Branch Prediction Game

The Troublemakers

Brody Skaleski, Josh Bowers, Anish Munimadugu, Ayan Deep Hazra

Project Final Report ECE 554 Spring 2023

Advisor: Eric Hoffman

University of Wisconsin-Madison

Table of Contents

1. ISA Table

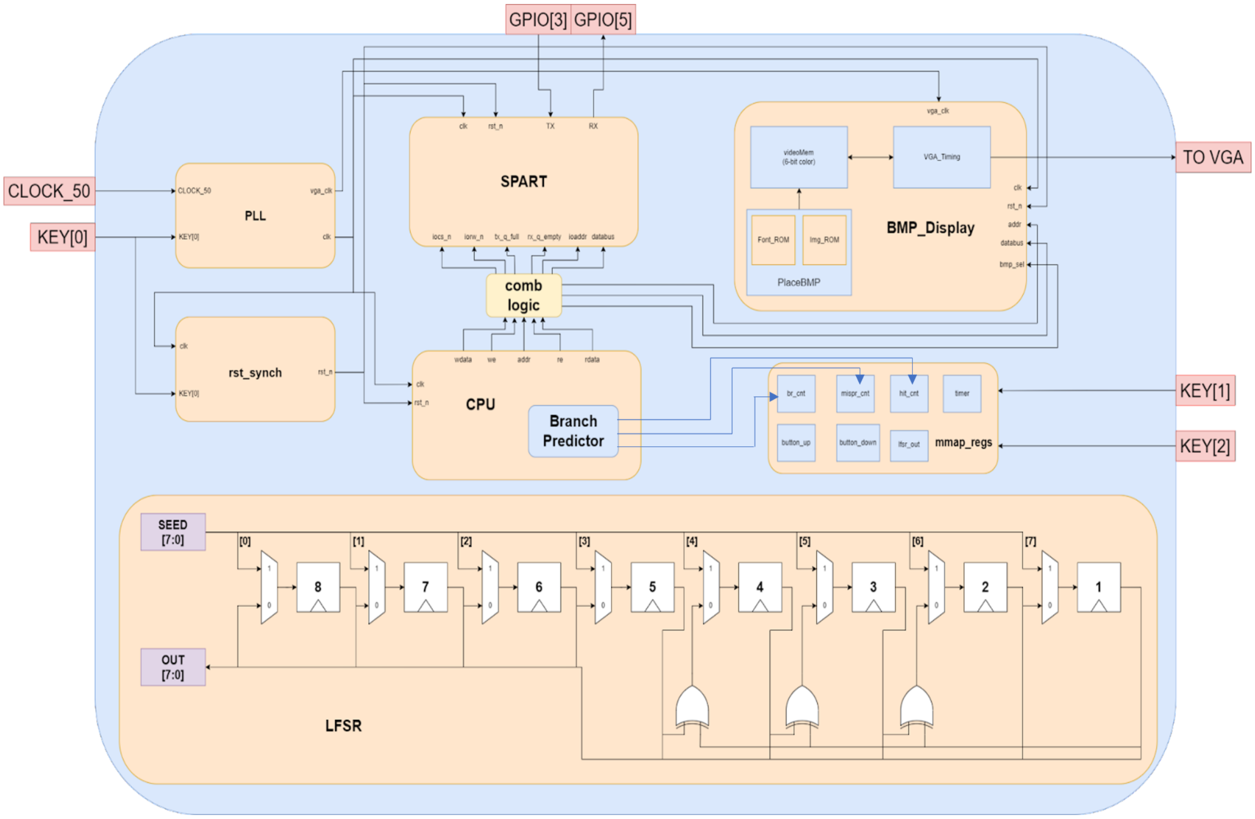
The first five bits of the encoding symbolize the opcode assigned to the instruction. dddd indicates in binary the register which serves as the destination (usually on the left hand side of the operation, unless otherwise specified). ssss refers to the first or only register in binary that serves as the input operand to the instruction. tttt may refer to an optional second operand depending on the instruction. dddd, ssss, tttt in the explanation column take random registers as examples to display applicability. iiii refers to an immediate if present in the instruction and is henceforth referred to as IMM for clarity.

