attoPong 1.0 (zdrojový kód v jazyce attoASM)

```
// Register Memory Allocation
PADDLE0_Y { 0 }
PADDLE1_Y { 1 }
BALL_X { 2 }
BALL_Y { 3 }
SCORE0 { 4 }
SCORE1 { 5 }
BALL_XSPD { 6 }
BALL_YSPD { 7 }
TEMP { 8 }

// For passing an argument to a symbol
ARG0 { 0 }
ARG1 { 0 }
ARG2 { 0 }
ARG3 { 0 }

// Auxiliary

EXE { CTRL+7(2) ! }
WriteTEMP
{
      ADDR [02, 8]
      CTRL [03, 7]
      EXE
}

// Output value from the Register memory at address given by ARG
OutputRegister
{
      ADDR [03, 8]
      DATA [ARG0]
      CTRL [01, 7] // write address
      EXE
      CTRL [0DH, 7]// output data
      EXE
      DATA 1(32)
}

// stop register output
StopRegister
{
  ADDR [03, 8]
  CTRL [0, 7]
  EXE
}

OUT2TEMP
{
   DATA 1(32)
   ADDR [05, 8] CTRL [01, 7] EXE // output OUT
   WriteTEMP
   ADDR [05, 8] CTRL [0, 7] EXE // stop the OUT output
}

// signed add two values together and store the result back
// ARG1 - first value
// ARG2 - second value
// ARG3 - result (where to write)
SADDStoreBack
{

      ARG0 {! ARG1 }
      OutputRegister
      WriteTemp
      ADDR [03, 8] CTRL [0, 7] EXE // stop the register output
      ARG0 {! ARG2 }
```

```
        OutputRegister

        // add them together
        ADDR [04, 8]
        CTRL [09, 7]
        EXE

        // store the result back
        ADDR [03, 8] CTRL [0, 7] EXE // stop the register output
        ADDR [03, 8] DATA [ARG3] CTRL [01, 7] EXE // address the proper location in the register
memory
        DATA 1(32)
        ADDR [05, 8] CTRL [1, 7] EXE // output the out
        ADDR [03, 8] CTRL [0EH, 7] EXE // write the value
        ADDR [05, 8] CTRL [0, 7] EXE // stop the out output
}

// **** KEYBOARD READ ****

ReadKey
{
        DATA 1(32)   // prepare for data exchange
        ADDR [0DH, 8]// address the input controller
        CTRL [08, 7] // read the key
        EXE
}

StopKey
{
        ADDR [0DH, 8]// address the input controller
        CTRL [0, 7]  // read the key
        EXE
}

ProcessAllKeys
{
        ARG0 {! PADDLE0_Y }
        ARG1 {! 22 } // the W key
        ARG2 {! 02 } // subtraction
        ProcessKey
        ARG0 {! PADDLE0_Y }
        ARG1 {! 18 } // the S key
        ARG2 {! 01 } // addition
        ProcessKey

        // second paddle
        ARG0 {! PADDLE1_Y }
        ARG1 {! 4 }  // the E key
        ARG2 {! 02 } // subtraction
        ProcessKey
        ARG0 {! PADDLE1_Y }
        ARG1 {! 3 }  // the D key
        ARG2 {! 01 } // addition
        ProcessKey
}

/*
                ARGUMENTS:
                ARG0 = address at the register memory to process
                ARG1 = code of the key
                ARG2 = command code for the ALU to calculate new value
*/

ProcessKey
{
        // Read the W key
        ReadKey
```

```
    WriteTEMP
    StopKey

    DATA [ARG1] // code of the W key
    // compare them using the ALU
    ADDR [04, 8]
    CTRL [28H, 7] // test for equality
    EXE

    // output result from the OUT
    ADDR [05, 8]
    CTRL [01, 7]
    EXE
    DATA 1(32)
    WriteTEMP    // and store it in the temp register
    // multiply by two, so it moves by two pixels
    ADDR [04, 8]
    CTRL [01, 7]
    EXE EXE
    WriteTEMP

    ADDR [05, 8]
    CTRL [0, 7]
    EXE // stop OUT output


    // load the value from the Register memory

    // address is prepared in the ARG0
    OutputRegister

    // calculate new value
    ADDR [04, 8]
    CTRL [ARG2, 7]
    EXE

    ADDR [03, 8]
    CTRL [0, 7]
