# 4COM1042 [Computing Platforms]

# Co-design Group Project C

# “The Game of Mancala”

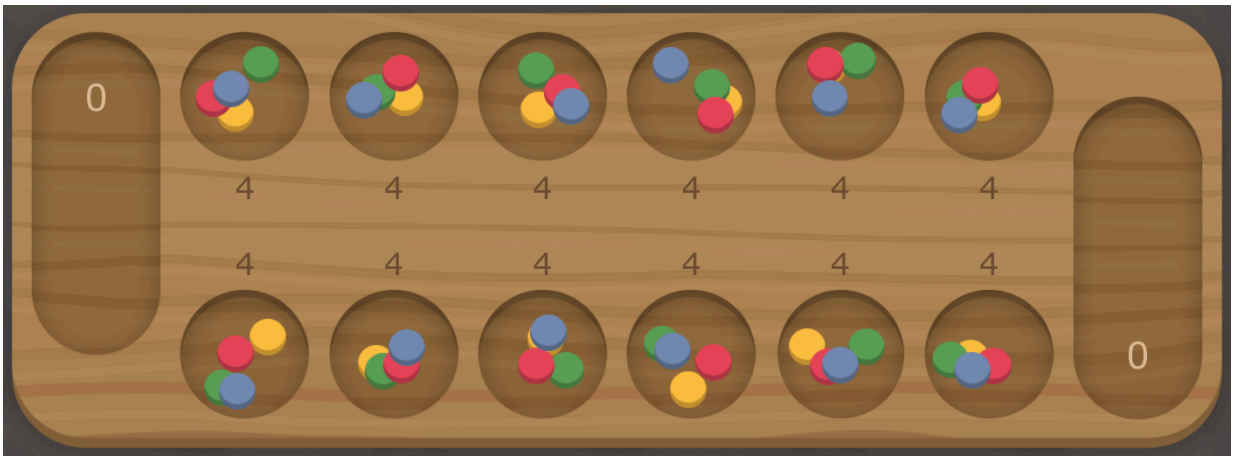
*Notes*

Alex Shafarenko, 2016

## Summary

This project implements a strategy game known most broadly as Mancala. On the hardware side the project builds a Mancala console interfaced with CdM-8. The console lacks intelligence (other than the detection of win or lose which relies on the wire-OR technique). The main challenge is thus to ensure that the gameplay is implemented in software. By the time that is done, there will be very little code memory left for strategy. The project may be limited in this part to a couple of simple heuristics, rather than anything in the spirit of minimax play, alpha-beta pruning etc.

## Preliminaries




The above is a computer rendition of the board for playing the classical game Mancala, taken from http://play-mancala.com/. It has 6 pits (or holes) on either side of the board and two stores, also known as “mancalas”.

The game originated, according to some theories, in ancient Egypt. Whether it is so or not, it is certainly thousands of years old. It has many versions, which are played all over the world, but most of all in Africa. See Appendix.

Here are the rules of the most common version:

1. Two players play the game facing each other. A player’s holes are in front of the player and the player’s mancala is to his or her right. Initially four seeds are placed in each hole. That amounts to 48 seeds in total. (They could be the same or different colours, a mixture of seeds and pebbles — it does not matter.) Players take turns to make a move.

2. The move consists in taking all the seeds out of one of the nonempty holes that belong to the player and spreading them anticlockwise. One seed is dropped in each hole or the player’s mancala on the way, but the opponent’s mancala is bypassed.

