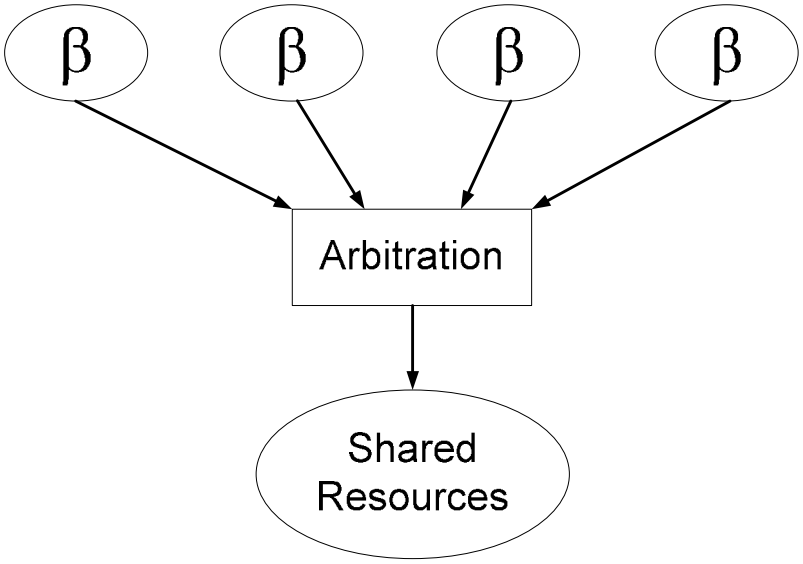
Multi-Core βeta Computer



By Christopher Celio
and Jonathan M. Long

Abstract
Due to fundamental physical constraints, the rate of growth of central processing unit clock speeds is beginning to fall short of the predictions set forth by Moore's Law. As a result, researchers are now exploiting the performance gains possible from multi-core processing. This project aims to introduce some of the difficulties presented by multi-core computing and what is needed to achieve an operational multi-core system.

Massachusetts Institute of Technology
6.111: Introductory Digital Systems Laboratory
TA: Amir Hirsch
May 17, 2007

Table of Contents

List of Figures and Tables

1 Introduction

This document describes the design, implementation details, and analysis of the multi-core Beta computer developed by Christopher Celio and Jonathan M. Long as the final laboratory project for 6.111, *Introductory Digital Systems Lab*, a course of the MIT Electrical Engineering & Computer Science Department offered during the Spring 2007 semester.

1.1 The Game of Life

As a test of the performance benefits of a multi-core system, our system plays the zero-player Game Of Life (GoL). The Game of Life is a simple mathematically model to simulate cellular life cycles. The playing field consists of a matrix of Boolean values. Each cell is an element in the matrix and can be either alive or dead. Each round, one must visit each cell, compute the number of adjoining neighbors that are alive, and deduce the next state of the cell. The playing field can easily be divided into sections, giving each core its own domain. A screenshot of the Game of Life, as generated by the Multi-Core Beta Computer.

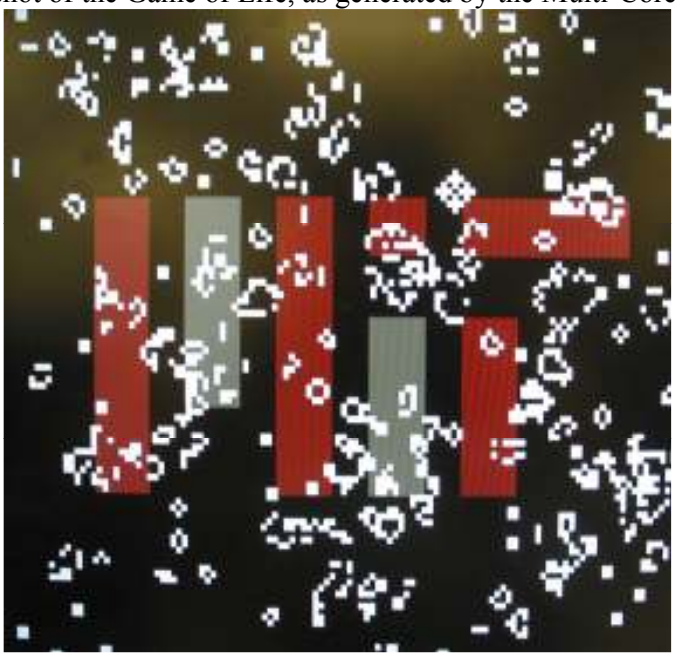


Figure 1: Screenshot of the Game of Life being played on the Multi-Core Beta Computer.

1.2 Design Overview

The Multi-core Beta Computer consists of four Beta cores. Each core has direct access to a private copy of instruction ROM and private memory for stack storage. Each Beta processor has access to "shared resources." Shares resources include the PS2 Driver, Character Buffer, the Game Of Life Buffer, and a small amount of memory space used for synchronizing the Betas called Sync RAM.

Because there is only one port for access to shared resources, a *memory arbiter* acts to coordinate access to the shared memory bus. If two processors attempt to access the shared resources at the same time, the arbiter connects one processor to the memory bus, and stalls the other processor.

