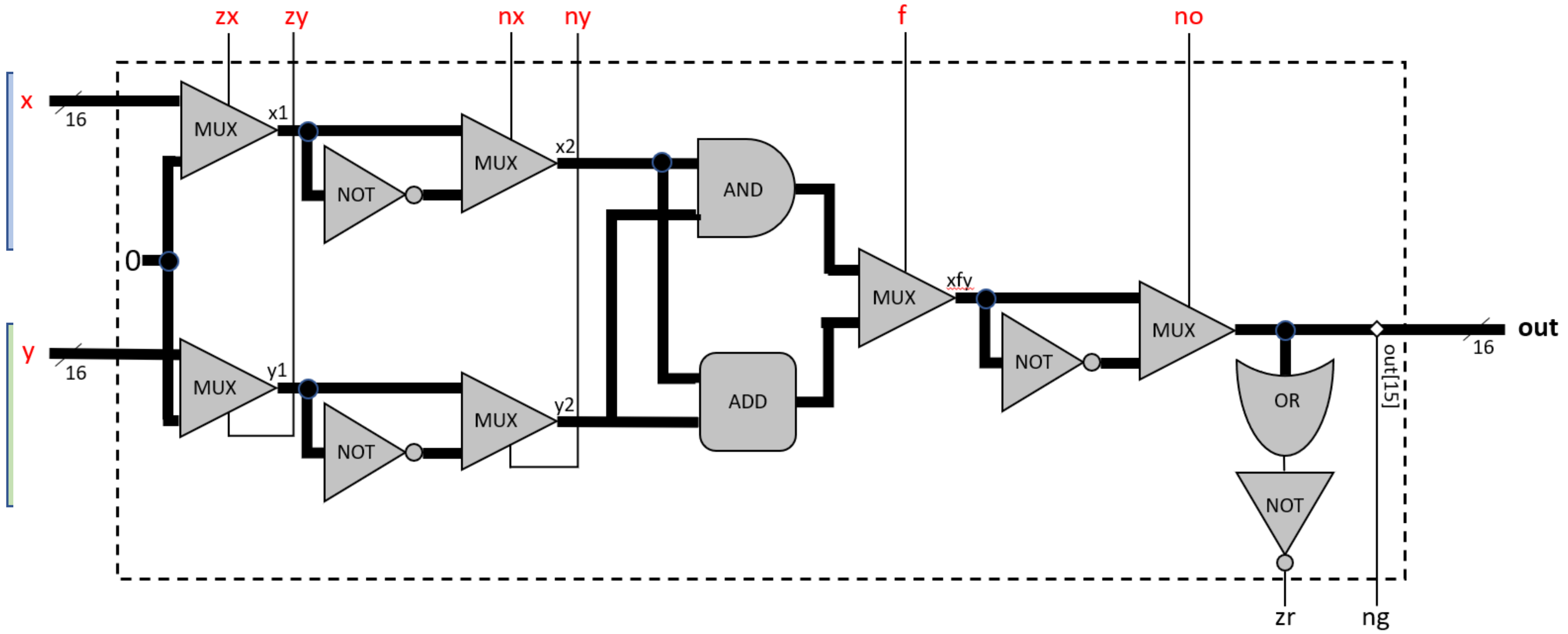




```
def ALU(x,y,zx,nx,zy,ny,f,no):
    zero = (0,)*len(x)
    x1 = op_mux16(zx,x,zero)
    x2 = op_mux16(nx,x1,op_not16(x1))
    y1 = op_mux16(zy,y,zero)
    y2 = op_mux16(ny,y1,op_not16(y1))
    ...
    return out,zr,ng
```

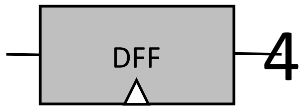
ALU



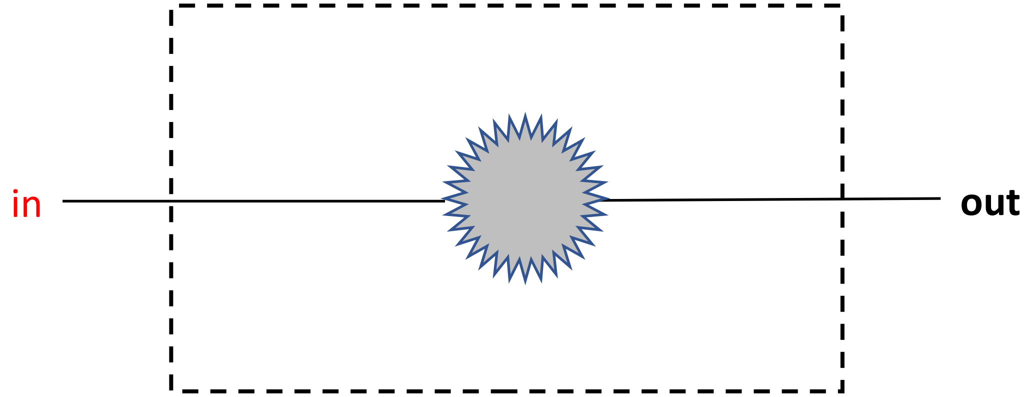


```
def ALU(x,y,zx,nx,zy,ny,f,no):
    ...
    xfy = op_mux16(f,op_and16(x2,y2),op_add16(x2,y2))
    out = op_mux16(no,xfy,op_not16(xfy))
    ng = out[15]
    zr = op_not(op_or16way(out))
    return out,zr,ng
```

Flip-Flop



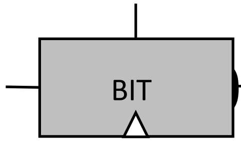
4 transistors

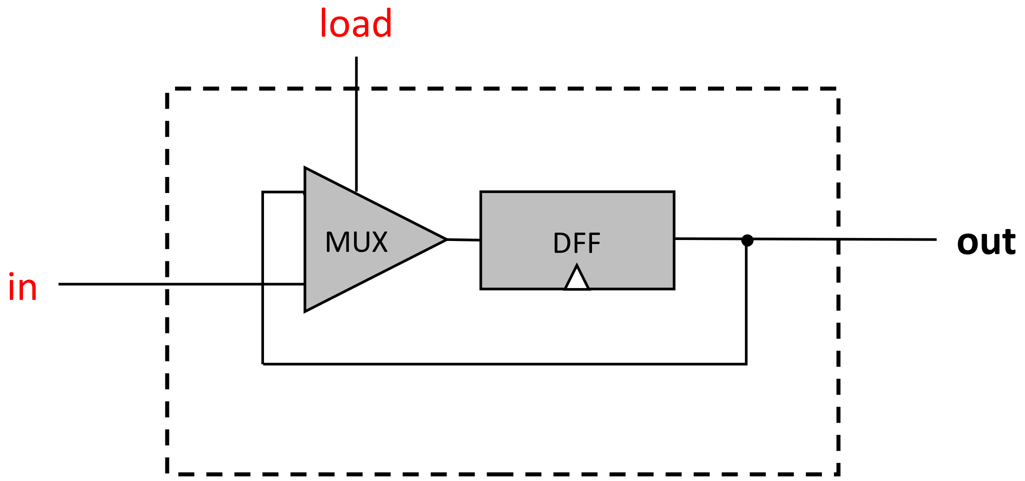


(circuit élémentaire)

out(t+1) = in(t)

Bit transistors

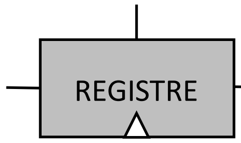




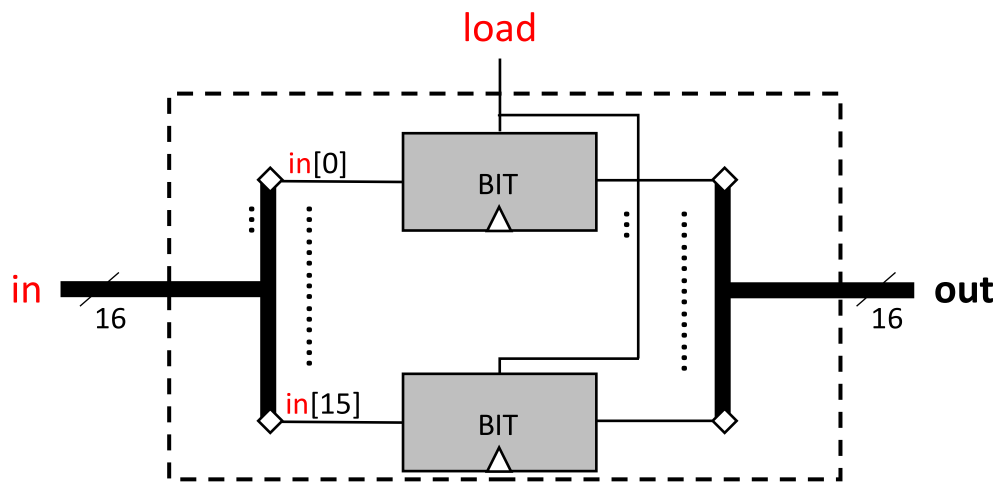
out(t+1) = in(t) if load(t) else out(t)

```
class binon:
    def __init__(self):
        self.val = 0
    def probe(self):
        return self.val
    def set(self,load,inp):
        self.val = op_mux(load,self.val,inp)
```

Registre



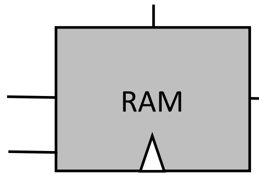
320 transistors



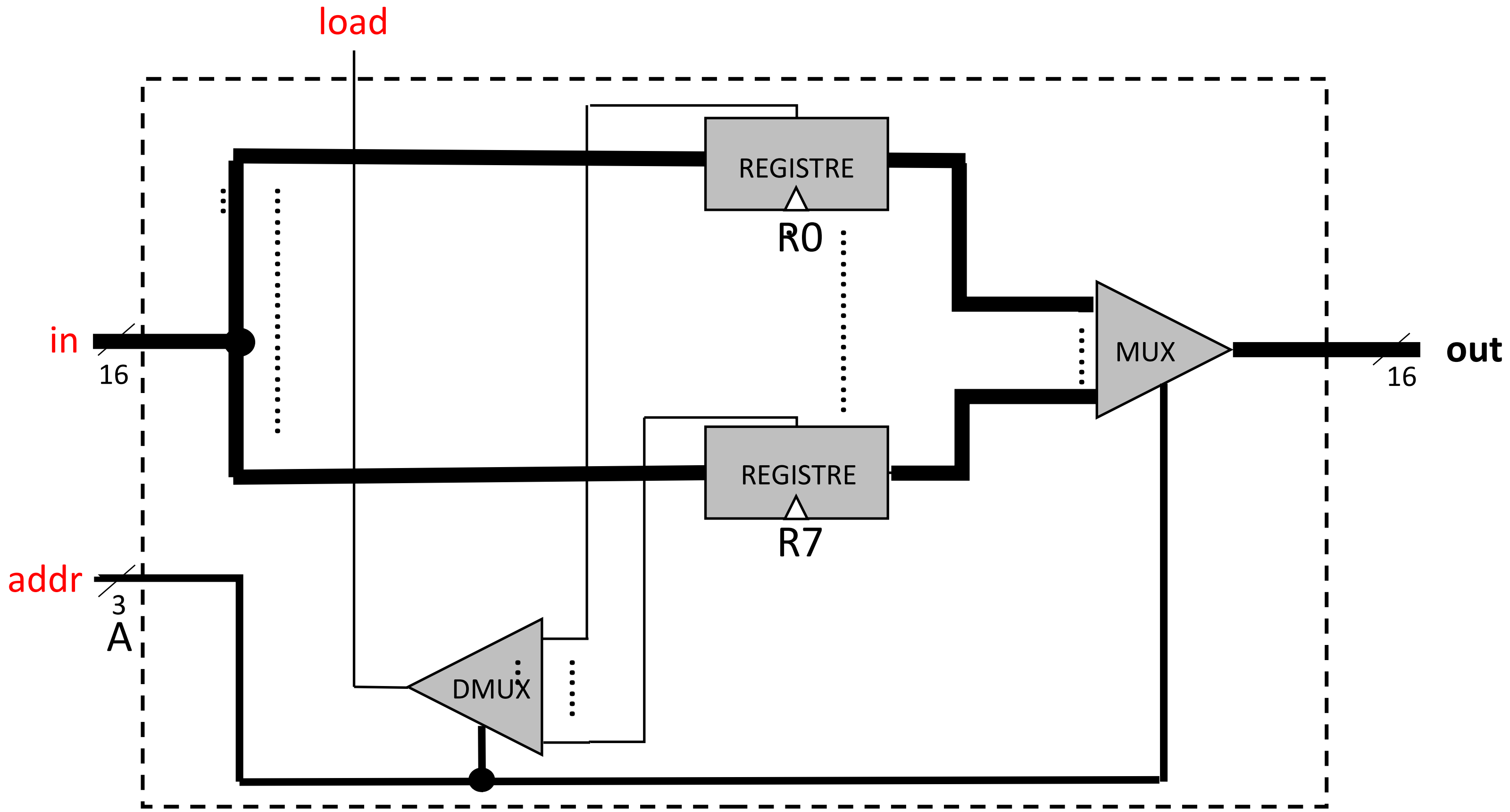
out(t+1) = in(t) if load(t) else out(t)

```
class registre:
    TAILLE=16
    def __init__(self):
        self.binons = tuple(binon() for _ in range(registre.TAILLE))
    def probe(self):
        return tuple(b.probe() for b in self.binons)
    def set(self,load,inp):
        for b,v in zip(self.binons,inp):
            b.set(load,v)
```

