4COM1042 [Computing Platforms]

Co-design Group Project A

“The Game of Noughts and Crosses”

Notes

Alex Shafarenko and Stephen Hunt, 2016

Overview

Your task for this project is to implement a *noughts and crosses* game (aka tic-tac-toe) in Logisim. You will develop a 3x3 game pad, connect it to a full-core CdM-8 system with a memory-mapped I/O, and write an assembly language program to manage the game. The full-core CdM-8 system employs separate code and data memory (Harvard architecture), so your program will need to be written with this in mind.

Each of the 9 cells on the game pad has sufficient electronics to drive a 4x4 pixel LED display, which can show a cross, or a nought, or an empty space. Each cell also has an input button that may be used by a human player to play his/her turn in that cell. The game pad has controller chips that can be used to connect it to the I/O bus of the CdM-8 processor, so that it may control the game.

The design brief contains specifications for a set of devices sufficient to build a game pad, and a schematic showing how to connect them together. However, you may design and implement your own game pad if you wish.

Basic system

Your program should implement the gameplay for noughts and crosses and have a very basic strategy. You should get it to

- deal with inputs in the form of button presses;
- generate outputs that cause the right LED displays to show the right patterns for noughts and crosses at the right times;
- keep track of which cells contain noughts, which contain crosses, and which are empty (because the overall state of the game cannot be read from the game pad);
- enforce turn-taking;
- choose where to play each nought in response to the human player's crosses (we suggest making this *very* basic at the outset: don't worry about programming a strategy for winning);
- determine when the game is over and the result (computer wins, computer loses, players draw) and give an indication of what the result is;

Whilst this is all pretty basic, the resources you have at your disposal are very limited indeed, so you will have to make sacrifices.

If in doubt, concentrate your effort on getting *something* working, no matter how simple.

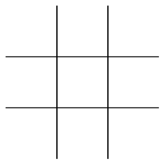
Suggested extensions

Once you have the basic game working you might like to extend it in some way. Just make sure you keep a *copy* of each working version!! You could try to

1. add a reset button that allows the human to return the game to its starting state (all cells empty, PC set to 00) at any time;
2. add an LED that lights up when the human player attempts an illegal move;
3. modify the game so that when either player wins a line is drawn through the three symbols in the winning line;
4. improve the game strategy of the computer (you choose how)
5. create a game that uses a 4x4 board.

A basic game of noughts and crosses

The game is very familiar. It is played on a 3 by 3 grid, usually drawn like this



We have simplified things a little for the project, and the rules your game should enforce are as follows:

1. The human player always plays first, and always plays crosses.

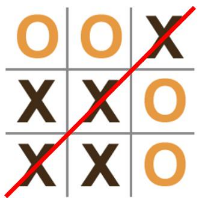
2. At each turn the human plays a cross in one of the empty grid squares. This square is now occupied, so is no longer available for play.

3. The computer responds by playing a nought in one of the remaining empty squares.

4. The players take turns, playing crosses (human) and noughts (computer) in the remaining empty squares until the game is finished.

5. A game finishes when any one of three situations occurs:
   i) there is a line of three crosses on the board (human wins)
   ii) there is a line of three noughts on the board (computer wins)
   ii) all nine squares are occupied and there is no line of three identical symbols (game drawn)

6. Any line of three identical symbols counts as a win for one of the players: horizontal, vertical or diagonal. Note, also, that it is possible for the game to be won even when all nine squares are occupied. See below for an example:



Those are all the rules the game has.

The project is not about producing a system that employs the best possible strategy: the resources at your disposal are insufficient for a full solution. 256 bytes is not enough code memory, and the Logisim simulation is too slow to make the full game playable by any but the most patient of players.

What we are looking for is a *working* noughts and crosses system that uses a simulated hardware game pad and maintains correct gameplay.

Hardware

The hardware employed for this project consists of a full-core split-memory CdM-8 system with an I/O bus, to which a 9 cell *game pad* is connected. The game pad uses one I/O address.

