University of Utah
ECE 3710 - Computer Design Lab

# Team FIFO - Scorched Earth Game

Final Report

Arthur Gerbelli
Frederico Butzke
Joel Hough

December 19th 2013

## Index

## 1 Abstract

This project implements a game in which two players take turns aiming, moving, and firing projectiles from tanks. The goal is to refine your aim each turn and be the first to hit the other player's tank.

A 16-bit CPU including register mapped hardware stack and an interrupt controller is at the core of the system. A VGA controller with a pixel FIFO attached to a graphics module specifically for this game renders the graphics. Timer based interrupts trigger software interrupt handlers to update the game state. A Super Nintendo controller is used for input.

The software is written in a Ruby based domain specific language implementing an assembly like language with pseudo operations, control blocks, and macros written in Ruby. The assembler supports forward referenced labels by using two passes.

## 2 Introduction

This project implements a game in which two tanks sit on opposite sides of some terrain. Players take turns aiming, moving, and firing their tanks. The first tank to hit the other one wins. The game was inspired by a similar game called Scorched Earth.

The system includes a 16-bit CPU, interrupt controller, hardware timers, SNES controller, and VGA system. All peripherals are access from the application code through memory mapping. The application is written in a Ruby based domain specific language implementing a macro assembler.

## 3 Hardware

The hardware for this project includes a baseline CR-16 CPU extended with an interrupt controller, hardware stacks mapped into the register file, and a buffered VGA controller.

Interrupts are used to make writing an animated game easier. Timer interrupts are used to trigger animation updates, while input interrupts change game state. The main loop of the application does not need to do any polling, instead relying entirely on interrupts.

Stacks simplify much of the application code. The dual stack architecture is inspired by Forth machine implementations. Many of the benefits of a stack machine, such as smaller opcode size and simplified pipelines, do not apply to this design since it is also a register machine. Stacks are vital in the implementation of nested interrupts and subroutine calls.

The buffered VGA controller decouples the system from the pixel clock, greatly simplifying synchronization between the system and the display. New graphics modules can be added by implementing a simple interface.

## 3.1 Memory

The entire application is stored in block ram. As a result, the memory interface is very simple. Reads and writes happen in one clock cycle, so the CPU does not need to stall on memory operations. The CPU is the only device that accesses the memory, so no arbiter or second port is needed. No distinction is made between program code and data memory. Peripherals are memory mapped above the block ram. The Table 1 presents the memory map.

**Table 1 - Memory Map**

| | |
|---|---|
| 0x0000-0x3FFF | RAM |
| 0x4001 | Interrupt Controller |
| 0x4002 | PRNG |
| 0x4004 | Timer 1 |
| 0x4005 | Timer 2 |
| 0x4008-0x400F | Tank Game Graphics |
| 0x4020 | Switches and Buttons |
| 0x4021 | LEDs |
| 0x4022 | 7-Segment |
| 0x4023 | SNES Buttons |

## 3.2 CPU

The CPU (fig 3.1) is a fairly standard implementation of the baseline CPU specification. Instructions take a variable number of cycles to execute, depending mostly on how many internal registers data needs to move through to reach a destination. An earlier design had a simple three-cycle execution model for all instructions. Unfortunately, the length of the data paths for some instructions caused the CPU to fail timing constraints with a 100Mhz clock, the speed of the onboard oscillator. It was also hard to alter and extend the instruction set since many instructions naturally have more than three steps. For these reasons, the control logic was rewritten to use a Moore style state machine and registers were added along the longer data paths to allow for higher clock rates in exchange for more clocks per instruction.

The process of executing an instruction generally follows a standard pattern. The instruction is fetched from memory and stored in an instruction register. The instruction is then decoded. Data needed for the instruction is loaded and moved to where it is needed.

This is where instructions differ from each other most. Some instructions, like mov, only read the B register while addi reads only A, sub reads A and B, and movi reads none. It is important for our register file to know which registers are read in order to support our register file mapped stacks (discussed later). To accommodate this, we have states that set register read signals for each of these cases.

Once the data is loaded, the data is processed. In the case of an ALU operation, there is a state for performing the operation and then one for storing the outputs of the ALU. The outputs stored from an ALU operation can be the flags generated by the ALU, the result generated by the ALU, or both. Flags are stored in a flag register, which is part of the PSR, and the result is stored back into the A register in the register file.

States exist to control each variety of instruction. Many states are unique to specific instructions. While they share earlier data loading states, how the data is used in different enough from other instructions to warrant special handling. clri, jal, and store all have a state dedicated to them at the end of the instruction execution pipeline. load has two states dedicated to its implementation.

The shortest instruction is an untaken branch, which only requires two clock cycles. The first step is a FETCH, which all instructions go through. The second is DECODE, which only interrupt servicing can skip. At this point, the CPU knows if the branch is taken or not and the PC has already been incremented. If the branch is not taken, the state goes back to FETCH for the next instruction.

The longest instructions are load and ALU operations that store the result to the register file, requiring five clocks. The process for a memory load starts like any other instruction, with a FETCH and a DECODE. From there, the address to load from is read from the register file during the LOAD_B state (which is also used for jump instructions). The LOAD_FROM_MEMORY state comes next and is when the memory is actually read. Once read, the data path between the memory read register and the register file is enabled during the MEM_TO_REGFILE state, concluding the execution of the load instruction.