RAM8

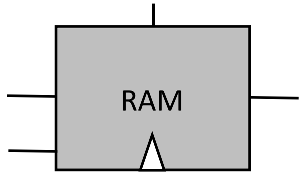


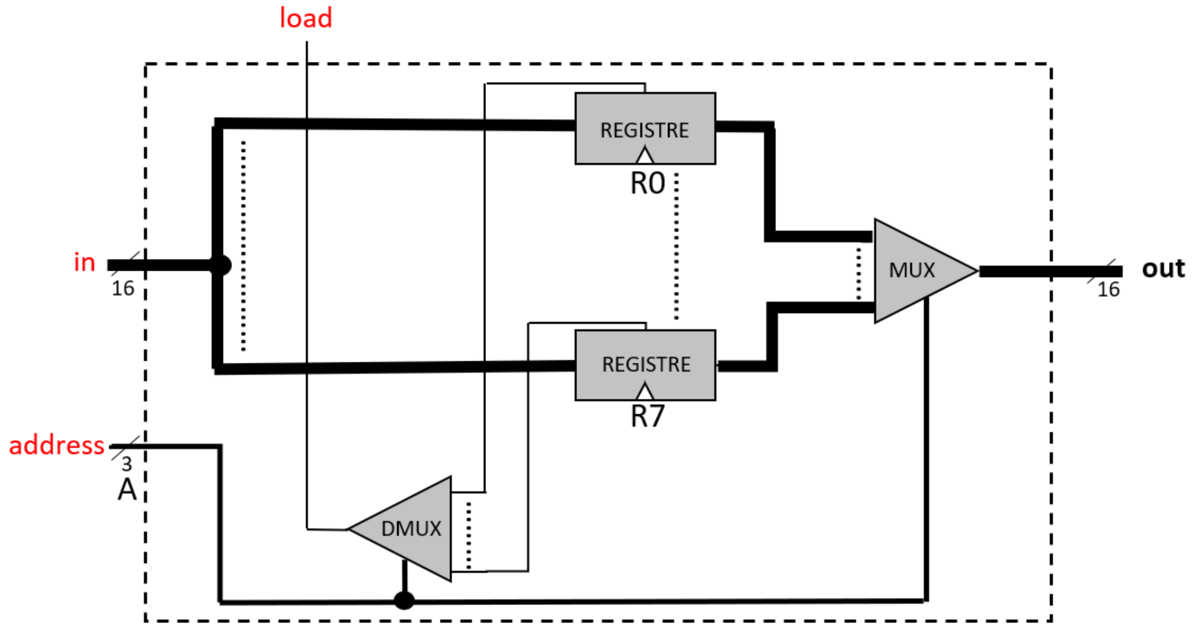
4422 transistors



```
class ram8:
    A=3
    TAILLE=2**A
    ...
```

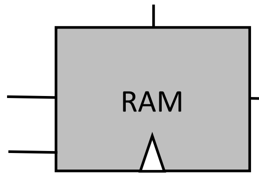
RAM8



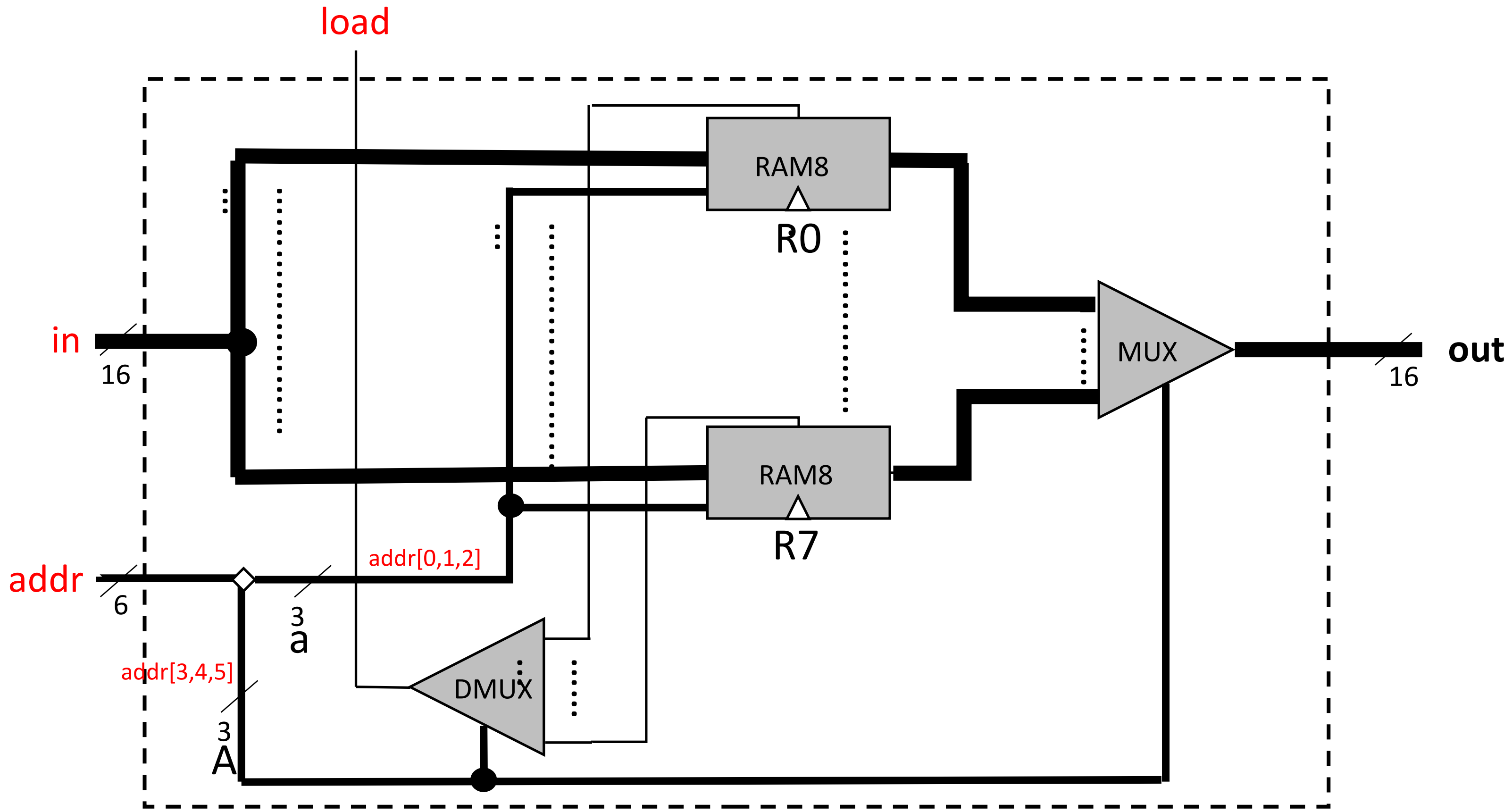


```
class ram8:
    ...
    def __init__(self):
        self.registres = tuple(registre() for _ in range(ram8.TAILLE))
    def probe(self,addr):
        return op_mux8way16(addr,*(r.probe() for r in self.registres))
    def set(self,load,inp,addr):
        loads = op_dmux8way(addr,load)
        for l,r in zip(loads,self.registres):
            r.set(l,inp)
```

RAM64

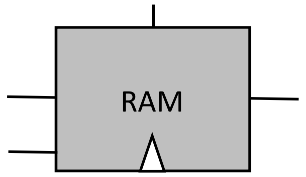


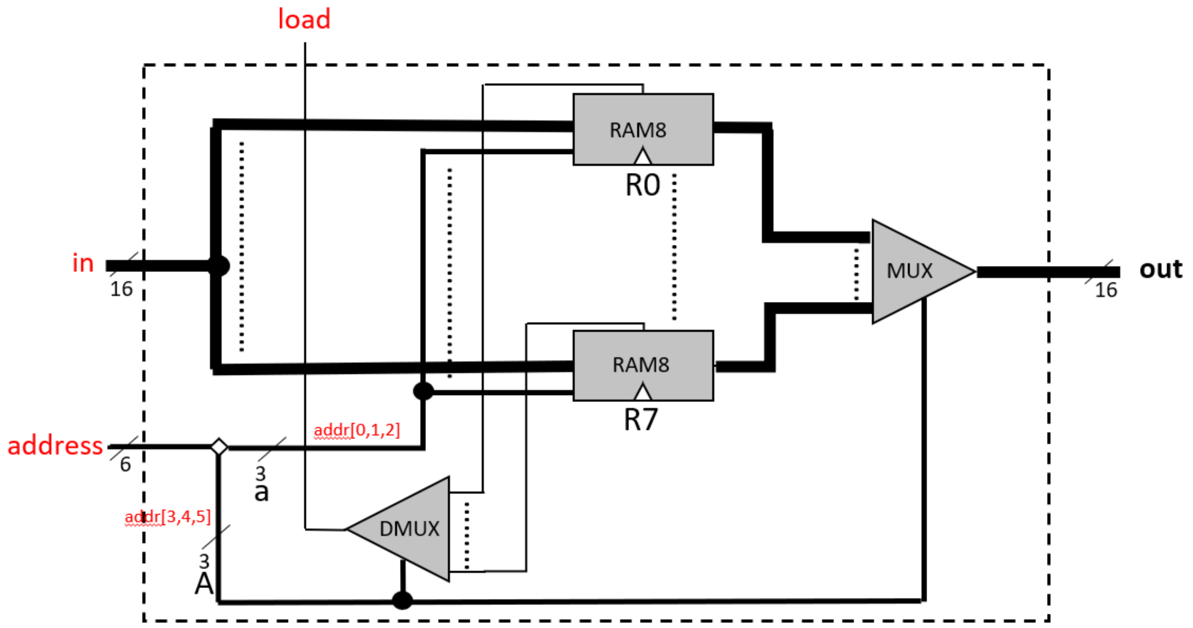
37268 transistors



```
class ram64:
    A,a=3,3
    TAILLE=2**A
    ...
```

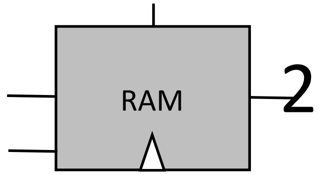
RAM64



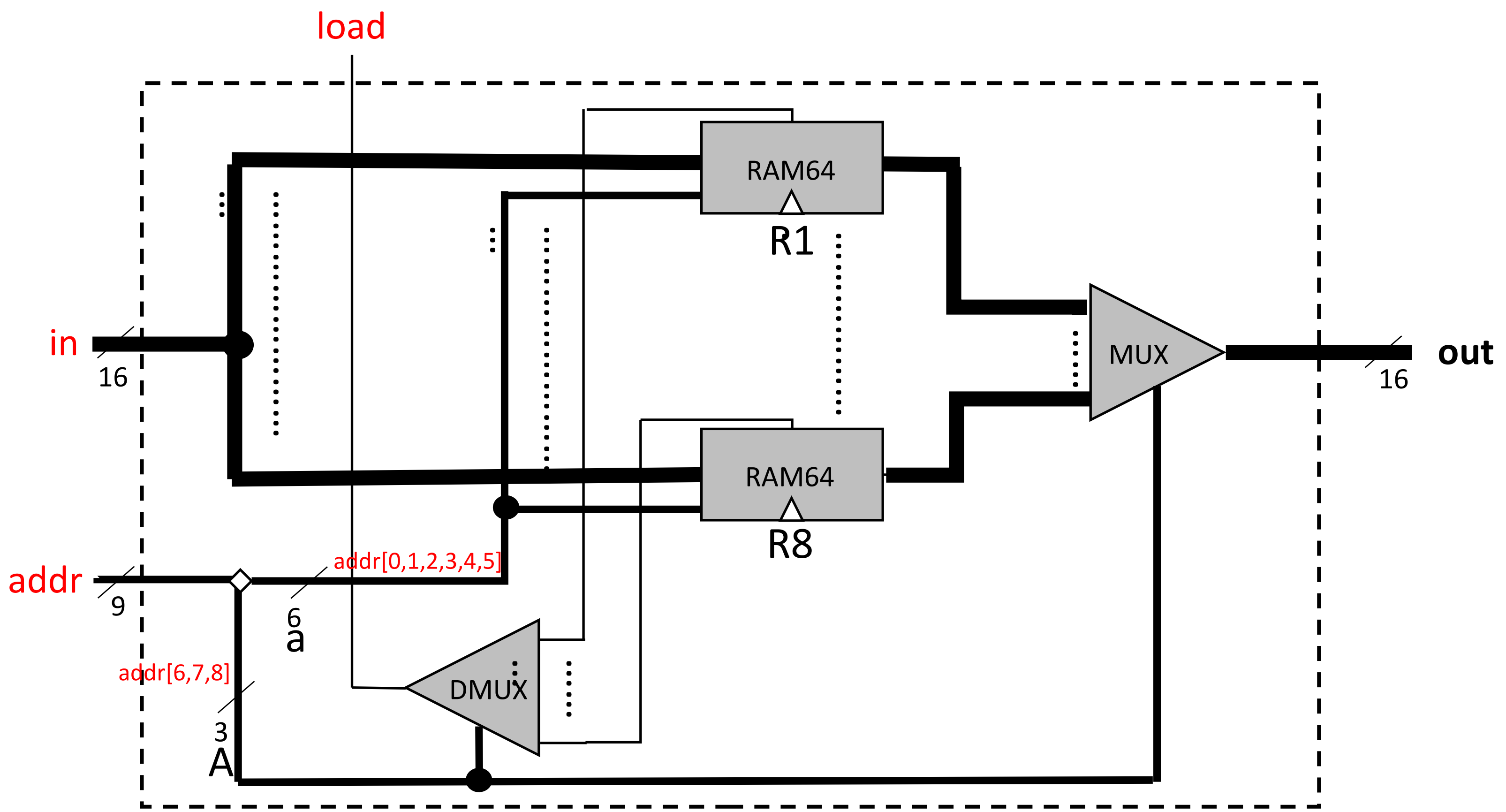


```
class ram64:
    ...
    def __init__(self):
        self.ram8s = tuple(ram8() for _ in range(ram64.TAILLE))
    def probe(self,addr):
        return op_mux8way16(addr[-ram64.A:],*(r.probe(addr[:-ram64.A]) for r in self.ram8s))
    def set(self,load,inp,addr):
        loads = op_dmux8way(addr[-ram64.A:],load)
        for l,r in zip(loads,self.ram8s):
            r.set(l,inp,addr[:-ram64.A])
```