    EXE // stop register memory outputting

    // ---- limit value to min 0, max 127 ----
    // copy it from the OUT to the TEMP
    DATA 1(32)
    ADDR [05, 8]
    CTRL [1, 7]
    EXE
    WriteTemp
    ADDR [05, 8]
    CTRL [0, 7]
    EXE

    // maximum
    DATA [128-18]
    ADDR [04, 8]
    CTRL [26H, 7]
    EXE

    // copy it from the OUT to the TEMP
    DATA 1(32)
    ADDR [05, 8]
    CTRL [1, 7]
    EXE
    WriteTemp
    ADDR [05, 8]
    CTRL [0, 7]
    EXE
```

```
        // minimum
        DATA [0]
        ADDR [04, 8]
        CTRL [24H, 7]
        EXE

        // new position is calculated, now store the value back
        DATA 1(32)
        ADDR [05, 8]
        CTRL [1, 7]
        EXE // output calculated value
        ADDR [03, 8] // register memory
        CTRL [0EH, 7]
        EXE // modified value is now written back

        // cleanup
        ADDR [05, 8]
        CTRL [0, 7]
        EXE // stop OUT output
}

// DRAWING

// draws the ball at current position - it's calculated automatically
DrawBall
{
        // ball is 6x6 px

        // get the Y position first and calculate proper address for the LCD
        ARG0 {! BALL_Y }
        OutputRegister
        // write it to the temp
        ADDR [02, 8]
        CTRL [03, 7]
        EXE

        // stop register output
        ADDR [03, 8]
        CTRL [0, 7]
        EXE

        // multiply by 128
        DATA [128]
        ADDR [04, 8]
        CTRL [03, 7]
        EXE

        // output the OUT
        ADDR [05, 8]
        CTRL [01, 7]
        EXE
        DATA 1(32)

        // write it to the temp
        WriteTEMP

        // stop the OUT output
        ADDR [05, 8]
        CTRL [0, 7]
        EXE

        // add the paddle X position to the address
        ARG0 {! BALL_X }
        OutputRegister

        ADDR [04, 8]
        CTRL [01, 7]
        EXE    // add the BALL_X to the address
        // OUT now contains the address, where drawing of the ball should start

        // stop register output
        ADDR [03, 8]
        CTRL [0, 7]
        EXE

        // output the OUT
        ADDR [05, 8]
        CTRL [01, 7]
        EXE

        // write the address to the LCD
        ADDR [0CH, 8]
        CTRL [01, 7]
        EXE          // write the new address

        // write it to the temp too (DrawRowNext requires it)
        WriteTEMP

        // stop the out output
        ADDR [05, 8]
        CTRL [0, 7]
        EXE

        ARG0 {! 00FF0000H }

        // draw 6 rows
        DrawRowNext DrawRowNext DrawRowNext
        DrawRowNext DrawRowNext DrawRowNext
}

// draws paddle at the current position - it needs to be set before this symbol is used
DrawPaddle
{
        // paddle is 6x18 px
        DATA 1(32)
        // store the starting value in the TEMP first
        ADDR [0CH, 8]
        CTRL [06, 7]
        EXE
        WriteTemp

        // start writing pixels
        ADDR [0CH, 8]
        CTRL [0, 7]
        EXE          // stop the data output first

        ARG0 {! 00FFFF00H }

        DrawRowNext DrawRowNext DrawRowNext DrawRowNext
        DrawRowNext DrawRowNext DrawRowNext DrawRowNext
        DrawRowNext DrawRowNext DrawRowNext DrawRowNext
        DrawRowNext DrawRowNext DrawRowNext DrawRowNext
        DrawRowNext DrawRowNext
}

// draw a row of pixels and move to the next one
// color is stored in ARG0
DrawRowNext
{
        DATA+8 [ARG0, 24]
        // write 6 pixels
        ADDR [0CH, 8]
        CTRL [03, 7]
        CTRL+7(12) !