Memory decoders act as the glue for the system. A *private memory decoder* coordinates data directly into and out of the Beta processor. For example, it is in charge of either connecting the processor to private memory, or routing the data to the *arbiter* if the processor is accessing shared memory.

A *shared memory decoder* decodes data from the *memory arbiter* and routes the data to the correct module, such as routing access to the Character Buffer when a processor would like to print to the screen.

A *display controller* accesses the second port of the Character and the Game Of Life Buffers. The *display controller* reads the data in the buffers, and translates the data to visuals to be displays on a VGA monitor.

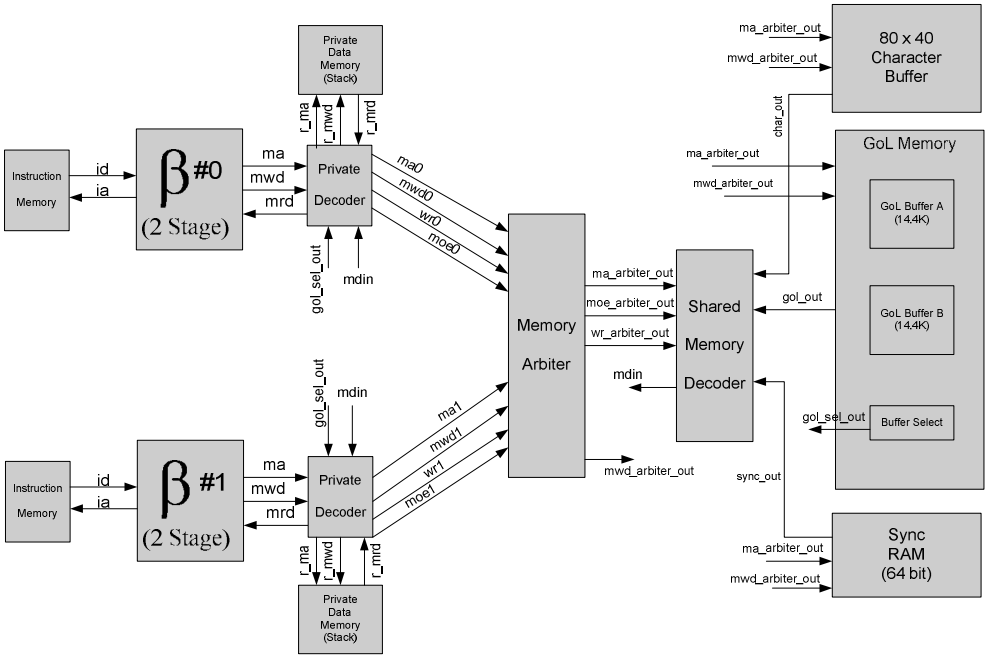


Figure 2: High Level Multi-Core Beta Computer block diagram. Note that only 2 Beta microprocessors are shown. Additional Betas are added similarly. All Betas communicate with the shared memory via the memory arbiter. Each Beta has two memory access points: one for its personal copy of the instruction memory and one for access shared memory.

2 Implementation Description

In this section, the design and implementation of the multi-core Beta computer modules are described.

2.1 The Harvard RISC Beta Processor

The Beta microprocessor is a construct used as a teaching tool to introduce Electrical Engineering & Computer Science students at MIT to processor design, instruction sets, and computer architecture. It is used in the 6.004: Computation Structures core curriculum class, and all students implement their own beta processor using a Hardware Description Language.

In building a multi-core computer in Verilog, we wanted to focus as little on software as possible. Implementing the Beta processor made the most sense because assembly editors and compilers were readily available.

The Reduced Instruction Set Computer, or RISC, Beta processor implements a set of simple instructions such as mathematical operations (with the exception of divide), logical operations, memory accesses, and branching instructions. All instructions are executed in a single cycle, with the exception of LD and LDR (memory reads), which take two cycles. The Harvard Architecture means that the processor has two ports for accessing memory, one port for instruction data and one port for data memory (See Section 2.1.2: Instruction Memory & Harvard Architecture).