RAM512

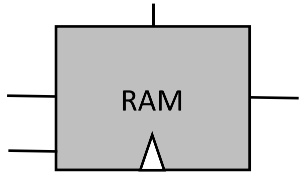


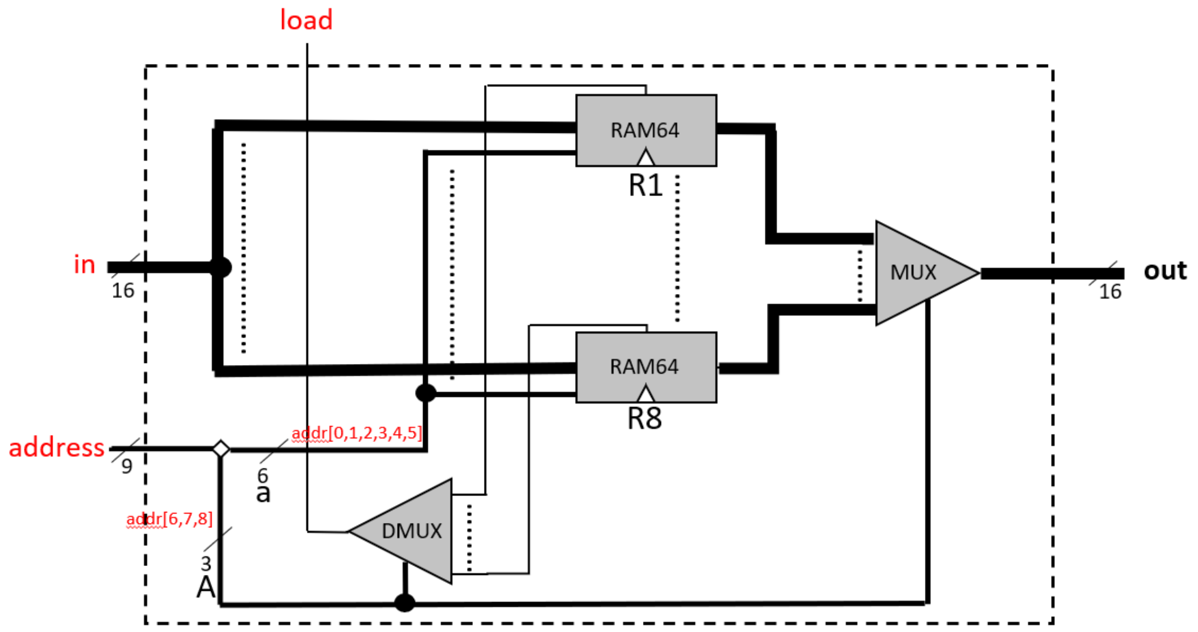
299766 transistors



```
class ram512:
    A,a=3,6
    TAILLE=2**A
    ...
```

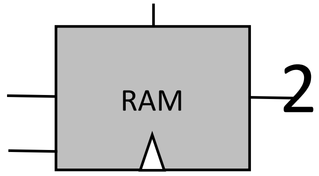
RAM512



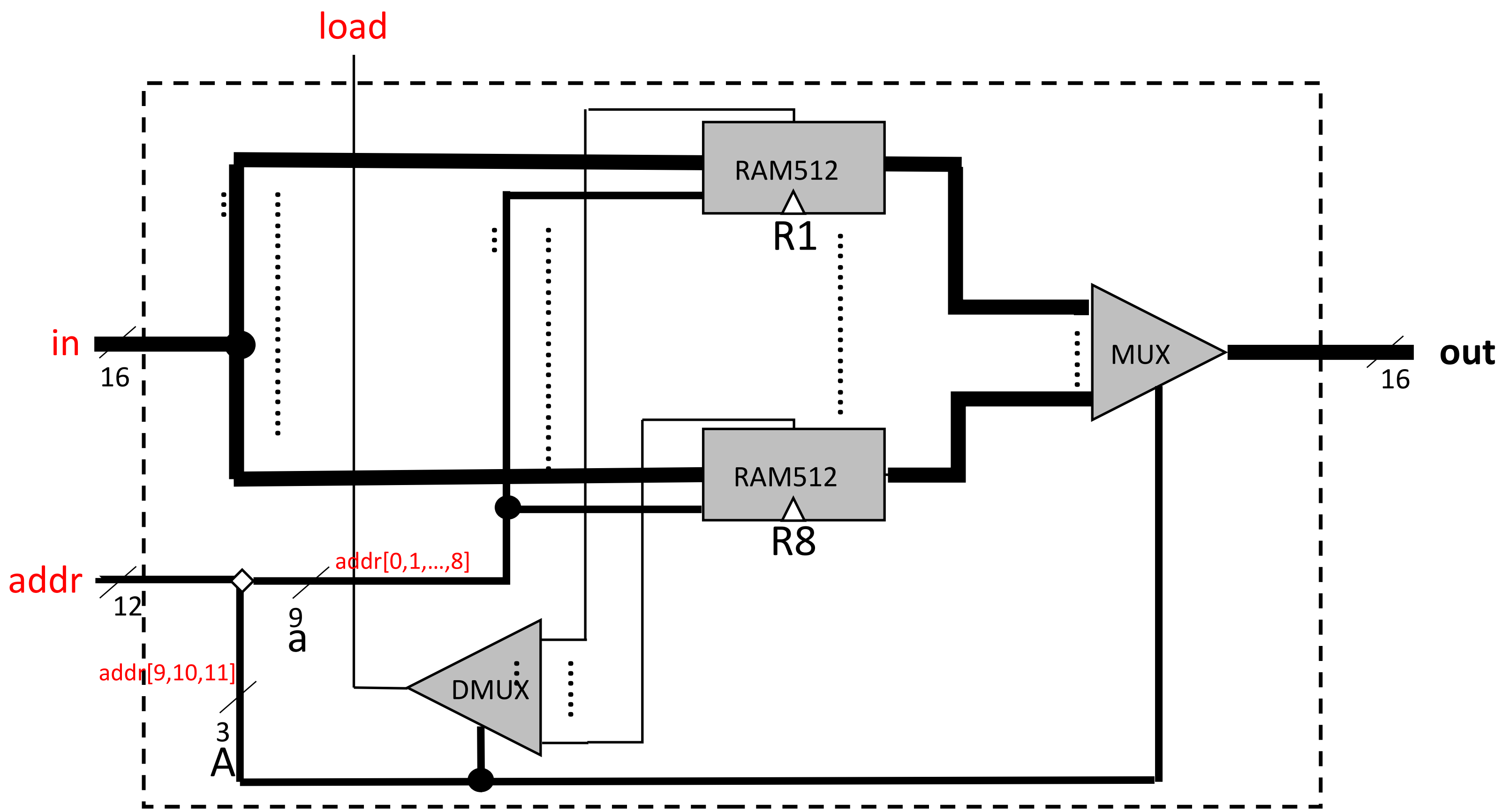


```
class ram512:
    ...
    def __init__(self):
        self.ram64s = tuple(ram64() for _ in range(ram512.TAILLE))
    def probe(self,addr):
        return op_mux8way16(addr[-ram512.A:],*(r.probe(addr[:-ram512.A]) for r in self.ram64s))
    def set(self,load,inp,addr):
        loads = op_dmux8way(addr[-ram512.A:],load)
        for l,r in zip(loads,self.ram64s):
            r.set(l,inp,addr[:-ram512.A])
```

RAM4K

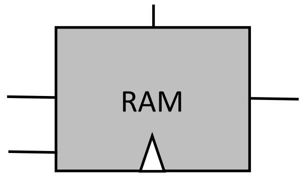


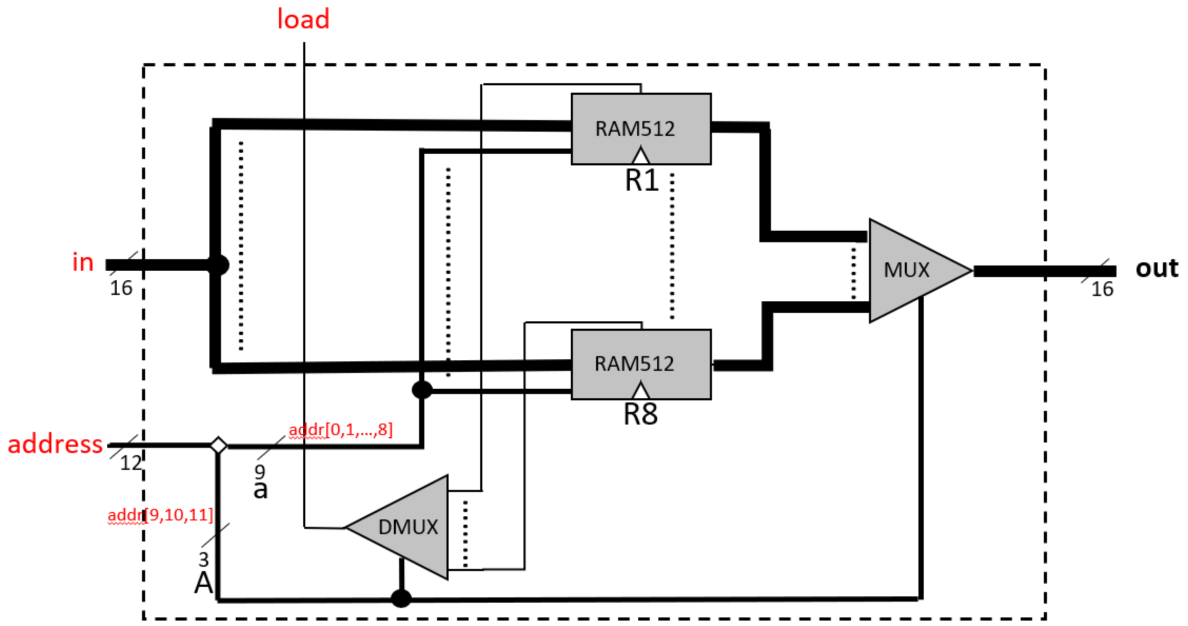
2399990 transistors



```
class ram4k:
    A,a=3,9
    TAILLE=2**A
    ...
```

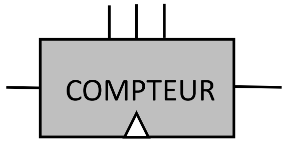
RAM4K



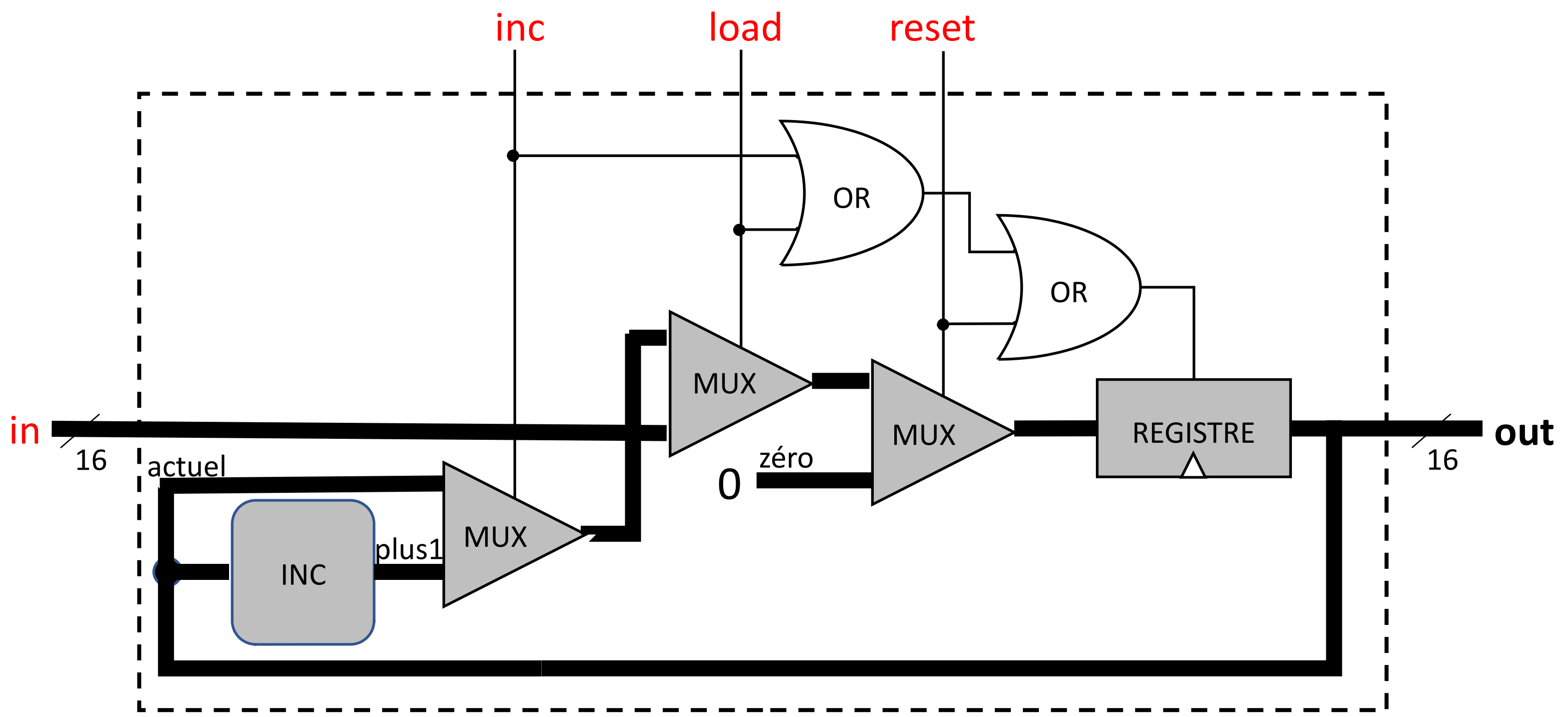


```
class ram4k:
    ...
    def __init__(self):
        self.ram512s = tuple(ram512() for _ in range(ram4k.TAILLE))
    def probe(self,addr):
        return op_mux8way16(addr[-ram4k.A:],*(r.probe(addr[:-ram4k.A]) for r in self.ram512s))
    def set(self,load,inp,addr):
        loads = op_dmux8way(addr[-ram4k.A:],load)
        for l,r in zip(loads,self.ram512s):
            r.set(l,inp,addr[:-ram4k.A])
```