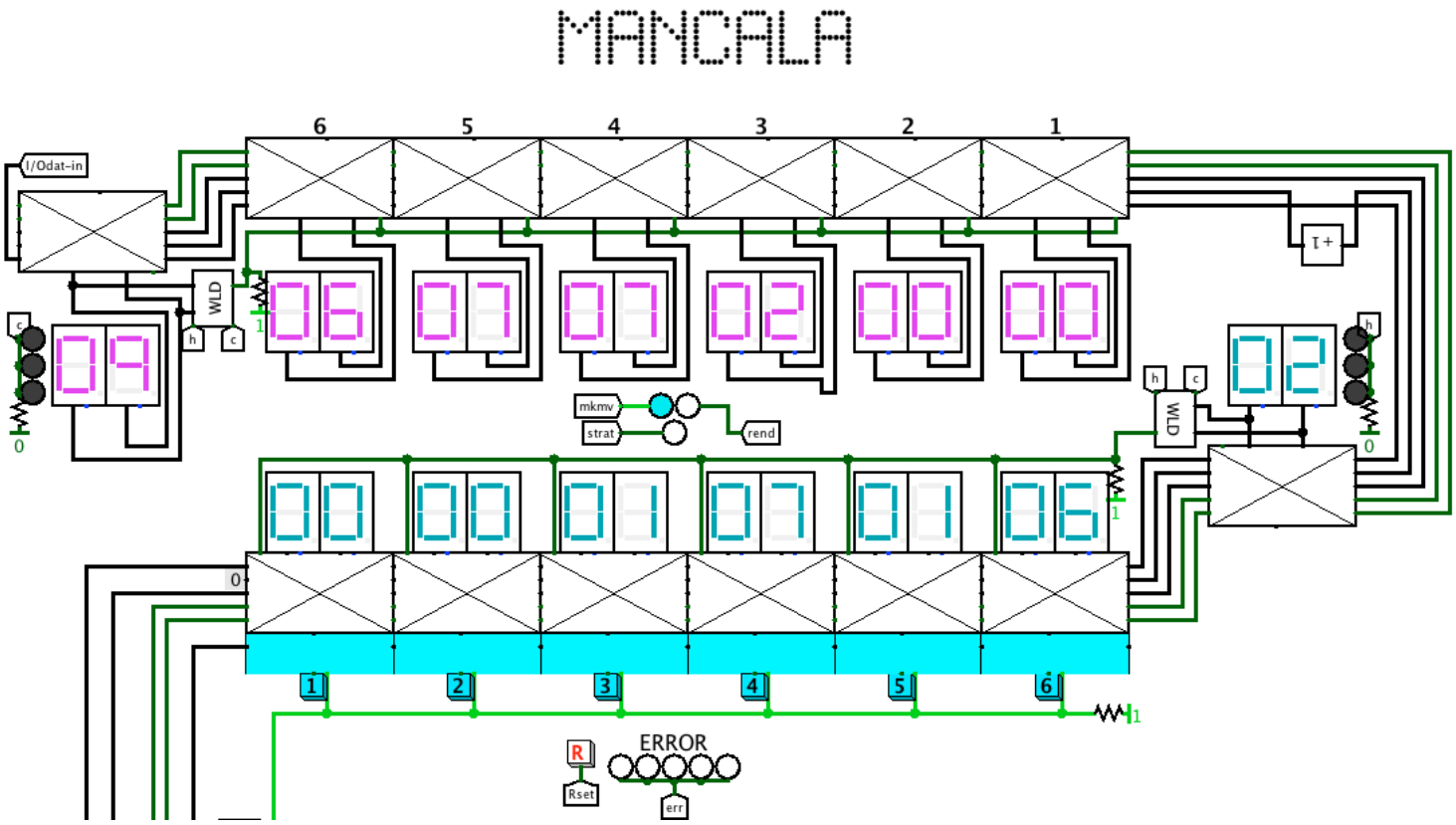
3. [Capture rule] If the last seed of a move drops in the player’s own empty hole and the opposite hole is nonempty, that seed and all the seeds from the opposite hole are captured and transferred to the player’s mancala.

4. [Repeat rule] If the last seed of a move drops in the player’s mancala the player makes another move.

The game stops when one of the players has no seeds/can’t make a move. If their mancala has 24 seeds, it is a draw, more: a win, less: a loss.

Those are in fact the rules of a version called Kalah(6,4), see https://en.wikipedia.org/wiki/Kalah

## Game Pad




The above is a Logisim implementation of a Mancala game pad. Only three uncomplicated chips need to be designed.

**Display driver.** The criss-crossed boxes are display drivers: they have a register inside that holds the value to be displayed. The chip takes all the values it needs from the west side; it passes them over to the east side for the next chip in chain. As we do in other projects, we maintain a cellular design here to avoid a jumble of wires. The chips are differentiated by a signal **pos** fed into the chip from the west (second pin from the top), which gets incremented before being passed over to the corresponding eastern pin. Here is a complete inventory of the west- and east-side pins, listed top down:

| Name | Description |
|---|---|
| data | Display data, 8 bit |
| pos | Chip position in chain, 4 bit |
| addr | Address (chip position) for read/write operations |
| clock | Trigger for writing into display register |
| in/out’ | Read/write selector |

Essentially the display driver is little more than an I/O register, similar to the one presented in the document "**Working with a full-core CdM-8 system**". Here are the add-ons:

The chip has two output pins on the north side, 4-bits each. Those drive hex displays (using only the decimal figures). The decimal to BCD conversion is done inside the chip (possibly by brute force, using Logisim's own division chip).