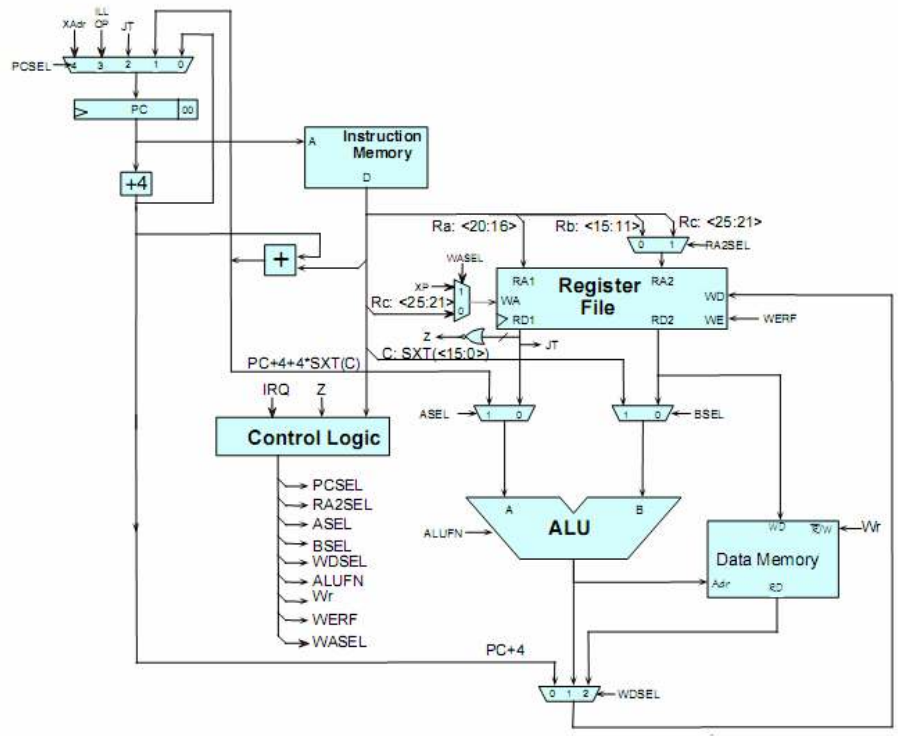


Figure 3: The 6.004 Beta processor. This is a single-stage, Harvard architecture processor designed to access asynchronous memory. It is very similar to our own implementation of the Beta processor. The main differences include the addition of stall logic, additional data lines for CPU ID# and Core Count, and changes to the Program Counter to interface with synchronous memory. Image courtesy of the 6.004 "Building a Beta" lecture notes.

The Beta has four main modules: the program counter, the register file, the ALU, and the control logic. The program counter is in control of setting the *Instruction Address* and sending it to the instruction memory.

The *Register File* holds 31 registers that act as small, 32-bit distributed RAM that can be accessed quickly by the processor. This is the best place to store temporary variables. Thirty-two registers can be addressed, but R31 always returns zero.

The *ALU*, or *Arithmetic Logic Unit*, is in the functional heart of the processor. It takes in two 32-bit inputs, and a 5-bit function select signal, and outputs a 32-bit signal. The *ALU* can perform addition, subtraction, multiplication (but no division), shift right, shift left, shift right arithmetic, AND, OR, XOR, , and comparison logic. The inputs to the ALU originate either from registers from the Register File, or as a sign extended literal from the instruction

data. The 32-bit output can either be sent out as the *memory address*, or multiplexed as write data to the *Register File.*

The *Control Logic* module is the brain of the processor. It takes in the 6-bit *opcode* from the 32-bit instruction data signal, and deduces the appropriate control signals for the processor. It also deals with external stall logic sent to it either by the user (to pause execution of the Beta) or from the arbiter (to stall the Beta until it is safe to access shared memory).

The Beta processor is well described through lecture notes and laboratory assignments found on the 6.004 website: http://6004.lcs.mit.edu/ . Though wire diagrams accurately detail most of the modules, a number of differences between the 6.004 HDL simulation environment and the 6.111 FPGA environment necessitated a number of modifications.

The biggest difference was that the Virtex BRAM modules used as memory for the project are synchronous modules. Therefore, the LD and LDR operations (memory reads) now take two cycles to complete: the first cycle generates the memory address, on the clock edge the value is latched into the memory module, and on the second cycle the value for the specified memory address is returned to the Beta processor. This also necessitated adding stall logic to the beta processor so that two cycle reads were stalled on the first cycle to allow time for the memory data to return to the Beta.

The stall logic also came in to use for another reason. In a multi-core environment, in which multiple processors are attempting to access limited resources, some beta processors need to be stalled until the resource becomes open.

To facilitate the operation of the processors in a multi-core environment, two additional instructions were added to the Beta instruction set: CPUID (retrieve the CPU's unique ID number) and NCORES (retrieve the total number of cores in the system). Once a processor knows its ID and the total number of cores, the software can appropriately control each processor to perform the tasks specific to it. For example, CPU#0 of two cores knows to control the top half of the Game of Life playing field, while CPU#1 of two cores takes the bottom half.

2.1.1 Instruction Memory & the Harvard Architecture

A naïve approach to a multi-core architecture would involve a single shared memory resource that all processors access. If the processors must share a single port of access, an Arbiter is needed to decide which processor gets assess, tie that processor to the memory bus, and stall all other processors attempting to access memory until it is their turn.

Unfortunately, the RISC Beta completes almost all of its instructions in a single cycle. Therefore, the processor must access memory every clock cycle to retrieve instruction data. If all processors had to access the same shared, single-port memory resource, there would be an unacceptable bottleneck. A multi-core system, in which each core must share instruction memory with all other cores, would effectively be reduced to a more expensive, threaded single-core system.

There are a number of solutions to this dilemma. One method involves using instruction caches to allow each processor to store a small set of instruction data. Unfortunately, this does not solve collisions during cache misses that will undoubtedly occur during interrupts and process changes.

Our solution was to go with a Harvard architecture, in which each processor is given its own private ROM copy of the instruction data, completely separate from data memory (See Section 2.5 Sync RAM for more details).