COMPTEUR



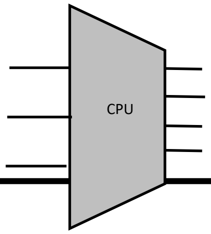
1452 transistors



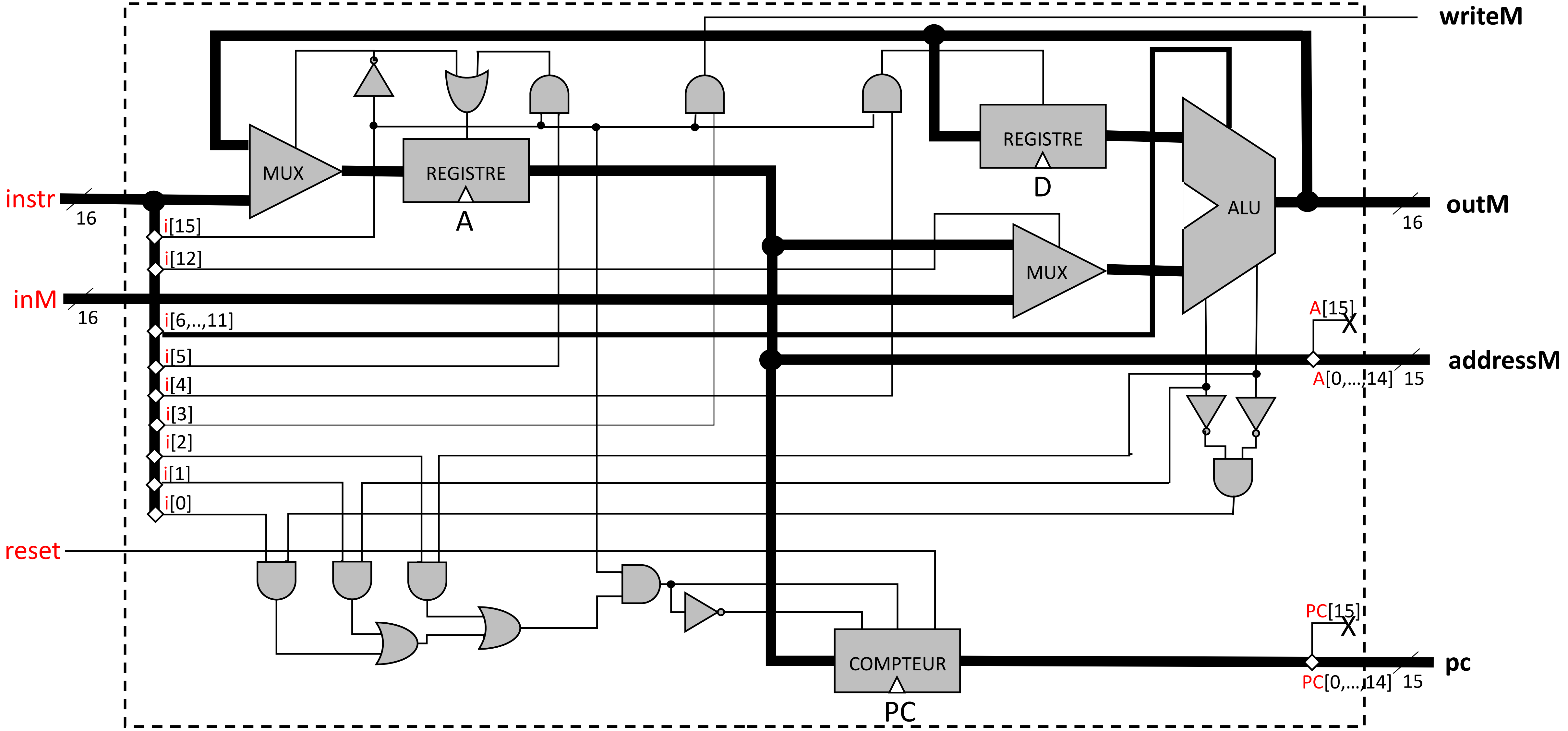
out(t+1) = 0 if reset(t) else
in(t) if load(t) else
out(t)+1 if inc(t) else
out(t)

```
class compteur(registre):
    def set(self,reset,load,inc,inp):
        actuel = self.probe()
        plus1 = op_inc16(actuel)
        zéro = (0,)*len(actuel)
        super().set(op_or(reset,op_or(load,inc)),
            op_mux16(reset,op_mux16(load,op_mux16(inc,actuel,plus1),inp),zéro))
```

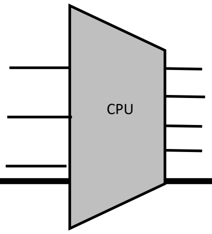
CPU

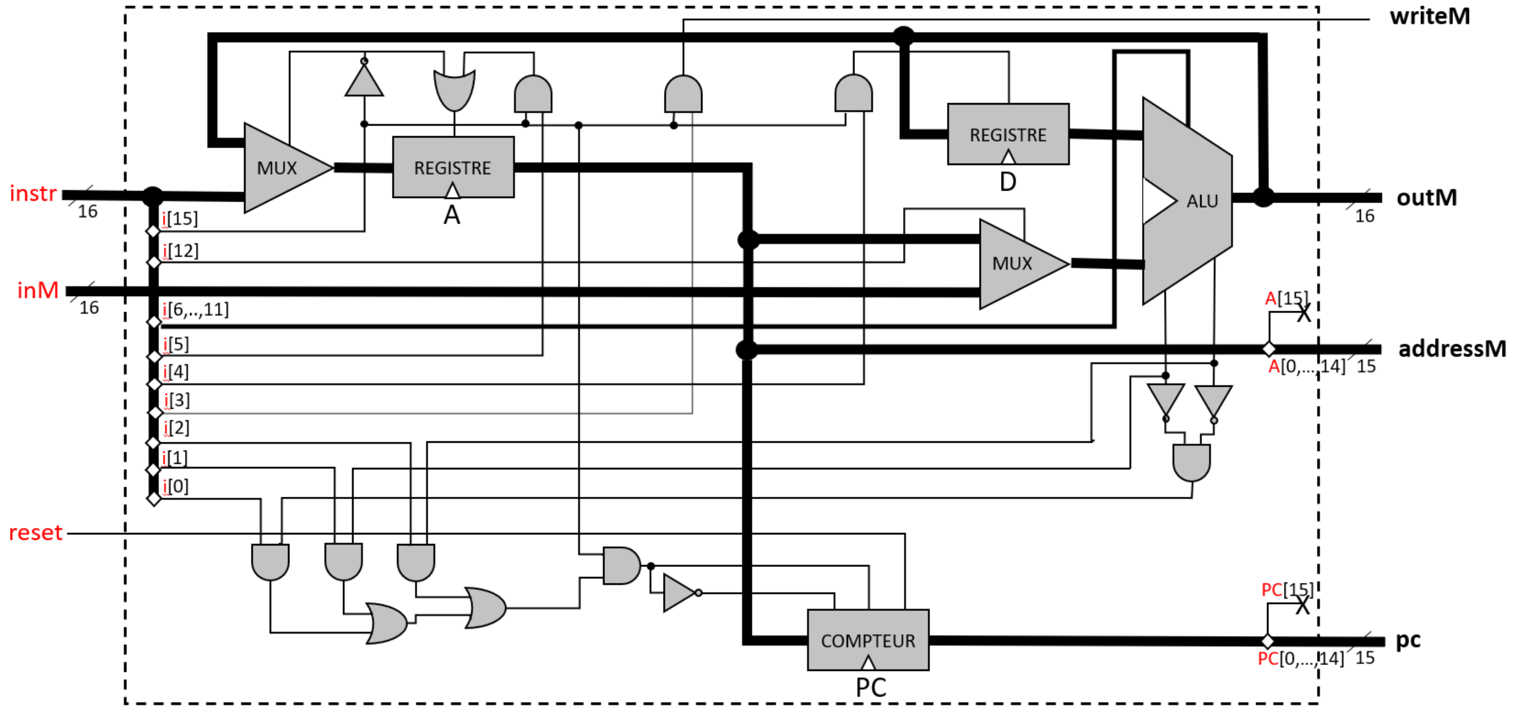


5510 transistors



CPU



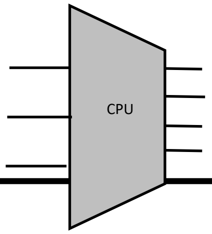


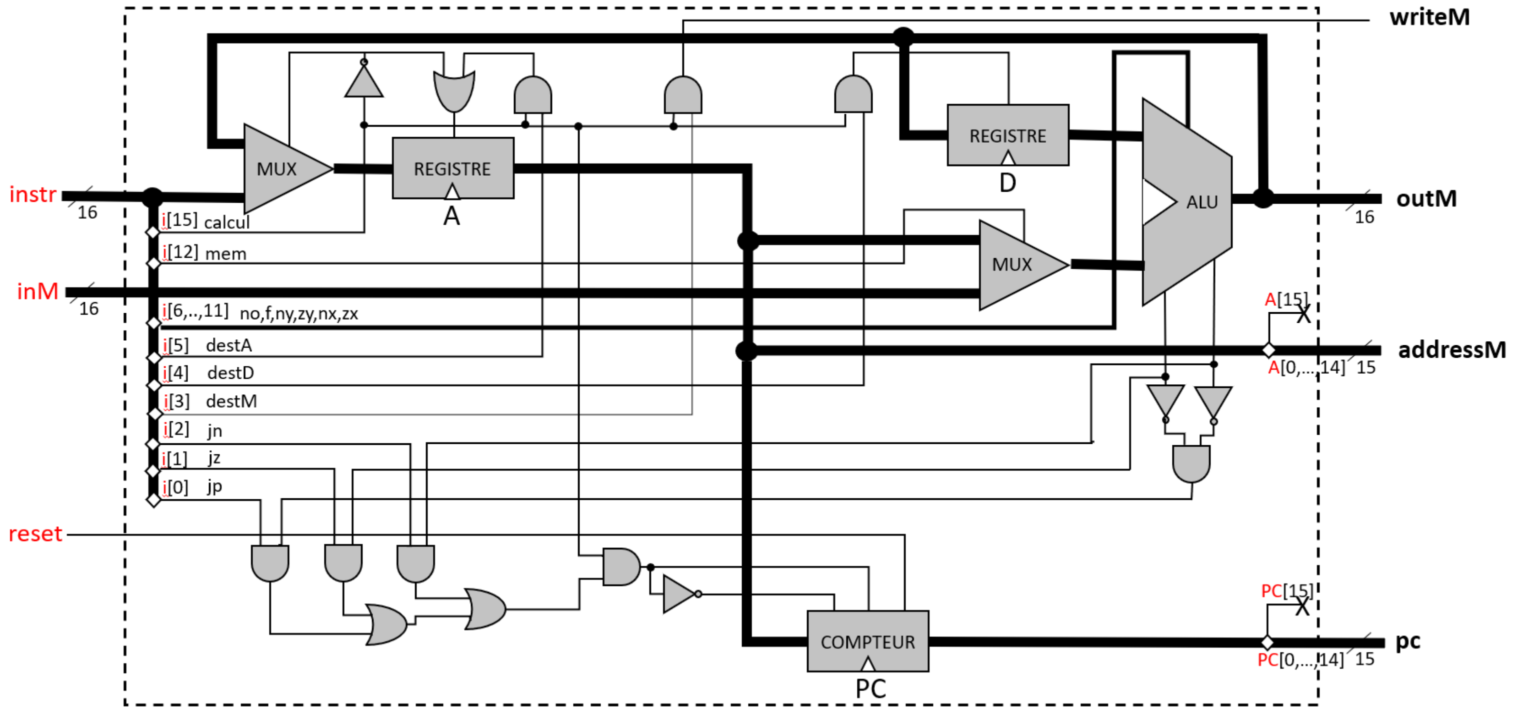
Registre A pour adresses
Registre D pour données
ALU pour calculs
Compteur PC pour instructions

Instructions encodées pour faire une operation ALU, ranger dans A/D/M et changer PC

```
class CPU:
    def __init__(self):
        self.A = registre()
        self.D = registre()
        self.PC = compteur()
    ...
```