Mnemonic	Encoding	Sample Instruction	Explanation	Comments
Add	00000 dddd ssss tttt	Add R3, R2, R1	R3 <= R2 + R1	ALU arithmetic. Leverages ALU unit to perform basic add, subtract, and and xor instructions and some variations on the same. Add, Addz, and Sub raise Z, N, and V flags for condition codes All other instructions in this group only raise the Z flag for condition codes
Addz	00001 dddd ssss tttt	ADDZ R6, R5, R4	R6 <= R5 + R4 only if Z=1	
Sub	00010 dddd ssss tttt	Sub R9, R8, R7	R9 <= R8 – R7	
And	00011 dddd ssss tttt	And R12, R11, R10	R12 <= R11 & R10	
nor	00100 dddd ssss tttt	nor R10, R12, R13	R10 <= ~(R12 | R13)	
Shift				
Sll	00101 dddd ssss iiii	Sll R1, R0, IMM	R1 <= R0 << IMM	
Srl	00110 dddd ssss iiii	Srl R1, R0, IMM	R1 <= R0 >> IMM	
Sra	00111 dddd ssss iiii	Srl R3, R2, IMM	R3 <= R2 >> IMM	
Load/Store				
Lw	01000 dddd ssss iiii	Lw R7, R6, 5	R7 <= mem[R6+5]	load reg into mem
Sw	01001 dddd ssss iiii	SW R12, R14, 13	Mem[R14 +13] <= R12	store reg into mem
Lhb	01010 dddd iiiiiiii	Lhb R13, 12	R13 <= {12, R13[7:0]}	load upper 8 bits into reg
Llb	01011 dddd iiiiiiii	Llb R12, 11	R12 <= sign-extend{11}	load lower 8 bits into reg. Auto sets high byte/top 8 bits to value of the 8th bit (sign extension) unless followed by a relevant Lhb instruction.
Branch				
Neq	01100 ccc iiiiiiiiiii	b neq, label	Branch if Z=0	Branch target address = (Address of branch instruction + 1) + offset. PC holds word addresses, each instruction is 1 word, offset is specified as the number of instructions with respect to
Eq	01100 ccc iiiiiiiiiii	B eq, label	Branch if Z=1	
Gt	01110 ccc iiiiiiiii	B gt, label	Branch if {Z,N}==2'b00	
Lt	01110 ccc iiiiiiiiiii	B lt, label	Branch if N=1	
Gte	01110 ccc iiiiiiiiiii	B gte, label	Branch if N=0	
Lte	01110 ccc iiiiiiiiiii	B lte, label	Branch if N=1 or Z=1	
Ovfl	01110 ccc iiiiiiiiiii	B ovfl, label	Branch if V=1	

Uncond	01110 ccc iiiiiiiiii	B uncond, label	Branch unconditionally	the instruction following the branch instruction.
Jump				
JAL	01101 iiiiiiiiiiii	JAL label	R15 <= (Address of jal instruction + 1) Jump to target address	Jump target address = (Address of jal instruction + 1) + offset
JR	01110 iiiiiiiiiiii	JR R15	Jump to target address given by contents of R15	Return from function calls.
NOOP	01111 xxxxxxxxxxxx	NOOP	No operation.	No operation.
Immediate				
ADDI	10000 dddd ssss iiii	ADDI R3, R2, IMM	R3 <= R2 + IMM	ALU arithmetic but uses an immediate value (1, 2, 10, tc) instead of the contents of a reg. ADDI and SUBI sign extended. others zero extended. ADDI and SUBI raises Z, N, and V flags for our condition codes Other immediate instructions raise Z flags for condition codes
SUBI	10001 dddd ssss iiii	SUBI R3, R2, IMM	R3 <= R2 - IMM	
XORI	10010 dddd ssss iiii	XOR R3, R2, IMM	R3 <= R2 ^ IMM	
ANDNI	10011 dddd ssss iiii	ANDNI R3, R2, IMM	R3 <= ~(R2 & IMM)	
ANDI	10100 dddd ssss iiii	ANDI, R3, R2, IMM	R3 <= R2 & IMM	
XORNI	10101 dddd ssss iiii	XORNI R2, R1, IMM	R2 <= ~(R1 | IMM)	
ORI	10110 dddd ssss iiii	ORI R2, R1, IMM	R3 <= R2 | IMM	
Misc.				
ANDN	10111 dddd ssss tttt	ANDN R1, R2, R3	R1 <= ~(R2 & R3)	ANDN calculates the NAND operation. ANDN only raises the Z flag.
MOVC	11000 dddd ssss iiii	Movc R3, R2, IMM	R3 <= InstrMem[R2 + IMM]	Movc stalls execution for one cycle. It retrieves data from instruction mem indexed by PC value stored in R2 added to IMM. Since our instructions are 17 bit and we retained a 16 bit register file, we forego MSB in return. MOVC raises Z, N, and V flags for condition codes.

