Hardware-C

```
part PassThrough
{
	public bit in;
	public bit out;

	out = in;
}


part MUX
{
	public bit control;
	public bit[2] inputs;
	public bit output;

	for (i; 0..2)
	{
		if (control == i)
			output = inputs[i];
	}
}
```

////\\\\////\\\\////\\\\////\\\\////\\\\////\\\\////\\\\////\\\\////\\\\////\\\\////\\\\////\\\\////\\\\////\\\\////\\\\////\\\\////\\\\////\\\\////\\\\////

I swear if you don't stop touching my code antelope  .

```
//half adder
bit myBitA;
bit myBitB;

bit result;
bit carry;
```

- I think the basic logical transistors can be operators or keywords instead of objects. So you'd write

~~ANDgate myAnd = new ANDgate(myBitA, myBitB);~~
~~ORgate myOr = new ORgate(myBitA, myBitB);~~

```
 bit myAnd = (myBitA AND myBitB);
```

- instead of each part object having an execute(), we should have a global proceedNextCycle(), similar to how MARS has stepthrough button

~~result = ANDgate.execute();~~

~~result = ORgate.execute();~~

- There should be an easy way to debugging code. I did it like above.

```
printf("calc_branchAddr branchAddr=: 0x%x", branchAddr);
printf("calc_jumpAddr pcPlus4: 0x%04x", pcPlus4);
```

- I think there should be an INSTRUCTION object (struct?) It should contain:
  - which control bits are turned, along with opcode/funct
  - instruction name in English (this would've made testcases way easier)
  - Its value in hex/binary

```
type Instruction{
string   name;
int      opcode, funct;
int      regDst,regWrite, ALUsrc, memRead, memWrite, memToReg, ALU.op;
int      ALU.bNegate, extra1, extra2, extra3;
}
```

- Instead of the beloved RussWires, we should have custom data-type to represent binary integers, like above

```
public class UnsignedBinary32 {
    private static final MAX_UNSIGNED32 = Math.pow(2, 32) - 1;
    private static final MIN_UNSIGNED32 = 0;

    private final int value;

    public Unsigned32(int value) {
        if (value > MAX_UNSIGNED32 || value < MIN_UNSIGNED32)
            throw new IllegalArgumentException("omg overflow: " + value);
        this.value = value;
    }

    public int getValue() {
        return value.toBinary(); //returns 0010001111000, not binaryArr[32]
    }
// ... other methods, such as equals(), comparison, add, sub, etc.
}
```

- I want to access RAM and play with the registers.

```
part Register{
register[] $s = new register[9];
register[] $t = new register[8];
//then I can do things like write back or add $t[0], $S[1], $zero
}

part RAM{
WORD[] RAM = new WORD[8192];
//I don't even know what this will look like, I just want to see the memory I'm writing to
}
```

////\\\\////\\\\////\\\\////\\\\////\\\\////\\\\////\\\\////\\\\////\\\\////\\\\////\\\\////\\\\////\\\\////\\\\////\\\\////\\\\////\\\\////

```
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
// Purpose:      Half-adder
// Authors:      Goose and HedgeHog
part halfAdder {
        public  bit input[2];
        public  bit carry;
        public  bit output;

        output = input[0] ^ input[1];

}
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

part halfAdder
{
        public bit A
        public bit B

        public bit sum
        public bit carry
```

```
        A (gate.xor) B -> sum
        A (gate.and) B -> carry

}
```

Suggestions:

- format for binary numbers in nibbles (2x1001_0100_0111_1111)
- assignment always goes left -> right with input on left and output on right
- logic represented as words (and, xor, nor, or)

//~~~~~~

```
part halfAdder {
        public bit input1;
        public bit input2;
        public bit output;
        public bit carryOut;

        // output = input1 ^ input2, carryOut = input1 && input2

}
```

```
part HalfAdder {
        public bit a;
        public bit b;
        public bit sum;
        public bit carry;

        sum = a^b;
        carry = a & b;
}
```

```
part HalfAdder
{
        public bit a;
        public bit b;
        public bit carryOut;
        public bit sum;

        sum = (a != b);
        carryOut = (a && b);

}
```

```
//-----------------------------//
part HalfAdder {
        //Can we parameterize parts?
        public bit a,b, sum, carry;
        sum = (a != b);
        carry = (a & b);
}

//-----------------------------//
```

```
-----------------------------------LINE-----------------------------------------
Part ONE_BIT_HALF_ADDER
{
        Public input_1;
        Public input_2;
        Public output;
        Public carry_out;

        Output = input_1 + input_2;
        If (input_1 == input_2 == 1)
        {
                Carry_out = 1;
        }
}

Part ONE_BIT_FULL_ADDER
{
        Public ONE_BIT_HALF_ADDER_1(input_1, input_2);
        Public carry_in = ONE_BIT_HALF_ADDER_1.carry_out;
        Public ONE_BIT_HALF_ADDER_2(input_1, input_2);
        Public output = ONE_BIT_HALF_ADDER_2.output;
        Public carry_out = ONE_BIT_HALF_ADDER_2.carry_out;

}
-----------------------------------END-------------------------------------------




_____Start_____
Part FullAddr
{
        Public bit carryIn;
        Public bit input[2];
        Public bit output;
        Public bit sum;

        Sum

}
```

_____end_____

----------Start----------
```
part HalfAdder {
	public bit A,B;
	public bit carryOut;
	public bit sumOut;

	carryOut = (A&B);
	sumOut = (A^B);
}
```
----------End----------

==========
#TrustInRuss |
==========

```
part halfAdder{
	public bit a;
	public bit b;

	public bit sum;
	public bit carry;

	sum=(!a & b) | (a & !b);
	carry= a & b;

}
```

NOBODY FOUND ME!!
I FOUND U.
OK :P

----- This is my stuff dudes! ---

```
Part halfAdder {
        Public bit in1;
        Public bit in2;
        Public  bit carry  = in1 & in2;
        Public bit sum = in1 ^ in2;
}
Part fullAdder {
        Public bit in1;
        Public bit in2;
        Public bit carry;
        Tm1 = halfAdder(in1, in2);
        Tm2 = halfAdder(carry, tm1.sum);
        Public bit sum = Tm2.sum;
        Public bit carry = Tm1.carry | Tm1.carry;
}
```

```
part HalfAdder
{
	public bit[2] inputs;
	public bit output;

	elseif(input[0] || input[1]
}
```

==========

```
part HalfAdder
{
   public bit a;
   public bit b;

   public bit s = a ^ b;
   public bit c = a & b;
}

part FullAdder
{
   public bit a, b, ci;

   public bit s, co;

   s = (a ^ b) ^ ci;
   co = (a & b) | ((a ^ b) & c)


}
```

==========

==========

```
part  halfAdder
{
	public bit a;
	public bit b;
	public bit output;
	Public bit carry;

	output = (a ^ !b) | (!a ^ b);
	carry = a ^ b;
}
```

==========

==========

```
part halfAdder {

	public bit a;
	public bit b;

	public XOR sum;
	public AND carryOut;

	sum.setIn1(a);
	sum.setIn2(b);
	sum.execute();

	carryOut.setIn1(a);
	carryOut.setIn2(b);
	sum2.execute();

}
```