CPU

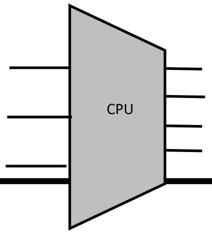


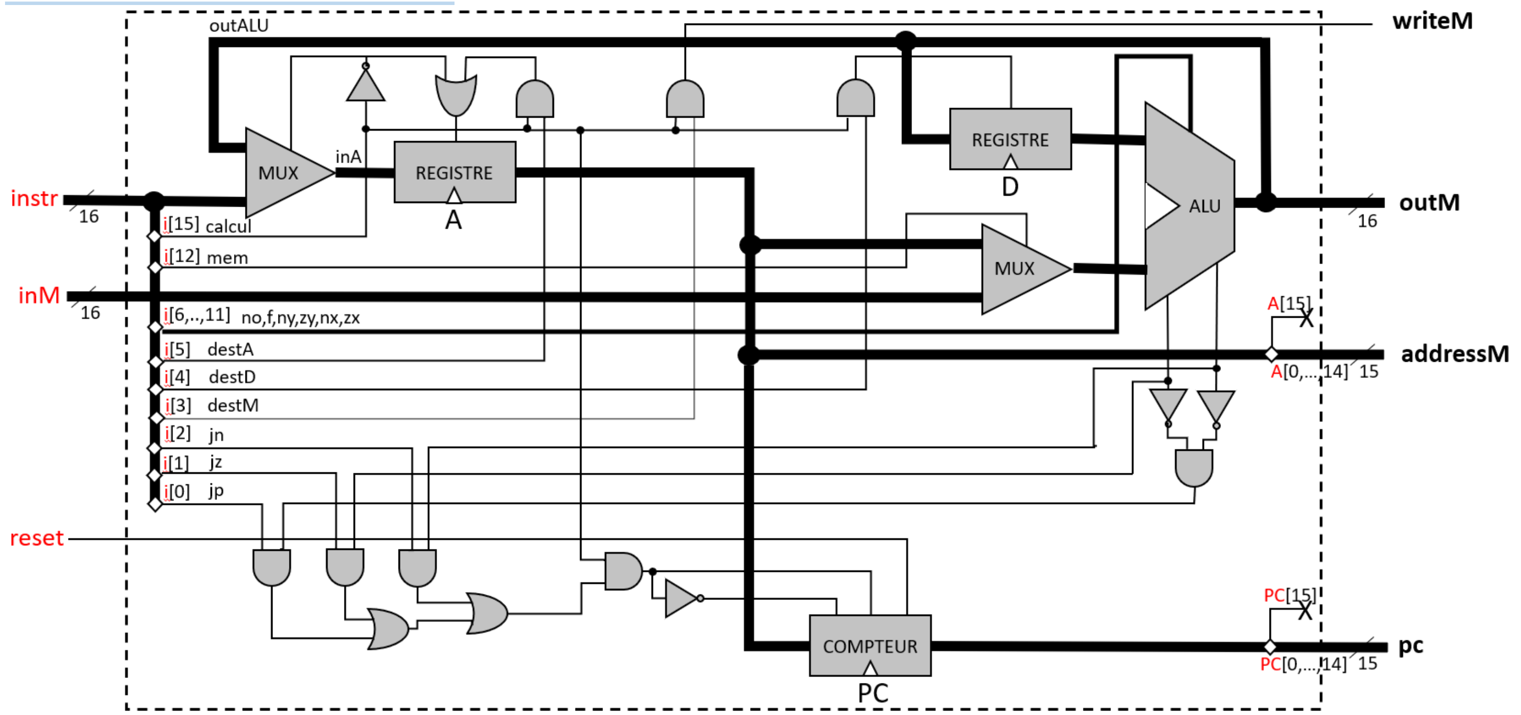


Décodage d'instruction:
0..2: condition de saut
3..5: destination de rangement du calcul
6..11: commande d'ALU
12: choix A/M pour donnée Y de l'ALU
15: selection instruction A/C

```
class CPU:
    ...
    def exec(self,instr,inM,reset):
        jp,jz,jn,destM,destD,destA,no,f,ny,zy,nx,zx,mem,_,_,calcul = instr
    ...
```

CPU

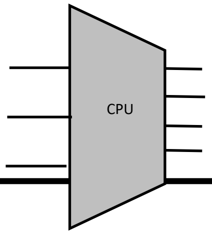


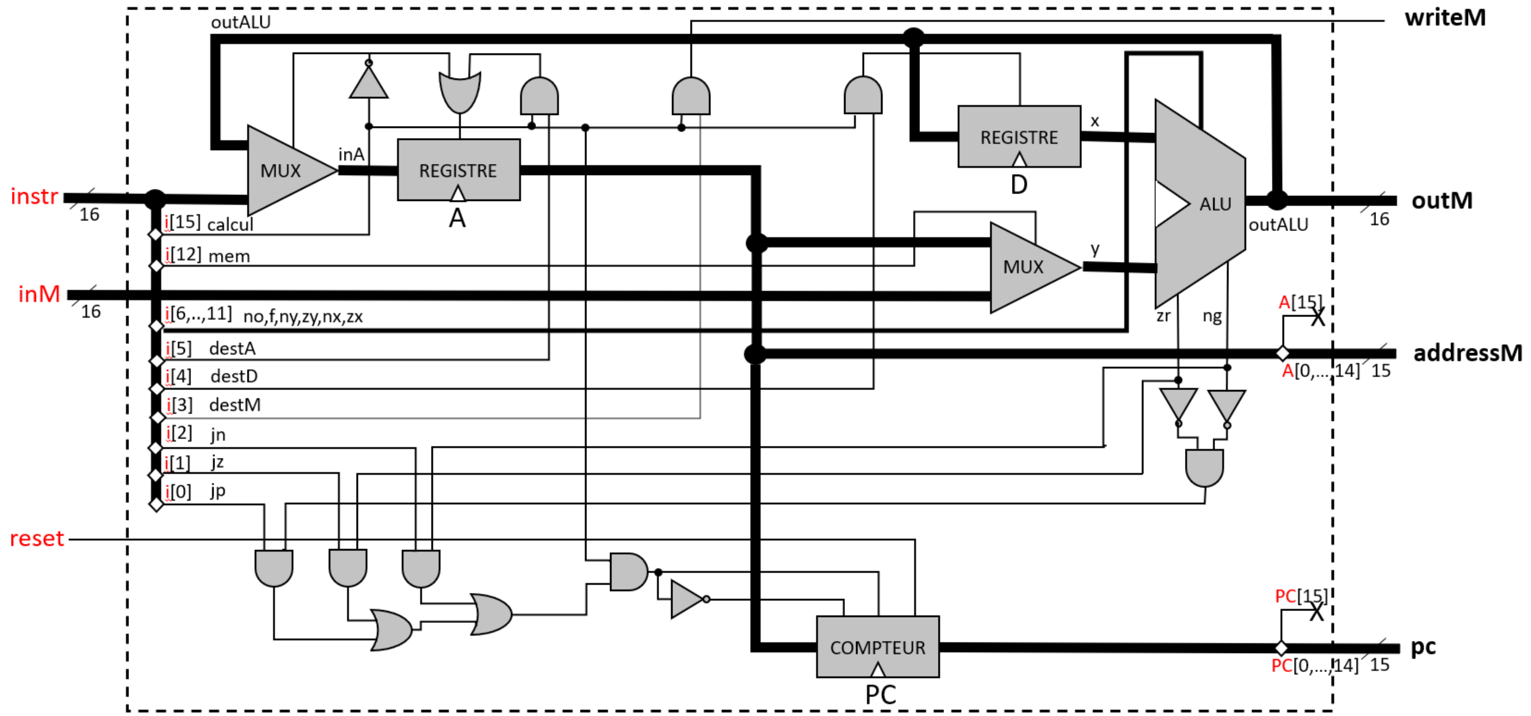


Instruction A:
range instr dans A

```
class CPU:
    ...
        inconnu = (0,)*len(inM) # inconnu mais sans importance
        outALU = inconnu
        inA = op_mux16(op_not(calcul),outALU,instr)
        self.A.set(op_not(calcul),inA)
    ...
```

CPU





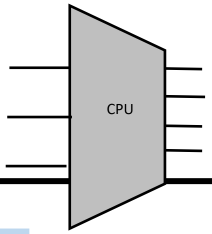
Instruction C:

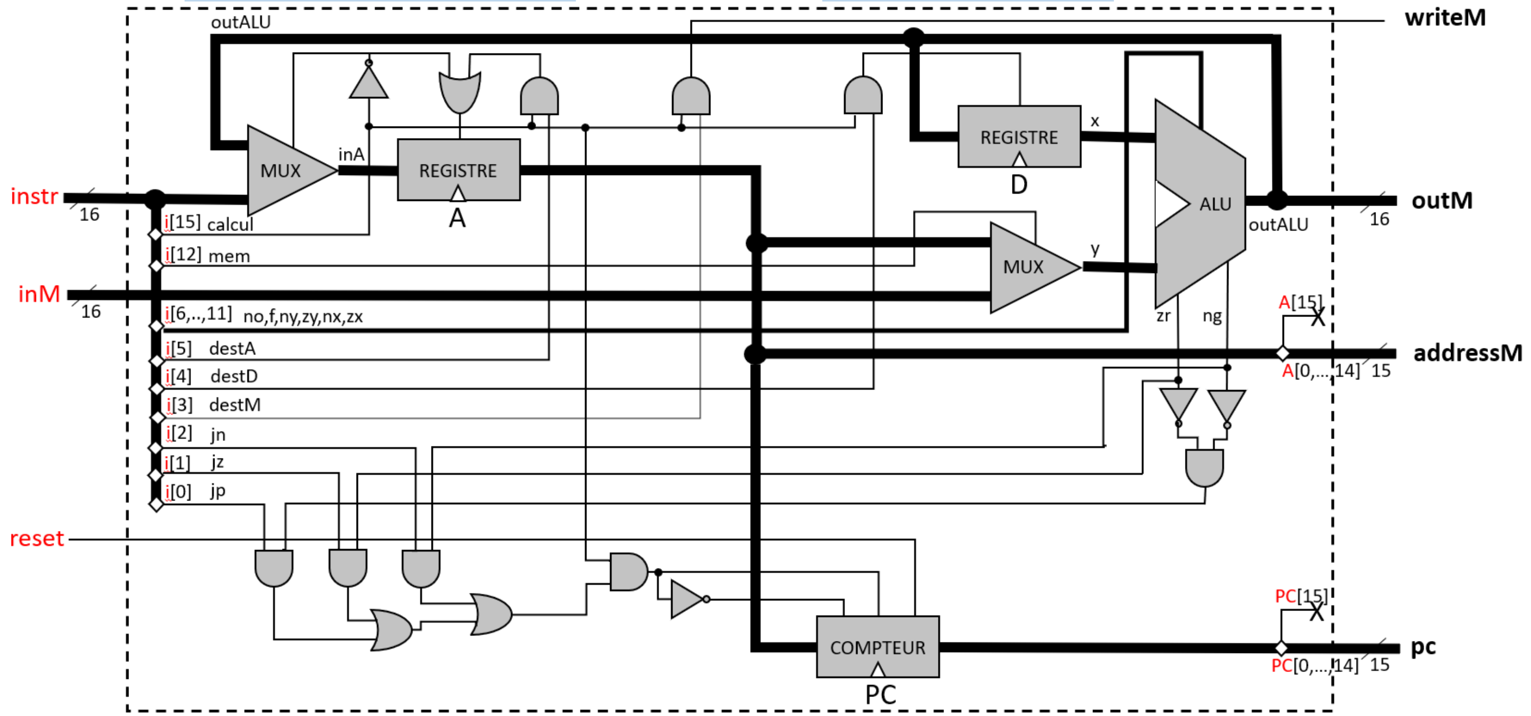
ALU calcule avec

- D
- inM si mem sinon A

```
class CPU:
    ...
        x = self.D.probe()
        y = op_mux16(mem,self.A.probe(),inM)
        outALU,zr,ng = ALU(x,y,zx,nx,zy,ny,f,no)
    ...
```

CPU

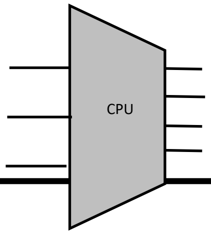


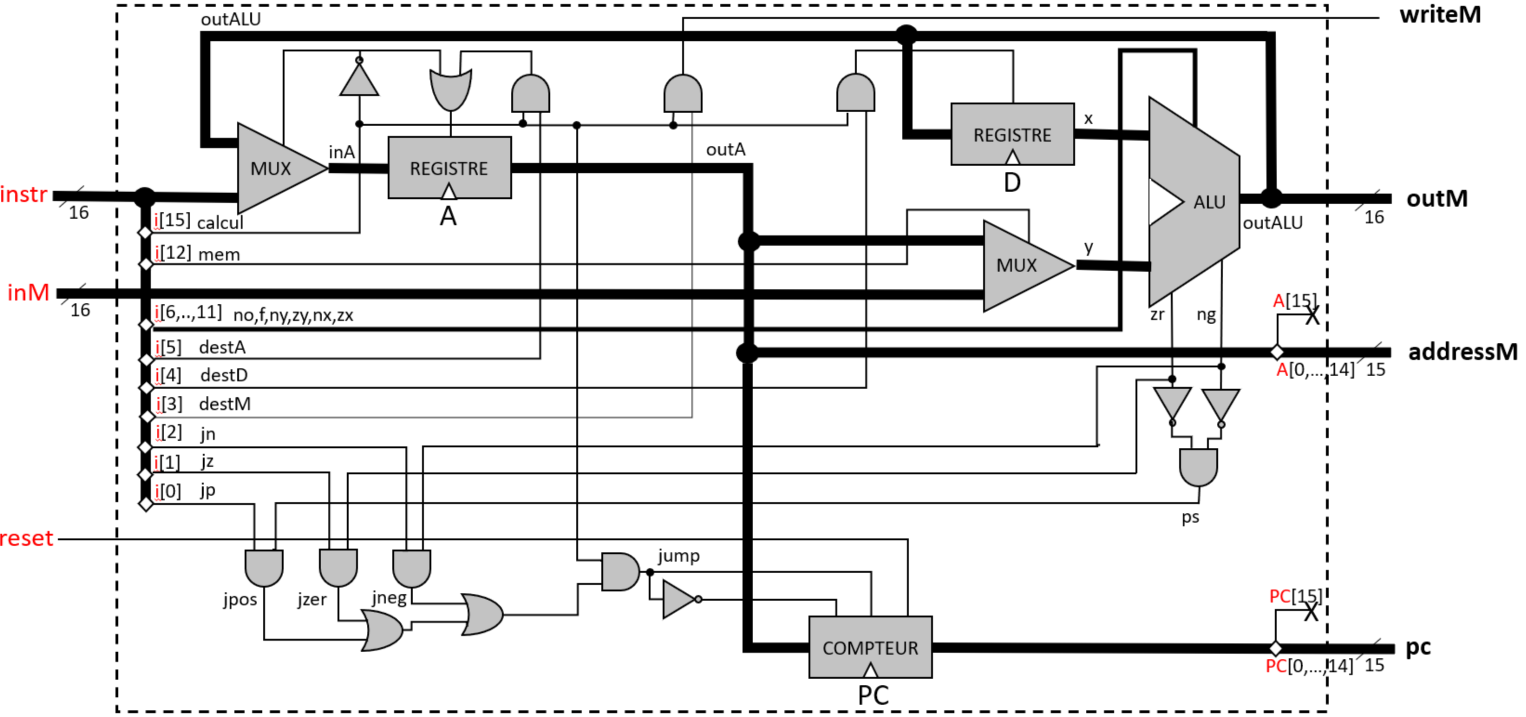


Instruction C:
résultat va dans
- D si destD
- A si destA

```
class CPU:
    ...
        inA = op_mux16(op_not(calcul),outALU,instr)
        self.A.set(op_and(calcul,destA),inA)
        self.D.set(op_and(calcul,destD),outALU)
    ...
```