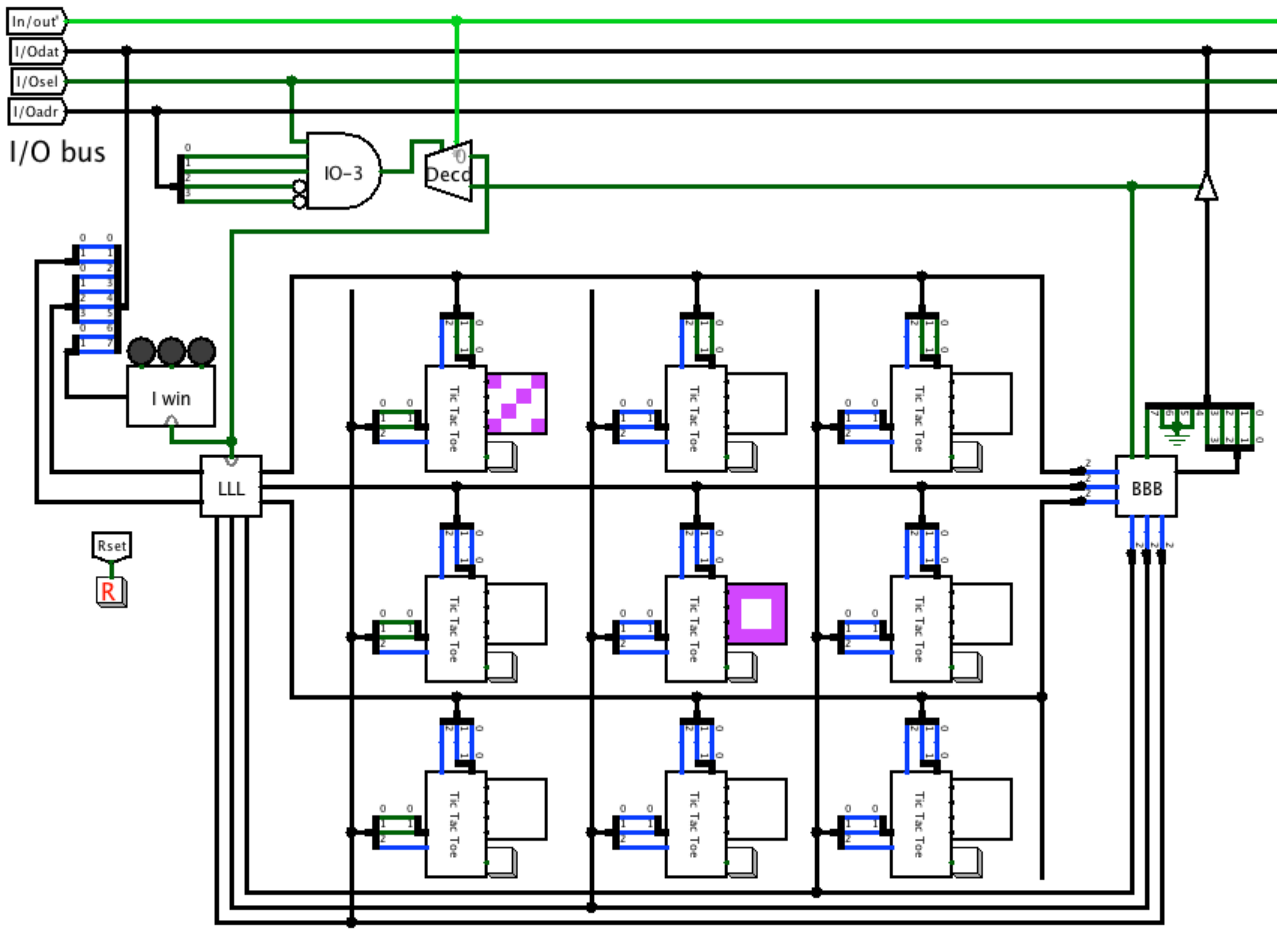
Game pad

Below we show a prototype game pad. It employs a *crossbar* arrangement, with three horizontal 3-bit buses (H0, H1 and H2) and three vertical 3-bit buses (V0, V1 and V2). Each cell is connected to one 3-bit horizontal bus, and to one 3-bit vertical bus.

Each cell has an x-coordinate (horizontal position, left to right) in the form of a 2-bit string, **xx**, representing 0, 1 or 2, and a 2-bit y-coordinate (vertical position from top to bottom), **yy**, representing 0,1 or 2. These correspond to the numbered buses, so the cell with **xx** = 0b00 and **yy** = 0b10 is connected to buses H0 and V2. The x- and y-coordinates are concatenated to give each cell a different four-bit address: **xxyy**.

In the circuit diagram below the cell displaying a cross has the address 0b0000, the cell displaying a nought has the address 0b0101, and the cell displaying a blank to the south of the cell displaying a nought has the address 0b0110.