UMUL	11001 dddd ssss tttt	UMUL R3, R2, R1	R3 <= unsigned(R2*R1)	Bitwise unsigned Multiplication. If overflow occurs then we discard the top 16 bits and raise the overflow flag. UMUL raises Z, N, and V flags for condition codes.
SMUL	11010 dddd ssss tttt	SMUL R3, R2, R1	R3 <= signed (lowest 8 bits(R2) * lowest 8 bits(R1))	8 bit by 8 bit signed bitwise multiplication. SMUL raises Z, N, V flags for condition codes.
XOR	11011 dddd ssss tttt	XOR R7, R8, R9	R7 <= R8 ^ R9	XOR calculates the XOR operation. XOR only raises the Z flag.
Other notes:	1. Flag registers are Z-zero, V-overflow(positive and negative), N-negative/sign 2. R0-R13 are general purpose registers 3. R15 is the return address for JAL. Shouldn't write to this register 4. R14 is our stack and base pointer 5. PUSH and POP are special keywords that the assembler will turn into an SW followed by ADDI and SUBI followed by LW respectively.			

2. Hardware Block Diagrams

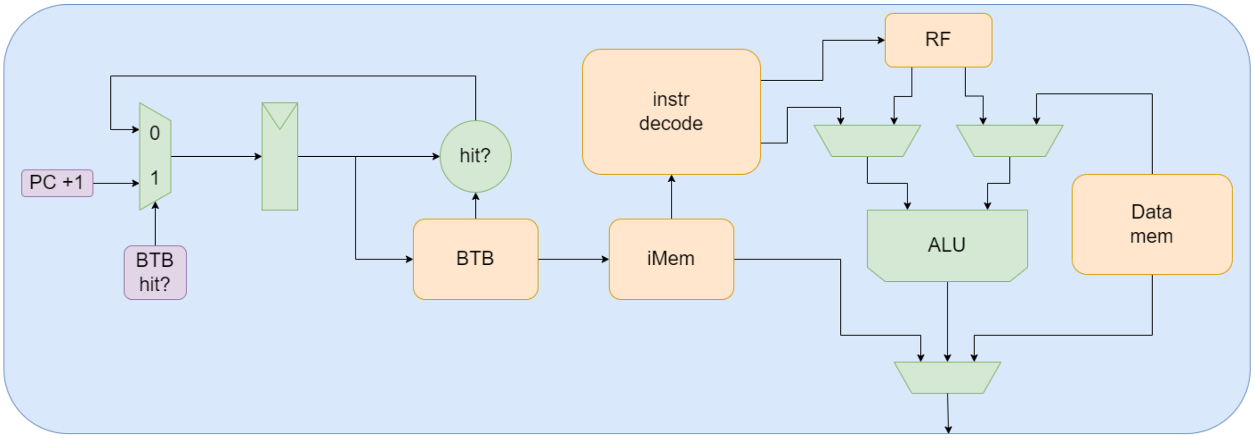


The Hardware Diagram is a high-level abstraction of our wrapper file. This file includes the instantiations of the main blocks that perform the many functions needed for our project.

- PLL - Takes in a 50MHz clock and outputs a clk signal that the rest of our logic can use.
- rst_synch - Takes in the raw signal from KEY0 and stabilizes it so the rest of our logic can use it.
- CPU - The “brains” of our project. Contains the logic that will control our project, as well as all the new ISA additions as well as our branch predictor.
- SPART - Logic used to allow branch predictor stats to be displayed on a terminal from the source computer. Stats displayed include prediction hits and prediction misses.
- BMP_Display - Logic used to display pictures on the screen for the user to be able to interact with. Can place a spaceship and a meteor that is used in the game, as well as clear the screen when resetting the game.
- Linear Feedback Shift Register (LFSR) - Pseudo random number generator to determine where a meteor will spawn on the screen. The LFSR used in our design computes the output in parallel with the shifting in order to reduce performance impact.
- Branch Predictor - Will either unconditionally take or not take a branch depending on the stats saved in the Branch Target buffer.
- Control Logic - Misc. logic needed for the SPART, BMP_display, and for mmap_regs

2.1 CPU

One complex functional unit we all have that needs further subdividing is the CPU. We present a block diagram below.