```

```
	// move to the next row
	DATA [128]
	ADDR [04, 8]
	CTRL [01, 7]
	EXE          // add 128 to the value

	DATA 1(32)
	ADDR [05, 8]
	EXE          // output it
	WriteTemp
	ADDR [0CH, 8]
	CTRL [01, 7]
	EXE          // write the new address
	ADDR [05, 8]
	CTRL [0, 7]
	EXE          // stop the output from the OUT
}

// write LCD paddle start position
// ARG0 - register address containing the position
// ARG1 - number to add to the start address (used to determine side)
LCDPaddleStart
{
	// output the start position from the register memory
	OutputRegister

	// write it to the temp
	ADDR [02, 8]
	CTRL [03, 7]
	EXE

	// stop register output
	ADDR [03, 8]
	CTRL [0, 7]
	EXE

	// multiply by 128
	DATA [128]
	ADDR [04, 8]
	CTRL [03, 7]
	EXE

	// output the OUT
	ADDR [05, 8]
	CTRL [01, 7]
	EXE
	DATA 1(32)

	// write it to the temp
	WriteTEMP

	// stop the OUT output
	ADDR [05, 8]
	CTRL [0, 7]
	EXE

	// now add the value in ARG1 (horizontal shift)
	DATA [ARG1]
	ADDR [04, 8]
	CTRL [01, 7]
	EXE

	// output the OUT
	ADDR [05, 8]
	CTRL [01, 7]
	EXE
```

```
       DATA 1(32)

       // write the address to the LCD
       ADDR [0CH, 8]
       CTRL [01, 7]
       EXE

}

UpdateBall
{
       // increment/decrement
       // add BALL_XSPD to the BALL_X
       ARG1 {! BALL_X }
       ARG2 {! BALL_XSPD }
       ARG3 {! ARG1 }
       SADDStoreBack

       // add BALL_YSPD to the BALL_Y
       ARG1 {! BALL_Y }
       ARG2 {! BALL_YSPD }
       ARG3 {! ARG1 }
       SADDStoreBack

       /* **********************
                    VERTICAL COLLISION
          ********************** */

       DATA [0]
       WriteTEMP    // temp contains minimal value
       ARG0 {! BALL_Y }
       OutputRegister
       // now compare them
       ADDR [04, 8]
       CTRL [25H, 7]
       EXE          // if value in TEMP is larger than BALL_Y, then one will be outputed to the
OUT

       ADDR [03, 8] CTRL [0, 7] EXE // stop register output
       CTRL [01, 7] DATA [TEMP] EXE    // address the cell for temporary data
       DATA 1(32) ADDR [05, 8] CTRL [01, 7] EXE // output the out
       ADDR [03, 8] CTRL [0EH, 7] EXE // write the value
       ADDR [05, 8] CTRL [0, 7] EXE // stop the OUT output


       DATA [128-6]
       WriteTEMP
       ARG0 {! BALL_Y }
       OutputRegister
       // now compare them
       ADDR [04, 8]
       CTRL [27H, 7]
       EXE          // if value in TEMP is smaller than BALL_X, then one will be outputed to the
OUT

       ADDR [03, 8] CTRL [0, 7] EXE // stop register output

       // copy OUT to the TEMP
       DATA 1(32)
       ADDR [05, 8] CTRL [1, 7] EXE // output the OUT
       WriteTEMP
       ADDR [05, 8] CTRL [0, 7] EXE // stop the OUT output


       // output the first value on the bus
       ARG0 {! TEMP }
       OutputRegister
       // now OR them, so 1 is outputted if at one of them is 1, otherwise zero
       ADDR [04, 8] CTRL [18H, 7] EXE
       ADDR [03, 8] CTRL [0, 7] EXE // stop the register output

       // now multiply by -1, so -1 is outputted, when position overflows, zero otherwise
       DATA [-1]
       WriteTEMP
       DATA 1(32)
       ADDR [05, 8] CTRL [01, 7] EXE // output the OUT
       ADDR [04, 8] CTRL [0BH, 7] EXE // signed multiply

       WriteTEMP
       ADDR [05, 8] CTRL [0, 7] EXE // stop the OUT output

       DATA [1]
       ADDR [04, 8] CTRL [29H, 7] EXE // copy 1 to the OUT only if TEMP is zero (so OUT now
contains either -1 or 1)

       // write back to the TEMP
       DATA 1(32)
       ADDR [05, 8] CTRL [1, 7] EXE // output the OUT
       WriteTEMP
       ADDR [05, 8] CTRL [0, 7] EXE // stop the OUT output

       // multiply it with the BALL_YSPD
       ARG0 {! BALL_YSPD }
       OutputRegister
       ADDR [04, 8] CTRL [0BH, 7] EXE // multiply them

       // store the result back
       ADDR [03, 8] CTRL [0, 7] EXE // stop the register output
       CTRL [01, 7] DATA [BALL_YSPD] EXE // address the cell with BALL_YSPD, because the new
value will be written there
       DATA 1(32)
       ADDR [05, 8] CTRL [1, 7] EXE // output the OUT, contaning the new value
       ADDR [03, 8] CTRL [0EH, 7] EXE // write the value
       ADDR [05, 8] CTRL [0, 7] EXE // stop OUT output

             // left paddle detection

              ARG1 {! PADDLE0_Y }
              ARG2 {!