2.1.2 Private Data Memory

The Operating System uses a regular 300Hz clock interrupt to switch between user processes. Both clock and keyboard interrupts also force a switch from user to kernel operations. To allow the processor to switch between different threads of operation, each process has its own stack space and user-state (register contents). Switching between processes requires storing the current user stack and loading a previous processor state. The stack is also used to pass information between functions such as parameters and return values. Thus, it is imperative that each processor has its own memory to store its stacks and user-states.

2.1.3 Software

All software was written in Beta Assembly code in the BSIM java program. The BSIM application can compile the assembly text file into a *.coe* file. Python scripts provided by Professor Terman could instantly convert *.coe* files into *.v* files, instantiating the appropriate number of Block RAM modules, initialized with the program memory. The Python scripts were modified by Matt Long as needed.

The Operation System is a modified version used as part of the class 6.004 to teach students about operation systems and kernels. The code was modified by Professor Chris Terman of 6.004 and 6.111 for use on an FPGA. The OS has a simple time-sharing kernel that can handle a number of user processes. The OS also has a set of supervisor calls which are used to interface with the keyboard input, the character buffer display, and semaphore control between user processes. None of this code was modified by us for our project.

We did however, completely rewrite two of the three user processes for our use. The former Pig Latin translation process now simply echoes what is typed by the user and then prints a role-call of all activated processors:

```
00000001> Greetings
GREETINGS: CPU_ID# 0 of 4
         : CPU_ID# 1 of 4
         : CPU_ID# 2 of 4
         : CPU_ID# 3 of 4
```

Figure 4. Typical Command line Output: The user can type in a message at the command line. Because Core#0 is given absolute preference by the memory arbiter, Core#0 captures all keyboard input. Upon hitting "ENTER", Core#0 echoes the message, and then appends its ID# and the core count. When Core#0 has finished, the remaining cores write their ID# and core count in turn. This demonstrates the synchronization of the cores, and the ability to write to the character buffer without glitches or collisions on the memory bus.

By having each core print to the screen, we could both verify that the core was alive and capable of writing to the shared character buffer. Also, by having each core write in sequence, we could demonstrate that the cores were synchronized and communicating between each other correctly. Furthermore, we could verify that each core correctly knew its own ID and the number of cores in the system.

The second user process we rewrote was used to play the Game of Life. Based on the Core ID# and the number of cores, each processor could calculate the bounds of the playing field it should play. The field was allocated by dividing the screen horizontally. For a quad-core system, the first core received the top quarter of the screen, the second core the second quarter, and so on. The playing field bounds for a single core is computed as follows:

Core ID#0: x = [0, width)
y = [y_min, y_max)

Where y_min = core_id * (HEIGHT / core_count)
Y_max = y_min + (HEIGHT / core_count)

Because divide operations were not allowed, a case-select statement was used to assign the y_min and y_max bounds. The Game of Life playing field is stored as two matrices in the *Game of Life* RAM module. Execution of a round involves reading the game state of each cell from one matrix, and writing the results, or the state of the next round, into the second matrix. A *Buffer Select Register* tells the *VGA Display Controller* which matrix to display to the monitor. The *Buffer Select Register* also helps the Beta cores remain synchronized. When the *Buffer Select Register* changes value, the cores know that a new round has begun (See Section 2.5 Sync RAM for more details).

2.2 Private Memory Decoder

For every Beta processor in a system, an accompanying private memory decoder is also instantiated. The private memory decoder multiplexes both outgoing (write) and incoming (read) operations from/to the associated Beta. For writes, the decoder routes high write-enable signals and write data to 1) the Beta's private data memory, 2) shared memory via the Memory Arbiter, or 3) the GoL buffer-select register (only accessed by Processor #0). For reads, the decoder redirects high memory output enable signals to the arbiter if an address within the appropriate address space is specified by the Beta. Furthermore, the decoder multiplexes incoming read-data (also based on memory address) from the three sources listed above. This memory address-space is translated as shown in Table 1.

Memory Address	Memory Element
0x00000000 – 0x00007FFF	Private Data Memory
0xFFFFFF00	GoL buffer-select Register
All other addresses	Shared Memory (Arbiter)

Table 1: Partition of the address space as seen by the private memory decoder.

2.3 Memory Arbiter

The memory Arbiter serves to connect all of the Beta processors to all Shared memory resources. All processors route their memory addresses, memory write data busses, and read request signals to the Arbiter. By monitoring each processor's write and read signals, the Arbiter can deduce which processors are trying to access shared memory. The Arbiter must then choose one processor to connect to the memory bus, and stall the other processors attempting to access shared memory. In the interests of time and simplicity, our Arbiter design gives preference to the processor with the lowest ID.

2.4 Shared Memory Decoder

The Shared Memory Decoder takes read/write requests from the Memory Arbiter and forwards them to the appropriated shared memory element. As with the Private Memory Decoder, forwarding is done by analyzing the memory address. In the case of writes, after deciding which memory element a read/write is intended for, the decoder forwards write-enable and write-data signals to the appropriate data element. In the case of reads, the decoder multiplexes read data based on address, selecting the data from the intended memory

element. The memory elements managed by the Shared Memory Decoder are shown in Table 2 along with their associated address space.