BTB - Cache that holds the PC of the branch in a cache. If hit, the branch is immediately taken

iMem - loads the hex file

Instr Decode - 17-bit instruction decoded. Output source buses for execute stage

RF - Triple ported register file. Two read ports, and one write port. Data is written on clock high, and read on clock low.

Data mem - Single ported, can read or write but not both in a single cycle. Precharge on clock high, read/write on clock low

Src-mux - determines if muxes are bypassed, or data from register file or data memory is used by the alu

ALU - Performs ADD, SUB, AND, NOR, SLL, SRL, or SRA based on func input. Provides OV and ZR outputs.

2.1.1 PC (Program Counter) Interface

Signal:	**Dir:**	**Description:**

Flow_change_ID_EX	In	Indicates PC will be updated due to branch or jump (from ID unit)
Stall_IM_ID	in	Asserted if stalling pipe due to load/use or MOVC-related stall
Dst_ID_EX[15:0], Dst_EX_DM[15:0]	In	Branch target address from ALU
Btb_hit	in	If asserted, PC should update to btb output
Btb_nxt_pc [15:0]	in	Output of BTB, indicates branch prediction result – target PC
Btb_hit_ID_EX	in	Piped in to determine if there was a branch misprediction (must flush)
LWI_instr_EX_DM	In	For MOVC
Pc[15:0]	out	Forms address to instruction memory and is selected from mux logic within the pc flop. Resets to 0, chooses pc_pre_movc_mux if there is no flow change, and output of dst mux if there is a flow change. If there is a MOVC command, it chooses pipelined dst mux signals.
Pc_ID_EX[15:0]	out	Piped version needed in EX stage for computing new branch target.
PC_EX_DM[15:0]	out	Piped to EX_DM for JAL instruction store in R15.
pc_pre_mux[15:0]	out	Used to index into the BTB

2.1.2 PC (Program Counter) Registers

Register:	Description:

pc, pc_ID_EX, pc_EX_DM All 16-bits	Described in the table above.
pc_IM_ID	First pipeline stage of nxt_pc, pipelines it into pc_ID_EX.
pc_pre_mux[15:0]	Pc register that stores dst_ID_EX if there is a flow change and nxt_pc if there is no flow change.

2.1.3 BTB (Branch Target Buffer) Interface

Signal:	**Dir:**	**Description:**
en	In	Enable branch prediction
PC [15:0]	in	Program counter used to index into BTB
flow_change_ID_EX	in	Whether a flow change occurred in the EX stage. Used to determine branch taken/not taken/ mispredictions etc
br_instr_ID_EX	in	Is the instruction in EX a branch or not? Used for btb write logic
pc_ID_EX [15:0]	in	Pipelined PC, for writing back to BTB
dst_ID_EX [15:0]	in	Alu output – destination after branch decision is made. Used to write back to BTB
target_PC [15:0]	out	Target PC prediction from the btb_mem.

2.1.4 BTB (Branch Target Buffer) Registers

Register:	Description:
[24:0] btb_mem [0:511]	512 deep branch target buffer memory unit. 25 bit cache lines encoded as below: TAG[24:18] || S[17] || V[16] || target_PC[15:0]
[24:0] btb_out	Single cache line output of the btb_mem
btb_hit_IF_ID, btb_hit_ID_EX	Hit in the BTB. Piped to EX to determine if misprediction.
sbit_IF_ID, sbit_ID_EX	Strong bit. If set for a given line, demote if misprediction. Another misprediction invalidates the line. Strong bit should be set for a given line if an accurate prediction is made. (btb_hit and branch is taken)
target_PC_IF_ID, target_PC_ID_EX	Pipelined target_PC signal. When the previous target_PC needs to be retained (misprediction but strong bit was high)

2.1.5 src_mux (Source Mux) Interface

Signal:	**Dir:**	**Description:**
[2:0] src0sel_ID_EX	In	1 bit wider than Eric's implementation because of more sources for the 0 src bus. We added a mux select that zero extends the 4-bit immediate for use in arithmetic immediate instructions (ADDI, SUBI).