                          DATA [6]
                          ADDR [04, 8]
                          CTRL [27H, 7]
                          EXE
                          }
             ARG3 {! 1 }
             PaddleBounce

             // right paddle detection

              ARG1 {! PADDLE1_Y }
              ARG2 {!
                          DATA [128-6-6]
                          ADDR [04, 8]
                          CTRL [25H, 7]
                          EXE
                          }
             ARG3 {! -1 }
             PaddleBounce

             DetectOutside
}

// detect if the ball left the area
DetectOutside
```

```
{
        // detect left outside
        ARG0 {! BALL_X }
        OutputRegister
        WriteTEMP
        StopRegister
        DATA [0]
        ADDR [04, 8] CTRL [27H, 7] EXE   // if ball left on the left, then OUT is 1
        OUT2TEMP

        // conditional jump
        DATA [LEFTLOSE%]
        ADDR [04, 8] CTRL [2AH, 7] EXE // if OUT is 1 then OUT will contain address of the 
LEFTLOSE
        DATA [LEFTNORMAL%]
        ADDR [04, 8] CTRL [29H, 7] EXE // if OUT is 0 then OUT will contain address of the 
LEFTNORMAL

        // output out
        DATA 1(32)
        ADDR [05, 8] CTRL [01, 7] EXE // output OUT
        ADDR [00, 8] CTRL [01, 7] EXE // write the new address

        LEFTLOSE%:
        CTRL+7 0 // to be safe
        ADDR [05, 8] CTRL [0, 7] EXE // stop OUT output
        ResetBall
        LeftLoseCode
        AJMP [END%, 15]
        AJMP+15(2) !

        LEFTNORMAL%:
        CTRL+7 0 // to be safe
        ADDR [05, 8] CTRL [0, 7] EXE // stop OUT output

        // detect right outside
        ARG0 {! BALL_X }
        OutputRegister
        WriteTEMP
        StopRegister
        DATA [128-6]
        ADDR [04, 8] CTRL [25H, 7] EXE   // if ball left on the right, then OUT is 1
        OUT2TEMP

        // conditional jump
        DATA [RIGHTLOSE%]
        ADDR [04, 8] CTRL [2AH, 7] EXE // if OUT is 1 then OUT will contain address of the 
RIGHTLOSE
        DATA [END%]
        ADDR [04, 8] CTRL [29H, 7] EXE // if OUT is 0 then OUT will contain address of the END

        // output out
        DATA 1(32)
        ADDR [05, 8] CTRL [01, 7] EXE // output OUT
        ADDR [00, 8] CTRL [01, 7] EXE // write the new address

        RIGHTLOSE%:
        CTRL+7 0 // to be safe
        ADDR [05, 8] CTRL [0, 7] EXE // stop OUT output
        ResetBall
        RightLoseCode

        END%:
        CTRL+7 0 // to be safe
        ADDR [05, 8] CTRL [0, 7] EXE // stop OUT output
}
```

```
LeftLoseCode
{
	// increment right score
	ARG0 {! SCORE1 }
	OutputRegister
	WriteTEMP
	StopRegister
	DATA [1] ADDR [04, 8] CTRL [01, 7] EXE // add one
	DATA [SCORE1] ADDR [03, 8] CTRL [01, 7] EXE // write the address
	ADDR [05, 8] DATA 1(32) CTRL [01, 7] EXE // output OUT
	ADDR [03, 8] CTRL [0EH, 7] EXE // write data
	ADDR [05, 8] CTRL [0, 7] EXE // stop OUT output
	UpdateText
}

RightLoseCode
{
	// increment right score
	ARG0 {! SCORE0 }