CPU





Instruction C:

saut si résultat satisfait

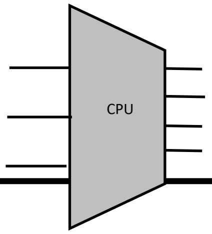
- zr et jz
- ng et jn
- ps et jp

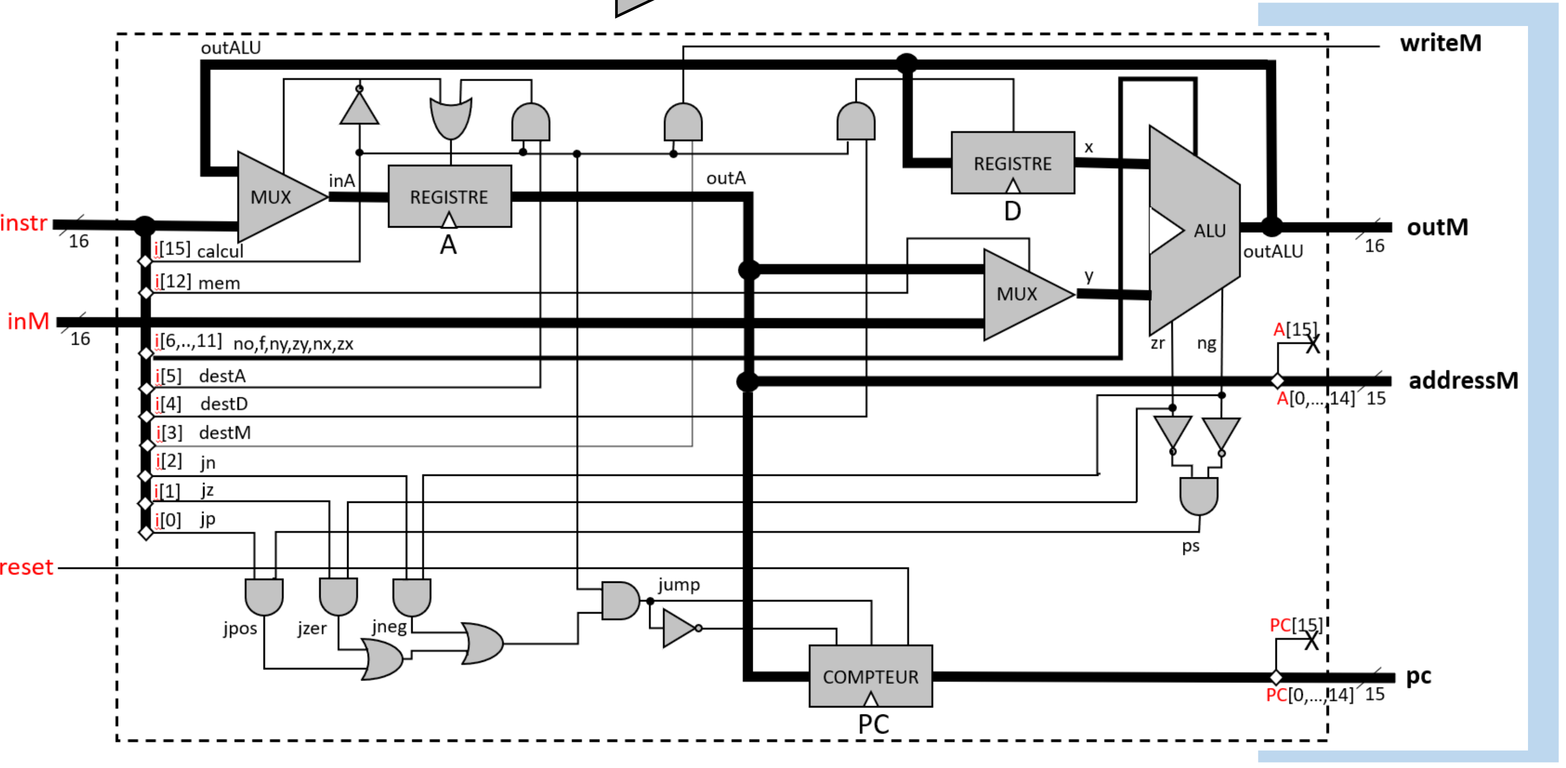
inc sinon

reset est prioritaire

```
class CPU:
    ...
        outA = self.A.probe()
        ps = op_and(op_not(ng),op_not(zr))
        jpos,jzer,jneg = op_and(jp,ps),op_and(jz,zr),op_and(jn,ng)
        jump = op_and(calcul,op_or(jzer,op_or(jpos,jneg)))
        self.PC.set(reset,jump,op_not(jump),outA)
    ...
```

CPU



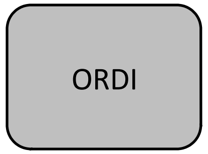


Instruction C:
  writeM si destM

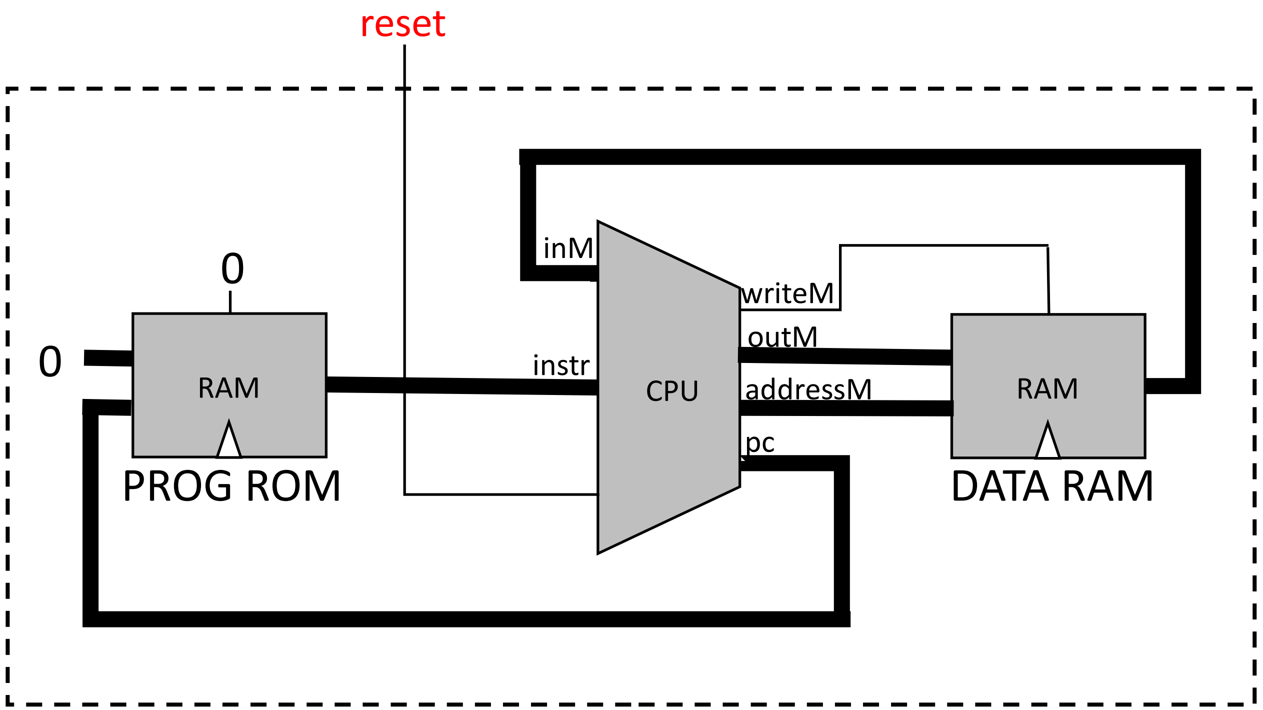
outM est outALU
addressM est A
pc est PC

```
class CPU:
    ...
        outM = outALU
        writeM = op_and(calcul,destM)
        addressM = outA[:-1]
        pc = self.PC.probe()[:-1]
        return writeM,outM,addressM,pc
```

ORDI



38409074 transistors



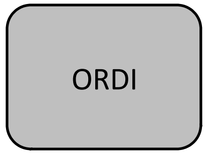
PROG ROM pour programme
DATA RAM pour données
CPU pour traitement

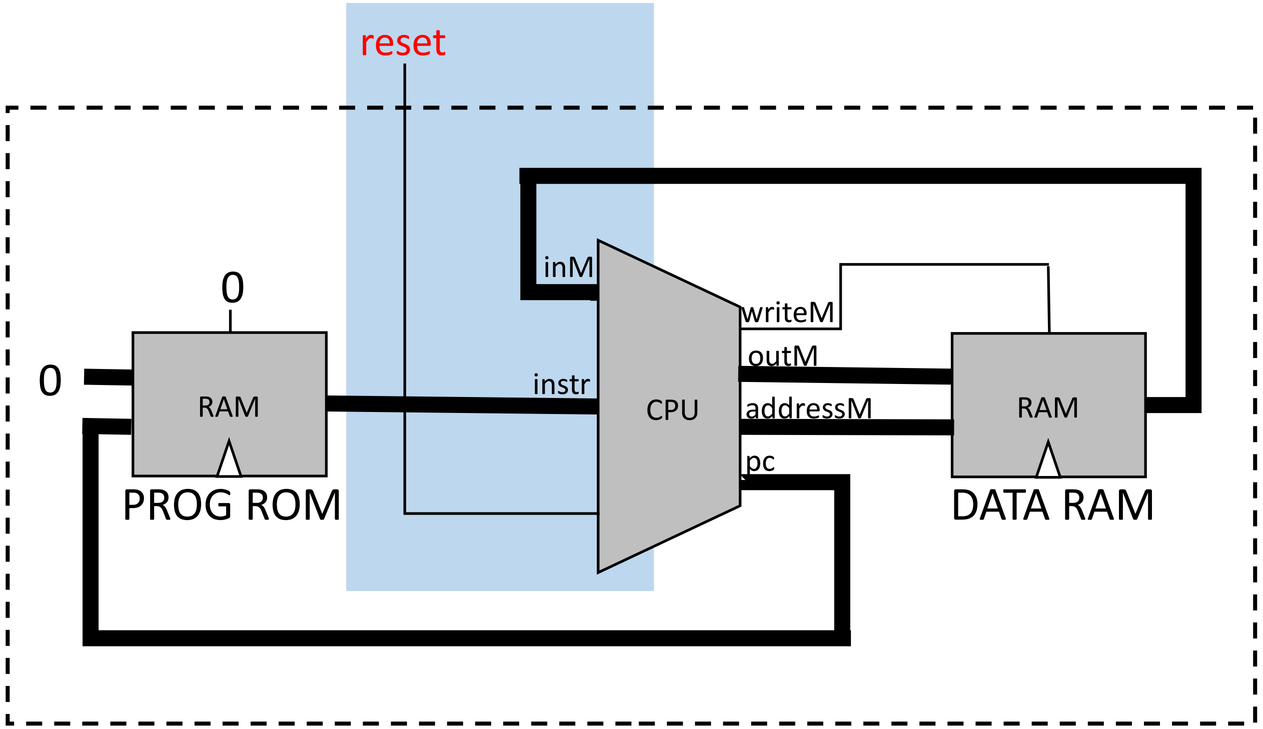
Boucle infinie:

- Entrées en CPU
- Traitement
- Sorties de CPU

```
class ordinateur():
    def __init__(self):
        self.CPU = CPU()
        self.RAM = ram()
        self.ROM = ram()
    def __str__(self):
        return “CPU:”+str(self.CPU)+”\nROM:”+self.ROM.dump()+”\nRAM:”+self.RAM.dump()
    ...
```