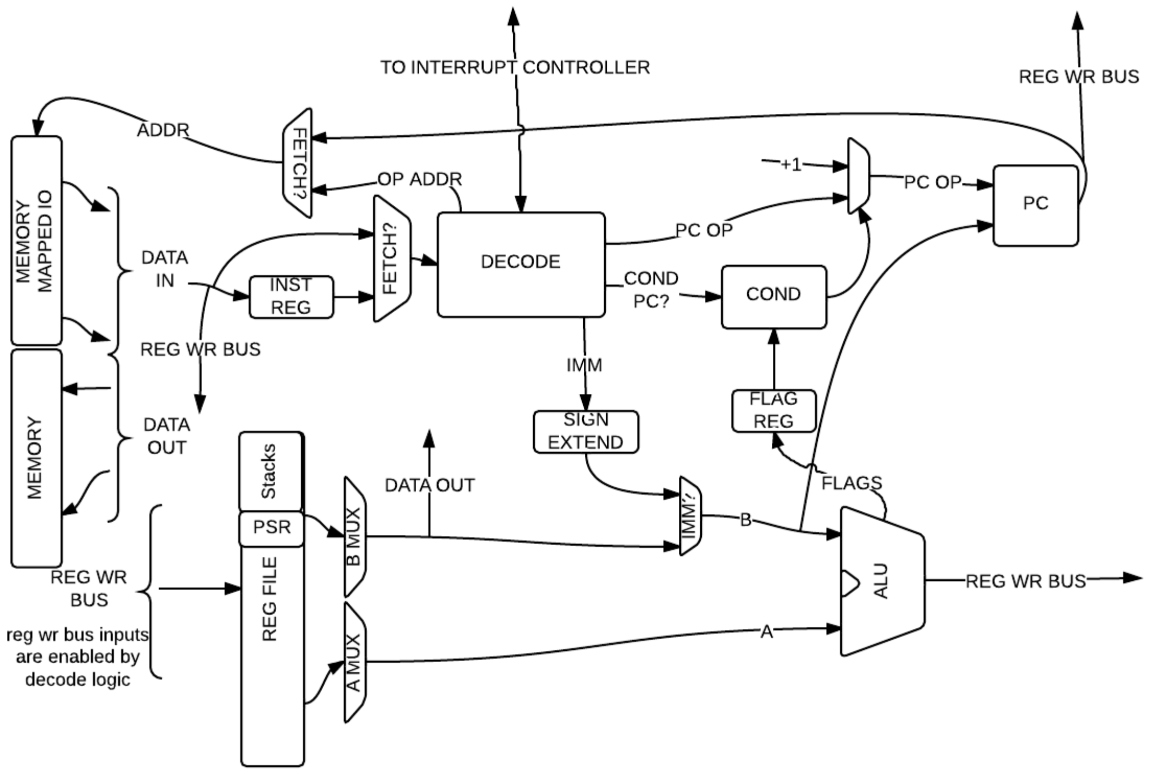

**Figure 3.1 - Central Processing Unit Block Diagram**

## 3.3 ALU

The ALU Flags and instructions were one of the first designs decisions the team has made. The first component that was implemented was the ALU. It has 16-bits data bus for the two inputs and the output, and also has 8-bits-width input for the OPCODES and 5-bits-width output for the flags.

- C Flag: It stands for carry. It is set to 1 when we get a carry on or a borrow resulting from arithmetic operations such as ADD and SUB, otherwise is set to zero.
- L Flag: It stands for low. It is used in CMP operations, it is set to 1 when we get the second operand lower than the first one in an unsigned compare operation, otherwise is set to zero.
- N Flag: It stands for negative. It is used in CMP operations, it is set to 1 when we get the second operand lower than the first one in a signed compare operation, otherwise is set to zero.
- C Flag: It stands for carry. It is set to 1 when we get a carry on or a borrow resulting from arithmetic operations such as ADD and SUB, otherwise is set to zero.

- F Flag: It is a general exception flag. It is set to 1 when a overflown (the result is truncated) occurs after perform an arithmetic operations such as ADD and SUB, otherwise is set to zero.

For this first design of ALU and Register files, we implemented the following instructions and theirs opcodes:

```
ADD   = 'b0101;
AND   = 'b0001;
CMP   = 'b1011;
LSH   = 'b1000;
LUI   = 'b1111;
MOV   = 'b1101;
OR    = 'b0010;
SUB   = 'b1001;
XOR   = 'b0011;
```

ADD:

Perform a signed add operation between the two operands and store the result in the destination.
Flag: C and F.
example: add Rsrc, Rdest -> Rdest = Rsrc + Rdest

AND:

Perform a logical bit AND operation between the two operands and store the result in the destination.
Flag: none.
example: and Rsrc, Rdest -> Rdest = Rsrc & Rdest

CMP:

Perform a both signed and unsigned comparison between the two operands at the same time, and update the flags.
Flag: N and L.
example: cmp Rsrc, Rdest -> if(Rsrc>Rdest ) N=1(signed) L=1(unsigned)

LSH:

Perform a logical bit shift, left shift if the operand is a positive number, filling the LSBs with zeros, or right shift if the operand is a negative number, filling the MSBs with zeros.
Flag: none.
example: LSH Ramt, Rdest -> Rdest = Rdest << Ramt if Ramt is positive
LSH Ramt, Rdest -> Rdest = Rdest >> |Ramt| if Ramt is negative

LUI:

Perform a 8-bit immediate value load operation on the upper 8 bits of the destination register.
example: lui imm, Rdest -> Rdest[15:8] = imm, Rdest[7:0] = old value

MOV:

Perform a move operation from the source register to the destination register.
example: mov Rsrc, Rdest -> Rdest = Rsrc

OR:

Perform a logical bit OR operation between the two operands and store the result in the destination.
Flag: none.
example: or Rsrc, Rdest -> Rdest = Rsrc | Rdest

SUB:

Perform a signed sub operation between the two operands and store the result in the destination.
Flag: C and F.
example: sub Rsrc, Rdest -> Rdest = Rsrc - Rdest

XOR:

Perform a logical bit XOR operation between the two operands and store the result in the destination.
Flag: none.
example: xor Rsrc, Rdest -> Rdest = Rsrc ^ Rdest

To test this important module we also made a testbench to perform test just on the ALU, this testbench runs test for every instruction, special cases and random cases. A special case occurs when we expect an overflow, add two highest possible values as an example. Even though our ALU pass ours special cases test, we have to perform a test with random value to test consistency. See code in alu_equiv_test.v, alu_random_test.v, alu_specific_test.v in appendix.