Bits 0 and 1 of each bus are used for output from the CPU to the attached cells and bit 2 is used for input from the attached cells to the CPU. Every cell is connected to a different *pair* of buses, so an *input from* a specific cell will be asserted on a unique pair of buses, and we can send an *output to* a specific cell by picking which pair of buses to assert.



Individual cells

Every cell contains a 4x4-pixel LED matrix display for output and a button for input. These are controlled by a **Tic Tac Toe Cell** (TTTC) chip which is connected to the buses.

- On the North side of the TTTC chip (running East to West) we have a 1-bit pin labelled **`hout`**, and a 2-bit pin labelled **`hin`**. All three bits are connected to the cell's horizontal bus (**`hout`** to wire 2 and **`hin`** to wires 0 and 1). Note that pins are always labelled from the perspective of the device that 'owns' them, so the pin labelled **`hout`** carries *output from* the TTTC chip, and **`hin`** carries *input to* the TTTC chip.
- On the West side of the TTTC chip there is a 2-bit **`vin`** and a 1-bit **`vout`**, for connection to the relevant vertical bus.
- A 4x4 LED matrix display is connected directly to the TTTC chip's output pins **`L0, L1, L2, L3`** (4 bits each) on its East side. These drive rows 0, 1, 2 and 3 of the matrix respectively, to display a nought, a cross, or a blank
- A button is provided for a human to play a cross in the cell. It is connected to a 1-bit input pin on the East side of the TTTC chip, labelled **`btn`**.

In reality a TTTC chip is actually just a convenient box into which two different functional units are *packaged*. One of these units is a combinational circuit that deals with a single button and the other unit contains both combinational and sequential elements and deals with a single 4x4 pixel LED display. These two units do not need to have any circuit elements in common.

The specification for a TTTC chip appears in the Appendix

Driving the game pad

In addition to the 9 TTTC chips there are two controller chips on the game pad, each of which is connected to all three horizontal buses and all three vertical buses.

Button input

The first of these is the **Button Press Capture** (BPC) chip. Its job is to capture and hold the address of the cell in which the most recent button press occurred. This data may then be read by the processor.

The BPC chip is sequential. It has a 4-bit register inside, which latches the grid address of the cell containing the last button that was pressed. Each of the pins **`H0`**, **`H1`**, **`H2`** and **`V0`**, **`V1`**, **`V2`** is connected to bit 2 on one of the six buses. The BPC chip is not connected to bits 0 and 1.

Symbol output

The second is the **Symbol Display Router** (SDR) chip. It is used to convey the ID of a symbol to be displayed on the game pad to the correct TTTC chip controlling a given cell. The symbol ID and the xy coordinate of the cell come from the processor.

The SDR chip is entirely combinational. It has six 2-bit output pins **`H0`**, **`H1`**, **`H2`** and **`V0`**, **`V1`**, **`V2`**, each of which is connected to bits 0 and 1 of one of the buses. It is positioned to the west of the crossbar.

When the processor needs to write a symbol (nought, cross or space) into the cell with coordinates xxyy it must send both a 2-bit symbol ID and the cell address xxyy to the SDR chip. The cell address is decoded by the SDR chip, and used to select the right horizontal and vertical bus pair for the cell, this routing the symbol ID to the correct TTTC chip.

How does the correct symbol ID get into the correct TTTC chip?

Bit 0 of the symbol ID is directed to bit 0 of the selected horizontal bus, and bit 1 of the symbol ID is directed to bit 0 of the selected vertical bus. So the least significant bits of the two buses carry the ID of the symbol to be displayed.

The TTTC chip also needs a trigger to latch the symbol ID into its internal 2-bit register. The trigger is provided on bit-1 of the relevant pair of horizontal and vertical buses at the same time, each of which is raised then dropped after the data has been asserted. This latches the data into the register. Only one cell on the crossbar will have bit-1 of both its horizontal and vertical buses raised at the same time, so only one cell will latch the data despite the fact that each bus is connected to 3 cells.

Where does the data come from?

It is asserted onto the **`I/Odat`** bunch of the CdM-8 I/O bus. Since the bunch is 8-bit, the question arises of how the necessary information is packed into the byte. Here is how:

bits	meaning
0 – 1	symbol ID
2 – 5	cell address
6 – 7	game score

The symbol ID and cell address are received by the SDR chip on the west side, via a 2-bit pin **`symID`** and a 4-bit pin `XY`.