Shared Memory Element	Address Space
Character Buffer	0xFFFF8000 – 0xFFFF8D00
Game of Life RAM A	0xFFFF8D01 – 0xFFFFC600
Game of Life RAM B	0xFFFFC601 – 0xFFFFFFEF
GoL Buffer Select Register	0xFFFFFF00
Sync RAM	0xFFFFFF01 – 0xFFFFFFF7
PS2 (Keyboard) Buffer	0xFFFFFFF8
System Timer	0xFFFFFFFC

Table 2: Partition of the shared address space amongst all shared memory elements as seen by the Shared Memory Decoder.

2.5 Sync Ram

For our system, a small amount of memory is needed for the Beta cores to communicate with one another for synchronization purposes. The Sync RAM is a module of sixty-four 1-bit registers, where each register can be addressed for reads and writes. The Sync RAM can be treated as a 64 element array of Boolean values. Two different methods were used for synchronization, one for The Game of Life process, and one for the command-line process.

In the command-line mode, when a user presses the "ENTER" key on the keyboard, the desired output is to have each core write to the character buffer to note its presence (See Figure 4). To prevent the output from being completely garbled, it was necessary to only have one core write to the character buffer at a time. When a user presses "ENTER", CPU#0 immediately begins writing to the display. When it finishes, it writes a Boolean *true* in its place in the table("CPU_List" is an abstraction for a Boolean array inside Sync RAM):

CPU_List[core_id] = "TRUE"

CPU#1 monitors the CPU_LIST[0] . When it sees CPU#0 has finished, it begins writing to the monitor. When CPU#1 finishes, it writes to the Sync RAM and CPU#2 begins writing. In this manner, each CPU monitors the Sync RAM and waits for the previous CPU to finish its task. The last CPU has the job of clearing the registers in Sync RAM used by the command-line process. When a CPU notices its own entry in the Sync RAM array has been cleared, it knows that the last CPU has finished, and all cores are ready for the next "ENTER" keystroke. The GameOfLife synchronization is handled a little differently. It is very important that all CPUs are synchronized, and do not begin a round until the previous round has finished. When a CPU has finished its round, it "Checks in" by writing *true* in its spot in the Sync_RAM. CPU#0 is tasked with monitoring the array and deducing when all processors have finished:

Once CPU#0 has found that all cores have finished, it clears the Sync RAM registers used for the GameOfLife, and then changes *Buffer Select Register*. All checked in cores (except CPU#0) monitor the *Buffer Select Register*, and when they realize its value has changed, they know to begin the next round:

//begin Round
Current_state = *Buffer_Select Register*

```
        .....
//perform Calculations
        Next_state = ~current_state;

        While(Next_state != Buffer_Select_Register) {
                //wait here until CPU#0 signals to begin next
                // round by modifying Buffer_Select_Register
        }
```

To prevent memory congestion in reading the *Buffer Select Register*, it is wired directly to each core's *private memory decoder*.

2.6 PS2 Driver & Keyboard Support

The keyboard used for the command line prompt was connected to the system through a PS2 driver written by Professor Terman. Keyboard support was beyond the scope of our project, but we did find a use for the keyboard as a debugging tool. For example, hitting the "ENTER" key released a key to execute a section of Process 1 code. Therefore, we could place our own code in the loop, such as flipping bits in the video buffer, to verify the Betas' ability to access specific memory locations.

Though we did not code the driver, we did spend time to insure that the Betas could properly talk to the PS2 driver as a shared memory resource. Unfortunately, the PS2 driver is not designed for multi-core use, and it would require a reworking of the OS keyboard service call to allow all Beta's access to the keystrokes entered by the user. The problem lies in the fact that the PS2 Driver's FIFO buffer has a *read_pointer* that is incremented every time it detects that it has been accessed. Therefore, the first Beta that reads the driver gets the character data. Adding a counter to the Driver does not fix the problem for two reasons: because of the complexities of how the driver deduces if it has been read, and because the arbiter may stall someone who is in the middle of accessing the PS2 Driver and may have to start again (this is also an artifact of the arbiter always gives preference to CPU#0).

2.7 32-bit Hex Display

The 32-bit Hex Display is Professor Terman's code, provided to 6.111 students as an excellent abstraction module to effortlessly send 32-bit signals for display on the 6.111 Lab-kit's Hex display.

The hex display was used almost exclusively to monitor the instruction addresses of each of the cores. Even at 27 Mhz, it was very easy to deduce the behavior of each core from the Hex Display. For example, a solid 0x0618 denotes an illegal opcode crash (usually from attempting to read an instruction from reserved memory space), while oscillations in the 1E80 region denote a processor waiting to receive the command from CPU#0 to begin the next round in the Game of Life.

2.8 VGA Display

The VGA Display subsystem retrieves and visualizes information from the Game of Life and character buffers. The display is set to toggle between a terminal mode, which serves as a command line prompt for keyboard interaction with the Betas, and the Game of Life mode,

which displays the 120 by 120 grid of cells. A block diagram of the VGA Display subsystem is shown in Figure 5.