## 3.4 Decode and Control

The decode logic makes use of the regular structure of the modified baseline CR-16 opcode in order to split the instruction up with combinational logic. Each part of the instruction is routed to where it might be needed immediately after being fetched. The control logic is implemented as a Moore type state machine (fig 3.2, 3.3). Each state controls various write enables and external signals to route data through the system. Several data paths were broken into multiple steps to improve timing performance.

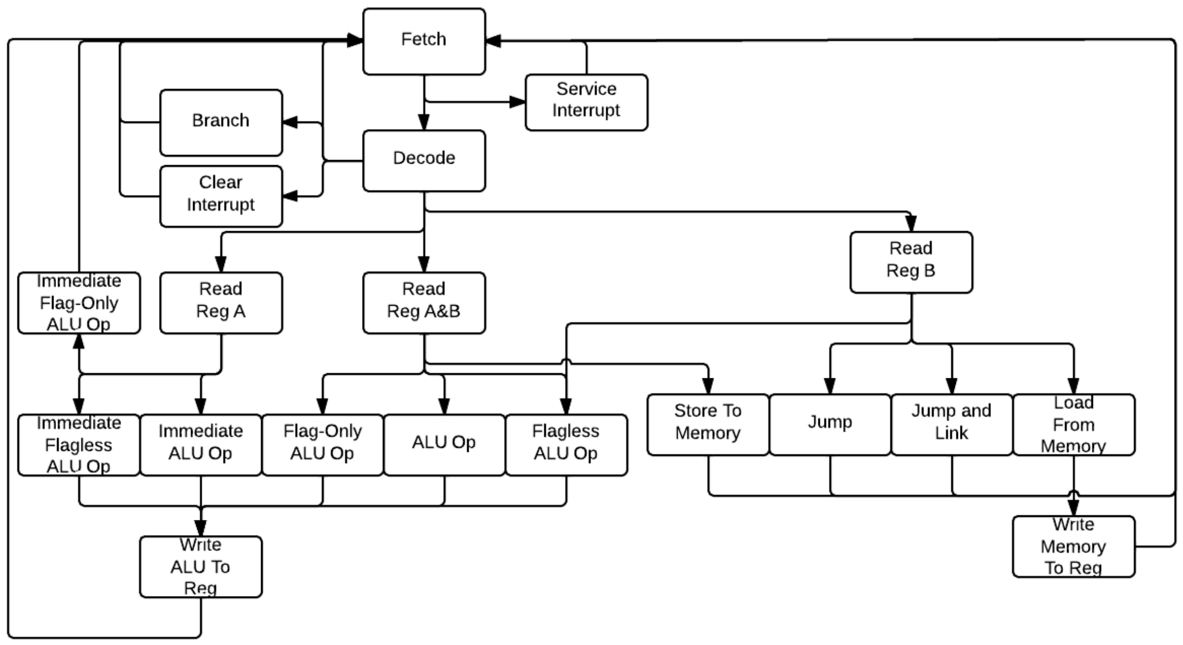

**Figure 3.2 - Control States**

## 3.5 Conditionals

Branches and jumps can be taken conditionally, depending on flags previously set by an ALU operation. A separate module examines the ALU flags and the cond bits of the current instruction to determine if the given condition is satisfied by the ALU flags. The result is fed into the PC module where it is used to select the appropriate PC operation.

## 3.6 PC

The PC has its own small ALU (fig 3.4). The 'A' input is the current PC. The 'B' input is tied to the main ALU's 'B' input in order to take advantage of the immediate value handling and B register selection. The PC ALU has four operations: A, A+1, A+B, B. The operation is normally A, causing no change to the PC. A+1 increments the PC to the next instruction. A+B is used for branches. B is used for jumps. The operation selected may depend on a flag condition. To account for this, a condition checking module reads the main ALU flags and a COND from the decode module and generates a COND_P signal if the current flags satisfy the COND. The decode module pulls the op code for the PC ALU from the instruction and generates a COND_PC signal when the PC op must be chosen conditionally. If the PC OP is conditional, COND_P is used to either select the PC OP from decode or A+1.

Since the B input of the PC ALU is from the main ALU B, is is a sign extended 16-bit value ready to add to the 16-bit unsigned PC. Signed verilog intrinsics are not used, so there are no conversation issues.