Indicating that the game is over

Bits 6 and 7 of the **`I/Odat`** bunch are used to drive the **Game State Display Driver** (GSDD) chip, which lights an LED indicting win/lose/draw when the game is over. During play the processor sets both bits to 0. On the last turn (the one that results in a win, a loss, or a draw) the processor outputs a 2-bit signal on these two wires that identifies which of the LEDs should be lit.

The GSDD chip is sequential, but pretty straightforward. It is triggered by the same signal as the SDR chip uses to latch the TTTCs, at which point the 2-bit string in bits 6 and 7 of **`I/Odat`** is latched into an internal 2-bit register. The value stored in the register is used to select which – if any – of the three LEDs is lit.

Finally, the IO bus is shared with the memory bus (it is *memory-mapped* I/O after all). This means that the wires that form the **`I/Odat`** bunch are constantly in use and the game pad only needs to do anything if it is commanded to do so. The command comes from the IO bus line **`I/Osel`**, which is raised when the processor requires I/O. At the same time the 4-bit address of the device to be used is asserted on the **`I/Oadr`** bunch. The address in the prototype implementation is chosen to be 0x3, but any 4-bit address would do as long as the program knows it, since the game machine will not have any other I/O devices.

Another line of the I/O bus, **`In/out'`**, is used as a selector: when *high* it signals that the processor is reading data (in), and when *low*, that it is writing data to the device (out). At the top of the diagram the gate and the decoder do the job of interfacing with the computer side (on the east). The correct combination of signals is fed to the clock input of the SDR chip (its north side), and this is the trigger that it passes on to the TTTC chip selected by bits 2..5 of the **`I/Odat`** bunch.

Suggested hardware design progression

1. The best way to start building the pad is to start with a single TTTC chip, attach test pins and LED matrix to it and see if you can latch spaces, noughts and crosses. Check that the button press has the desired effect on the outputs.

2. Build the SDR chip and connect it to the crossbar of Tic-Tac-Toes. Check that you can make any one of the 9 displays change its content.

3. Now build the BPC chip and connect it to the crossbar. See if you can add a circuit that places a cross in the cell that you press the button in. This would require the additional circuit (just a few splitters and some wires) to be placed between BPC and SDR chips.

4. Now complete the interface by adding the and gate, the controlled buffer and the decoder as indicated in the circuit diagram. Once you have these in place you can connect your device to the I/O bus.

5. Write a program that places a nought or a cross in an arbitrary cell. Load it in the CdM-8 instruction ROM and run it. See if it works. If it does not work, clock through it by pressing on the processor clock. Watch the registers and the bus values.

6. Tell your team mates that they can start testing *their* software for the gameplay. Be prepared to show them they are wrong if their software cannot drive your game pad. Meanwhile quickly build and test the GSDD chip and connect it to the design.

Software

The data structure

The board may be represented by a 3x3 array; however, this would make it difficult to address individual cells (would require either multiplication by 3 or a table look-up). You may be short of instruction memory but you have plenty of data memory at your disposal, so we suggest you use a 4x4 array and ignore the fact that almost half of it will go unused.

It makes sense to place the array at the bottom end of data memory, from address 0.

Each element of the array represents the contents of one square. We suggest you use an integer in the range 0 to 2 (0=space, 1=nought, 2=cross).

Main loop

```
1. Wait for button press (use a loop to keep checking)
2. Get cell coordinates for the button
3. If cell already occupied (contains ‘nought’ or ‘cross’)
      a. Ignore button and go to 1
4. Mark new content of cell in game array and output a ‘cross’ to the cell
5. If game is not over (lost or drawn)
      a. Work out where computer will move
      b. Mark new content of cell in game array and output a ‘nought’ to the cell
6. If game is over (won, lost or drawn)
      a. Display outcome on “I win”
      b. Halt
   else go to 1
```

Subroutines

You may not need these as *subroutines* but it helps to structure and debug your code.

1) A subroutine that determines the game state (win, lose or draw). It will scan the game array and determine whether it contains a line of three crosses, or a line of three noughts, or a ‘full house’ of nine noughts and crosses.

One way to spot lines is to include a table in the program which lists all possible lines of three in terms of cell numbers (converted from their 4-bit addresses), and make comparisons with the current board state

```
table:    # each triplet below represents a line of three cells
   dc 0,1,2          # horizontal lines
   dc 4,5,6
   dc 8,9,10
   dc 0,4,8          # vertical lines
   dc 1,5,9
   dc 2,6,10
   dc 0,5,10         # diagonal lines
   dc 8,5,2
```

You will require the ‘ldc’ instruction to read table elements from code memory, but otherwise the technique is the same as that of using the ‘ld’ instruction.