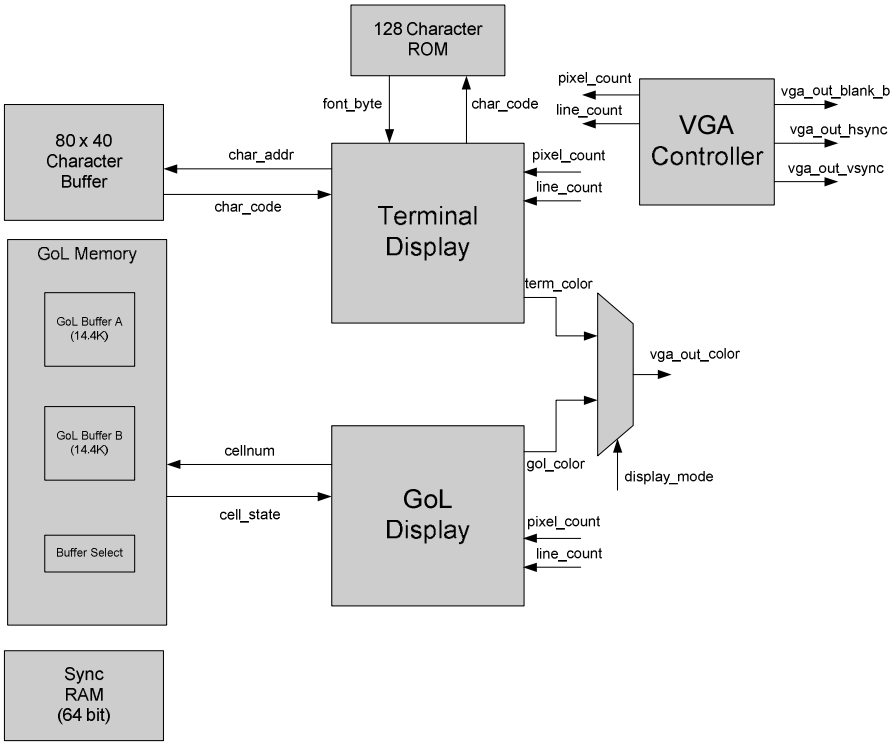


Figure 5: The VGA Display system. Each display module has a dedicated read-only access port to the appropriate shared memories. Use of a Font ROM allows for straightforward modification of font styles. Outputs of each display mode are continually computed to allow for seamless display transitioning.

2.8.1 VGA Controller

The VGA Controller module takes the *pixel_clock* and *reset_sync* signals as inputs. From these inputs, the controller generates the following signals: p*ixel_count, line_count, blank_b, hsync, vsync*. Of these outputs, *blank_b* is wired directly to the *VGA_Out* signals used to drive a VGA monitor. The raw *hsync* and *vsync* signals are passed through a simple sync delay module that delays the signals by two clock cycles. This is done to allow the VGA color and blanking data to pass through a digital-to-analog block before being rejoined by the synching signals. The *pixel_count* and *line_count* signals are both 10-bit signals that are used by the Terminal Display and Game of Life Display modules (described below) to calculate what color data is sent for each pixel of the display. All of the module's outputs are derived from the *pixel_clock* input as well as the VGA timing parameters for a 640 x 480 display running at a 75 hz refresh rate, as shown in Table 3.

Parameter	Value
PIXELS	800
LINES	525
HVID	640
HFRONT	16
HSYNC	96
VVID	480
VFRONT	11

VSYNC	2

Table 3: VGA Timing parameters. Note that HBACK and VBACK parameters are also defined, but never used in calculation and are thus not shown in the above table.

In the event of a HIGH *reset_sync* signal, the VGA Controller module is designed to produce a blank (black) display. While an 800 x 600 pixel display was originally planned, peculiarities of the particular monitor model being used for the project and a lack of clear monitor documentation prevented the use of an 800 x 600 display. However, due to the modularity of the VGA controller implementation, modifications to achieve any desired VGA display size and refresh rate can be achieved simply by making appropriate changes to the values in the table above.

2.8.2 Terminal Display

The Terminal Display module constantly displays the contents of the 80 x 40 character buffer, one of the memory modules within the system's shared resources. The Terminal Display module takes as inputs the *pixel_clock, pixel_count,* and *line_count* signals. With a display size of 640 x 480 pixels and an 80 x 40 character display, each character occupies an 8 x 12 region of the screen. The Terminal Display module continually calculates 1) which line of characters (0 – 39), 2) which character within that line (0-79), 3) which row of pixels within that character (0-11), and 4) which pixel of that row (0-7) corresponds to the given screen position specified by *pixel_count* and *line_count* values.
From the line and character values calculated as described above, a character address is calculated by the formula:

$$\text{charAddress} = (80 \times \text{Line}) + \text{Character}$$

Note that addresses are referenced from the top-left corner of the display, *i.e.* the address of the character in the top-left corner is zero and the address of the bottom-right character is 3,199. Since the VGA protocol scans horizontally, the address scheme of the character buffer behaves similarly in that the address increments from left to right across each line of characters. The calculated character address is then sent to the Terminal Display's dedicated port of the character RAM module. Since the display is only interested in reading character codes and not writing them, this is a read-only port. Note that in order to allow enough time to finish determining the color output data, character addresses sent to the character RAM are actually calculated from the "next line" and "next character" rather than the currently specified position.