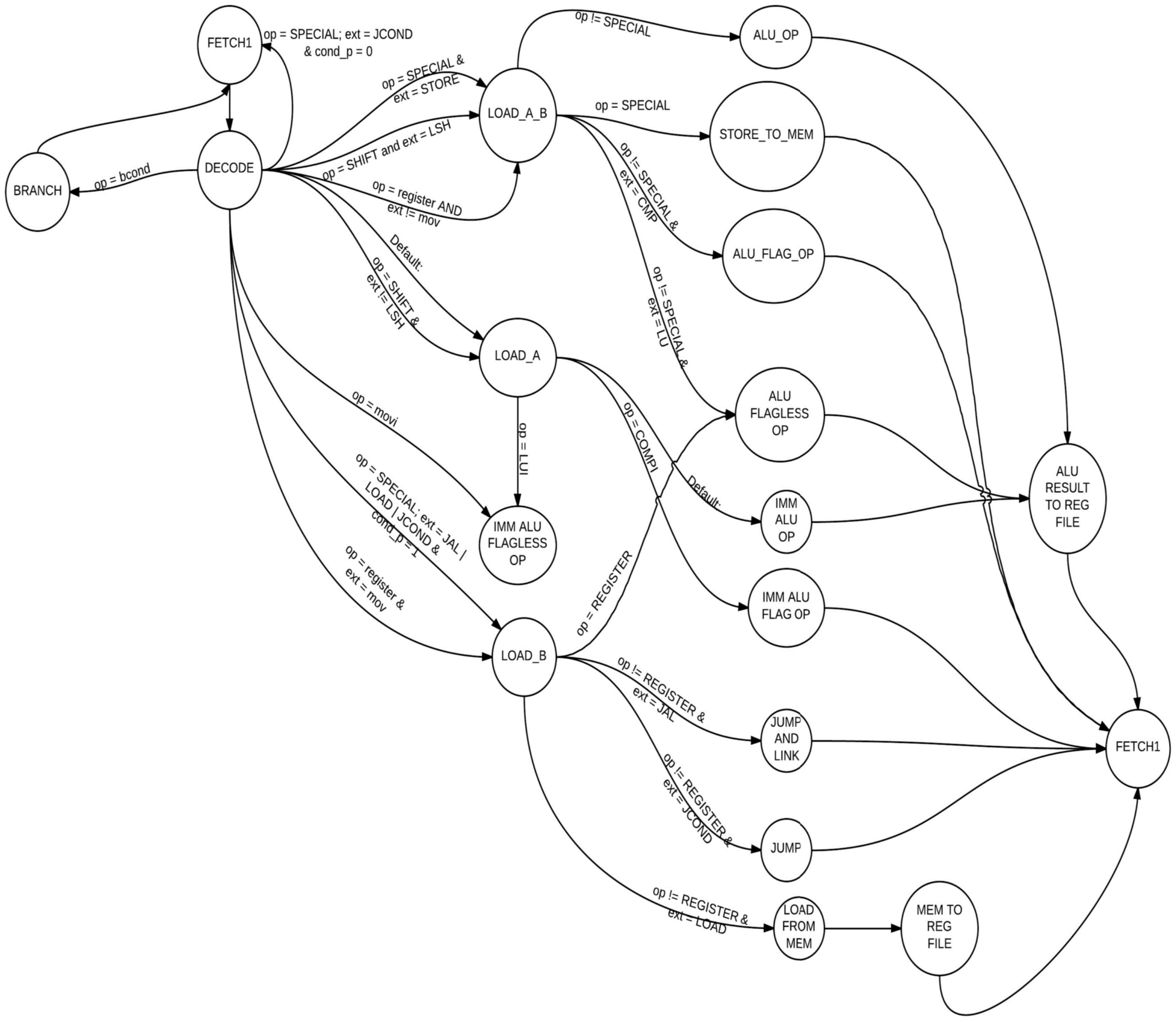

**Figure 3.3 - Control Finite State Machine and its firing signals.**

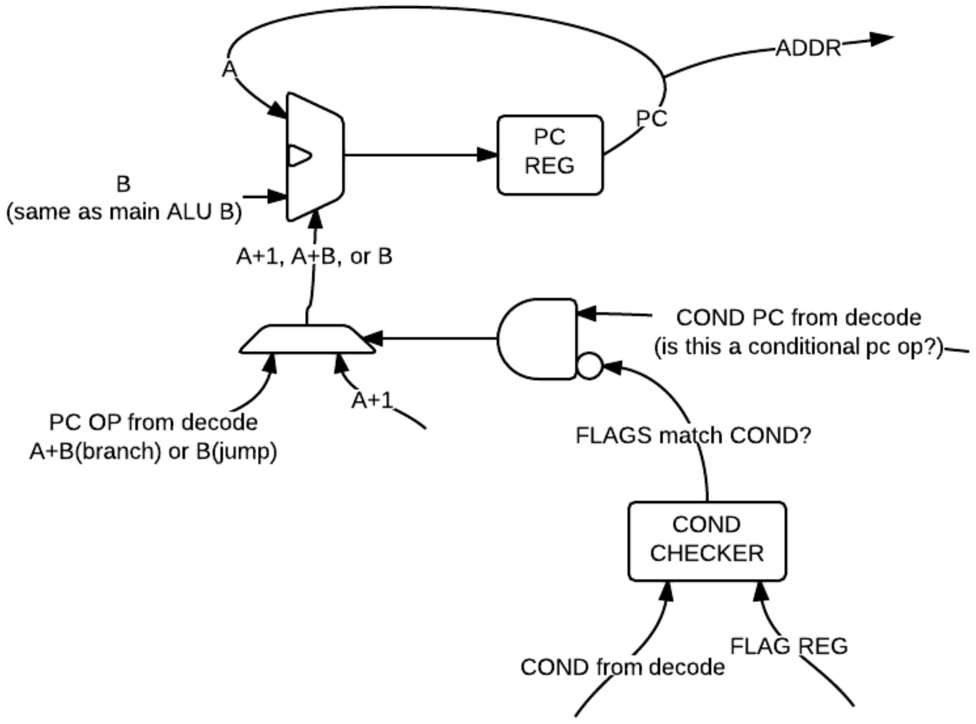

**Figure 3.4 - PC ALU**

## 3.7 Register File

The register file presents two read ports and one write port for an array of 16x16-bit words. Of these 16 register, 13 are plain registers implemented as distributed RAM.

One register acts as the PSR and has another read port that feeds the conditional module and a write port for accepting ALU flags. Write conflicts do not occur because the CPU is never in a state where it is writing to a register while writing ALU flags.

Two registers are hardware stacks (described in detail later). Operations which read from these registers implicitly pop the stack, while writes implicitly push. An operation which does both modifies the top off the stack. For example, `addi ps, 1` (ps being the parameter stack register) increments the value at the top of the stack, while `movi ps, 1` pushes 1 onto the stack.

The register file was originally a more standard asynchronous read configuration until timing issues were encountered. Now the register file is a synchronous design, requiring a clock cycle for reading or writing.

## 3.8 Stacks

Two hardware stacks are mapped into the register file: the parameter stack and the return stack. This configuration was inspired by the design of Forth machines. The return stack is used for return addresses (although it can be used for other purposes), often used with a jump and link instruction. The parameter stack is used for everything else, such as temporary storage or passing parameters.

Since the stacks are implicitly pushed and popped on access, signals need to be generated on such accesses. The control state machine tracks which registers are being accessed and generates the necessary signal to enable pushing or popping. The stacks are 64 words deep, each.

## 3.9 Interrupts

The CPU has some extensions to support the interrupt system. A new instruction, clri, has been added. When executed, the CPU generates a clear interrupt signal which is routed to the interrupt controller.

A pending interrupt flag and pending interrupt id register have been added. When the interrupt controller signals an interrupt, the pending flag is set and the id of the interrupt is stored.