	OutputRegister
	WriteTEMP
	StopRegister
	DATA [1] ADDR [04, 8] CTRL [01, 7] EXE // add one
	DATA [SCORE0] ADDR [03, 8] CTRL [01, 7] EXE // write the address
	ADDR [05, 8] DATA 1(32) CTRL [01, 7] EXE // output OUT
	ADDR [03, 8] CTRL [0EH, 7] EXE // write data
	ADDR [05, 8] CTRL [0, 7] EXE // stop OUT output
	UpdateText
}

UpdateText
{
	ADDR [0BH, 8] CTRL [09, 7] EXE // address the text display and reset it first
	ARG0 {! strInfo}
	CopyStr
	ARG0 {! strScore0 }
	CopyStr
	ARG0 {! SCORE0 }
	TwoDigitsFromReg
	ARG0 {! endLine }
	CopyStr
	ARG0 {! strScore1 }
	CopyStr
	ARG0 {! SCORE1 }
	TwoDigitsFromReg
	ARG0 {! endLine }
	CopyStr
	ARG0 {! strInfo2 }
	CopyStr
}

// write two digits to the text display from the register memory at address in ARG0
TwoDigitsFromReg
{
	// first digit
	DATA [10]
	WriteTEMP
	OutputRegister
	ADDR [04, 8] CTRL [05, 7] EXE // divide it by 10
	StopRegister
	OUT2TEMP
	ADDR [04, 8] DATA [30H] CTRL [01, 7] EXE // add the value of '0' to it to produce a digit 
character
	ADDR [05, 8] DATA 1(32) CTRL [01, 7] EXE // output out
	ADDR [0BH, 8] CTRL [03, 7] EXE // write the character
	ADDR [05, 8] CTRL [0, 7] EXE // stop the OUT output
```

```
        // second digit
        DATA [10]
        WriteTEMP
        OutputRegister
        ADDR [04, 8] CTRL [06, 7] EXE // module it by 10
        StopRegister
        OUT2TEMP
        ADDR [04, 8] DATA [30H] CTRL [01, 7] EXE // add the value of '0' to it to produce a digit
character
        ADDR [05, 8] DATA 1(32) CTRL [01, 7] EXE // output out
        ADDR [0BH, 8] CTRL [03, 7] EXE // write the character
        ADDR [05, 8] CTRL [0, 7] EXE // stop the OUT output
}

// copy zero terminated string to the display
// ARG0 - start address in the attocode memory
CopyStr
{
        ADDR [01, 8] DATA [ARG0] CTRL [01, 7] EXE // address start of the string

        LOOP%:
        DATA 0(24)1(8)
        ADDR [01, 8] CTRL [03, 7] EXE // output character
        // determine if it's a zero - then end the loop
        WriteTEMP
        ADDR [01, 8] CTRL [0, 7] EXE // stop output
        DATA 1(24)
        ADDR [04, 8]
        DATA [END%] CTRL [29H, 7] EXE // copy the END address if TEMP is zero (zero terminated
string)
        DATA [CONTINUE%] CTRL [2AH, 7] EXE // copy when TEMP is non-zero (contains character)

        // write the address
        DATA 1(32)
        ADDR [05, 8] CTRL [01, 7] EXE // output the OUT
        ADDR [0, 8] CTRL [01, 7] EXE // write new address

        CONTINUE%:
        CTRL+7 0
        ADDR [05, 8] CTRL [0, 7] EXE // stop the OUT output

        // copy the character to the text memory
        ADDR [02, 8] CTRL [04, 7] EXE    // output value from the TEMP (the character)
        ADDR [0BH, 8] CTRL [03, 7] EXE // write the character and move to the next one
        ADDR [02, 8] CTRL [0, 7] EXE // stop the TEMP output

        ADDR [01, 8] CTRL [07, 7] EXE // move to the next character

        AJMP [LOOP%, 15] AJMP+15(2) !    // maintain the cycle

        END%:
        CTRL+7 0