After receiving an 8-bit character code from the character RAM, the Terminal Display queries its own Font ROM to determine how to represent the character on screen. Rather than fetching data for the entire 8 x 12 character, a single 8-bit row of data is retrieved one at a time. This is done using the row values calculated from the given *pixel_count* and *line_count*. Once a given row of a character (*i.e.* a "font byte") is retrieved from the ROM, the ON/OFF status of the current pixel is found by analyzing the proper bit of the row. If the bit is HIGH, the foreground color, white, is output. If the bit is LOW, the background color, blue, is output.

Since seven of the eight bits of a character code are used to specify a character (the high-order bit is used to indicate if the color scheme should be reversed), the Font ROM allows for up to 128 (2^7) distinct character definitions. Thus, the total needed size of the font ROM is 128 x 8 x 12 = 12288 bits. On the 6.111 lab-kit, this can be realized within a single block RAM module.

2.8.3 Game of Life Display

The Game of Life Display (GoL Display) module is responsible for continually representing the state of the GoL buffer. Similarly to the Terminal Display module, the GoL Display module takes the *pixel_clock, pixel_count,* and *line_count* signals as inputs. The module continually determines within which cell of the 120 x 120, if any, is being specified by *pixel_count* and *line_count*. Each cell is represented by a 4 x 4 square on the monitor, resulting in a 480 x 480 pixel square being used by the grid. Cells are identified by address calculated as follows:

$$\text{cellAddress} = (120 \times \text{Line}) + \text{Column}$$

Cell addresses are referenced from the top-left corner of the game board and increment horizontally across the board from left to right. With a 120 x 120 grid, there are 14,400 cells to be displayed with cell addresses ranging from 0 to 14,399. The calculated cell address is then sent to a dedicated read-only port of the GoL memory unit. On the next positive clock edge, the single bit of data stored at the reference address (the LIVE/DEAD state of the cell) is returned to the GoL Display module. The pixel color output by the GoL Display module is chosen based on the live/dead status: white for live cells, black for dead cells. To allow for processing time, as with the Terminal Display, the address of the "next line" and "next column" is actually used to calculate the cell address that is sent to the GoL memory. Thus, the live/dead status will be available during the next cycle (*i.e.* when the "next cell" has become the current cell) to determine color output.

Outside of the 480 x 480 grid, an eight pixel wide wall is drawn (in white) to delineate the edge of the game. Beyond the wall is unused space that, with additional time, could be used to display statistics about the game such as generations per second, total generations calculated, etc. Note that these statistics are currently being displayed on the 16 character hex-display.

2.8.4 VGA Top Module

The VGA Top Module primarily serves as the interconnect of the VGA Controller, Terminal Display, and GoL Display modules. Also within the VGA Top module is a small amount of logic responsible for multiplexing the RGB color data generated by the Terminal and GoL Displays. An input driven by a switch on the 6.111 lab-kit is used as the selector bit to toggle between terminal and game display modes.

As a last minute addition, in order to show school pride, supplementary logic was added in the VGA Top Module to generate the MIT logo. The logo is centered within the 480 x 480 Game of Life field and can be toggled on/off via a switch on the 6.111 lab-kit.

2.9 Game of Life Memory

The state of the Game of Life is stored between two identical RAM modules. Two RAMs are needed so that a previous generation can be read from one RAM while the next generation is calculated and then stored in the other. Furthermore, two initial game states are stored in ROMs, one of which is loaded into the GoL RAM modules upon system startup/reset. A GoL memory manager module and its supporting modules are implemented to support RAM toggling and loading the RAMs from ROMs.

2.9.1 Game of Life RAM

The GoL Memory module instantiates the two symmetric GoL RAMs named *bufferA* and *bufferB*. The RAMs have dual access ports: 1) a read-only port for the VGA Display subsystem and 2) a read-write port accessed by the system Beta cores. While there are two RAMs in the module, only one address, data-in, and data-out bus is provided to the outside world. To choose which of the two RAMs is read/written by a Beta, a distinct 32-bit memory address is associated with each RAM. There addresses are decoded by the Shared Memory Decoder (see Section 2.4) and translated into two buffer-select bits, one for each RAM. For proper operation only one of the two buffer-select bits should be HIGH at any given time. The buffer-select bits are used to enable their associated RAMs for writing as well as multiplex among the data retrieved from each RAM for reading.

To decide which of the two RAMs the VGA Display receives its data from, a *ram_select* register is periodically written to by a Beta. Every read request from the VGA Display is processed by both RAMs, the data retrieved is then multiplexed by the *ram_select* register before being returned.

2.9.2 Game of Life ROMs

Two Game of Life states are stored in corresponding ROM modules. The GoL ROM modules consist of a single read-only port used by the GoL Memory Manager upon a system boot/reset to load an initial state into the GoL Memory module. Each state is stored in the lower 14,400 bits of the BRAMs that realize the two ROMs. A dual "Gosper's Gun" configuration is stored in one ROM and was chosen since the configuration ensures a game evolution that persists indefinitely. The other ROM stores a state which is simply a randomly chosen series of ones and zeros.