2.1.6 Dst_mux (Destination Mux) Interface

Signal:	**Dir:**	**Description:**
[16:0] instr	In	instr value of the requested instruction through MOVC.
LWI_instr_EX_DM	in	Pipelined signal that indicates to dst_mux that it has to load instr into the register file.

2.1.7 ID (Instruction Decode) Interface

Signal:	**Dir:**	**Description:**
[16:0] instr	In	17-bit instruction from IM.

2.1.8 ID (Instruction Decode) Registers

Register:	Description:
[2:0] src0sel_ID_EX	1-bit wider than Eric’s implementation because of more sources for the 0 src bus.
LWI_instr, LWI_instr_ID_EX, LWI_instr_EX_DM	1-bit pipelined value that kickstarts a MOVC instruction.

2.1.9 IM (Instruction Memory) Interface

Signal:	**Dir:**	**Description:**
[16:0] instr	out	Increased bit width to 17 to accommodate 17-bit instructions.

2.1.10 IM (Instruction Memory) Registers

Register:	Description:
[16:0] instr_mem [0:2047]	17-bit instruction memory structure that holds all the instructions from hex.

2.1.11 ALU Interface

Signal:	Dir:	Description:
[3:0] func	in	4-bit func increased from 3-bit in Eric’s implementation. Contains a bitwise map of current ALU operations to perform. By default is ADD.

2.2 Branch Target Buffer - BTB

2.1.11 BTB Block Diagram and working

The complex and fully original unit added to our design is the Branch Target Buffer, which is a dynamic branch predictor unit. At its core, it contains a 256 deep cache. Each cache line contains a 25 bit entry that is encoded as follows:

TAG [24:18]	S[17]	V[16]	TARGET PC [15:0]

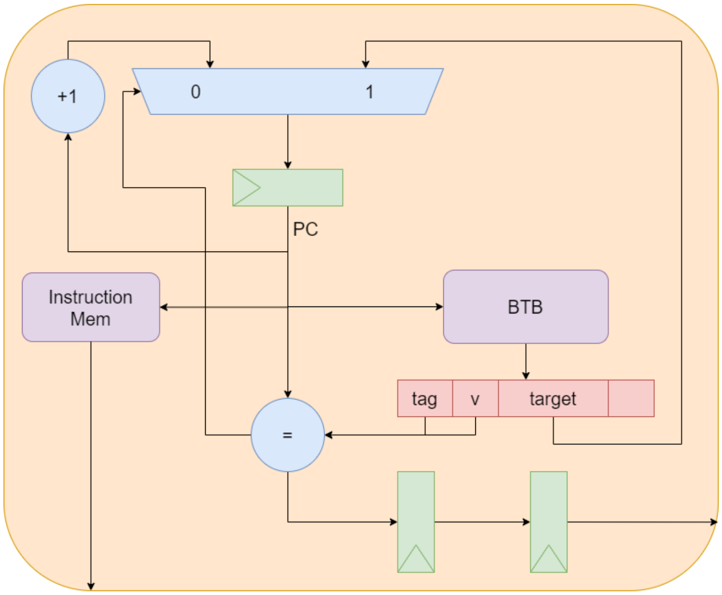
On reset, the entire cache is reset to 0 and every cache line is set to hex 0.

The 3 inputs from ID that the BTB borrows and acts on are the flow_change_ID_EX, stall_IM_ID and br_instr_ID_EX, which signal if there has been a change in flow, a stall and if any branch instruction has been executed or not, respectively.

The BTB is indexed into using the current PC to determine if this instruction has a prediction available. This will be the case if the program has encountered this particular branch instr before and remembers that it was a branch taken case.

- If there's a hit in the BTB, the next PC is fetched accordingly (assuming branch is taken). A couple cycles later, in the EX stage on branch decision, if this turns out incorrect then the line is invalidated, to prevent this misprediction from happening again.
- If there's a miss in the BTB, meaning the CPU should assume branch not taken, then the PC simply increments to PC+2. Again, in the EX stage, if this is proven incorrect, the BTB is then written to in order to remember that this PC took the branch and the next time, we will assume the branch taken.

