MC404 - Organição Básica de Computadores

Resumo Teórico

23 de agosto de 2021

Guilherme Nunes Trofino
217276

Conteúdo

Instruções Comandos necessários para que o simulador seja aberto a partir do terminal:

```
cd classes/mc404/nand2tetris/tools/
chmod 755 HardwareSimulator.sh
./HardwareSimulator.sh

chmod 755 CPUEmulator.sh
./CPUEmulator.sh
```

1. Projeto 1

1.1. Not

Definição [Funcionamento]

Tabela Verdade Esta porta lógica possuirá a seguinte tabela verdade:

in	out
0	1
1	0

Tabela 1.1: Tabela Verdade Not

Representação Esta porta lógica pode ser expressa pelo seguinte circuito:

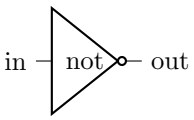

Figura 1.1: Porta Lógica Not

Implementação Esta porta lógica pode ser implementada de acordo com o seguinte código:

```
// 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/Not.hdl

/**
 * Not gate:
 * out = not in
 */

CHIP Not {
    IN in;
    OUT out;

    PARTS:
    Nand (a=in, b=in, out=out);
    //author: tr0fin0
}
```

1.2. And

Definição [Funcionamento]

Tabela Verdade Esta porta lógica possuirá a seguinte tabela verdade:

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

Tabela 1.2: Tabela Verdade And

Representação Esta porta lógica pode ser expressa pelo seguinte circuito:

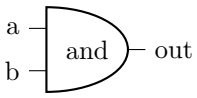

Figura 1.2: Porta Lógica And

Implementação Esta porta lógica pode ser implementada de acordo com o seguinte código:

```
// 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/And.hdl

/**
 * And gate:
 * out = 1 if (a == 1 and b == 1)
 *       0 otherwise
 */

CHIP And {
    IN a, b;
    OUT out;

    PARTS:
    Nand(a=a, b=b, out=outNand);
    Not(in=outNand, out=out);
    //author: tr0fin0
}
```

1.3. Or

Definição [Funcionamento]

Tabela Verdade Esta porta lógica possuirá a seguinte tabela verdade:

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

Tabela 1.3: Tabela Verdade Or

Representação Esta porta lógica pode ser expressa pelo seguinte circuito:

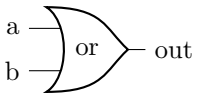

Figura 1.3: Porta Lógica Or

Implementação Esta porta lógica pode ser implementada de acordo com o seguinte código:

```
// 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/Or.hdl

 /**
 * Or gate:
 * out = 1 if (a == 1 or b == 1)
 *       0 otherwise
 */

CHIP Or {
    IN a, b;
    OUT out;

    PARTS:
    Not(in=a, out=notA);
    Not(in=b, out=notB);
    Nand(a=notA, b=notB, out=out);
    //author: tr0fin0
}
```

1.4. Xor

Definição [Funcionamento]

Tabela Verdade Esta porta lógica possuirá a seguinte tabela verdade:

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

Tabela 1.4: Tabela Verdade Xor

Representação Esta porta lógica pode ser expressa pelo seguinte circuito:

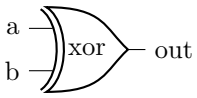

Figura 1.4: Porta Lógica Xor

Implementação Esta porta lógica pode ser implementada de acordo com o seguinte código:

```
// 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/Xor.hdl

/**
 * Exclusive-or gate:
 * out = not (a == b)
 */

CHIP Xor {
    IN a, b;
    OUT out;

    PARTS:
    Not(in=a, out=notA);
    Not(in=b, out=notB);
    And(a=a,    b=notB, out=aAndnotB);
    And(a=notA, b=b,    out=notAAndB);
    Or(a=aAndnotB, b=notAAndB, out=out);
    //author: tr0fin0
}
```

1.5. Mux

Definição [Funcionamento]

Tabela Verdade Esta porta lógica possuirá a seguinte tabela verdade:

s	a	b	out
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

Tabela 1.5: Tabela Verdade Mux

Representação Esta porta lógica pode ser expressa pelo seguinte circuito:

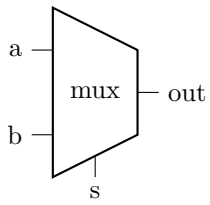

Figura 1.5: Porta Lógica Mux

Implementação Esta porta lógica pode ser implementada de acordo com o seguinte código:

```
// 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/Mux.hdl

/**
 * Multiplexor:
 * out = a if sel == 0
 *       b otherwise
 */

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

    PARTS:
    Not(in=sel, out=notSel);
    And(a=a,   b=notSel, out=aAndNotSel);
    And(a=sel, b=b,      out=selAndB);
    Or(a=aAndNotSel, b=selAndB, out=out);
    //author: tr0fin0
}
```

1.6. DMux

Definição [Funcionamento]

Tabela Verdade Esta porta lógica possuirá a seguinte tabela verdade:

s	in	a	b
0	0	0	0
0	1	1	0
1	0	0	0
1	1	0	1

Tabela 1.6: Tabela Verdade DMux

Representação Esta porta lógica pode ser expressa pelo seguinte circuito:

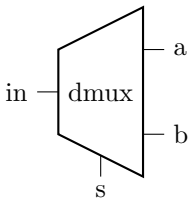

Figura 1.6: Porta Lógica DMux

Implementação Esta porta lógica pode ser implementada de acordo com o seguinte código:

```
// 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;

    PARTS:
    Not(in=sel, out=notSel);
    And(a=in, b=notSel, out=a);
    And(a=in, b=sel,    out=b);
    //author: tr0fin0
}
```

1.7. Not16

Definição [Funcionamento]

Tabela Verdade Esta porta lógica possuirá a seguinte tabela verdade:

in[i]	out[i]
0	1
1	0

Tabela 1.7: Tabela Verdade Not16

Representação Esta porta lógica pode ser expressa pelo seguinte circuito:

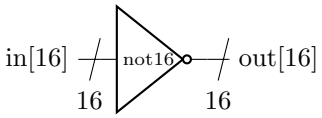

Figura 1.7: Porta Lógica Not16

Implementação Esta porta lógica pode ser implementada de acordo com o seguinte código:

```
// 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/Not16.hdl

/**
 * 16-bit Not:
 * for i=0..15: out[i] = not in[i]
 */

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

    PARTS:
    Not(in=in[0],  out=out[0]);
    Not(in=in[1],  out=out[1]);
    Not(in=in[2],  out=out[2]);
    Not(in=in[3],  out=out[3]);
    Not(in=in[4],  out=out[4]);
    Not(in=in[5],  out=out[5]);
    Not(in=in[6],  out=out[6]);
    Not(in=in[7],  out=out[7]);
    Not(in=in[8],  out=out[8]);
    Not(in=in[9],  out=out[9]);
    Not(in=in[10], out=out[10]);
    Not(in=in[11], out=out[11]);
    Not(in=in[12], out=out[12]);
    Not(in=in[13], out=out[13]);
    Not(in=in[14], out=out[14]);
    Not(in=in[15], out=out[15]);
    //author: tr0fin0
}
```

1.8. And16

Definição [Funcionamento]

Tabela Verdade Esta porta lógica possuirá a seguinte tabela verdade:

a[i]	b[i]	out[i]
0	0	0
0	1	0
1	0	0
1	1	1

Tabela 1.8: Tabela Verdade And16

Representação Esta porta lógica pode ser expressa pelo seguinte circuito:

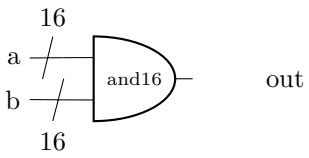

Figura 1.8: Porta Lógica And16

Implementação Esta porta lógica pode ser implementada de acordo com o seguinte código:

```
// 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/And16.hdl

/**
 * 16-bit bitwise And:
 * for i = 0..15: out[i] = (a[i] and b[i])
 */

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

    PARTS:
    And(a=a[0],  b=b[0],  out=out[0]);
    And(a=a[1],  b=b[1],  out=out[1]);
    And(a=a[2],  b=b[2],  out=out[2]);
    And(a=a[3],  b=b[3],  out=out[3]);
    And(a=a[4],  b=b[4],  out=out[4]);
    And(a=a[5],  b=b[5],  out=out[5]);
    And(a=a[6],  b=b[6],  out=out[6]);
    And(a=a[7],  b=b[7],  out=out[7]);
    And(a=a[8],  b=b[8],  out=out[8]);
    And(a=a[9],  b=b[9],  out=out[9]);
    And(a=a[10], b=b[10], out=out[10]);
    And(a=a[11], b=b[11], out=out[11]);
    And(a=a[12], b=b[12], out=out[12]);
    And(a=a[13], b=b[13], out=out[13]);
    And(a=a[14], b=b[14], out=out[14]);
    And(a=a[15], b=b[15], out=out[15]);
    //author: tr0fin0
}
```

1.9. OR16

Definição [Funcionamento]

Tabela Verdade Esta porta lógica possuirá a seguinte tabela verdade:

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

Tabela 1.9: Tabela Verdade OR16

Representação Esta porta lógica pode ser expressa pelo seguinte circuito:

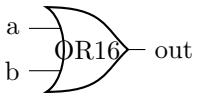

Figura 1.9: Porta Lógica OR16

Implementação Esta porta lógica pode ser implementada de acordo com o seguinte código:

```
// 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/Or16.hdl

/**
 * 16-bit bitwise Or:
 * for i = 0..15 out[i] = (a[i] or b[i])
 */

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

    PARTS:
    Or(a=a[0],  b=b[0],  out=out[0]);
    Or(a=a[1],  b=b[1],  out=out[1]);
    Or(a=a[2],  b=b[2],  out=out[2]);
    Or(a=a[3],  b=b[3],  out=out[3]);
    Or(a=a[4],  b=b[4],  out=out[4]);
    Or(a=a[5],  b=b[5],  out=out[5]);
    Or(a=a[6],  b=b[6],  out=out[6]);
    Or(a=a[7],  b=b[7],  out=out[7]);
    Or(a=a[8],  b=b[8],  out=out[8]);
    Or(a=a[9],  b=b[9],  out=out[9]);
    Or(a=a[10], b=b[10], out=out[10]);
    Or(a=a[11], b=b[11], out=out[11]);
    Or(a=a[12], b=b[12], out=out[12]);
    Or(a=a[13], b=b[13], out=out[13]);
    Or(a=a[14], b=b[14], out=out[14]);
    Or(a=a[15], b=b[15], out=out[15]);
    //author: tr0fin0
}
```

1.10. MUX16

Definição [Funcionamento]

Tabela Verdade Esta porta lógica possuirá a seguinte tabela verdade:

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

Tabela 1.10: Tabela Verdade MUX16

Representação Esta porta lógica pode ser expressa pelo seguinte circuito:

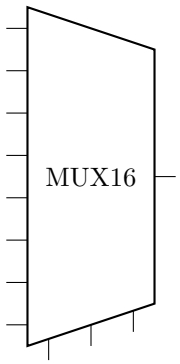

Figura 1.10: Porta Lógica MUX16

Implementação Esta porta lógica pode ser implementada de acordo com o seguinte código:

```
// 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/Mux16.hdl

/**
 * 16-bit multiplexor:
 * for i = 0..15 out[i] = a[i] if sel == 0
 *                        b[i] if sel == 1
 */

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

    PARTS:
    Mux(a=a[0],  b=b[0],  sel=sel, out=out[0]);
    Mux(a=a[1],  b=b[1],  sel=sel, out=out[1]);
    Mux(a=a[2],  b=b[2],  sel=sel, out=out[2]);
    Mux(a=a[3],  b=b[3],  sel=sel, out=out[3]);
    Mux(a=a[4],  b=b[4],  sel=sel, out=out[4]);
    Mux(a=a[5],  b=b[5],  sel=sel, out=out[5]);
    Mux(a=a[6],  b=b[6],  sel=sel, out=out[6]);
    Mux(a=a[7],  b=b[7],  sel=sel, out=out[7]);
    Mux(a=a[8],  b=b[8],  sel=sel, out=out[8]);
    Mux(a=a[9],  b=b[9],  sel=sel, out=out[9]);
```

```
    Mux(a=a[10], b=b[10], sel=sel, out=out[10]);
    Mux(a=a[11], b=b[11], sel=sel, out=out[11]);
    Mux(a=a[12], b=b[12], sel=sel, out=out[12]);
    Mux(a=a[13], b=b[13], sel=sel, out=out[13]);
    Mux(a=a[14], b=b[14], sel=sel, out=out[14]);
    Mux(a=a[15], b=b[15], sel=sel, out=out[15]);
    //author: tr0fin0
}
```

1.11. OR8WAY

Definição [Funcionamento]

Tabela Verdade Esta porta lógica possuirá a seguinte tabela verdade:

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

Tabela 1.11: Tabela Verdade OR8WAY

Representação Esta porta lógica pode ser expressa pelo seguinte circuito:

Figura 1.11: Porta Lógica OR8WAY

Implementação Esta porta lógica pode ser implementada de acordo com o seguinte código:

```
// 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/Or8Way.hdl

/**
 * 8-way Or:
 * out = (in[0] or in[1] or ... or in[7])
 */

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

    PARTS:
    Or(a=in[0], b=in[1], out=orA);
    Or(a=in[2], b=in[3], out=orB);
    Or(a=in[4], b=in[5], out=orC);
    Or(a=in[6], b=in[7], out=orD);
    Or(a=orA, b=orB, out=orE);
    Or(a=orC, b=orD, out=orF);
    Or(a=orE, b=orF, out=out);
    //author: tr0fin0
}
```

1.12. MUX4WAY16

Definição [Funcionamento]

Tabela Verdade Esta porta lógica possuirá a seguinte tabela verdade:

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

Tabela 1.12: Tabela Verdade MUX4WAY16

Representação Esta porta lógica pode ser expressa pelo seguinte circuito:

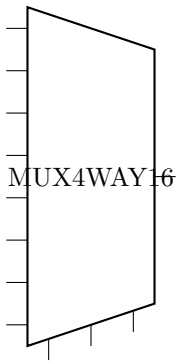

Figura 1.12: Porta Lógica MUX4WAY16

Implementação Esta porta lógica pode ser implementada de acordo com o seguinte código:

```
// 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/Mux4Way16.hdl

/**
 * 4-way 16-bit multiplexor:
 * out = a if sel == 00
 *       b if sel == 01
 *       c if sel == 10
 *       d if sel == 11
 */

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

    PARTS:
    Mux16(a=a, b=b, sel=sel[0], out=muxAB);
    Mux16(a=c, b=d, sel=sel[0], out=muxCD);
    Mux16(a=muxAB, b=muxCD, sel=sel[1], out=out);
    //author: tr0fin0
}
```

1.13. MUX8WAY16

Definição [Funcionamento]

Tabela Verdade Esta porta lógica possuirá a seguinte tabela verdade:

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

Tabela 1.13: Tabela Verdade MUX8WAY16

Representação Esta porta lógica pode ser expressa pelo seguinte circuito:

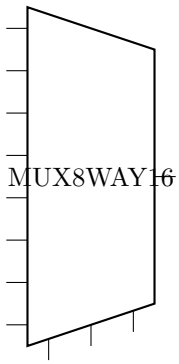

Figura 1.13: Porta Lógica MUX8WAY16

Implementação Esta porta lógica pode ser implementada de acordo com o seguinte código:

```
// 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/Mux8Way16.hdl

/**
 * 8-way 16-bit multiplexor:
 * out = a if sel == 000
 *       b if sel == 001
 *       etc.
 *       h if sel == 111
 */

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

    PARTS:
    Mux4Way16(a=a, b=b, c=c, d=d, sel=sel[0..1], out=muxABCD);
    Mux4Way16(a=e, b=f, c=g, d=h, sel=sel[0..1], out=muxEFGH);
    Mux16(a=muxABCD, b=muxEFGH, sel=sel[2], out=out);
    //author: tr0fin0
}
```

1.14. DMUX4WAY

Definição [Funcionamento]

Tabela Verdade Esta porta lógica possuirá a seguinte tabela verdade:

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

Tabela 1.14: Tabela Verdade DMUX4WAY

Representação Esta porta lógica pode ser expressa pelo seguinte circuito:

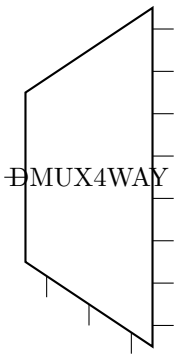

Figura 1.14: Porta Lógica DMUX4WAY

Implementação Esta porta lógica pode ser implementada de acordo com o seguinte código:

```
// 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/DMux4Way.hdl

/**
 * 4-way demultiplexor:
 * {a, b, c, d} = {in, 0, 0, 0} if sel == 00
 *                {0, in, 0, 0} if sel == 01
 *                {0, 0, in, 0} if sel == 10
 *                {0, 0, 0, in} if sel == 11
 */

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

    PARTS:
    Not(in=sel[0], out=ns0);
    Not(in=sel[1], out=ns1);
    And(a=ns0, b=ns1, out=ns0ns1);
    And(a=sel[0], b=ns1, out=s0ns1);
    And(a=ns0, b=sel[1], out=ns0s1);
    And(a=sel[0], b=sel[1], out=s0s1);
    And(a=in, b=ns0ns1, out=a);
    And(a=in, b=s0ns1, out=b);
    And(a=in, b=ns0s1, out=c);
```

```
28      And(a=in, b=s0s1, out=d);
29      //author: tr0fin0
30  }
```

1.15. DMUX8WAY

Definição [Funcionamento]

Tabela Verdade Esta porta lógica possuirá a seguinte tabela verdade:

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

Tabela 1.15: Tabela Verdade DMUX8WAY

Representação Esta porta lógica pode ser expressa pelo seguinte circuito:

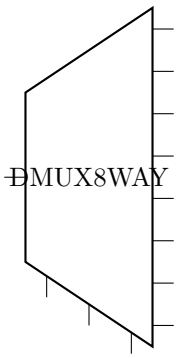

Figura 1.15: Porta Lógica DMUX8WAY

Implementação Esta porta lógica pode ser implementada de acordo com o seguinte código:

```
// 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/DMux8Way.hdl

/**
 * 8-way demultiplexor:
 * {a, b, c, d, e, f, g, h} = {in, 0, 0, 0, 0, 0, 0, 0} if sel == 000
 *                            {0, in, 0, 0, 0, 0, 0, 0} if sel == 001
 *                            etc.
 *                            {0, 0, 0, 0, 0, 0, 0, in} if sel == 111
 */

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

    PARTS:
    Not(in=sel[0], out=ns0);
    Not(in=sel[1], out=ns1);
    Not(in=sel[2], out=ns3);
    And(a=ns0, b=ns1, out=ns0ns1);
    And(a=sel[0], b=ns1, out=s0ns1);
    And(a=ns0, b=sel[1], out=ns0s1);
    And(a=sel[0], b=sel[1], out=s0s1);
    And(a=ns0ns1, b=ns3, out=ns0ns1ns2);
    And(a=s0ns1, b=ns3, out=s0ns1ns2);
```

```
    And(a=ns0s1, b=ns3, out=ns0s1ns2);
    And(a=s0s1, b=ns3, out=s0s1ns2);
    And(a=ns0ns1, b=sel[2], out=ns0ns1s2);
    And(a=s0ns1, b=sel[2], out=s0ns1s2);
    And(a=ns0s1, b=sel[2], out=ns0s1s2);
    And(a=s0s1, b=sel[2], out=s0s1s2);
    And(a=in, b=ns0ns1ns2, out=a);
    And(a=in, b=s0ns1ns2, out=b);
    And(a=in, b=ns0s1ns2, out=c);
    And(a=in, b=s0s1ns2, out=d);
    And(a=in, b=ns0ns1s2, out=e);
    And(a=in, b=s0ns1s2, out=f);
    And(a=in, b=ns0s1s2, out=g);
    And(a=in, b=s0s1s2, out=h);
    //author: tr0fin0
}
```

2. Projeto 2

2.1. HalfAdder

Definição [Funcionamento]

Tabela Verdade Esta porta lógica possuirá a seguinte tabela verdade:

a	b	sum	out
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Tabela 2.1: Tabela Verdade HalfAdder

Representação Esta porta lógica pode ser expressa pelo seguinte circuito:

Figura 2.1: Porta Lógica HalfAdder

Implementação Esta porta lógica pode ser implementada de acordo com o seguinte código:

```
// 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/02/HalfAdder.hdl

/**
 * Computes the sum of two bits.
 */

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

    PARTS:
    Xor(a=a, b=b, out=sum);
    And(a=a, b=b, out=carry);
    //author: tr0fin0
}
```

2.2. FullAdder

Definição [Funcionamento]

Tabela Verdade Esta porta lógica possuirá a seguinte tabela verdade:

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

Tabela 2.2: Tabela Verdade FullAdder

Representação Esta porta lógica pode ser expressa pelo seguinte circuito:

Figura 2.2: Porta Lógica FullAdder

Implementação Esta porta lógica pode ser implementada de acordo com o seguinte código:

```
// 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/02/FullAdder.hdl

/**
 * Computes the sum of three bits.
 */

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

    PARTS:
    HalfAdder(a=a,     b=b, sum=sumAB, carry=carryA);
    HalfAdder(a=sumAB, b=c, sum=sum,   carry=carryB);
    Or(a=carryA, b=carryB, out=carry);
    //author: tr0fin0
}
```

2.3. Add16

Definição [Funcionamento]

Tabela Verdade Esta porta lógica possuirá a seguinte tabela verdade:

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

Tabela 2.3: Tabela Verdade Add16

Representação Esta porta lógica pode ser expressa pelo seguinte circuito:

Figura 2.3: Porta Lógica Add16

Implementação Esta porta lógica pode ser implementada de acordo com o seguinte código:

```
// 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/02/Adder16.hdl

/**
 * Adds two 16-bit values.
 * The most significant carry bit is ignored.
 */

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

    PARTS:
    HalfAdder(a=a[0],  b=b[0],          sum=out[0],  carry=c0);
    FullAdder(a=a[1],  b=b[1],  c=c0,   sum=out[1],  carry=c1);
    FullAdder(a=a[2],  b=b[2],  c=c1,   sum=out[2],  carry=c2);
    FullAdder(a=a[3],  b=b[3],  c=c2,   sum=out[3],  carry=c3);
    FullAdder(a=a[4],  b=b[4],  c=c3,   sum=out[4],  carry=c4);
    FullAdder(a=a[5],  b=b[5],  c=c4,   sum=out[5],  carry=c5);
    FullAdder(a=a[6],  b=b[6],  c=c5,   sum=out[6],  carry=c6);
    FullAdder(a=a[7],  b=b[7],  c=c6,   sum=out[7],  carry=c7);
    FullAdder(a=a[8],  b=b[8],  c=c7,   sum=out[8],  carry=c8);
    FullAdder(a=a[9],  b=b[9],  c=c8,   sum=out[9],  carry=c9);
    FullAdder(a=a[10], b=b[10], c=c9,   sum=out[10], carry=c10);
    FullAdder(a=a[11], b=b[11], c=c10,  sum=out[11], carry=c11);
    FullAdder(a=a[12], b=b[12], c=c11,  sum=out[12], carry=c12);
    FullAdder(a=a[13], b=b[13], c=c12,  sum=out[13], carry=c13);
    FullAdder(a=a[14], b=b[14], c=c13,  sum=out[14], carry=c14);
    FullAdder(a=a[15], b=b[15], c=c14,  sum=out[15], carry=c15);
    //author: tr0fin0
}
```

2.4. Inc16

Definição [Funcionamento]

Tabela Verdade Esta porta lógica possuirá a seguinte tabela verdade:

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

Tabela 2.4: Tabela Verdade Inc16

Representação Esta porta lógica pode ser expressa pelo seguinte circuito:

Figura 2.4: Porta Lógica Inc16

Implementação Esta porta lógica pode ser implementada de acordo com o seguinte código:

```
// 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/02/Inc16.hdl

/**
 * 16-bit incrementer:
 * out = in + 1 (arithmetic addition)
 */

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

    PARTS:
    HalfAdder(a=in[0],  b=true, sum=out[0],  carry=c0);
    HalfAdder(a=in[1],  b=c0,   sum=out[1],  carry=c1);
    HalfAdder(a=in[2],  b=c1,   sum=out[2],  carry=c2);
    HalfAdder(a=in[3],  b=c2,   sum=out[3],  carry=c3);
    HalfAdder(a=in[4],  b=c3,   sum=out[4],  carry=c4);
    HalfAdder(a=in[5],  b=c4,   sum=out[5],  carry=c5);
    HalfAdder(a=in[6],  b=c5,   sum=out[6],  carry=c6);
    HalfAdder(a=in[7],  b=c6,   sum=out[7],  carry=c7);
    HalfAdder(a=in[8],  b=c7,   sum=out[8],  carry=c8);
    HalfAdder(a=in[9],  b=c8,   sum=out[9],  carry=c9);
    HalfAdder(a=in[10], b=c9,   sum=out[10], carry=c10);
    HalfAdder(a=in[11], b=c10,  sum=out[11], carry=c11);
    HalfAdder(a=in[12], b=c11,  sum=out[12], carry=c12);
    HalfAdder(a=in[13], b=c12,  sum=out[13], carry=c13);
    HalfAdder(a=in[14], b=c13,  sum=out[14], carry=c14);
    HalfAdder(a=in[15], b=c14,  sum=out[15], carry=c15);
    //author: tr0fin0
}
```

2.5. ALU

Definição [Funcionamento]

Tabela Verdade Esta porta lógica possuirá a seguinte tabela verdade:

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

Tabela 2.5: Tabela Verdade ALU

Representação Esta porta lógica pode ser expressa pelo seguinte circuito:

Figura 2.5: Porta Lógica ALU

Implementação Esta porta lógica pode ser implementada de acordo com o seguinte código:

```
// 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/02/ALU.hdl

/**
 * The ALU (Arithmetic Logic Unit).
 * Computes one of the following functions:
 * x+y, x-y, y-x, 0, 1, -1, x, y, -x, -y, !x, !y,
 * x+1, y+1, x-1, y-1, x&y, x|y on two 16-bit inputs,
 * according to 6 input bits denoted zx,nx,zy,ny,f,no.
 * In addition, the ALU computes two 1-bit outputs:
 * if the ALU output == 0, zr is set to 1; otherwise zr is set to 0;
 * if the ALU output < 0, ng is set to 1; otherwise ng is set to 0.
 */

// Implementation: the ALU logic manipulates the x and y inputs
// and operates on the resulting values, as follows:
// if (zx == 1) set x = 0        // 16-bit constant
// if (nx == 1) set x = !x       // bitwise not
// if (zy == 1) set y = 0        // 16-bit constant
// if (ny == 1) set y = !y       // bitwise not
// if (f == 1)  set out = x + y  // integer 2's complement addition
// if (f == 0)  set out = x & y  // bitwise and
// if (no == 1) set out = !out   // bitwise not
// if (out == 0) set zr = 1
// if (out < 0) set ng = 1

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

    OUT
```

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

    PARTS:
    Mux16(a=x, b=false, sel=zx, out=xA);
    Mux16(a=y, b=false, sel=zy, out=yA);

    Not16(in=xA, out=NotXA);
    Not16(in=yA, out=NotYA);

    Mux16(a=xA, b=NotXA, sel=nx, out=xB);
    Mux16(a=yA, b=NotYA, sel=ny, out=yB);

    Add16(a=xB, b=yB, out=AddXY);
    And16(a=xB, b=yB, out=AndXY);

    Mux16(a=AndXY, b=AddXY, sel=f, out=outA);
    Not16(in=outA, out=outNotA);
    Mux16(a=outA, b=outNotA, sel=no, out[15]=ng, out[0..7]=outL, out[8..15]=outR,
    out=out);

    Or8Way(in=outL, out=zrA);
    Or8Way(in=outR, out=zrB);
    Or(a=zrA, b=zrB, out=zrC);
    Not(in=zrC, out=zr);

    //author: tr0fin0
}
```

3. Projeto 3

3.1. Bit

Definição [Funcionamento]

Tabela Verdade Esta porta lógica possuirá a seguinte tabela verdade:

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

Tabela 3.1: Tabela Verdade Bit

Representação Esta porta lógica pode ser expressa pelo seguinte circuito:

Figura 3.1: Porta Lógica Bit

Implementação Esta porta lógica pode ser implementada de acordo com o seguinte código:

```
// 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/03/a/Bit.hdl

/**
 * 1-bit register:
 * If load[t] == 1 then out[t+1] = in[t]
 *                 else out does not change (out[t+1] = out[t])
 */

CHIP Bit {
    IN in, load;
    OUT out;

    PARTS:
    Mux(a=outDFF, b=in, sel=load, out=outMux);
    DFF(in=outMux, out=out, out=outDFF);

    //author: tr0fin0
}
```

3.2. Register

Definição [Funcionamento]

Tabela Verdade Esta porta lógica possuirá a seguinte tabela verdade:

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

Tabela 3.2: Tabela Verdade Register

Representação Esta porta lógica pode ser expressa pelo seguinte circuito:

Figura 3.2: Porta Lógica Register

Implementação Esta porta lógica pode ser implementada de acordo com o seguinte código:

```
// 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/03/a/Register.hdl

/**
 * 16-bit register:
 * If load[t] == 1 then out[t+1] = in[t]
 * else out does not change
 */

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

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

    //author: tr0fin0
}
```

3.3. RAM8

Definição [Funcionamento]

Tabela Verdade Esta porta lógica possuirá a seguinte tabela verdade:

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

Tabela 3.3: Tabela Verdade RAM8

Representação Esta porta lógica pode ser expressa pelo seguinte circuito:

Figura 3.3: Porta Lógica RAM8

Implementação Esta porta lógica pode ser implementada de acordo com o seguinte código:

```
// 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/03/a/RAM8.hdl

/**
 * Memory of 8 registers, each 16 bit-wide. Out holds the value
 * stored at the memory location specified by address. If load==1, then
 * the in value is loaded into the memory location specified by address
 * (the loaded value will be emitted to out from the next time step onward).
 */

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

    PARTS:
    DMux8Way(in=load, sel=address, a=loadA, b=loadB, c=loadC, d=loadD, e=loadE,
    f=loadF, g=loadG, h=loadH);
    Register(in=in, load=loadA, out=regA);
    Register(in=in, load=loadB, out=regB);
    Register(in=in, load=loadC, out=regC);
    Register(in=in, load=loadD, out=regD);
    Register(in=in, load=loadE, out=regE);
    Register(in=in, load=loadF, out=regF);
    Register(in=in, load=loadG, out=regG);
    Register(in=in, load=loadH, out=regH);
    Mux8Way16(a=regA, b=regB, c=regC, d=regD, e=regE, f=regF, g=regG, h=regH,
    sel=address, out=out);

    //author: tr0fin0
}
```

3.4. RAM64

Definição [Funcionamento]

Tabela Verdade Esta porta lógica possuirá a seguinte tabela verdade:

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

Tabela 3.4: Tabela Verdade RAM64

Representação Esta porta lógica pode ser expressa pelo seguinte circuito:

Figura 3.4: Porta Lógica RAM64

Implementação Esta porta lógica pode ser implementada de acordo com o seguinte código:

```
// 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/03/a/RAM64.hdl

/**
 * Memory of 64 registers, each 16 bit-wide. Out holds the value
 * stored at the memory location specified by address. If load==1, then
 * the in value is loaded into the memory location specified by address
 * (the loaded value will be emitted to out from the next time step onward).
 */

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

    PARTS:
    DMux8Way(in=load, sel=address[0..2], a=loadA, b=loadB, c=loadC, d=loadD,
    e=loadE, f=loadF, g=loadG, h=loadH);
    RAM8(in=in, load=loadA, address=address[3..5], out=ramA);
    RAM8(in=in, load=loadB, address=address[3..5], out=ramB);
    RAM8(in=in, load=loadC, address=address[3..5], out=ramC);
    RAM8(in=in, load=loadD, address=address[3..5], out=ramD);
    RAM8(in=in, load=loadE, address=address[3..5], out=ramE);
    RAM8(in=in, load=loadF, address=address[3..5], out=ramF);
    RAM8(in=in, load=loadG, address=address[3..5], out=ramG);
    RAM8(in=in, load=loadH, address=address[3..5], out=ramH);
    Mux8Way16(a=ramA, b=ramB, c=ramC, d=ramD, e=ramE, f=ramF, g=ramG, h=ramH,
    sel=address[0..2], out=out);

    //author: tr0fin0
}
```

3.5. RAM512

Definição [Funcionamento]

Tabela Verdade Esta porta lógica possuirá a seguinte tabela verdade:

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

Tabela 3.5: Tabela Verdade RAM512

Representação Esta porta lógica pode ser expressa pelo seguinte circuito:

Figura 3.5: Porta Lógica RAM512

Implementação Esta porta lógica pode ser implementada de acordo com o seguinte código:

```
// This file is part of the materials accompanying the book
// "The Elements of Computing Systems" by Nisan and Schocken,
// MIT Press. Book site: www.idc.ac.il/tecs
// File name: projects/03/b/RAM512.hdl

/**
 * Memory of 512 registers, each 16 bit-wide. Out holds the value
 * stored at the memory location specified by address. If load==1, then
 * the in value is loaded into the memory location specified by address
 * (the loaded value will be emitted to out from the next time step onward).
 */

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

    PARTS:
    DMux8Way(in=load, sel=address[0..2], a=loadA, b=loadB, c=loadC, d=loadD,
    e=loadE, f=loadF, g=loadG, h=loadH);
    RAM64(in=in, load=loadA, address=address[3..8], out=ramA);
    RAM64(in=in, load=loadB, address=address[3..8], out=ramB);
    RAM64(in=in, load=loadC, address=address[3..8], out=ramC);
    RAM64(in=in, load=loadD, address=address[3..8], out=ramD);
    RAM64(in=in, load=loadE, address=address[3..8], out=ramE);
    RAM64(in=in, load=loadF, address=address[3..8], out=ramF);
    RAM64(in=in, load=loadG, address=address[3..8], out=ramG);
    RAM64(in=in, load=loadH, address=address[3..8], out=ramH);
    Mux8Way16(a=ramA, b=ramB, c=ramC, d=ramD, e=ramE, f=ramF, g=ramG, h=ramH,
    sel=address[0..2], out=out);

    //author: tr0fin0
}
```

3.6. RAM4K

Definição [Funcionamento]

Tabela Verdade Esta porta lógica possuirá a seguinte tabela verdade:

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

Tabela 3.6: Tabela Verdade RAM4K

Representação Esta porta lógica pode ser expressa pelo seguinte circuito:

Figura 3.6: Porta Lógica RAM4K

Implementação Esta porta lógica pode ser implementada de acordo com o seguinte código:

```
// 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/03/b/RAM4K.hdl

/**
 * Memory of 4K registers, each 16 bit-wide. Out holds the value
 * stored at the memory location specified by address. If load==1, then
 * the in value is loaded into the memory location specified by address
 * (the loaded value will be emitted to out from the next time step onward).
 */

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

    PARTS:
    DMux8Way(in=load, sel=address[0..2], a=loadA, b=loadB, c=loadC, d=loadD,
    e=loadE, f=loadF, g=loadG, h=loadH);
    RAM512(in=in, load=loadA, address=address[3..11], out=ramA);
    RAM512(in=in, load=loadB, address=address[3..11], out=ramB);
    RAM512(in=in, load=loadC, address=address[3..11], out=ramC);
    RAM512(in=in, load=loadD, address=address[3..11], out=ramD);
    RAM512(in=in, load=loadE, address=address[3..11], out=ramE);
    RAM512(in=in, load=loadF, address=address[3..11], out=ramF);
    RAM512(in=in, load=loadG, address=address[3..11], out=ramG);
    RAM512(in=in, load=loadH, address=address[3..11], out=ramH);
    Mux8Way16(a=ramA, b=ramB, c=ramC, d=ramD, e=ramE, f=ramF, g=ramG, h=ramH,
    sel=address[0..2], out=out);

    //author: tr0fin0
}
```

3.7. RAM16K

Definição [Funcionamento]

Tabela Verdade Esta porta lógica possuirá a seguinte tabela verdade:

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

Tabela 3.7: Tabela Verdade RAM16K

Representação Esta porta lógica pode ser expressa pelo seguinte circuito:

Figura 3.7: Porta Lógica RAM16K

Implementação Esta porta lógica pode ser implementada de acordo com o seguinte código:

```
// 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/03/b/RAM16K.hdl

/**
 * Memory of 16K registers, each 16 bit-wide. Out holds the value
 * stored at the memory location specified by address. If load==1, then
 * the in value is loaded into the memory location specified by address
 * (the loaded value will be emitted to out from the next time step onward).
 */

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

    PARTS:
    DMux4Way(in=load, sel=address[0..1], a=loadA, b=loadB, c=loadC, d=loadD);
    RAM4K(in=in, load=loadA, address=address[2..13], out=ramA);
    RAM4K(in=in, load=loadB, address=address[2..13], out=ramB);
    RAM4K(in=in, load=loadC, address=address[2..13], out=ramC);
    RAM4K(in=in, load=loadD, address=address[2..13], out=ramD);
    Mux4Way16(a=ramA, b=ramB, c=ramC, d=ramD, sel=address[0..1], out=out);

    //author: tr0fin0
}
```

3.8. PC

Definição [Funcionamento]

Tabela Verdade Esta porta lógica possuirá a seguinte tabela verdade:

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

Tabela 3.8: Tabela Verdade PC

Representação Esta porta lógica pode ser expressa pelo seguinte circuito:

Figura 3.8: Porta Lógica PC

Implementação Esta porta lógica pode ser implementada de acordo com o seguinte código:

```
// 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/03/a/PC.hdl

/**
 * A 16-bit counter with load and reset control bits.
 * if      (reset[t] == 1) out[t+1] = 0
 * else if (load[t] == 1)  out[t+1] = in[t]
 * else if (inc[t] == 1)   out[t+1] = out[t] + 1  (integer addition)
 * else                    out[t+1] = out[t]
 */

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

    PARTS:
    Inc16(in=outR, out=outI);

    Mux16(a=outR,    b=outI,  sel=inc,   out=outInc);
    Mux16(a=outInc,  b=in,    sel=load,  out=outLoad);
    Mux16(a=outLoad, b=false, sel=reset, out=outReset);

    Register(in=outReset, load=true, out=out, out=outR);

    //author: tr0fin0
}
```

4. Projeto 4

4.1. Linguagem de Máquina Hack

Definição Linguagem básica utilizada para manipular as portas lógicas implementadas anteriormente, consistindo de comandos binários com palavras de 16-bit. Há duas possíveis instruções suportadas por esta linguagem, descritas a seguir:

1. **A Instructions:** Registrador A assume o valor de `value`, representado pelo seguinte comando:

```
@value
```

Este comando também possuirá uma representação em binário de 16-bit, descrito pela seguinte notação:

$$\underbrace{0}_{\text{instrução A}} \quad \underbrace{000000000000000}_{\texttt{value} \text{ representado em 15-bit}} \tag{4.1}$$

2. **C Instructions:** Realiza a operação `comp`, armazena o resultado em `dest` e poderá realizar um deslocamento de acordo com a condição imposta por `jump`;

```
dest = comp ; jump
```

Este comando também possuirá uma representação em binário de 16-bit, descrito pela seguinte notação:

$$\underbrace{1}_{\text{instrução C}} \quad \underbrace{11}_{\text{não utilizado}} \quad \underbrace{a\ c1\ c2\ c3\ c4\ c5\ c6}_{\texttt{comp} \text{ bits}} \quad \underbrace{d1\ d2\ d3}_{\texttt{dest} \text{ bits}} \quad \underbrace{j1\ j2\ j3}_{\texttt{jump} \text{ bits}} \tag{4.2}$$

Onde:

(a) `comp`: Representa todas as possíveis operações que a `ALU` poderá realizar, expressas a seguir:

Operação	Resultado		c1	c2	c3	c4	c5	c6
Tornar Zero	0		1	0	1	0	1	0
Tornar Um	1		1	1	1	1	1	1
Tornar −Um	−1		1	1	1	0	1	0
Manter	D		0	0	1	1	0	0
	A	M	1	1	0	0	0	0
Negar	!D		0	0	1	1	0	1
	!A	!M	1	1	0	0	0	1
Oposto	−D		0	0	1	1	1	1
	−A	−M	1	1	0	0	1	1
Incrementar	D+1		0	1	1	1	1	1
	A+1	M+1	1	1	0	1	1	1
Decrementar	D−1		0	0	1	1	1	0
	A−1	M−1	1	1	0	0	1	0
Somar	D+A	D+M	0	0	0	0	1	0
Subtrair	D−A	D−M	0	1	0	0	1	1
	A−D	M−D	0	0	0	1	1	1
`AND`	D&A	D&M	0	0	0	0	0	0
`OR`	D | A	D | M	0	1	0	1	0	1
	a==0	a==1						

Tabela 4.1: Operações `ALU`

Note que estas entradas corresponderam as entradas da `ALU`.

(b) `dest`: Representa o destino do resultado da operação realizada, expressa a seguir:

Operação	Armazena	d1	d2	d3
`null`	Descarta Resultado	0	0	0
`M`	`RAM[A]`	0	0	1
`D`	Registrador D	0	1	0
`MD`	`RAM[A]` e Registrador D	0	1	1
`A`	Registrador A	1	0	0
`AM`	Registrador A e `RAM[A]`	1	0	1
`AD`	Registrador A e Registrador D	1	1	0
`ADM`	Registrador A, `RAM[A]` e Registrador D	1	1	1

Tabela 4.2: Destinos de `C`

Note que estas entradas corresponderam respectivamente aos loads de cada armazenador; `A.load = d1`, `D.load = d2` e `M.load = d3`.

(c) `jump`: Representa qual a condição para o fluxo do programa, expresso a seguir:

Operação	Resultado	j1	j2	j3
`null`	no jump	0	0	0
`JGT`	if out > 0 jump	0	0	1
`JEQ`	if out $= 0$ jump	0	1	0
`JGE`	if out ≥ 0 jump	0	1	1
`JLT`	if out < 0 jump	1	0	0
`JNE`	if out $\neq 0$ jump	1	0	1
`JLE`	if out ≤ 0 jump	1	1	0
`JMP`	unconditional jump	1	1	1

Tabela 4.3: Condições de `jump`

Note que estas entradas corresponderam respectivamente aos resultados da `ALU`; `not(ng) = j1`, `zr = j2` e `ng = j3`.

4.2. Mult

Definição

Implementação Este código pode ser implementada de acordo com o seguinte código:

```
// 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/04/Mult.asm

// Multiplies R0 and R1 and stores the result in R2.
// (R0, R1, R2 refer to RAM[0], RAM[1], and RAM[2], respectively.)

// Put your code here.

@R0
D=M
@i
M=D  //i = R0
@sum
M=0  //sum = 0

(LOOP)
@i
D=M
@STOP
D;JEQ //if i = 0 goto stop
@R1
D=M
@sum
M=M+D  //sum = sum + R1
@i
M=M-1 //i = i - 1
@LOOP
0;JMP

(STOP) //stop label
@sum
D=M
@R2
M=D    //RAM[2] = sum

(END) //end label
@END
0;JMP
```

4.3. Fill

Definição

Implementação Este código pode ser implementada de acordo com o seguinte código:

```
// 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/04/Fill.asm

// Runs an infinite loop that listens to the keyboard input.
// When a key is pressed (any key), the program blackens the screen,
// i.e. writes "black" in every pixel;
// the screen should remain fully black as long as the key is pressed.
// When no key is pressed, the program clears the screen, i.e. writes
// "white" in every pixel;
// the screen should remain fully clear as long as no key is pressed.