2.9.3 Game of Life Load Engine

The GoL Load Engine is responsible for loading the state stored in one of the two GoL ROMs into the GoL memory upon system boot/reset. The engine is triggered by a "start" signal received from a level-to-pulse converter which has the system reset signal as its input (note that the reset signal is asserted on system boot as well as the manual reset button). Thus, on the negative edge of the reset signal, the engine is triggered to begin loading. To ensure that writing to the GoL RAMs does not interfere with Beta operations, a busy signal is output to stall all Beta cores for the duration of the load.

Once started, the engine performs the load by incrementing through all 14,400 cell addresses. Each address is first sent to the GoL ROM selected by a user switch. On the next clock cycle the same address is then sent to the GoL RAM along with the data output from the ROM and a HIGH write-enable signal. Thus, a load takes approximately 14,400 clock cycles to complete. After loading the last address, the engine's busy signal is unasserted, allowing the Betas to begin functioning.

2.10 Lab-kit Top Module

The primary role of the top module is to serve as the interconnect for all the modules described above. Additionally, the top module assigns all lab-kit inputs and outputs to their appropriate signals. Outside of these two standard tasks, the top module contains logic for several additional features.

First, the top module contains a clock multiplexer that allows dynamic clock rate shifting. This feature was added so that a suitable clock rate, *i.e.* one that is slow enough to prevent memory read/write glitches yet fast enough to produce reasonable game performance, could be found without the need to recompile the project simply for the sake of changing the system clock. Second, the top module generates timer interrupt signals needed by the Beta microprocessors for process time-sharing.

Finally, the top module calculates performance statistics of the Game of Life. First, the total number of generations calculated by the system is calculated by counting the number of times the GoL ram select register goes high. This value is displayed on the four left most digits of the character display on the 6.111 lab-kit. Second, a measure of generations per second is computed. The generations per second metric is calculated by registering the generation count every second and then taking the difference between the current generation count and the previous generation count. Thus, the generations per second count is updated once per second rather than continuously. The generations per second is also shown in the 16 digit character display on the 6.111 lab-kit.

3 Project Results

As whole, our project was very successful at providing compelling visual evidence of the system performance gains from multi-core processing. Dynamically changing between one, two, and four cores resulted in a linear speed-up of the number of Game of Life generations calculated per second. Being able to visually observe these system speed-ups on screen provides conclusive evidence that huge performance gains are possible in the field of multi-core processing. It is important to note that software capable of taking full advantage of a multi-core architecture is of critical importance to exact any performance gains.

While we had hoped to demonstrate as many as eight or sixteen cores functioning in parallel, getting four cores to behave correctly without causing a system crash proved to be a difficult task in itself.

4 Conclusions & Future Work

Though there are a number of improvements we would like to make to the system, we still conclude that our project was a total success. Throughout the project, we remained reserved about the chances of implementing a fully working single-core system that married a Beta processor with a video buffer and Display Controller, knowing that many projects fail to successfully integrate all parts.

Though we synthesized and tested an octo-core version of our system, we realized that our simple memory arbiter was not adequate for more than four cores. By giving absolute preference to the lowest core ID#, the higher numbered cores found themselves locked out of accessing shared resources.

Our current architecture can comfortably fit eight cores before using up all available Block RAM memory (and thirty percent of LUT's). A more miserly use of BRAMs could dramatically reduce the overhead involved and allow for the maximum use of LUTs. We treated all BRAMs serving as instruction memories as single-port memory blocks from the point of view of the Beta processors. However, it would be possible to halve the number of BRAMs used for instruction memory by dual porting the instruction memory ROMs. Caching could also further reduce the footprint of the instruction memories by letting more betas share instruction memory. Each beta also has a private copy of data memory. In the interest of time, this copy of data memory is a complete copy of instruction memory. This

could be easily reduced to a few hundred bytes by accurately measuring the amount of stack space each user process needs, and changing the memory addresses in software to more efficiently utilize a smaller private memory space. Therefore, we estimate, with careful memory control, that as many as 24 Beta processors could be implemented on the 6.111 FPGA Lab-kit. However, through personal experience, we believe the limiting factor is memory access of a single-port of shared resources, and the routing lengths for all of the cores to be routed to the memory arbiter. Unfortunately, adding new cores to the system comes at a cost. It costs more LUTs, more memory, longer and more difficult routings to reach the shared resources, more collisions in accessing shared resources, and slower clock speeds.

Our design worked optimally with four cores at a little under 27Mhz. However, adding more cores would quickly require a complete architecture redesign from the ground up. System architects who hope to leverage more cores will have to use more novel methods of sharing resources and synchronizing cores.

5 Acknowledgements

Many thanks are in order for Professor Chris Terman for providing guidance, software compilation tools/scripts, keyboard driver, and extensive understanding of the Beta microprocessor. Thanks also to our project TA, Amir Hirsh, for his guidance in designing and implementing our project. Also, thanks to the entire Spring 2007 6.111 staff for providing an exceptionally well taught course in digital design principles. Finally, thanks to John and Jamie Long for taking the time to edit this report and make it more readable.