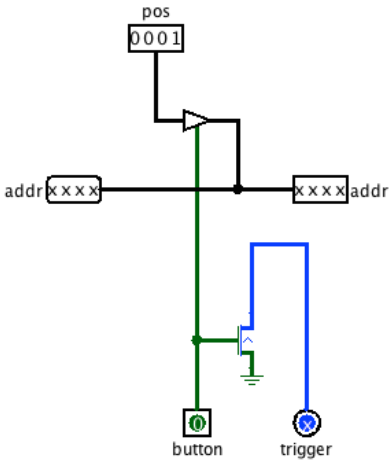
There is an additional output pin on the north side, Z, which is there to help detect that the player's holes are all empty using wire-OR. That helps to establish that the game has ended. It can be done in software of course, but the reference implementation ran out of code memory ☺. The Z pin is asserted with 0 if the register content is greater than 0 or left floating otherwise.

The display driver chip has a 4-bit pin on the south side, on which it asserts its position in the chain. This is required by the chip that is attached to the south edge of the display driver: the button driver, which we will consider next, but first here is the pseudo-code specification of the display driver:

```
Chip Specification
   Name: Display driver
   Depends on combinational chips: BCD(8->4,4)

I/O Pins
   Input:   data(8), pos(4), addr(4), clock, dir      # West side
   Output:  dataout(8)=data, posout(4),               # East side
   Output:  addrout(4)=addr, clockout=clock,dirout=dir#    "
   Output:  posoutsouth(4)            # South side, for btn driver
   Output:   tens(4),units(4)         # North side
   Output:   Z                        # North side

Internal
   Signals: mypos(4), out_condition
   Latches: register content(6)

Combinational
   mypos = pos+1
   out_condition= clock ∧ dir ∧ (mypos=addr)

   dataout = if out_condition then content.bitextender(5->8,Zero)
             else float # no conflict with input pin 'data'

   Z = if content≠0 then 0 else float

   (tens, units) = BCD (content)

Behaviour
   on falling edge of clock do:
                     content := data.select(0:5)
   enabled by: dir' ∧ mypos=addr     # dir=in/out' of I/O bus
End Chip Specification
```

There is no conflict between the combinational result on **`dataout`** and the direct connection to **`data`** declared in the header because **`out_condition`** is up only when the latter floats.

**Button driver.** This is a very small circuit (the one coloured in cyan in the circuit diagram on page 4).




All it does is assert what it receives from the north on the west pin when the button is pressed (the west of the two pins on the south side is for connecting the button to). Also, when the button is pressed the chip grounds the trigger pin on its south side. The trigger pin is intended for participating in the wire-OR that delivers the trigger for latching the button code (in this case, the chip position). The document "**Working with a full-core CdM-8 system**" provides sufficient information about how buttons are interfaced with processing circuitry to make any further explanations here redundant. Here is the pin-out of the chip:

| Name | Description |
|---|---|
| pos | Input, north side, chip position |
| addr | Input, east side, address bus for asserting pos on |
| addr | Output, west side, address bus for asserting pos on |
| trigger | Output, south side, trigger for latching button pos |
| button | Input, south side, button terminal |

**WLD (Win-Lose-Draw) chip.** The reference implementation uses this chip to establish the outcome of the game. The circuit inputs the wire-OR signal from the display drivers on the home side to establish that all holes are empty and so the game has ended. It further inputs the indicator values of the player's own mancala driver to see how many seeds are found there. Those together establish the outcome of the game. The outcome is signalled to the other WLD chip to make it indicate the outcome on behalf of the other player.

The present brief does not give the specification of the WLD chip: the project team is asked to design their own solution, whether in software or in hardware. Even the complete omission of this feature does not invalidate the design as it is

understood that the outcome of the game is evident to both players, so the feature is merely an enhancement.