ORDI

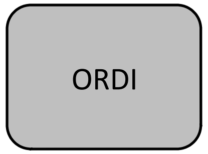


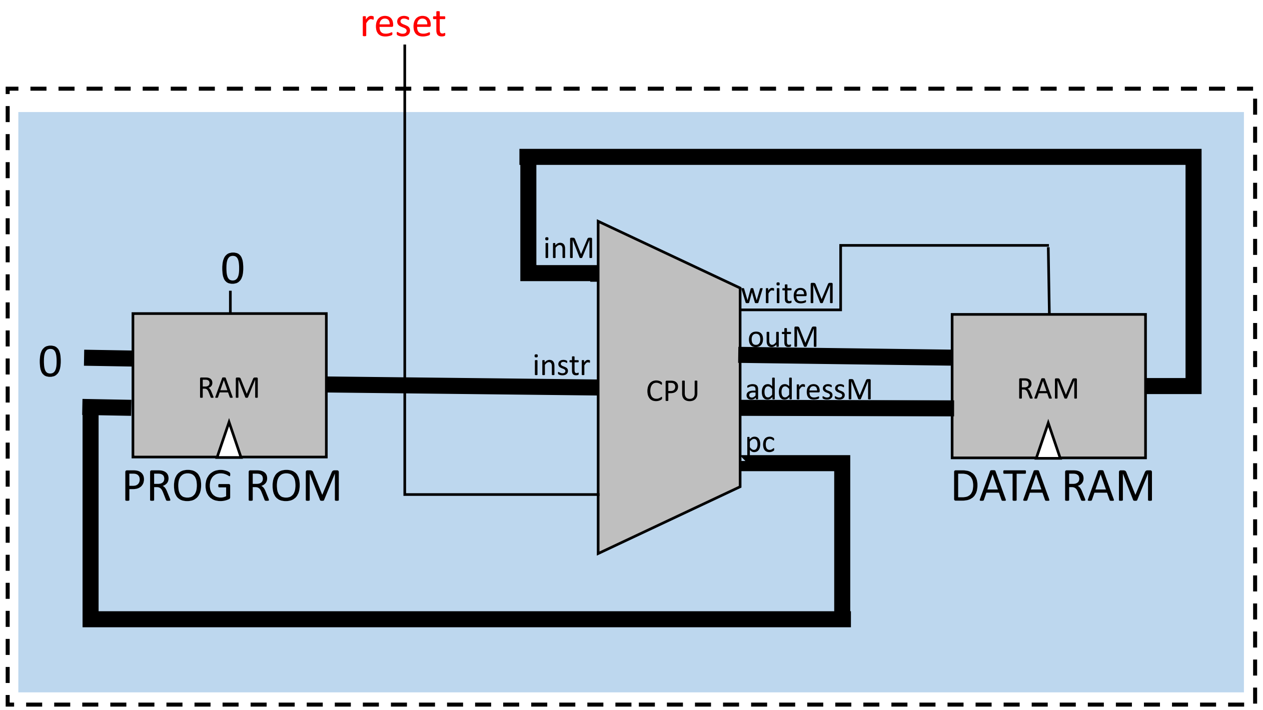


Reset pour faire repartir le CPU de l’instruction à l’adresse 0

```
class ordinateur():
    ...
    def reset(self):
        inconnu = (0,)*16 # inconnu mais sans importance
        inM, instr, reset = inconnu, inconnu, 1
    ...
```

ORDI

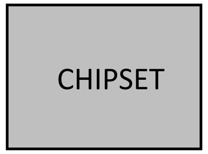




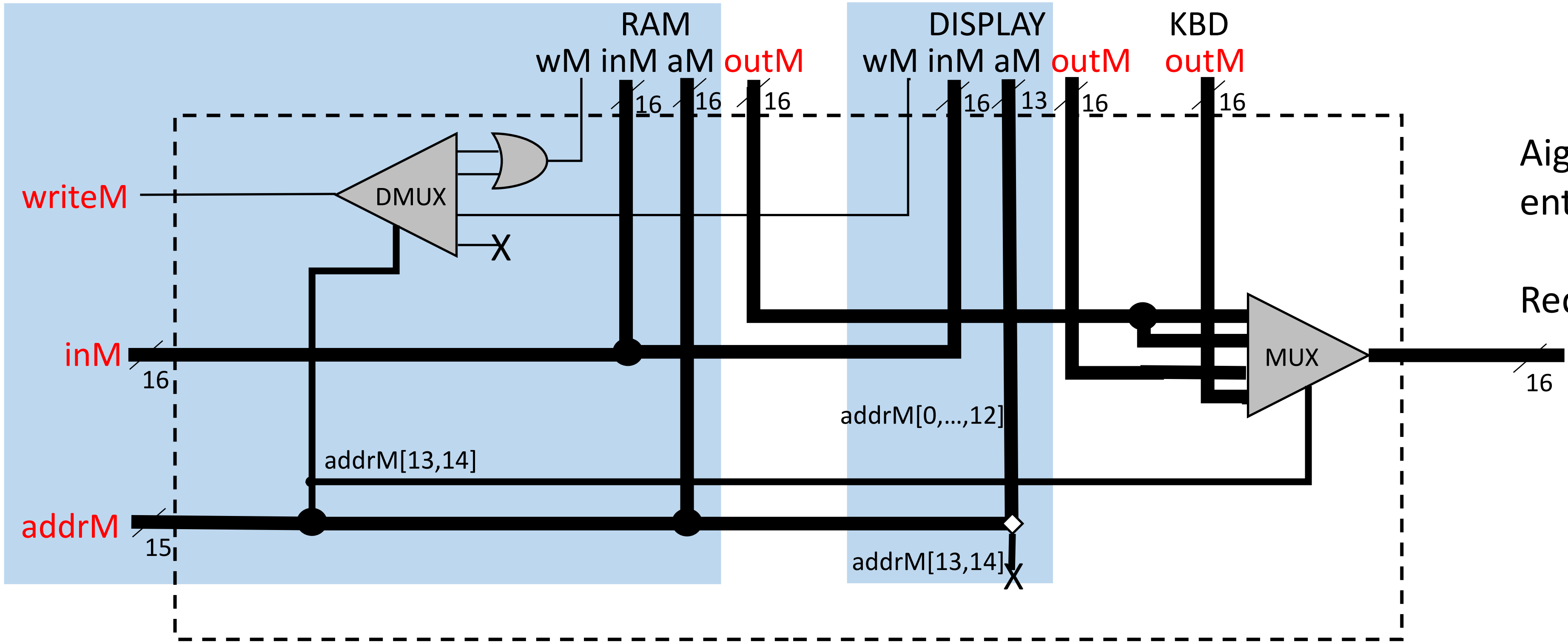
Boucle infinie:
- Traitement
- Écriture en mémoire des sorties de CPU
- Lecture de mémoire pour entrées en CPU

```
class ordinateur():
    ...
        while True:
            writeM, outM, addressM, pc = self.CPU.exec(instr,inM,reset)
            self.RAM.set(writeM,outM,addressM)
            inM, instr, reset = self.RAM.probe(addressM), self.ROM.probe(pc), 0
```

CHIPSET



798 transistors

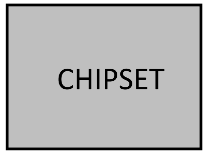


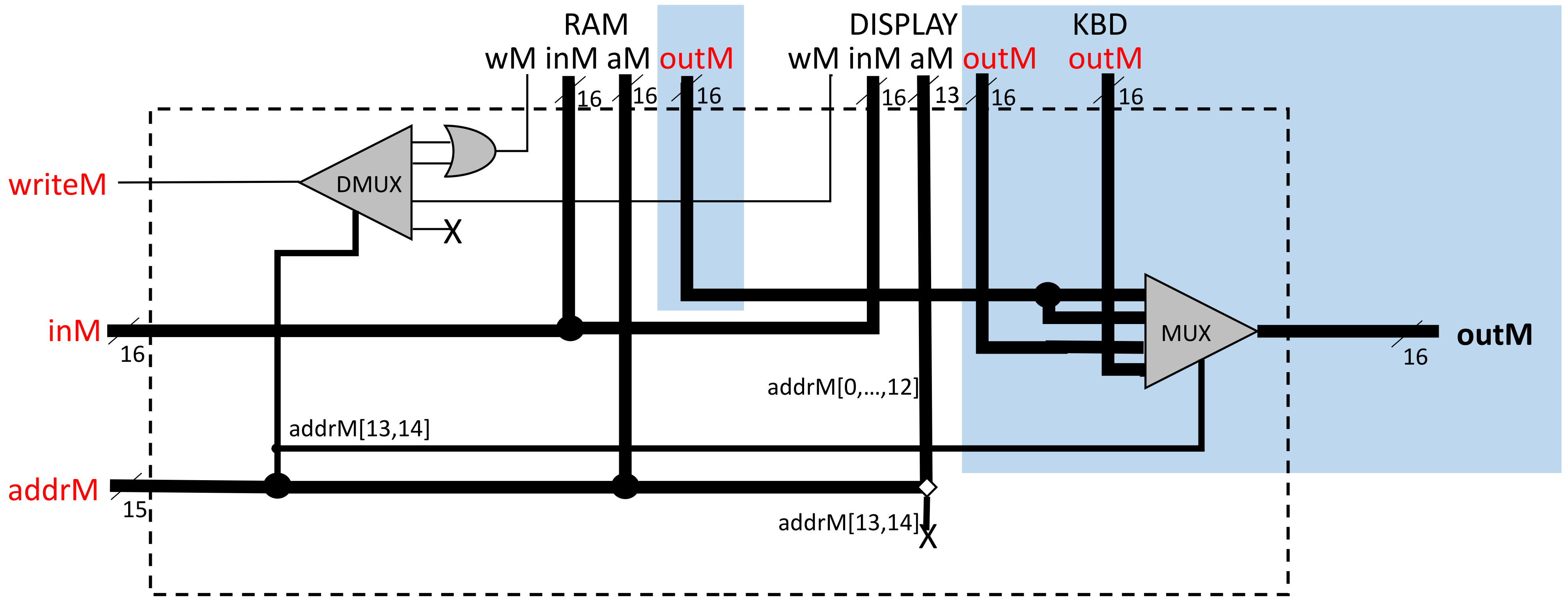
Aiguilleur en function d'adresse entre Mémoire/Écran/Clavier

Redirige entrée

```
def chipsetIN(writeM,inM,addrM):
    sel = addrM[-2:]
    RAMwriteM0, RAMwriteM1, DISPwriteM, _ = op_dmux4way(sel,writeM)
    RAMwriteM = op_or(RAMwriteM0,RAMwriteM1)
    return RAMwriteM, inM, addrM, DISPwriteM, inM, addrM[:-2]
```

CHIPSET

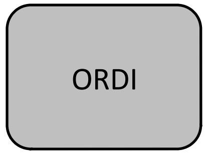


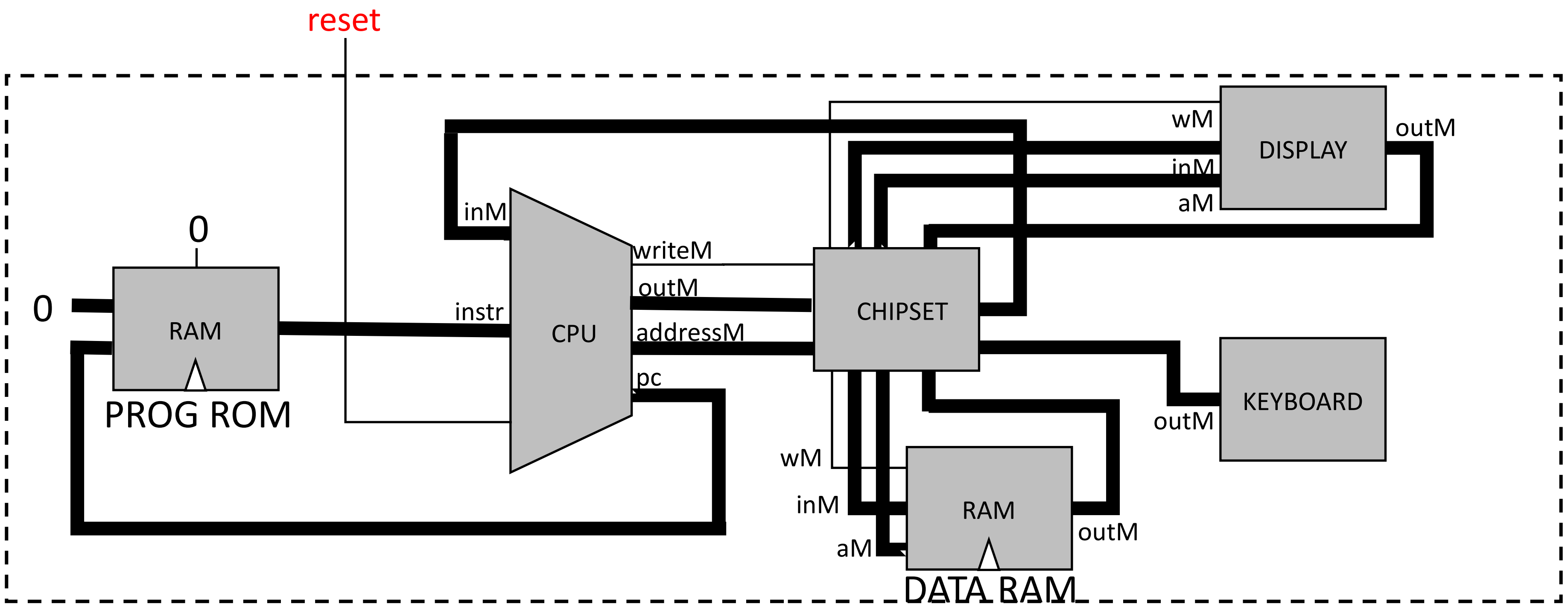


Redirige sortie

```
def chipsetOUT(addrM,RAMoutM,DISPoutM,KBDoutM):
    sel = addrM[-2:]
    return op_mux4way16(sel,RAMoutM,RAMoutM,DISPoutM,KBDoutM)
```