// Put your code here.
(START)
@SCREEN
D=A
@addr
M=D  //addr = 16384

@8192
D=A
@i
M=D   //i = 8192

@KBD  //KBD = 0, no key pressed
D=M
@WHITE
D;JEQ //if KBD = 0 goto WHITE

(BLACK)
@i
D=M
@START
D;JEQ // if i=0 goto START

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

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

@i
M=M-1 // i = i - 1

@BLACK
0;JMP

(WHITE)
@i
D=M
@START
D;JEQ // if i=0 goto START

@addr
A=M
```

```
M=0   // RAM[addr] = 0000000000000000

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

@i
M=M-1 // i = i - 1

@WHITE
0;JMP
```

5. Projeto 5

5.1. Arquitetura de Computadores

Definição Estruturação e construção de microcontroladores, definida teoricamente por Alan Turing e estabelecida na prática por John Von Neumann, estabelecendo diferentes blocos básicos:

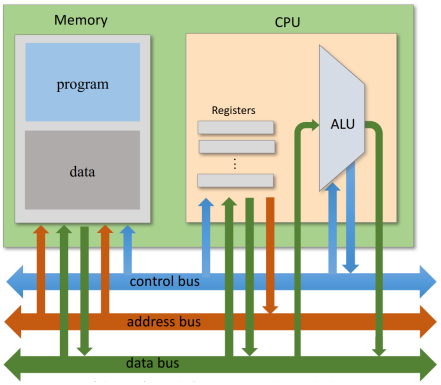

Figura 5.1: Hack Processor

Onde:

1. **Memória:** Responsável por armazenar variáveis do código, separadas em:
   (a) `Data`;
   (b) `Programa`;
2. **CPU:** Controle das operações realizadas por meio dos seguintes componentes:
   (a) `Registradores`;
   (b) `ALU`;

Estes componentes se comunicam através de 3 vias principais, cada qual responsável por transportar uma parcela das informações do sistema:

1. **Address Bus:** Transporta os enderenços envolvidos no comando;
2. **Control Bus:** Transporta as instruções a serem executadas;
3. **Data Bus:** Transporta as informações a serem utilizadas;

Fetching Armazenar a localização da próxima instrução na entrada do endereço de memória e obter a instrução através da leitura dessa memória alocada.

5.2. Memory

Definição [Funcionamento]

Implementação Este componente pode ser implementada de acordo com o seguinte código:

```
// 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/05/Memory.hdl

/**
 * The complete address space of the Hack computer's memory,
 * including RAM and memory-mapped I/O.
 * The chip facilitates read and write operations, as follows:
 *     Read:  out(t) = Memory[address(t)](t)
 *     Write: if load(t-1) then Memory[address(t-1)](t) = in(t-1)
 * In words: the chip always outputs the value stored at the memory
 * location specified by address. If load==1, the in value is loaded
 * into the memory location specified by address. This value becomes
 * available through the out output from the next time step onward.
 * Address space rules:
 * Only the upper 16K+8K+1 words of the Memory chip are used.
 * Access to address>0x6000 is invalid. Access to any address in
 * the range 0x4000-0x5FFF results in accessing the screen memory
 * map. Access to address 0x6000 results in accessing the keyboard
 * memory map. The behavior in these addresses is described in the
 * Screen and Keyboard chip specifications given in the book.
 */

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

    PARTS:
    Not(in=address[13], out=notA13);
    Not(in=address[14], out=notA14);

    And(a=address[14], b=notA13, out=nA13AndA14);

    And(a=nA13AndA14, b=load, out=loadScreen); //confirmation for Screen
    And(a=notA14,     b=load, out=loadRAM);    //confirmation for RAM

    RAM16K(in=in, load=loadRAM,    address=address[0..13], out=outRAM);
    Screen(in=in, load=loadScreen, address=address[0..12], out=outSCR);
    Keyboard(out=outKBD);

    //address: 000000000000000  sel == 0 0 a: 0 0 RAM
    //address: 010000000000000  sel == 0 1 b: 0 1 RAM
    //address: 100000000000000  sel == 1 0 c: 1 0 screen
    //address: 110000000000000  sel == 1 1 d: 1 1 keyboard
    Mux4Way16(
        a=outRAM,
        b=outRAM,
        c=outSCR,
        d=outKBD,
        sel[1]=address[14],
        sel[0]=address[13],
        out=out);

    //author: tr0fin0
}
```

5.3. CPU

Definição [Funcionamento]

Implementação Este componente pode ser implementada de acordo com o seguinte código:

```
// 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/05/CPU.hdl

/**
 * The Hack CPU (Central Processing unit), consisting of an ALU,
 * two registers named A and D, and a program counter named PC.
 * The CPU is designed to fetch and execute instructions written in
 * the Hack machine language. In particular, functions as follows:
 * Executes the inputted instruction according to the Hack machine
 * language specification. The D and A in the language specification
 * refer to CPU-resident registers, while M refers to the external
 * memory location addressed by A, i.e. to Memory[A]. The inM input
 * holds the value of this location. If the current instruction needs
 * to write a value to M, the value is placed in outM, the address
 * of the target location is placed in the addressM output, and the
 * DM control bit is asserted. (When DM==0, any value may
 * appear in outM). The outM and DM outputs are combinational:
 * they are affected instantaneously by the execution of the current
 * instruction. The addressM and pc outputs are clocked: although they
 * are affected by the execution of the current instruction, they commit
 * to their new values only in the next time step. If reset==1 then the
 * CPU jumps to address 0 (i.e. pc is set to 0 in next time step) rather
 * than to the address resulting from executing the current instruction.
 */

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:

    //instruction:
    // 5 43 2 109876 543 210
    // 0 00 0 000000 000 000
    // i    a cccccc ddd jjj
    //        123456 123 123
    // 1 11 0 001100 000 001

    //confirmation of writeM, from Memory M
    And(a=instruction[15], b=instruction[3], out=writeM);

    //chose instruction or keep data
    Not(in=instruction[15], out=notI15);
    Mux16(a=outALU, b=instruction, sel=notI15, out=inAR);

```

```
    //confirmation of writeA, from Register A
    Or(a=notI15, b=instruction[5], out=selAR);
    ARegister(in=inAR, load=selAR, out=outAR, out[0..14]=addressM);

    //confirmation of writeD, from Register D
    And(a=instruction[15], b=instruction[4], out=selDR);
    DRegister(in=outALU, load=selDR, out=outDR);

    //chose registerA or memoryM
    And(a=instruction[15], b=instruction[12], out=selAM);
    Mux16(a=outAR, b=inM, sel=selAM, out=outAM);

    //confirmation of ALU commands
    And(a=instruction[15], b=instruction[11], out=inZX);
    And(a=instruction[15], b=instruction[10], out=inNX);
    And(a=instruction[15], b=instruction[9],  out=inZY);
    And(a=instruction[15], b=instruction[8],  out=inNY);
    And(a=instruction[15], b=instruction[7],  out=inF);
    And(a=instruction[15], b=instruction[6],  out=inNO);
    ALU(
        x=outDR,
        y=outAM,
        zx=inZX,
        nx=inNX,
        zy=inZY,
        ny=inNY,
        f =inF ,
        no=inNO,
        out=outALU,
        out=outM,
        zr=outZR,
        ng=outNG
    );

    Not(in=outNG, out=notNG);
    Not(in=outZR, out=notZR);
    And(a=notNG, b=notZR, out=outPS);

    //jump instruction
    And(a=instruction[0], b=outPS, out=isPS); //if outM > 0: outPS = 1
    And(a=instruction[1], b=outZR, out=isZR); //if outM = 0: outZR = 1
    And(a=instruction[2], b=outNG, out=isNG); //if outM < 0: outNG = 1

    Or(a=isPS, b=isZR, out=isPZ);
    Or(a=isPZ, b=isNG, out=isJP);

    //confirmation of jump
    And(a=instruction[15], b=isJP, out=loadPC);

    PC(in=outAR, load=loadPC, inc=true, reset=reset, out[0..14]=pc);
    //author: tr0fin0
}
```

5.4. Computer

Definição [Funcionamento]

Implementação Este componente pode ser implementada de acordo com o seguinte código:

```
// 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/05/Computer.hdl

/**
 * The HACK computer, including CPU, ROM and RAM.
 * When reset is 0, the program stored in the computer's ROM executes.
 * When reset is 1, the execution of the program restarts.
 * Thus, to start a program's execution, reset must be pushed "up" (1)
 * and "down" (0). From this point onward the user is at the mercy of
 * the software. In particular, depending on the program's code, the
 * screen may show some output and the user may be able to interact
 * with the computer via the keyboard.
 */

CHIP Computer {

    IN reset;

    PARTS:
    Memory(
        in=outM,
        load=writeM,
        address=addressM,
        out=inM
    );

    CPU(
        inM=inM,
        instruction=outROM,
        reset=reset,
        outM=outM,
        writeM=writeM,
        addressM=addressM,
        pc=outPC
    );

    ROM32K(
        address=outPC,
        out=outROM
    );

    //author: tr0fin0
}
```

6. Projeto 6

Definição Conjunto gratuito e aberto RISC de Instruction Set Architecture, ou seja, conjunto de regras de desenvolvimento de software e hardware. Recomenda-se a utilização do seguinte Simulador de RISC-V.

6.1. Registradores RISC-V

Definição Processodores elaborados sobre esta Arquitetura possuirão 32 registradores responsáveis por desempenhar funções especificas, cuja descrição seguem abaixo:

Nome	Descrição
zero	Valor Fixo em 0
t0-t6	Valores Temporários
s0-s11	Valores Salvos
a0-a7	Parâmetros e Valores de Retorno de Funções
ra	Endereço de Retorno de Função
sp	Apontador de Pilha

Tabela 6.1: Registradores RISC-V

6.2. Formato de Funções

Definição Há diferentes estruturas de funções que podem ser empregadas, entre as principais estruturas de funções tem-se os seguintes formatos básicos onde `func` representa uma função genérica, como descrito a seguir:

1. **3 Arguments Functions:** Registrador `s0` assume o valor de `s1 func s2`, representado pelo seguinte comando:

```
func s0, s1, s2
```

2. **2 Arguments Functions:** Registrador `s0` assume o valor de `s1 func 1`, onde `1` será nomeado imediato, representado pelo seguinte comando:

```
func s0, s1, 1
```

3. **1 Arguments Functions:** Registrador `s0` assume o valor de `1`, onde `1` será nomeado imediato, representado pelo seguinte comando:

```
func s0, 1
```

6.3. Instruções Aritméticas

Definição Há diferentes funções aritméticas que permitem realizar matemática aritmética simples na arquitetura `load/store`, entre as principais instruções tem-se como descrito a seguir:

1. **ADD Instruction:** Armazenada a soma de `rs1 + rs2` no registrador `rd`, representado pelo seguinte comando:

```
ADD rd, rs1, rs2
```

2. **ADDI Instruction:** Armazenada a soma de `rs1 + imm` no registrador `rd`, representado pelo seguinte comando:

```
ADDI rd, rs1, imm
```

3. **SUB Instruction:** Armazenada a subtração de `rs1 - rs2` no registrador `rd`, representado pelo seguinte comando:

```
SUB rd, rs1, rs2
```

6.4. Instruções Lógicas

Definição Há diferentes funções lógicas que permitem realizar operações simples na arquitetura `load/store`, entre as principais instruções tem-se como descrito a seguir:

1. **XOR Instruction:** Armazenada a lógica de `rs1 XOR rs2` no registrador `rd`, representado pelo seguinte comando:

```
XOR rd, rs1, rs2
```

2. **OR Instruction:** Armazenada a lógica de `rs1 OR rs2` no registrador `rd`, representado pelo seguinte comando:

```
OR rd, rs1, rs2
```

3. **OR Instruction:** Armazenada a lógica de `rs1 AND rs2` no registrador `rd`, representado pelo seguinte comando:

```
AND rd, rs1, rs2
```

Nota-se que todas as instruções acima descritas possuem variação imediata, ou seja, podem receber alternativamente um valor imediato para realizar a operação.

6.5. Instruções de Deslocamento

Definição Há diferentes funções que permitem deslocar lateralmente bits na arquitetura `load/store`, entre as principais instruções tem-se como descrito a seguir:

1. **SLL Instruction:** Armazena no registrador `rd` o deslocamento de `rs2` bits para esquerda do valor que se encontra em `rs1`, representado pelo seguinte comando:

```
SLL rd, rs1, rs2
```

   Este comando multiplica o valor de `rs1` por 2^x, onde x representa o valor de `rs2`.

2. **SRL Instruction:** Armazena no registrador `rd` o deslocamento de `rs2` bits para direita do valor que se encontra em `rs1`, representado pelo seguinte comando:

```
SRL rd, rs1, rs2
```

   Este comando divide o valor de `rs1` por 2^x, onde x representa o valor de `rs2`.

3. **SRA Instruction:** Armazena no registrador `rd` o deslocamento de `rs2` bits para direita do valor que se encontra em `rs1`, representado pelo seguinte comando:

```
SRA rd, rs1, rs2
```

   Este comando, diferentemente do SRL, replica o valor mais significativo do valor, garantindo que o complemento de dois seja conservado.

Nota-se que todas as instruções acima descritas possuem variação imediata, ou seja, podem receber alternativamente um valor imediato para realizar a operação, sendo o método normalmente mais utilizado.

6.6. Instruções de Memória

Definição Há diferentes funções que permitem acessar e escrever na memória na arquitetura `load/store`, entre as principais instruções tem-se como descrito a seguir:

1. **LW Instruction:** Armazena no registrador `rd` a leitura do endereço dentro de `rs1 + imm`, representado pelo seguinte comando:

```
LW rd, rs1, imm
```

2. **SW Instruction:** Armazena no registrador `rd` a escrita do endereço de `rs1`, representado pelo seguinte comando:

```
SW rd, rs1, imm
```

Neste conjunto de instruções será necessário fornecer a localização do vetor utilizado com relação a seu início. Como `32 bits = 4 bytes` os imediatos fornecidos as funções serão múltiplos do tamanho de palavra que estiver sendo lida, como representado pela seguinte tabela:

Linguagem C	Variáveis em RISC-V	Tamanho em Bytes
bool	byte	1
char	byte	1
short	halfword	2
int	word	4
long	word	4
void	unsigned word	4

Tabela 6.2: Variavéis RISC-V

6.7. Instruções de Comparação

Definição Há diferentes funções que permitem, limitadamente, comparar valores na arquitetura `load/store`, entre as principais instruções tem-se como descrito a seguir:

1. **SLT Instruction:** Armazena no registrador `rd` a comparação se o valor em `rs1` é menor do que o valor em `rs2`, representado pelo seguinte comando:

```
SLT rd, rs1, rs2
```

   Este, comando apresentaram as variações imediatas, não sinalizadas e a combinação entre imediata e não sinalizada.

6.8. Instruções de Salto Condicional

Definição Há diferentes funções que permitem, com base em uma comparação, ir para outra posição de memória na arquitetura `load/store`, entre as principais instruções tem-se como descrito a seguir:

1. **BEQ Instruction:** Desloca-se para a posição de `imm` se o valor de `rs1 == rs2`, representado pelo seguinte comando:

```
BEQ rs1, rs2, imm
```

2. **BNE Instruction:** Desloca-se para a posição de `imm` se o valor de `rs1 != rs2`, representado pelo seguinte comando:

```
BNE rs1, rs2, imm
```

3. **BLT Instruction:** Desloca-se para a posição de `imm` se o valor de `rs1 < rs2`, representado pelo seguinte comando:

```
BLT rs1, rs2, imm
```

   Este, comando apresentará a variação não sinalizada.

4. **BGE Instruction:** Desloca-se para a posição de `imm` se o valor de `rs1 >=rs2`, representado pelo seguinte comando:

```
BGE rs1, rs2, imm
```

   Este, comando apresentará a variação não sinalizada.

6.9. Códigos Básicos

Definição Há diferentes estruturas comumente empregadas em código, entre os principais métodos tem-se como descrito a seguir:

1. **if Estrutura:** Realização de uma comparação e separação do código, representado pelos seguintes comandos:

```c
#include<stdio.h>
int main()
{
    int t0 = 9;
    int t1 = 0;
    int t2 = 5;

    if (t0 == t2)
    {
        t1 += 7;
    } else {
        t1 += 15;
    }
}
```

```
main:
    addi t0, zero, 9
    addi t1, zero, 0
    addi t2, zero, 5

    bne t0, t2, else
    addi t1, t1, 7
    j fim

else:
    addi t1, t1, 15

fim:
    jr ra
```

2. **while Estrutura:** Realização de `loop` do código, representado pelos seguintes comandos:

```c
#include<stdio.h>
int main()
{
    int t0 = 20;
    int t1 = 10;

    while (t0 != t1)
    {
        t0 += 2;
        t1 += 3;
    }
}
```

```
main:
    addi t0, zero, 20
    addi t1, zero, 10

while:
    beq t0, t1, fim
    addi t0, t0, 2
    addi t1, t1, 3
    j while

fim:
    jr ra
```

3. **for Estrutura:** Realização de `loop` do código, representado pelos seguintes comandos:

```c
#include<stdio.h>
int main()
{
    int t0 = 0;
    int t1 = 0;
    int t2 = 100;

    for (t0 = 0; t0 < t1; t0++)
    {
        t2 += t0;
    }

}
```

```
main:
    addi t0, zero, 0
    addi t1, zero, 0
    addi t2, zero, 100

for:
    bge t1, t2, fim
    addi t0, t0, t1
    addi t1, t1, 1
    j for

fim:
    jr ra
```

6.10. Programas

1. **Triângulo**:

```
#  --------------------------
#  posible triangule code
#  --------------------------
#  s0 side A of the triangule
#  s1 side B of the triangule
#  s2 side C of the triangule

## if a0 == 0 impossible triangule
#        == 1   possible triangule

#  for testing, uncomment and put the following
#  lines on the main code:
#     addi s0, zero, 3
#     addi s1, zero, 4
#     addi s2, zero, 6
main:

    add t0, s0, s1      # t0 = s0 + s1
    sub t0, t0, s2      # t0 = s0 + s1 - s2
    bge zero, t0, else # if t0 >= 0 then else

    add t0, s1, s2      # t0 = s1 + s2
    sub t0, t0, s0      # t0 = s1 + s2 - s0
    bge zero, t0, else # if t0 >= 0 then else

    add t0, s2, s0      # t0 = s2 + s0
    sub t0, t0, s2      # t0 = s2 + s0 - s1
    bge zero, t0, else # if t0 >= 0 then else

    # passed all restrictions
    addi a0, zero, 1 # a0 = 1
    j end             # go to end

else:
    add a0, zero, zero # a0 = 0

end:
    jr ra
```

2. **Multiplicação**:

```
# -------------------------------
# multiplication with deslocation
# -------------------------------
#
# s0 integer number >= 0
# s1 integer number >= 0
# a0 result of s0*s1
#
# operation done without multiplication instruction
#
# example:
#     0 0 0 0 1 0 0 1: 09, a0
#     0 0 0 0 0 1 0 1: 05, a1
#     ---------------
#     0 0 0 0 1 0 0 1: 09 a1
#   0 0 0 0 0 0 0 0
# 0 0 0 0 1 0 0 1    : 36 a1
#     ---------------
#     0 0 1 0 1 1 0 1: 45

#    a0 a1 t0 t1
# t0  9  5  0  0
# t1  9  5  0  0

multiply:
    add t0, zero, zero # t0 = zero + zero

    deslocation:
        andi t1, a1, 1 # t1 = 1, if a1 is even (a1[LSB])
                       # t1 = 0, if a1 is odd  (a1[LSB])
        srai a1, a1, 1 # right shift of a1 by 1, divide by 2

        beq t1, zero, pass # if t1 == zero then pass
        add t0, t0, a0     # t0 = t0 + a0

    pass:
        slli a0, a0, 1             # left shift of a0 by 1, multiple by 2
        bne  a1, zero, deslocation # if a1 != zero then deslocation
        add  a0, t0, zero          # a0 = t0 + zero
        jr ra                      # return

main:

    jal multiply

    jr ra
```

7. Projeto 7

Definição Assim como outras linguagens o RISC-V seguem uma sequência de convenções, regras estabelecida entre os usuários, para utilização e aplicação desta linguagem. Recomenda-se que sejam seguidas para garantir a compreensão de seu código.

7.1. Endereçamento da Memória

Definição Sequência utilizada para armazenar uma palavra de `4 bytes`, ou seja `32 bits`, na memória, diferenciando a ordem de leitura e escrita através dos seguintes métodos:

1. **Big Endian**: Aloca o `byte` mais significativo, `MSB`, primeiro;
2. **Little Endian**: Aloca o `byte` menos significativo, `LSB`, primeiro;

Esta configuração demonstra apenas como o processor lerá cada palavra de sua memória, iniciando pelo MSB ou pelo LSB. Desta maneira pode-se considerar o seguinte exemplo:

$$\underbrace{\text{H}}_{\text{0x48}}\ \underbrace{\text{E}}_{\text{0x45}}\ \underbrace{\text{L}}_{\text{0x4C}}\ \underbrace{\text{L}}_{\text{0x4C}}\ \underbrace{\text{O}}_{\text{0x4F}}\ \underbrace{!}_{\text{0x21}}\ \underbrace{}_{\text{0x00}} \tag{7.1}$$

Código Little Endian:

```
1    .word 0x4C4C4548
2    .word 0x0000214F
```

Código Big Endian:

```
1    .word 0x48454C4C
2    .word 0x4F210000
```

Memória:

Adress	Little Endian	Big Endian
0	H	
1	E	
2	L	!
3	L	O
4	O	L
5	!	L
6		E
7		H

Tabela 7.1: Estrutura de Memoria em RISC-V

Note que está diferença não influência para a leitura do código, pois isto influenciará apenas o processamento do processador. RISC-V é padronizado em `Little Endian`, desta forma será necessário atentar-se quando outros dispositivos sejam `Big Endian`.

7.2. Execução de Funções

Definição Trechos de código que executam uma tarefa específica organizadas separadamente para facilitar seu reuso e legibilidade dentro das convenções utilizadas, abaixo serão listados as principais:

1. **Atribuição de Variável**: Parâmetros utilizados na função serão sequênciados a partir de `a0`, `a1`, ...;
2. **Retorno de Variável**: Valores retornados na função serão sequênciados a partir de `a0`, `a1`, ...;

Note que quando uma função for executada variáveis atualmente armazenadas em registradores podem ser perdidas, pois durante a execução da função estes locais de memória podem ser acessados e reescritos. Desta maneira recomenda-se salvar as variáveis necessárias na pilha, deslocando o espaço de memória quando não for mais necessário.

7.3. Pilha

Definição Espaço da memória reservado para armazenar elementos momentaneamente cujo controle partirá do usuário. Nesta arquitetura o registrador `sp` sempre apontará para o último elemento da Pilha.

Organização de Memória Processadores terão suas memórias organizadas com objetivo de otimizar sua utilização, desta maneira será divida nos seguintes espaços:

1. **Pilha**: Inicia-se no fim da memória e cresce para baixo;
2. **Heap**: Inicia-se no começo da memória e cresce para cima;

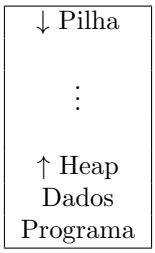

Tabela 7.2: Estrutura de Memoria em RISC-V

Note que desta forma há diferentes combinações possíveis entre as memórias, sendo apenas limitadas pela outra. Assim, o **Heap** poderá crescer até encontrar-se com a **Pilha** e vice-versa, possibilitando flexibilixar e maximizar a utilização da memória disponível. Caso deseja-se alocar espaço na **Pilha** recomenda-se utilizar a seguinte abordagem:

Início:

```
1    addi sp, sp, -8
2    sw   ra, 0(sp)
3    sw   s0, 4(sp)
```

Final:

```
1    lw   s0, 4(sp)
2    lw   ra, 0(sp)
3    addi sp, sp, 8
```

7.4. Programas

1. **Menor Valor Vetor**:

```
# ------------------------
# smaller value in vector
# ------------------------
#
# menorVetor(int *v, int n)
# int *v, pointer to vector
# int  n, size of vector

#  the first argument is a pointer to
#  where the array starts in the memory
#  to compair you need the go throw the
#  array getting every value and comparing
#  with the rest saving the smaller

MenorVetor:
    add s0, zero, a0   # s0 = *v, save pointer
    add t0, zero, zero # i  = 0,  start counter
    lw  t1, 0(s0)      # min = 0, save first value

    loop:
        lw  a0, 0(s0)    # a0 = v[j]
        bge a0, t1, pass # if v[j] ge min then pass

        add t1, zero, a0 # min = v[j]

    pass:
        addi s0, s0, 4   # j++, advance in stack
        addi t0, t0, 1   # i++, advance in counter
        blt t0, a1, loop # if i lt n then loop

    add a0, t1, zero # save value
    jr ra            # return
```

2. **Soma Vetores**:

```
# --------------
# sum of vectors
# --------------
#
# somaVetor(int *a, int *b, int *c, int n)
# int *a, pointer to vector with n int's
# int *b, pointer to vector with n int's
# int *c, pointer to vector with n int's
# int n,  size of vector's

# a[i] = b[i] + c[i]


SomaVetor:
    # inicialize

    # save inicial pointers
    add s0, zero, a0 # s0 = *A[j]
    add s1, zero, a1 # s1 = *B[j]
    add s2, zero, a2 # s2 = *C[j]
    # save vectors size
    add s3, zero, a3 # s3 =  n

    # vector values registers
    add t0, zero, zero # t0 = A[j]
    add t1, zero, zero # t1 = B[j]
    add t2, zero, zero # t2 = C[j]
    # size counter
    add t3, zero, zero # i = 0

    loop:
        lw t1, 0(s1) # load B[j] value
        lw t2, 0(s2) # load C[j] value

        add t0, t1, t2 # store total value
        sw  t0, 0(s0)  # A[j] = B[j] + C[j]

        addi s0, s0, 4 # *A[j] = A[j+1]
        addi s1, s1, 4 # *B[j] = B[j+1]
        addi s2, s2, 4 # *C[j] = C[j+1]

        addi t3, t3, 1 # i = i + 1

        blt t3, s3, loop # if i lt n then loop

    jr ra # return
```

8. Projeto 8

Defingão Linguagens Assembly são diretamente interligadas com o hardware envolvido, trazendo limitações a execução de instruções. Desta maneira, existem maneiras de armazenar e manipular variáveis com `32 bits` através do espaço reservado a instruções dentro dos registradores.

8.1. Constantes de 32 bits

Defição Instruções que manipulam valores imediatos só utilizam os `12 bits inferiores` dos registradores para armazenar os valores, pelas restrições trazidas pelas implementações de operações. Desta maneira pode-se utilizar as seguintes instruções para manipular os `20 bits superiores` quando necessário:

$$\underbrace{0000\ 0000\ 0000\ 0000\ 0000}_{\texttt{20 bits superiores}} \quad \underbrace{0000\ 0000\ 0000}_{\texttt{12 bits inferiores}} \tag{8.1}$$

Note que os espaços foram incluidos para facilitar o entendimento dos bits e devem ser desprezados durante a execução das instruções. As seguinte instruções permintem a manipulação dos registradores:

1. **LUI Instruction:** Adiciona-se o valor da posição de `imm` de `20 bits` nos bits superiores da posição `rs1`, representado pelo seguinte comando:

```
1    LUI rs1, imm
```

   Note que o comando `ADDI` adicionará um valor de `12 bits` nos bits inferiores da posição desejada.

8.2. Manipulação de Bits

Definição Operações lógicas com os bits armazenados em um registrador permitem extrair resultados importantes sobre seus valores. Algumas lógicas comumente aplicadas estão descritas a seguir:

1. **OR Instruction:** Realizar esta operação terá o seguinte efeito sobre os bits armazenados:

OR	a[i]	imm
1	a[i]	1
a[i]	a[i]	0

Tabela 8.1: Operação OR

2. **AND Instruction:** Realizar esta operação terá o seguinte efeito sobre os bits armazenados:

AND	a[i]	imm
a[i]	a[i]	1
0	a[i]	0

Tabela 8.2: Operação AND

3. **XOR Instruction:** Realizar esta operação terá o seguinte efeito sobre os bits armazenados:

XOR	a[i]	imm
$\overline{\text{a[i]}}$	a[i]	1
a[i]	a[i]	0

Tabela 8.3: Operação XOR

Isso possibilita os seguintes resultados:

1. **Número Par:** Análisa-se o último bit do valor no registrador `rs1` e armazena 1 se for par na posição `rd`, representado pelo seguinte comando:

```
ANDI rd, rs1, 1
```

   Note que o comando poderia ser aplicado para diferentes multiplicidades, basta alterar o imediato.

2. **Zerar Bits:** Análisa-se os bits do valor no registrador `rs1` e zera aqueles nas posições com 0 do `imm`, armazenando o resultado em `rd`, representado pelo seguinte comando:

```
ANDI rd, rs1, 0xFF0
```

   Note que o comando poderia ser aplicado para diferentes bits do código com números binários ou hexadecimais, basta alterar o imediato.

3. **Ativar Bits:** Análisa-se os bits do valor no registrador `rs1` e ativa aqueles nas posições com 1 do `imm`, armazenando o resultado em `rd`, representado pelo seguinte comando:

```
ORI rd, rs1, 0xFF0
```

   Note que o comando poderia ser aplicado para diferentes bits do código com números binários ou hexadecimais, basta alterar o imediato.

4. **Inverter Bits:** Análisa-se os bits do valor no registrador `rs1` e inverte aqueles nas posições com 1 do `imm`, armazenando o resultado em `rd`, representado pelo seguinte comando:

```
XORI rd, rs1, 0xFF0
```

   Note que o comando poderia ser aplicado para diferentes bits do código com números binários ou hexadecimais, basta alterar o imediato.

8.3. Representação de Caracteres

Definição Convenção necessária para representação de strings dentro das instruções binárias convertendo valores em caracteres distintos. Estas convenções mudaram ao longo do desenvolvimento de software, onde as seguintes convenções são as mais relevantes:

1. **ASCII:** Representação utilizando `7 bits`, mais simples e antiga codificação de caracteres, onde o sexto bit indica, no caso das letras, se é maíusculo e minusculo.

2. **ISO8859:** Representação utilizando `8 bits` de codificação de caracteres, onde o acentuações mais utilizadas foram implementadas.

3. **UTF:** Representação utilizando quantidades variáveis de bits, onde uma referência principal é implementada para facilitar sua expansão.

No RISC-V a manipulação de strings se dá pela manipulação de bytes na memória, desta maneira os seguintes comandos podem ser empregrados:

1. **LBU Instruction:** Carrega o valor de um byte não sinalizado no registrador `rs1` deslocado pelo imediato `imm` e o armazena na posição `rd`, representado pelo seguinte comando:

```
LBU rd, x(rs1)
```

2. **LHU Instruction:** Carrega o valor de um half byte não sinalizado no registrador `rs1` deslocado pelo imediato `imm` e o armazena na posição `rd`, representado pelo seguinte comando:

```
LHU rd, x(rs1)
```

3. **SBU Instruction:** Armazena o valor carregado da posição `rs2` deslocado pelo imediato `imm` de um byte no registrador `rs1`, representado pelo seguinte comando:

```
SBU rd, x(rs1)
```

4. **SHU Instruction:** Armazena o valor carregado da posição `rs2` deslocado pelo imediato `imm` de um half byte no registrador `rs1`, representado pelo seguinte comando:

```
    SHU rd, x(rs1)
```

Note que nestes comandos o final das strings será indicado por `" 0"`.

8.4. Funções de Strings

Definição Há muitos funções utéis para manipulações de strings que não são padrões ao RISC-V, assim algumas das mais empregadas serão empregadas a seguir:

1. **int strlen:** Retorna o tamanho da `string`:

```
# int strlen(const char *str)
strlen:
    addi t0, zero, 0 # i lenght of string

    loop:
        lbu t1, 0(a0)      # char = string[i]
        beq t1, zero, end # if char == \0 then end
        addi t0, t0, 1     # i++
        addi a0, a0, 1     # a0++
        j loop

    end:
        addi a0, t0, 0 # a0 = i
ret
```

2. **char *strcpy:** Copia uma `string` de um endereço para outro:

```
# char *strcpy(char *destination, const char *source)
strcpy:
    add t0, a0, zero # t0 = *destination

    loop:
        lbu t1, 0(a1)        #
        sbu t1, 0(a0)        # a0 = a1
        addi a0, a0, 1       # a0++
        addi a1, a1, 1       # a1++
        bne t1, zero, loop # if schar != \0 then end

    add a0, t0, zero # a0 = *destination

ret
```

3. **int strcmp:** Compara uma `string` de um endereço com outra:

```
# int strcmp(const char *str1, const char *str2)
strcmp:

    loop:
        lbu t0, 0(a0)        # t0 = str1[i]
        lbu t1, 0(a1)        # t1 = str2[i]
        addi a0, a0, 1       # str1++
        addi a1, a1, 1       # str2++
        bne t0, t1, end    # if str1[i] != str2[i] then end
        bne t0, zero, loop # if str1[i] != \0       then loop

        addi a0, zero, 1 # a0 = 1, str1 == str2
    ret

```

```
    end:
        addi a0, zero, 0 # a0 = 0, str1 != str2
```

4. **int strcat:** Concatena uma `string` de um endereço com outra:

```

# char *strcat(char *destination, const char *source)
strcat:
    addi t0, a0, 0 # t0 = *dest

    loop:
        lbu  t1, 0(a0)      # dest[i]
        addi a0, a0, 1      # dest++
        bne  t1, zero, loop # if dest[i] != \0 then loop

    copy:
        lbu  t2, 0(a1)      #
        sbu  t2, 0(a0)      #
        addi a0, a0, 1      # dest++
        addi a1, a1, 1      # sour++
        bne  t2, zero, copy # if sour[i] != \0 then copy

    addi a0, t0, 0 # a0 = *dest
```

8.5. Chamadas de Sistema

Definição Interações entre o código em assembly e o restante do sistema depende do sistema operacional empregado. Neste caso o simulador será capaz de realizar as seguintes operações:

Syscall	`t0`	Descrição
Imprime `inteiro`	1	Imprime o valor de `a0` no console como `inteiro`
Imprime `caracter`	2	Imprime o valor de `a0` no console como `caracter`
Imprime `string`	3	Imprime a `string` `a0` com tamanho a1 no console
Lê `inteiro`	4	Lê `inteiro` do console e retorna em `a0`
Lê `caracter`	5	Lê `caracter` do console e retorna em `a0`
Lê `string`	6	Lê `string` de tamanho a1 e retorna em `a0`
`SBRK`	7	`a0`>0, Aloca bytes de memória e retorna ponteiro em `a0`
		`a0`<0, Desaloca bytes de memória e retorna ponteiro em `a0`

Tabela 8.4: Syscall RISC-V

Onde o seguinte procedimento será necessário:

1. Coloque o número da `systemcall` no registrador `t0`;
2. Coloque os argumentos em `a0`, `a1` ...;
3. Execute a instrução `ecall`
4. Caso haja retorno, estará em `a0`;

8.6. Programas

1. **Maiuscula:**

```
# ---------
# Maiuscula
# ---------
#
# char *Maiuscula(char *s)
#   char *s, pointer to string
#
# receives a string ended in \0
# and return the same string with
# all it is letters upper case
#
#     .TEXT:
#         .word 0x61605E5C
#         .word 0x796E6462
#         .word 0x00007B7A

Maiuscula:
    addi t0, a0, 0          # save *string
    addi t2, zero,  97      # a in ASCII
    addi t3, zero, 122      # z in ASCII

    loop:
        lbu t1, 0(a0)       #    char = string[i]
        beq t1, zero, end   # if char == \0 then end

        addi a0, a0, 1      # a0++

        blt t1, t2, loop    # if char  a then loop
        bgt t1, t3, loop    # if char  z then loop

        lbu  t1,  -1(a0)    #      char = string[i]
        addi t1, t1, -32    #       "A" = "a" - 32
        sb   t1,  -1(a0)    # string[i] = CHAR

        j loop

    end:
        addi a0, t0, 0      # return *string
        jr ra
```

2. **Imprime Maiuscula:**

```
# ----------------
# ImprimeMaiuscula
# ----------------
#
# void *ImprimeMaiuscula(char *s)
#   char *s, pointer to string
#
# receives a string ended in \0
# and print the same string with
# all it is letters upper case
#
# assuming that the Maiuscula
# is implemented
#
#     .TEXT:
#         .word 0x61605E5C
#         .word 0x796E6462
#         .word 0x00007B7A

ImprimeMaiuscula:
    addi sp, sp, -4
    sw   ra, 0(sp)

    call Maiuscula

    addi t1, a0, 0          # *string

    loop:
        lbu t2, 0(t1)       #    char = string[i]
        beq t2, zero, end   # if char == \0 then end

        addi t1, t1, 1      # a0++

        addi t0, zero, 2    # syscall: print caracter
        addi a0, t2, 0      # caracter to print
        ecall

        j loop

    end:
        lw   ra, 0(sp)
        addi sp, sp,4
        jr ra
```

3. **Palindrome:**

```
# ----------
# Palindrome
# ----------
#
# int Palindrome(char *s)
#   char *s, pointer to string
#
# receives a string ended in \0
# return 1 if the string is     palindrome
#        0 if the string is not palindrome
#
#     .TEXT:
#         .word 0x61605E5C
#         .word 0x796E6462
#         .word 0x00007B7A

Palindrome:
    addi t0, a0, 0          # *s[0]

    loop0:
        lbu t1, 0(a0)       #    char = s[i]
        beq t1, zero, end0  # if char == \0 then end0

        addi a0, a0, 1      # a0++
        j loop0

    end0:
        addi t1, a0, -1     # *s[n-1]

    addi a0, zero, 1;       # a0 = 1

    loop1:
        lbu t2, 0(t0)       # s[i]
        lbu t3, 0(t1)       # s[n-i-1]
        beq t2, zero, end2  # if s[i] == zero then end2

        addi t0, t0,  1     # t0++
        addi t1, t1, -1     # t1--

        bne t2, t3, end1;   # if s[i] != s[n-i-1] then end1
        j loop1

    end1:
        addi a0, zero, 0;   # a0 = 0

    end2:
        jr ra
```

9. Projeto 9

Definição Acesso a memória em processadores Assembly será limitada, pois será responsabilidade integral do programador. Registradores podem ser acessados e manipulados em diferentes trechos do código, descartando valores anteriormente armazenados. Desta maneira variáveis devem ser manipuladas com cuidado.

9.1. Variáveis

Definição Locais na memória utilizados para armazenar valores necessários no decorrer dos programs. Há diferentes formais de realizar essa alocação que variam de acorodo com a plataforma utilizada, sendo as mais utilizadas para o simulador utilizado as demonstradas a seguir:

1. **Constantes:** Armazenadas na região de dados da memória e não poderam ser alteradas, implementadas da seguinte maneira:

```
    .section .rodata
    vetor:
        .word 0
        .word 1
        .word 2
```

   Note que neste caso `.word` armazena uma palavra de `4 bytes`, espaço necessário para armazenar um `int`.

2. **Variáveis Globais:** Armazenadas na região de dados da memória e poderam ser alteradas, implementadas da seguinte maneira:

```
    .section .data
    vetor:
        .word 0
        .word 1
        .word 2
```

   Note que neste caso o `vetor` será um ponteiro para o primeiro endereço de memória que compõem o vetor. Será necessário percorrer suas posições para ler os valores armazenados, desta maneira o tamanho deste deverá ser informado breviamente para que valores indesejados não sejam acessados.

3. **Variáveis Locais:** Armazenadas na pilha da memória e poderam ser alteradas, implementadas da seguinte maneira:

```
    addi sp, sp, -16
    sw ra, 12(sp)
```

   Note que `addi` deslocará o ponteiro da pilha pela quantidade de espaço desejada pelo usuário, neste caso `4 words`. Além disso, reposiciona-se o ponteiro `ra`, restando espaço para alocação de `3 words`. Ao final da execução do código será necessário restaurar a posição da pila e recuperar as variáveis nela armazenadas.

4. **Struct:** Armazena várias variáveis em conjunto sequencial onde cada variável deverá ser considerada, implementadas da seguinte maneira:

```

    sw ra, 12(sp)
```

   Note que torna-se comum realizar o **padding** dos dados, isto é, arredondar o espaço necessário para cade variável para múltiplos de 4 bytes para facilitar o percurso ao longo da memória.

9.2. Exceções

Definição Eventos que podem causar a transferência da execução para outra parte do código, tipicamente para o Sistema Operacional. No RISC-V faz-se a seguinte diferenciação:

1. **Exceções:**
   (a) `Definição:` Causas internas ao CORE;
   (b) `Exemplo:` Divisão por zero;
2. **Interrupções:**
   (a) `Definição:` Causas externas ao CORE;
   (b) `Exemplo:` Movimento do Mouse;

Note que interrupções são uma alternativa para o tratamento de preriféricos, processando suas informações apenas quando estas são requisitadas que poderá vir do software ou do hardware como descrito abaixo:

1. **Totalmente em Software:**
   (a) `Definição:` Única rotina chamada para qualquer evento externo e deve consultar todos os periféricos para descobrir o que aconteceu;
2. **Híbrido:**
   (a) `Definição:` Única rotina chamada para qualquer evento externo e recebe um registrador indicando o causador da interrupção;
3. **Auxiliada em Hardware:**
   (a) `Definição:` Rotinas diferentes chamadas para cada evento externo facilitando como o software é escrito;

Cada diferente abordagem poderá implementar uma rotina adequada, sendo as principais: Endereço Único, armazena um único endereço; Tratador Individualizado, armazena um endereço base e outro para cada evento possível num vetor de endereços.

9.3. Control and Status Registers

Definição Registrador além dos 32 disponíveis para manipulação responsáveis por armazenar informações sobre o processador e controlam operações do controlador, possibilitando os seguintes comandos:

1. **CSRR:** Le o valor de um registrador reservado `mscratch` e armazena no registrador `rd` como representado a seguir:

```
1    csrr rd, mscratch
```

2. **CSRW:** Escreve em um registrador reservado `mscratch` o valor de um registrador `rd` como representado a seguir:

```
1    csrw mscratch, rd
```

3. **CSRRW:** Troca o valor em um registrador reservado `mscratch` com o valor de um registrador `rd` como representado a seguir:

```
1    csrw rd, mscratch, rd
```

9.4. Programas

1. **Leitura e Impressão:**

```
.section .text
# insert function here
    function:
        addi t0, zero, 4            # syscall: 4 read integer
        ecall

        addi t1, a0,    0           # t0 = n    number of strings
        addi t2, zero, 20           # t2 = s    size   of strings
        addi t3, zero,  0           # t3 = i    counter


        read:
            sub  sp, sp, t2         # allocating space in stack
            addi a0, sp,  0

            addi t0, zero, 6        # syscall: 6 read string
            add  a1, zero, t2       # syscall: a1 size of string = s
            ecall

            addi t3, t3, 1          # i++
            beq t3, t1, print       # if i == n then read
            j read


        print:
            addi a0, sp,  0         # str[n-i-1]
            add  sp, sp, t2         # desallocatin space in stack

            addi t0, zero, 3        # syscall: 3 print string
            add  a1, zero, t2       # syscall: a1 size of string = s
            ecall

            addi t0, zero, 2        # syscall: 2 print caracter
            addi a0, zero, 13       # caracter \n
            ecall

            addi t3, t3, -1         # i--
            beq  t3, zero, end      # if i == 0 then end
            j print

        end:
            jr ra



    main:
        addi sp, sp, -4             # allocating space in stack
        sw   ra, 0(sp)              # saving ra address

        call function               # call of function to test

        lw   ra, 0(sp)              # retrieving ra address
        addi sp, sp,  4             # desallocatin space in stack
        jr ra
```