**Data structure**. The Mancala pad implements in hardware the data structure of the state of the game. The state is represented as a size-16 array of 5-bit unsigned integers, in which two elements are unused: 0 and 8. Elements 1 to 6 represent the holes that belong to the human player; elements 9 to 14 are the robotic player's holes. Both sets of holes are laid out in anticlockwise order. For example, hole 2 is opposite to hole 13 and hole 3 to 12. Elements 7 and 15 are the human's and the opponent's mancalas. Such an arrangement makes it easy to determine which hole is opposite to which: it only takes a moment to realise that the holes that are opposite one another have array indices that are **1's complements** of each other. For example, $2_{10}=0010_2$ and $13_{10}=1101_2$ and either bit string becomes the other after a bit-flip.

| Index | Bit pattern | Description | Index | Bit pattern | Description |
|---|---|---|---|---|---|
| 0 | 0000 | <unused> | 8 | 1000 | <unused> |
| 1 | 0001 | Human's hole 1 | 9 | 1001 | Opponent's hole 1 |
| 2 | 0010 | Human's hole 2 | 10 | 1010 | Opponent's hole 2 |
| 3 | 0011 | Human's hole 3 | 11 | 1011 | Opponent's hole 3 |
| 4 | 0100 | Human's hole 4 | 12 | 1100 | Opponent's hole 4 |
| 5 | 0101 | Human's hole 5 | 13 | 1101 | Opponent's hole 5 |
| 6 | 0110 | Human's hole 6 | 14 | 1110 | Opponent's hole 6 |
| 7 | 0111 | Human's mancala | 15 | 1111 | Opponent's mancala |

Finding the opposite hole is necessary to implement the capture rule (Rule 3). Notice that the index gap between the human's mancala and the opponent's first hole requires an incrementer to be inserted after the mancala display driver, see the NE corner of the circuit diagram on page 4.

## Interfacing the game pad with the system

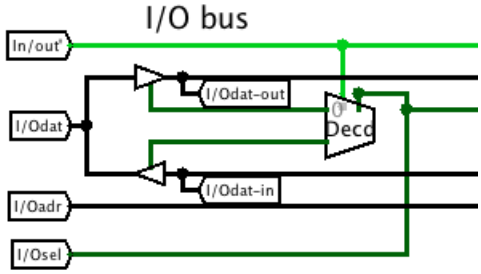
Since the game console is not expected to have any I/O devices other than the game pad, you may commandeer the complete I/O segment of the CdM-8 system (16 addresses) to represent the above data structure. In order to avoid copying to and from computer memory, we will use the I/O registers *exactly* as memory, i.e. will not only write into them, but also read from them when the program needs to act according to the gameplay.

One of the consequences of such a complete memory mapping is that the display drivers do all I/O address and direction decoding acting as mini-interfaces. The reference implementation utilises the unused address 0xf0, marked in the above table as the human's unused hole, to support *reading* the button keyboard (in order to receive the human's move) and *writing* to the three LEDs one can see on in the diagram on p.4: the "make move" light (cyan), the

“rendering” light (blue) and the “strategising” light (magenta), the last one indicating that the program is busy figuring out the next move. When the human presses the button by an empty hole, the “error” LEDs light up. All four bits for controlling the lights come from a register associated with IO-0, which is part of the game pad interface.

Notice that *all* the above lights are optional. The only essential function of IO-0 is to relay button presses to the program.

**Simulation problems.** Despite the straightforwardness of the design, there is a subtle problem that is due to a simulation error of Logisim. The north-west mancala driver has the data output, which needs to be connected to the interface. Connecting that output straight to the input of human hole 1, even though there is nothing but wire going around the pad to make a closed circuit, is perceived by Logisim as a positive feedback on the signal; it causes “oscillations”. In order to avoid those, the reference implementation utilises a split **`I/Odat`** bunch as follows:




Here the single **`I/Odat`** bunch is split (not de-multiplexed, since the derived bunches are going in opposite directions) into an **`I/Odat-in`** and **`I/Odat-out`** which are never connected together, and which are linked with the system’s **`I/Odat`** by two controlled buffers. As a result, the gamepad (on the east side) utilises a five-signal I/O bus, rather than the standard four-signal one. This solution is instructive in and by itself: when interfacing with third-party equipment, system bus designers often include defensive circuitry of the above sort to ensure that if the interface misbehaves the problem can quickly be located and made safe.