During the DECODE state, the CPU asserts an interrupt request signal. On the next clock cycle, the interrupt controller will assert an interrupt if one is pending. Since the CPU has already moved on with the current instruction execution, the interrupt will not be handled until the next instruction cycle.

A SERVICE_INTERRUPT state has been added. When the CPU is in the FETCH state, if an interrupt is pending, the SERVICE_INTERRUPT state is executed instead of DECODE for the instruction that was fetched. This new state asserts an acknowledge signal for the interrupt controller as well as indicating the interrupt id being acknowledged. From there, something very much like a jump and link to an interrupt vector is executed. The pending flag is cleared, also.

## 3.10 Interrupt Controller

The interrupt controller has interrupt signal lines which it will logical or with a pending interrupt register each clock cycle. If the CPU asserts an interrupt request, the most significant pending interrupt bit id is sent to the CPU (if one is set and it is higher that the highest interrupt currently being handled) along with an interrupt signal. If the CPU asserts an interrupt acknowledge, the bit corresponding to the acknowledged interrupt id is set in a handling register and cleared in the pending register, effectively moving the interrupt from pending to handling. If the CPU asserts the clear interrupt signal, the highest handling interrupt bit is cleared.

The reason for the elaborate song and dance (fig 3.5) is to allow for nested, prioritized interrupts. By the time the CPU issues a clear interrupt signal, it has no idea which interrupt it is currently handling. The interrupt controller needs to keep track of this at all times, even if, say, the CPU is executing a clri instruction when a high priority interrupt is asserted, in order to clear the correct interrupt.

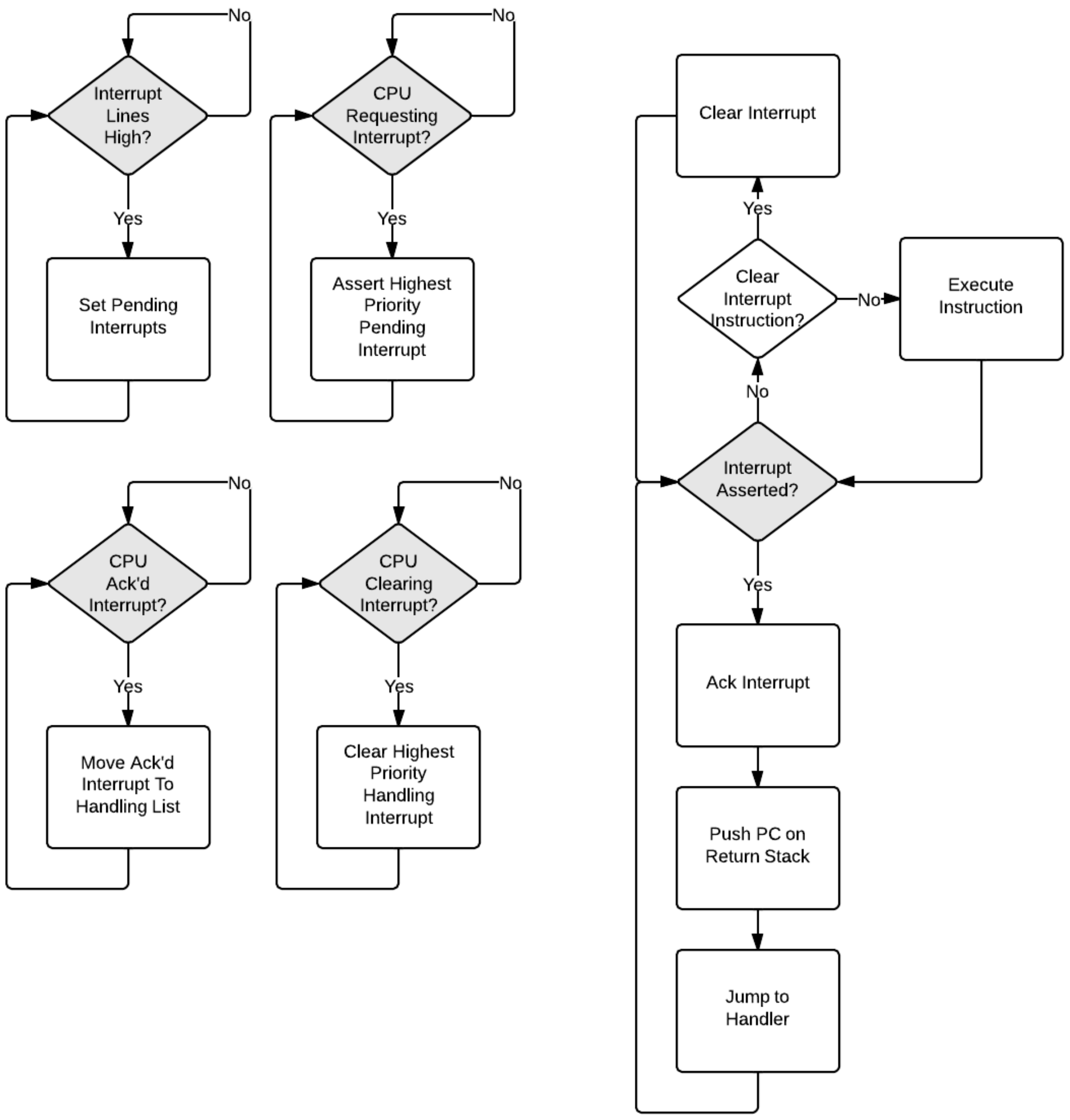

**Figure 3.5 - Interrupt Controller/CPU Interaction Flowchart**

## 3.11 VGA Controller

The VGA controller (fig 3.6) makes use of a pixel FIFO in order to separate the pixel clock and system clock domains. In doing so, it becomes very easy to accommodate pixel data sources with varying pipeline lengths. The scene generator waits for a pixel request signal from the controller, and begins pushing pixels into the FIFO some time later. The scene generator is responsible for keeping track of which pixel is being drawn; the VGA controller simply pops one pixel off the FIFO when it needs to draw one with no idea what coordinate it is drawing.