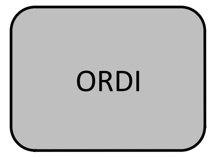
ORDI IO

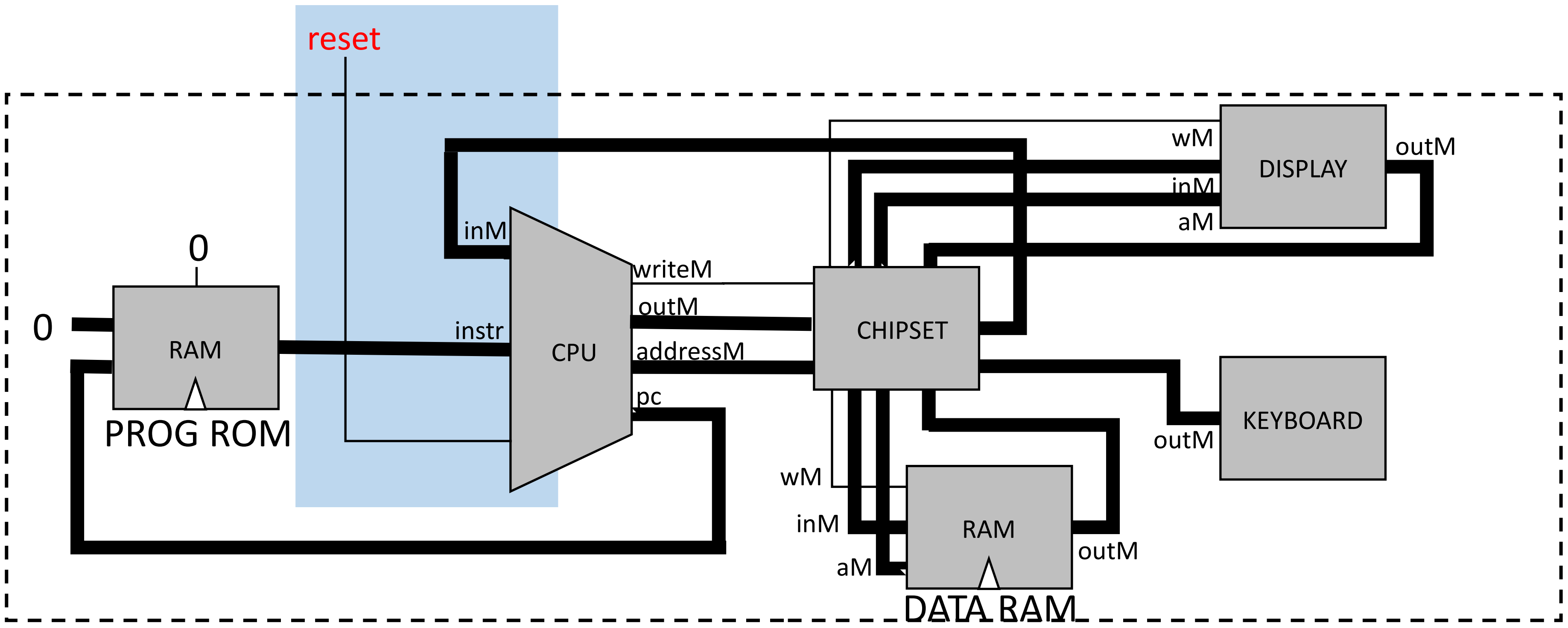




```
class ordinateurIO(ordinateur):
    def __init__(self):
        super().__init__()
        self.IO = IO() # DISPLAY et KEYBOARD
    ...
```

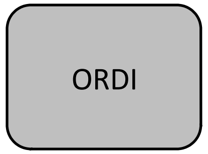
ORDI IO

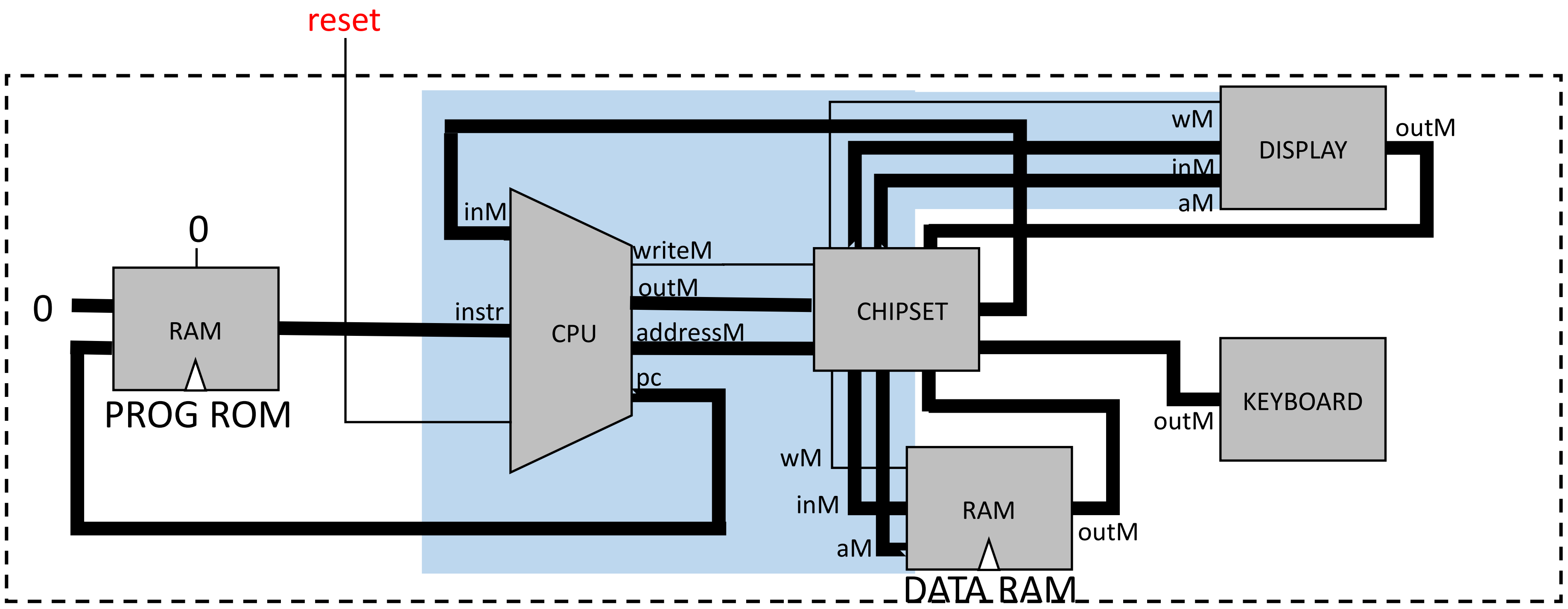




```python
class ordinateurIO(ordinateur):
    ...
    def reset(self):
        inconnu = (0,)*16 # inconnu mais sans importance
        inM, instr, reset = inconnu, inconnu, 1
    ...
```

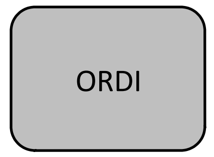
ORDI IO

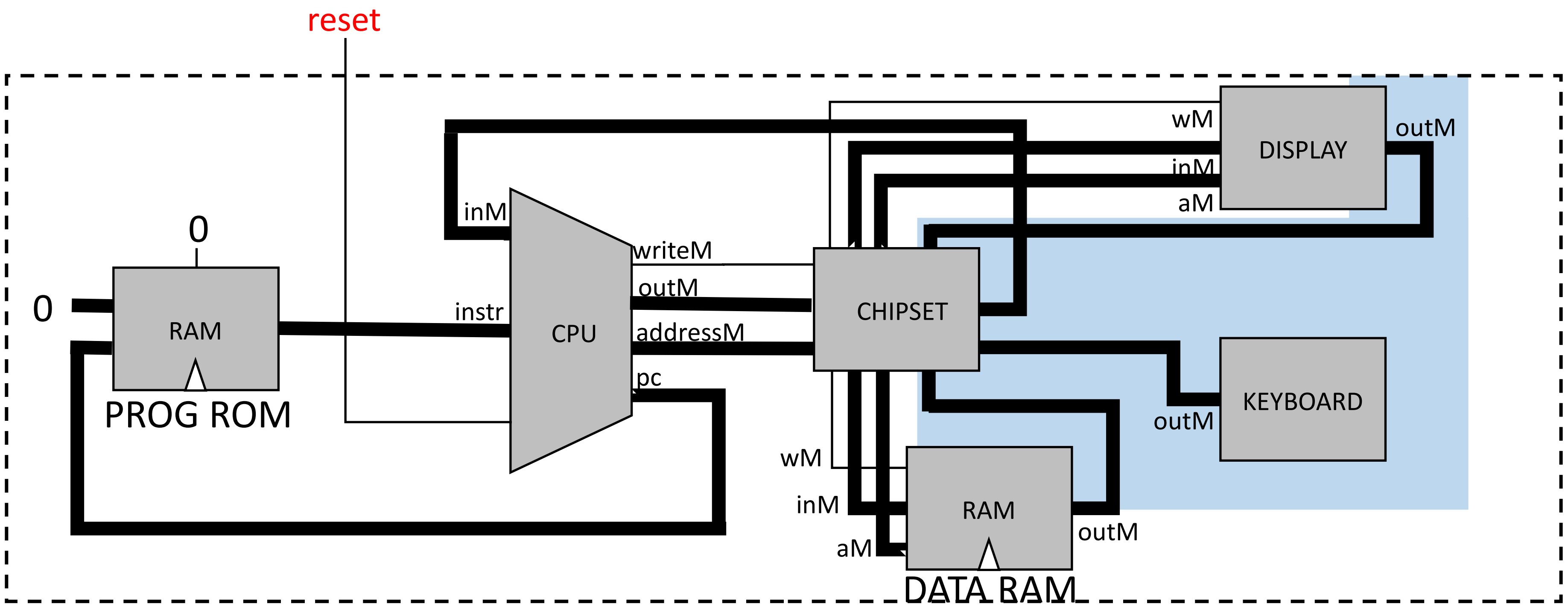




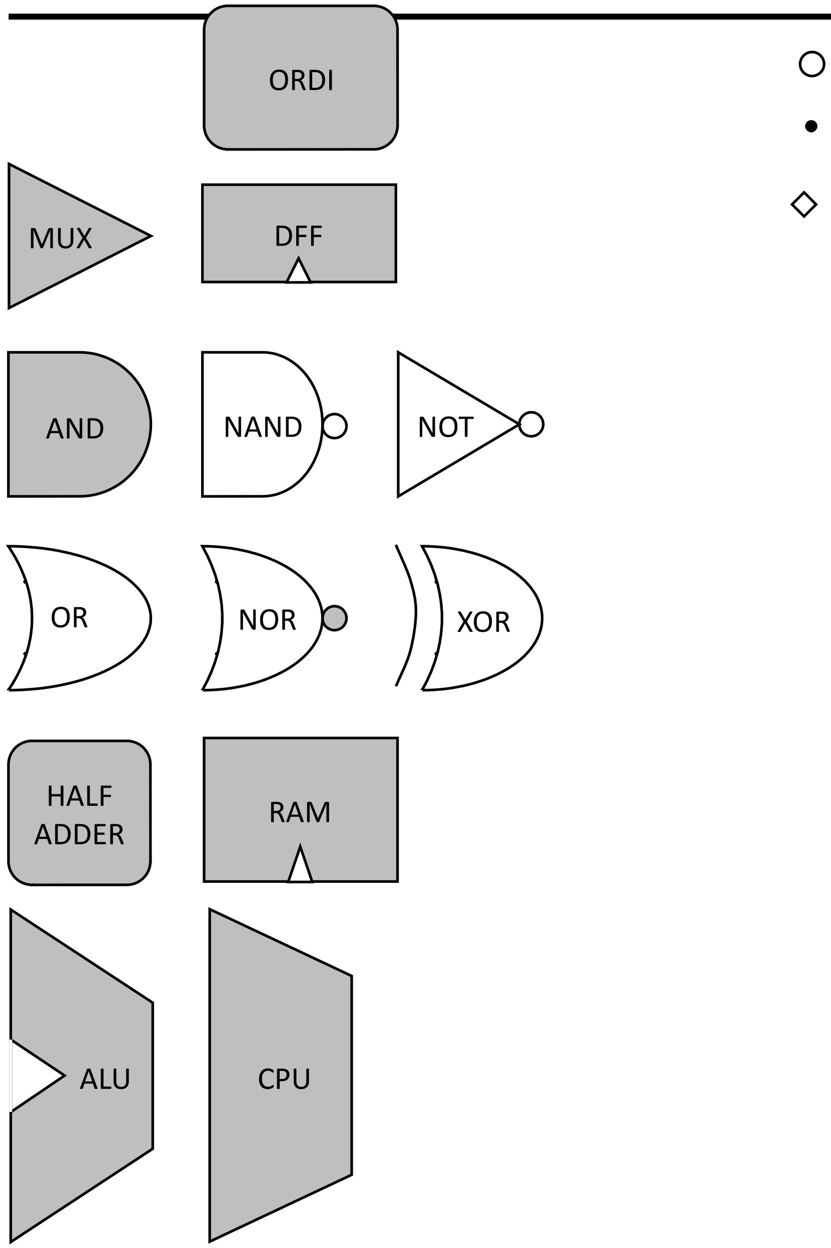
```
class ordinateurIO(ordinateur):
    ...
        while True:
            writeM, outM, addressM, pc = self.CPU.exec(instr,inM,reset)
            RAMwM, RAMinM, RAMaM, DISPwM, DISPinM, DISPaM = chipsetIN(writeM,outM,address)
            self.RAM.set(RAMwM,RAMinM,RAMaM)
            self.IO.set(DISPwM,DISPinM,DISPaM)

    ...
```

ORDI IO





```
class ordinateurIO(ordinateur):
    ...
            inM = chipsetOUT(addressM,self.RAM.probe(RAMaM),self.IO.probe(DISPaM),self.IO.key())
            instr = self.ROM.probe(pc)
            reset = 0
```


ORDI
MUX
DFF
AND
NAND
NOT
OR
NOR
XOR
HALF ADDER
RAM
ALU
CPU