2) A subroutine that determines in which cell the computer should play its next nought. This also needs to scan the game array. At its simplest it may just choose the first empty cell it finds. A more complex algorithm might play defensively (block the human from making a line of three), or even attempt to make a line of three for the computer.

Suggested software design progression

1. Use the **cocoemu** emulator for debugging your code thoroughly before interfacing it with the game pad. You should use the I/O address 0xf3 as if it were a normal memory address and simulate button presses by placing a constant in it. Make 2 versions, one with “ld” instructions to read the table of triples, and the other with “ldc”. Cocoemu does not implement the ldc instruction as it emulates a reduced core processor, only the full core) does.

2. Write a subroutine for placing noughts and crosses onto the gamepad and replace it by a testing version, which places them in a memory array 4x4 of the same shape as the gamepad. Then you will be able to satisfy yourself that the gameplay is correct and the strategy works before your co-design partners in the Hardware Department construct a usable pad.

3. Write a test suite (just a few little programs) to test that the pad works when the Hardware Department releases it. Do it in advance, having discussed the interface spec with them first (in case they are not following our very sensible design recommendations). Make sure you can argue with them that it is all their fault if it doesn’t work, and have evidence to support your findings.

4. Remember that it is more important that you have a working program than it is that you have a brilliant one. Start low and be constantly aware of the schedule and its deadlines. Also remember that nothing stimulates a hardware designer more than the fact that some software is available and is ready to be deployed on their product.

GOOD LUCK!

Appendix: Chip specifications

The **TTTC** chip *(labelled TicTacToe)* contains two separate functional units: one unit is a purely combinational circuit that deals with a single button and the other contains both combinational and sequential elements and deals with a single 4x4 pixel LED display.

```
=======================================================================
Chip Specification
   Name: Tic-Tac-Toe Cell (TTTC)

I/O Pins
   Inputs:   vin(2), hin(2), btn
   Outputs:  vout, hout, L0(4), L1(4), L2(4), L3(4)

Internal
   Signals:  symID(2), addr(2), bitmap(16)
   Latches:  register(2) sym_reg
   ROMs:     symbols(2->16)          # A look-up table of 16-bit strings
                                     # each with a 2-bit address

ROM contents
   symbols [0] =                     # Fill in bit-string for blank here
   symbols [1] =                     #  and for nought here
   symbols [2] =                     #  and for cross here
   symbols [3] =                     # This table entry is not used,
                                     # but it must contain something
                                     # (ROM cells cannot be empty)

Combinational
   symID.split(0) = hin.split(0)     # LSbs of hin & vin supply the ID
   symID.split(1) = vin.split(0)     # of the symbol to be displayed
                                     # This will be latched into sym_reg
                                     # (see below)
   addr = sym_reg                    # Address in ROM is ID of symbol
                                     # to display
   bitmap = symbols[addr]            # bitmap is read from ROM using
                                     # that address
   L0 = bitmap.split(0:3)            # Outputs for 4 rows display
   L1 = bitmap.split(4:7)            # are obtained from the
   L2 = bitmap.split(8:11)           # 4 bits per row...
   L3 = bitmap.split(12:15)

   hout = if btn then 0 else float   # use an N-type transistor
   vout = if btn then 0 else float   # for these

Behaviour
     on rising_edge of (hin.split(1) ∧ vin.split(1))
     do:
       sym_reg := symID              # Latch new symbol ID from CPU

End Chip Specification
========================================================================
```

The **BPC** chip *(labelled BBB)* is sequential. It has a 4-bit register inside, which latches the number of the last button pressed on the pad. The chip has input pins **`H0`**, **`H1`**, **`H2`** and **`V0`**, **`V1`**, **`V2`**, each connected to bit-2 of one of the horizontal or vertical buses. It is not connected to bits 0 and 1 of any bus.