When the pixel request is deasserted, the scene generator should stop sending pixels. There is enough room in the FIFO for the generator to output more pixels while it clears its pipeline.

If the FIFO underflows, a resync signal is asserted. The generator is expected to flush its pipeline and reset its counters when this happens. When the controller again asserts the pixel request signal, it will be expecting the new pixels to be for the start of the next frame.

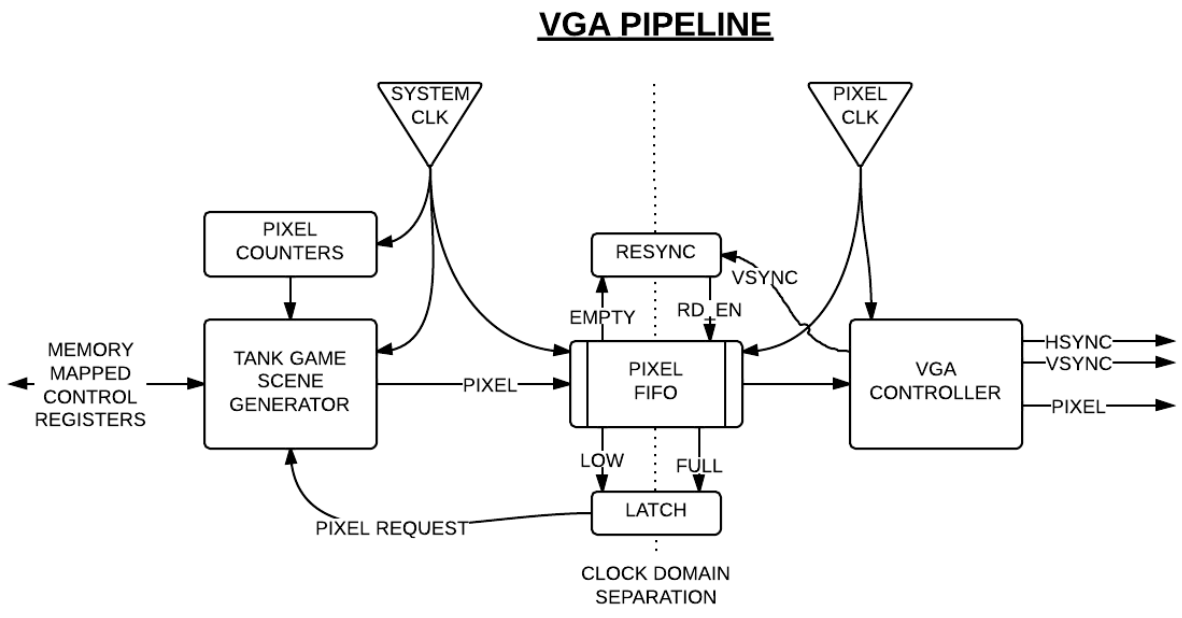

**Figure 3.6 - VGA Controller**

## 3.12 Peripherals

All peripherals were accessed through memory mapping. Additionally, some peripherals had interrupt lines running to the interrupt controller.

## 3.13 Memory Mapped IO

### 3.13.1 Timers

Two timers are present in the system. Their counter registers are mapped into memory so that they can be set from the application code. When a counter is not zero, it decrements once per millisecond, triggered by a clock divider dividing the system clock. When a counter transitions to zero, an interrupt is triggered for that timer. It is up to the application code to reload the counter register in the interrupt handler if a repeating timer is desired.

#### 3.13.2 PRNG

A linear feedback shift register (LFSR) is mapped into memory for use as a pseudo random number generator (PRNG). The LFSR is shifted once per system clock, which makes the random numbers much less predictable to the application code. The presence of user input interrupts makes the random number generation non-deterministic, since an unknown number of clock cycles could be used to handle user input at any time. The LFSR is 17 bits wide, with 16 exposed through the memory mapping. This allows the PRNG to return 0 without locking up the LFSR with a zero state, since the 17th unread bit will be 1.

#### 3.13.3 Onboard IO

Buttons, switches, LEDs and the 7-segment display on the Nexys 3 board are mapped to control registers so that the application code can read the inputs and display output.

### 3.14 Tank Game Graphics

A dedicated graphics module is included to render the tank game. It includes two hardware sprites for the tanks, one bullet sprite, and a ground height map. Control registers are mapped to allow setting the position of the sprites, as well as orientation for the tank sprites. A ground X index register can be set, after which a ground height register can be read or written to get or set the height of the ground at that column.

The graphics module also contains a PRNG to generate frame locked noise for the ground texture.

### 3.15 SNES Controller

A Super Nintendo Controller (SNES) was used as the main interface between the game and the player. The controller is responsible for the tank control, moving, aiming and firing, and also start a new scenario.

We built a memory mapped module which follows the SNES Controller protocol in order to communicate with the controller and get the buttons states and fires an interrupt request when a state of any button changes, then our interrupt handler can read the buttons states from the memory mapped register in this module.

The interrupt request fires a pulse only when a button state changes, which means that by holding the same button it will only fire one pulse.

## 4. Software

The software for the application is written in a domain specific language (DSL) hosted in the Ruby programming language. The application is written in an interrupt driven style where the main thread of execution waits for interrupts in a busy loop after initializing the application. The application code is written in Ruby code that looks like an assembly language describing the application, but is actually a program that assembles the application hex data when executed.

Ruby was chosen for a hosting language because of its support for writing natural looking DSLs, its meta-programming facilities, and its extremely fast development cycle. These features mean that the code for the application looks a lot like a normal assembly language in its basic form, but more powerful structures like conditional blocks can be made. All of the power of Ruby can be used during assembly. This made several tasks, such as generating sine tables or aliasing registers, trivially easy.

## 4.1 Program

The program (fig 4.1) begins with an interrupt vector section that has a branch for each one of sixteen interrupts. This code will be jumped to by the CPU when an interrupt occurs. After that is the initialization code where the terrain is generated, the tanks are placed, and the timers are initialized. After the initialization code is an infinite busy wait loop that will be executed whenever an interrupt is not being handled. All other game logic happens in interrupt handlers.