## Software

The software represents the main challenge of the Mancala project. At the heart of the reference implementation is the subroutine **`mkmove`**, which makes a move based on the state of the pad and the index of the hole, both being supplied to it in (general purpose) registers. The reference implementation uses address 0 of the data memory (as the most easily accessible one, it only requires a clear register to do ld/st) to keep the current move’s own mancala index. The implementation also permanently assigns the address of the gamepad to one of the registers (we chose r1) which is kept intact throughout the program (with the exception of some save/restore intervals). The subroutine responds with a

flag (C,Z or combination thereof) that indicates that a move has resulted in the last seed going to the own mancala, hence requires a repetition; or that the move was illegal since the hole had no seeds.

Another important subroutine is **`capture`**, which checks if the last move went into an empty own hole and at the same time the opposite hole had seeds. It then implements the capture rule (Rule 3).

The main program is a single loop in which **`mkmove`** is called until it indicates that the move is not to be repeated by the player, at which point **`capture`** is called to see if seeds can be captured after the last move. Then a handover takes place and the other player makes a move. If the hardware implements the four status lights mentioned above then the main loop contains instructions to write into the status register to turn these lights on and off.

**Strategy.** Given the severe limitation of code memory, any strategy will be accepted, including the simplest one: mirror play: the computer player attempts to make the same move as the human if the move is legal on its behalf. If it is not, the computer player makes the first legal move it manages to find. The reference implementation went a bit further than that:

- the computer first tries to determine if any move leads to a repeat move. If so, that move is taken;
- otherwise, the computer tries to determine if any move leads to capture. If there is one, it is taken
- otherwise the computer determines which move keeps the maximum number of seeds on its side. That move is taken unconditionally.

With a bit of code tuning and optimisation it might be possible to do even more than that (but not much more). The project team may decide to explore the above options (or any other strategy) if there is time.

## Suggested design progression

The software and hardware parts of the project are almost completely decoupled from one another. Here is one way the project can be approached:

1. Design and implement the **`mkmove`** subroutine and test it using **cocoemu** and ordinary memory. There should be no difference at all if the state of the game is kept at 0xf0 (don't forget to initialise the SP to 0xf0 though). Check that the move is correct under all circumstances, the opposite player's mancala is bypassed and that the distribution of seeds continues for more than one full round as long as seeds are available.

2. At the same time start designing the hardware, progressing from a single display driver fully tested using pins to 14 display drivers in a chain wired to pins at both ends and tested for writing. When you are satisfied that the bus protocol works on the pad and you can write numbers to each

register and they are displayed correctly, see if the read operation works.

3. With the pad working and the main component of gameplay implemented, complete the software. You are able to continue with **cocoemu** since all you are interested in is a single move and you can always simulate button presses by including a `dc` in the program covering the relevant I/O address.

4. Now build and test the interface. Write sufficient tests to show that the game pad works correctly under the control of a program: you should be able to write into any I/O register and read from it at any time, and it should be possible to loop on the button register 0xf0 until a positive number appears in it which corresponds to the button pressed.

5. Finally combine the hardware and the software and do overall testing. Play the game and pick up on anything that needs to be improved.

6. If you have time left (assuming your allowances for the preparation of the report and the final demo are set realistically), do some advanced features, starting with any improvements of the strategy.

# Appendix: Quotation ☺

*"… As I approached and took my seat, I noticed the ground before him. He had dug a series of small little holes in the dirt, and it almost looked like he was cleaning them out… quickly picking up little beans and rocks from each of the pockets.*

*"What's going on here?" I said out loud in English. He looked back up at me and reached out with a clenched fist, trying to share the debris."Okaaaaay" I whispered, reluctantly accepting my handful. Then he motioned toward the holes again, demonstrating that I could add my beans and rocks back to each of the pockets.*
*"Put these here?" I confirmed. He didn't understand the words, but watched my hands and nodded. First hole, second hole, third hole… slowly we filled each pocket with the same amount of pieces. Then it finally hit me. "Are we playing Mancala?!" I asked excitedly. He smiled and made his first move. That was my morning. We played a full game of Mancala, laughing together as the sun came up on this small Malian village. I never knew his name, and he never knew mine. I didn't speak his language, and he didn't speak mine. And none of it mattered. For thirty minutes the things that disconnected us were made insignificant by something that did connect us, and that was all we needed to enjoy the moment. For one morning, we were just two friends playing a game as old as time."*

– Tyler Riewer
`http://www.tylerriewer.com`