        ADDR [05, 8] CTRL [0, 7] EXE // stop the OUT output
}

ResetBall
{
        ADDR [03, 8]
        CTRL [01, 7] DATA [BALL_X] EXE DATA [64] CTRL [0EH, 7] EXE

        ARG0 {! BALL_Y }
        OutputRegister
        WriteTEMP
        StopRegister

        ADDR [03, 8] CTRL [01, 7] DATA [BALL_XSPD] EXE // address BALL_XSPD
        ADDR [02, 8] CTRL [04, 7] EXE // output TEMP
        ADDR [03, 8] DATA 0(31)1 CTRL [0EH, 7] EXE    // write the value
        ADDR [02, 8] CTRL [0, 7] EXE // stop TEMP output

        ADDR [03, 8] CTRL [01, 7] DATA [BALL_YSPD] EXE // address BALL_YSPD
        ADDR [02, 8] CTRL [04, 7] EXE // output TEMP
        ADDR [03, 8] DATA 0(30)10 CTRL [0EH, 7] EXE   // write data
        ADDR [02, 8] CTRL [0, 7] EXE // stop TEMP output

        ADDR [03, 8] CTRL [01, 7] DATA [BALL_Y] EXE  // address BALL_Y
        ADDR [02, 8] CTRL [04, 7] EXE // output TEMP
        ADDR [03, 8] DATA 1(32) CTRL [0EH, 7] EXE
        ADDR [02, 8] CTRL [0, 7] EXE // stop TEMP output

        // now alter the BALL_XSPD and BALL_YSPD
        ARG0 {! BALL_XSPD }
        OutputRegister
        WriteTEMP
        StopRegister
        // copy either 1 or -1
        ADDR [04, 8] DATA [1] CTRL [29H, 7] EXE
        ADDR [04, 8] DATA [-1] CTRL [2AH, 7] EXE
        DATA [BALL_XSPD] ADDR [03, 8] CTRL [01, 7] EXE // address register
        ADDR [05, 8] DATA 1(32) CTRL [1, 7] EXE // output out
        ADDR [03, 8] CTRL [0EH, 7] EXE // write the new value
        ADDR [05, 8] CTRL [0, 7] EXE // stop the OUT output

        ARG0 {! BALL_YSPD }
        OutputRegister
        WriteTEMP
        StopRegister
        // copy either 1 or -1
        ADDR [04, 8] DATA [1] CTRL [29H, 7] EXE
        ADDR [04, 8] DATA [-1] CTRL [2AH, 7] EXE
        DATA [BALL_YSPD] ADDR [03, 8] CTRL [01, 7] EXE // address register
        ADDR [05, 8] DATA 1(32) CTRL [1, 7] EXE // output out
        ADDR [03, 8] CTRL [0EH, 7] EXE // write the new value
        ADDR [05, 8] CTRL [0, 7] EXE // stop the OUT output
}

// handle boucing from either paddle
// ARG1 - which paddle
// ARG2 - X axis detect code (only set DATA and do ALU stuff, TEMP contains ball X)
// ARG3 - new direction
PaddleBounce
{
        // first, calculate if it's in the range of the paddle (below and above paddle's size)
        ARG0 {! ARG1 }
        OutputRegister
        WriteTEMP
        StopRegister
        DATA [-5]
        ADDR [04, 8] CTRL [09H, 7] EXE // subtract 5 from the paddle Y (so it can bounce from the
edge)
        OUT2TEMP
        // TEMP now contains the upper position, now check if it's above ball position
        ARG0 {! BALL_Y }
        OutputRegister
        ADDR [04, 8] CTRL [27H, 7] EXE // check if the BALL_Y is below PADDLE_Y
        StopRegister

        // store it in TEMP location in the register memory
        ADDR [03, 8] DATA [TEMP] CTRL [01, 7] EXE // write the address
        DATA 1(32) ADDR [05, 8] CTRL [01, 7] EXE // output OUT
        ADDR [03, 8] CTRL [0EH, 7] EXE   // the result is now stored