The ground is initialize with a simple Brownian method using the PRNG. Optionally, the ground can be added to a sine table, giving a hill. Each time the game is restarted, a new random terrain is generated.

Two main interrupts drive the game: the animation timer interrupt and the movement timer interrupt. The animation timer interrupt handles most of the games logic including bullet updates and collisions. The movement timer handles level restarts and updating the tanks' aim and position.

The bullet has a very simple physics system. The bullet's current X and Y position, X and Y speed, and acceleration due to gravity are stored in memory. Each animation update, the bullet's position is incremented by its speed and its speed incremented by its acceleration. These values are stored in two words, one for the screen space, whole pixel coordinate, and one for a fractional component. This allows the bullet to have a speed that can range from many pixels per animation update to many animation updates per pixel moved. Some application logic is dedicated to handling this fixed-point coordinate system.

Collisions are likewise simple. Since the terrain is exposed to the application as a 1D heightmap, collision with the ground is just a matter of checking to see if the height of the ground in the bullet's column is greater that the bullet's Y value. A bounding box around each tank is checked to see if the bullet is within it which registers as a collision. The bullet may also collide with the left or right edge of the screen.

On a collision, one of three code outcomes occur. If the bullet collides with the screen edge, the bullet is removed and it becomes the next player's turn. If a tank is hit, the other player win code is executed and the game stops. If the ground is hit, the ground explosion function is called and it becomes the next player's turn.

Ground explosions iterate over the ground height map around the collision point, subtracting a value calculated to resemble removing a circular chunk of dirt around the bullet. This leaves round craters and also allows digging through hills (very slowly).

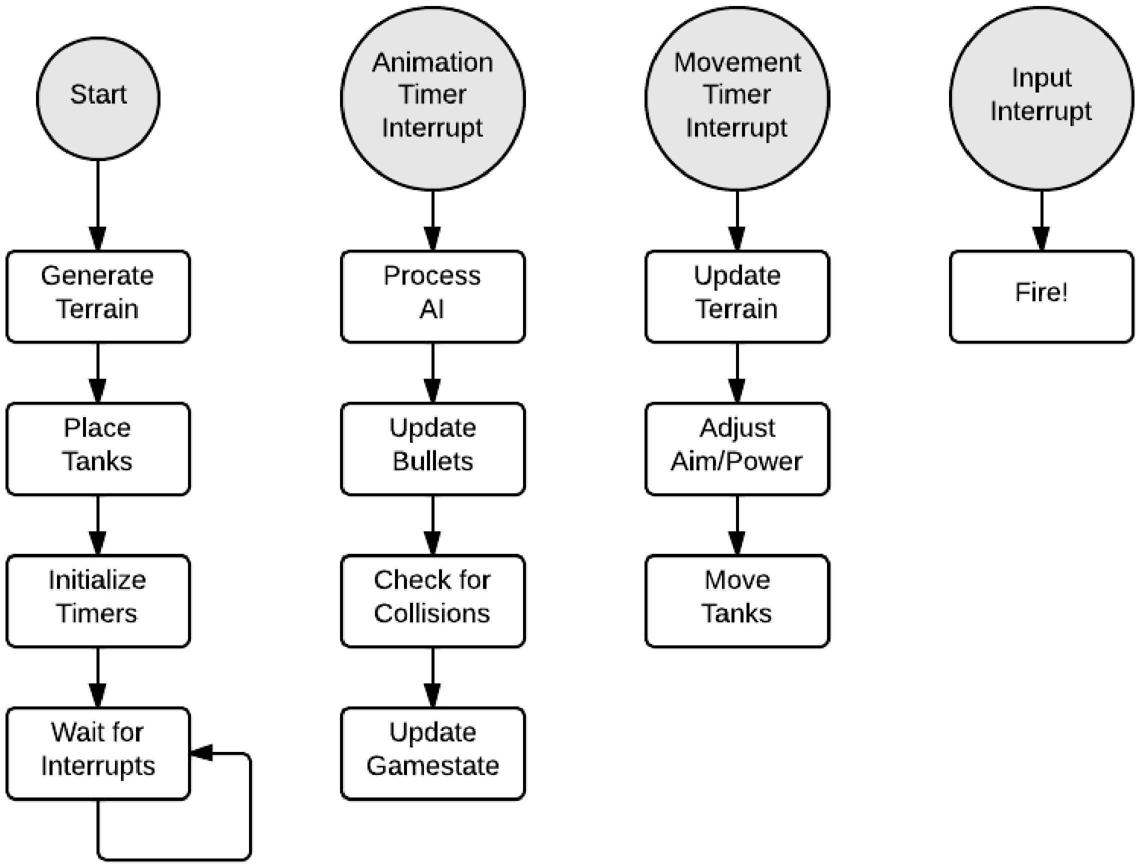

**Figure 4.1 -Simplified Application Flowchart**

## 4.2 Stacks

The availability of hardware stacks was originally to support the construction of a Forth system. Forth would have provided an easy to use on-board development environment and a higher level language to develop with.

The assembler delivered such an easy to use and powerful macro system that Forth was not worth implementing. Still, the stacks make many non-Forth tasks much easier. While there was some “Forth style” concatenative functions in the application code, the most use was in creating pseudo operations that did not need to worry about registers. Additionally, the return stack was vital for the nested interrupt and subroutine calling style of the program.

```
# stack annotations are formatted as:
# <vars expected on the stack when called> -- <vars put on the stack by this function>
func :random_shot do # -- power angle
       ...
end
func :deltas do # power angle -- dy dx
       ...
end
func :set_bullet_speed do # dy dx --
       ...
end
call :random_shot
call :deltas
call :set_bullet_speed
```

**Figure 4.2 - Concatenative Style Stack Functions**

Pseudo operations that needed some working registers were able to use the parameter stack, return stack, or both. For example, the tbit pseudo operation was able to test a bit in a register without altering the register or any temporaries because it had easy access to a stack.

```
def tbit(reg, n)
  mov ps, reg
  lshi ps, -n
  _andi ps, 1
  cmpi ps, 1
end
```

**Figure 4.3 - tbit Pseudo Operation Using the Parameter Stack**

## 4.3 Assembler

The assembler is a fairly simple two pass macro assembler. It includes support for labels which can be referenced before their definition, comments which are assembled into the resulting hex file, conditional blocks which assemble to the necessary series of branches and labels, function definition blocks that assemble register spilling and restoring code around the function code, assembling arbitrary data into the resulting hex file, and detailed error messages for programming errors like trying to branch too far, using an out-of-range immediate, or trying to use a register instead of an immediate.