The selector *weak-if* used here treats *floating* as false. When V0 is *low* V0' must be *high*, but V1 and V2 will be *floating*, which means V1' and V2' will be *floating* too. Pull resistors must therefore be used, either to drive V1 and V2 *high*, or to drive V1' and V2' *low*.

```
========================================================================
Chip Specification
   Name: Button Press Capture (BPC)

I/O Pins
   Inputs:   H0, H1, H2, V0, V1, V2, reset
   Outputs:  ready, XY(4)

Internal
   Signals:  xcoord(2), ycoord(2), XY(4), coord(4), b_press, btn, rst
   Latches:  register(4) XY_reg, RS ready_flag

Combinational
   ycoord = weak-if V0' then 0b10          # These three are NOT
            weak-if V1' then 0b01          # conflicted
            weak-if V2' then 0b00          # because only one V can
                                           # be low at a time

   xcoord = weak-if H0' then 0b10          # Similarly
            weak-if H1' then 0b01          # only one H can
            weak-if H2' then 0b00          # be low at a time

   b_press = (V0^V1^V2)' ^ (H0^H1^H2)'     # b_press is high when both
                                           # a V and an H are low
   coord.split(0:1) = ycoord
   coord.split(2:3) = xcoord

   XY = XY_reg
   ready = ready_flag

Behaviour
     on rising edge of b_press do:
                    pulse  btn
     on falling edge of reset do:
                    pulse rst

     on rising_edge of b_press do:    # When button is depressed
                    XY_reg := coord   # store XY coord of that button
                                      # in XY_reg

     on btn do:                       # When a button is pressed
              set ready_flag          #  the BBB's ready flag is raised
     on rst do:                       # When the BBB is reset by a read
              reset ready_flag        #  its ready flag is lowered

End Chip Specification
========================================================================
```

The **SDR** chip *(labelled LLL)* is entirely combinational. It is used to convey the ID of a symbol to be displayed on the game pad to the correct TTTC chip controlling a given cell. The symbol ID and the xy coordinate of the cell come from the processor. This chip has six 2-bit output pins **H0**, **H1**, **H2** and **V0**, **V1**, **V2**, each of which is connected to bits 0 and 1 of one of the buses. It is positioned to the west of the crossbar.

```
========================================================================
Chip Specification
   Name: Symbol Display Router (SDR)

I/O Pins
   Inputs:   symID(2), XY(4), clock
   Outputs:  H0(2), H1(2), H2(2), V0(2), V1(2), V2(2)

Internal
   Signals:  xcoord(2), ycoord(2)

Combinational
   xcoord = XY.split(0:1)           # XY carries the x,y coordinates of
   ycoord = XY.split(2:3)           # the TTTC chip to be used

   # symID collects the identity of the symbol to be displayed
   # The XY coordinates are used to route the combination of symID
   # and clock pulse to the correct TTTC chip
   # Five TTTC chips will receive a clock pulse, but only one will
   # receive simultaneous clock pulses on both hin and vin. This will
   # cause it to latch the ID of the symbol to be displayed

   H0.split(0) = if xcoord is 0b00 then symID.split(0) else float
   H0.split(1) = if xcoord is 0b00 then clock else float
   H1.split(0) = if xcoord is 0b01 then symID.split(0) else float
   H1.split(1) = if xcoord is 0b01 then clock else float
   H2.split(0) = if xcoord is 0b10 then symID.split(0) else float
   H2.split(1) = if xcoord is 0b10 then clock else float

   V0.split(0) = if ycoord is 0b00 then symID.split(1) else float
   V0.split(1) = if ycoord is 0b00 then clock else float
   V1.split(0) = if ycoord is 0b01 then symID.split(1) else float
   V1.split(1) = if ycoord is 0b01 then clock else float
   V2.split(0) = if ycoord is 0b10 then symID.split(1) else float
   V2.split(1) = if ycoord is 0b10 then clock else float

End Chip Specification
========================================================================
```

The **GSDD** chip *(labelled I Win)* is a simple sequential circuit containing a latch that holds the current state of the game from the computer's perspective (*in progress*, *won*, *drawn*, or *lost*). This chip drives a 3-LED display containing a green, an amber and a red LED. When the game state is *in progress* all LEDs are off. When the game state is *won* the green LED is lit and the others are off. When the game state is *drawn* the yellow LED is lit and the others are off. When the game state is *lost* the red LED is lit and the others are off.

```
========================================================================
Chip Specification
   Name: Game State Display Driver (GSDD)

I/O Pins
   Inputs:   score(2), clock
   Outputs:  win, lose, draw

Internal
   Latches:  register(2) state

Combinational
   lose = if state is 0b00 then 1 else 0
   win  = if state is 0b01 then 1 else 0
   draw = if state is 0b11 then 1 else 0

Behaviour
   on rising_edge of clock do:
                       state := score

End Chip Specification
========================================================================
```