        ADDR [05, 8] CTRL [0, 7] EXE // stop OUT output

        // BOTTOM OF THE PADDLE
```

```
	ARG0 {! ARG1 }
	OutputRegister
	WriteTEMP
	StopRegister
	DATA [18]
	ADDR [04, 8] CTRL [09H, 7] EXE // add 18 to the value (paddle is 18 pixels tall)
	OUT2TEMP
	// TEMP now contains the bottrom possition, now check if it's below ball position
	ARG0 {! BALL_Y }
	OutputRegister
	ADDR [04, 8] CTRL [25H, 7] EXE
	StopRegister
	OUT2TEMP

	// now AND both these together - they both must be true
	ARG0 {! TEMP }
	OutputRegister
	ADDR [04, 8] CTRL [17H, 7] EXE // Logical AND
	StopRegister

	// store the result in TEMP location once again, because it will be needed soon
	ADDR [03, 8] DATA [TEMP] CTRL [01, 7] EXE // write the address
	ADDR [05, 8] CTRL [1, 7] EXE // output out
	DATA 1(32)
	ADDR [03, 8] CTRL [0EH, 7] EXE // write the value
	ADDR [05, 8] CTRL [0, 7] EXE // stop the OUT output

	// now detect, if the ball is touching the paddle on the X axis
	ARG0 {! BALL_X }
	OutputRegister
	WriteTemp
	StopRegister
	ARG2 // detection is handled by an external code
	OUT2TEMP

	// now AND it with the value in the TEMP, to produce final value, determining whether or
not to bounce
	ARG0 {! TEMP }
	OutputRegister
	ADDR [04, 8] CTRL [17H, 7] EXE   // AND
	StopRegister
	OUT2TEMP

	// now calculate new BALL_XSPD based on the calculated conditional value
	DATA [ARG3]
	ADDR [04, 8] CTRL [2AH, 7] EXE // if TEMP is nonzero, copy value from DATA to the OUT
	ARG0 {! BALL_XSPD }
	OutputRegister
	ADDR [04, 8] CTRL [29H, 7] EXE // copy current speed if TEMP is zero (no collision -
maintain regular speed)
	StopRegister

	// write the calculated speed to the BALL_XSPD
	ADDR [03, 8] CTRL [1, 7] DATA [BALL_XSPD] EXE // address the propel cell
	DATA 1(32) ADDR [05, 8] CTRL [1, 7] EXE // output the OUT
	ADDR [03, 8] CTRL [0EH, 7] EXE // write the value
	ADDR [05, 8] CTRL [0, 7] EXE // stop the OUT output
}

/*	************************************
				PROGRAM START
	************************************ */

// INITIALIZE EVERYTHING

0 0(64)
```

```
// enable double buffering
ADDR [0CH, 8]
CTRL [0BH, 7]
EXE

ADDR [03, 8] CTRL [01, 7] DATA [PADDLE0_Y] EXE DATA [64-9] CTRL [0EH, 7] EXE
ADDR [03, 8] CTRL [01, 7] DATA [PADDLE1_Y] EXE DATA [64-9] CTRL [0EH, 7] EXE
ADDR [03, 8] CTRL [01, 7] DATA [SCORE0] EXE DATA [0] CTRL [0EH, 7] EXE
ADDR [03, 8] CTRL [01, 7] DATA [SCORE1] EXE DATA [0] CTRL [0EH, 7] EXE

ResetBall
UpdateText

LOOP:
// cleanup after jump
CTRL+7 0
ADDR [0, 8] CTRL [0, 7] EXE

// game logic
UpdateBall
ProcessAllKeys
ARG0 {! PADDLE0_Y }
ARG1 {! 0 }
LCDPaddleStart
DrawPaddle
ARG0 {! PADDLE1_Y }
ARG1 {! 128-6 }
LCDPaddleStart
DrawPaddle
DrawBall

// switch buffer
ADDR [0CH, 8]
CTRL [0CH, 7]
EXE
CTRL [09, 7]
EXE

// long jump
DATA [LOOP]
ADDR [0, 8] CTRL [01, 7] EXE

strInfo:
"              attoPong 1.0              " $00
strInfo2:
"Programmed by Tomas \"Frooxius\" Mariancik" $00
strScore0:
"           Player 0 score: " $00
strScore1:
"           Player 1 score: " $00
endLine:
"           " $00
```