The assembler works by defining functions that, when called, append a deferred instruction hex generation function to a list of such deferred functions that represent the program being assembled. Inside of an assemble block, the application programmer uses these function to write the application code. The PC is tracked during this process, and any labels introduced are associated with the proper PC. Pseudo operation and control blocks are expanded to machine operations at this point so that the list of deferred generators correspond 1-to-1 with machine instructions. When the end of the application code is reached, all labels have been resolved and hex generation can begin. The deferred functions are called and hex is generated with labels replaced with real memory offsets. The generated hex is printed to a file suitable for loaded with Verilog's $readmemh() function.

Extending the assembler is a simple process. Since Ruby is an interpreted language, no compilation step is needed to build the assembler. Adding pseudo instructions or more advanced control structures is as easy as editing the assembler code file and rerunning the assembler, which takes less than a second on modern computer when assembling the application code for this project.

```
def movwi(reg, w) # define the pseudo-op movwi for loading an immediate 16-bit word into
a register
  # check word range while assembling and show a reasonable error message
  throw "'word' is out of range" if w >= 2**16 || w < -2**15
  movi reg, w & 0xff # this & happens during assembly
  lui reg, (w >> 8) & 0xff if w > 0x7f || w < -128 # this conditional and this math
happens during assembly
end
```

**Figure 4.4 - Defining a Simple Pseudo Operation**

The assembler is 439 lines of Ruby code. The application code implementing the game is 767 lines of application specific Ruby code which assembles to 1,623 assembly instructions and 8,924 words of variable memory and lookup tables.

```
def preserve(*regs)
  regs.each {|r| mov rs, r}
  yield
  regs.reverse.each {|r| mov r, rs}
end
preserve r0, r1 do # pushes r0 and r1 onto the return stack
  ... code modifying r0 and r1 here ...
end # pops r0 and r1 from the return stack
```

**Figure 4.5  - Defining a Control Block with Advanced Ruby Features**

```
func :fire, r0 do # define a label :fire here, push r0 (implicit ‘preserve’)
  state = r0 # create an alias for r0
  lload state, :game_state # load a word into state (r0) from the labeled memory location
  cmpi state, @game_states.index(:player1_aiming) # cmp state with a constant from a
table
  eq? do # if eq, execute this block. otherwise branch over it
    switch 15 # cmp switch register with constant 15
    on? do # if on, execute this block
      call :random_shot # call a subroutine at the labeled memory location
      lstor ps, :tank1_power # push the value at the labeled memory location onto the
parameter stack
      lstor ps, :tank1_angle # push the value at the labeled memory location onto the
parameter stack
    end
...
```

**Figure 4.6 - A Typical Piece of Application Code**

## 5. Division of Labour

In this section it is described the labour division of the team during this course in a way that was possible to develop hardware and software using the best practices and the best knowledge of each member.

By far Joel had the best experience and knowledge for describing and doing the system. He has great knowledge of Ruby, system architecture and how software is integrated with hardware. He had described the Verilog, CPU, interrupts, stacks, VGA, assembler, application code.

Fred had worked in the description of the Verilog modules in the assignments, but after that the team decided to keep with Joel`s since they were more beautifully described. Also he worked in the writing tasks for the reports, in the system validation by doing testbenches and helped Arthur with the controller.

Arthur had also described the verilog, the SNES Controller and Tank glyphs. Also some testbenches during the CPU modules and worked in the reports.

## 6. Conclusion

Looking at the work develop it is possible to conclude that hardware and software should be developed together. Changes in one might effect the other and vice versa. This project had that integration between the two "interfaces" of a computational system.

This project had a dedicated hardware to implement the Tank game, but the same time it had a general purpose architecture that could handle any applications. In addition it became clear that design patterns in the modules description can affect the performance and therefore it was necessary to plan well the modules and describe them using simple language which allowed the Xilinx to perform its optimizations and improve the architecture performance.

Working in a complete design flow, hardware and software, was a great experience for the them since some of team members had never experienced that flow, now they know how an entire system works. The team conclude saying this class was amazing and definitely this experience was unique for our careers and personal knowledge.

## 7. Future Work

While having a dedicated graphics module was a good decision for this game, having a more general purpose frame buffer base graphical system would be a good to have.

An audio system could be added. Using the interrupt system would allow interesting hardware/software audio system hybrids. For example, software could feed hardware audio buffers periodically from an audio system interrupt.

It would be interesting to see how efficient a Forth implementation would be on this system. The register mapped stacks were designed specifically to support Forth operations. No further hardware changes should need to be implement to create a Forth system, although many stack machine specific optimizations such as smaller opcodes and simplified data paths could be implemented.

The CPU is entirely unoptimized. There is no pipelining, no variable width opcodes, no branch prediction. Each of these would be a worthy future endeavour.

## 8. References

VGA Timing
tinyvga.com
http://tinyvga.com/vga-timing

Linear Feedback Shift Registers
Wikipedia
http://en.wikipedia.org/wiki/Linear_feedback_shift_register

Forth
“Starting Forth” by Leo Brodie
http://www.forth.com/starting-forth/

Forth Implementation
“Moving Forth” by Brad Rodriguez
http://www.bradrodriguez.com/papers/moving1.htm

Stack Machine Design
“Stack Computers: the new wave” by Philip J. Koopman, Jr.
http://www.ece.cmu.edu/~koopman/stack_computers/

SNES Controller protocol
http://www.gamefaqs.com/snes/916396-snes/faqs/5395