The "Strong Bit" is an additional layer that effectively works as a 3 state machine to prevent stray evictions or allocations from the BTB. Each line has a strong bit. When a prediction is proven correct it's labeled a "Strong" prediction. Now, a single misprediction will simply demote the cache line to a regular prediction instead of directly invalidating the line. Another accurate prediction will set the Strong bit again.



Currently, the cache needs to be reset when the FPGA is physically reset. This is done by passing an async reset to the 256 deep BTB, which is around the upper limit of a synthesizable memory module size that can be asynchronous reset.

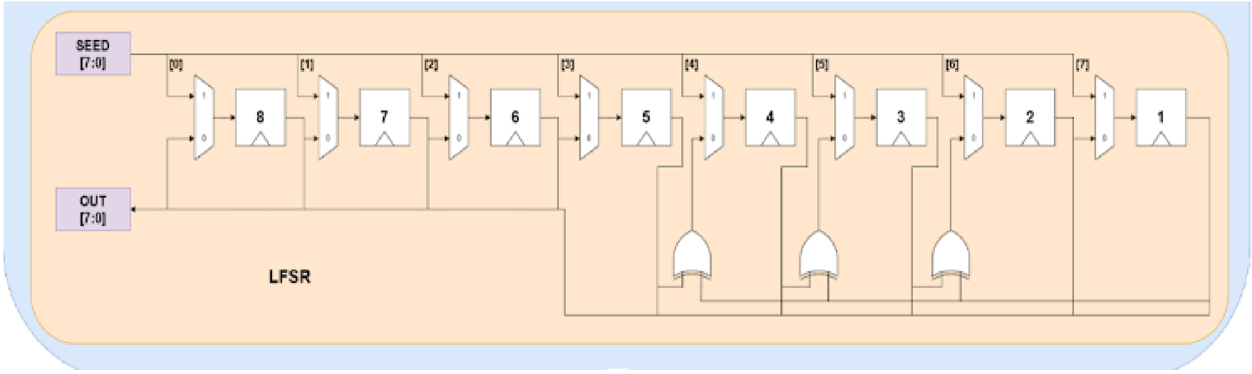
Another feature of the BTB that was included was the reset. One of the switches [SW[0]] was mapped to be a switch that can turn Branch prediction on or off. This was done using an enable signal and a dedicated register in the ASM.

2.1.11 BTB Statistics

We keep track of branch prediction statistics using control signals that are ultimately ported out of the BTB into the memory mapped registers. There, they are used to increment the particular statistics. Primarily, we keep a record of the total number of branches, the number of hits to the BTB and the number of mispredictions caused as a result of our BTB.

2.3 Linear-Feedback Shift Register - LFSR

A linear-feedback shift register (LFSR) is a shift register whose input bit is a linear function of its previous state. The most commonly used linear function of single bits is exclusive-or (XOR). In our implementation, a XOR function was used to generate pseudorandom behavior in the stored bits.



More specifically, we included a XOR gate ahead of the inputs to the 2nd, 3rd and 4th Registers in our design. We fed registers 1 and 3 as the inputs to the XOR of the 2nd, 1 and 4 as inputs to the XOR of the 3rd and 1 and 5 as the inputs to the XOR of the 4th. This ensures that the outputs of the registers are loosely related to each other. The other registers are all fed random outputs of the remaining registers.

In our ASM code, a random seed was assigned to the LFSR's initial condition, which served as the starting point for the LFSR. In our game, the output of the LFSR would ultimately dictate what lane a meteor would shoot out of.

2.4 Memory Mapped Registers Module

Since our project is very application heavy, we needed several registers that would keep track of various inputs and outputs of the FPGA execution. We decided to place all of these together in the mmap_reg module.

Memory Address in Hex	Description
0xC00F	Keeps track of VGA module's status in clearing out screen.
0xC010	Counter for the number of branches in BTB.

0xC011	Counter for the number of mispredictions in BTB.
0xC012	Counter for the number of hits in BTB.
0xC013	Timer that counts up.
0xC014	Button Up [KEY[1]]
0xC015	Button Down [KEY[2]]
0xC016	LFSR output.

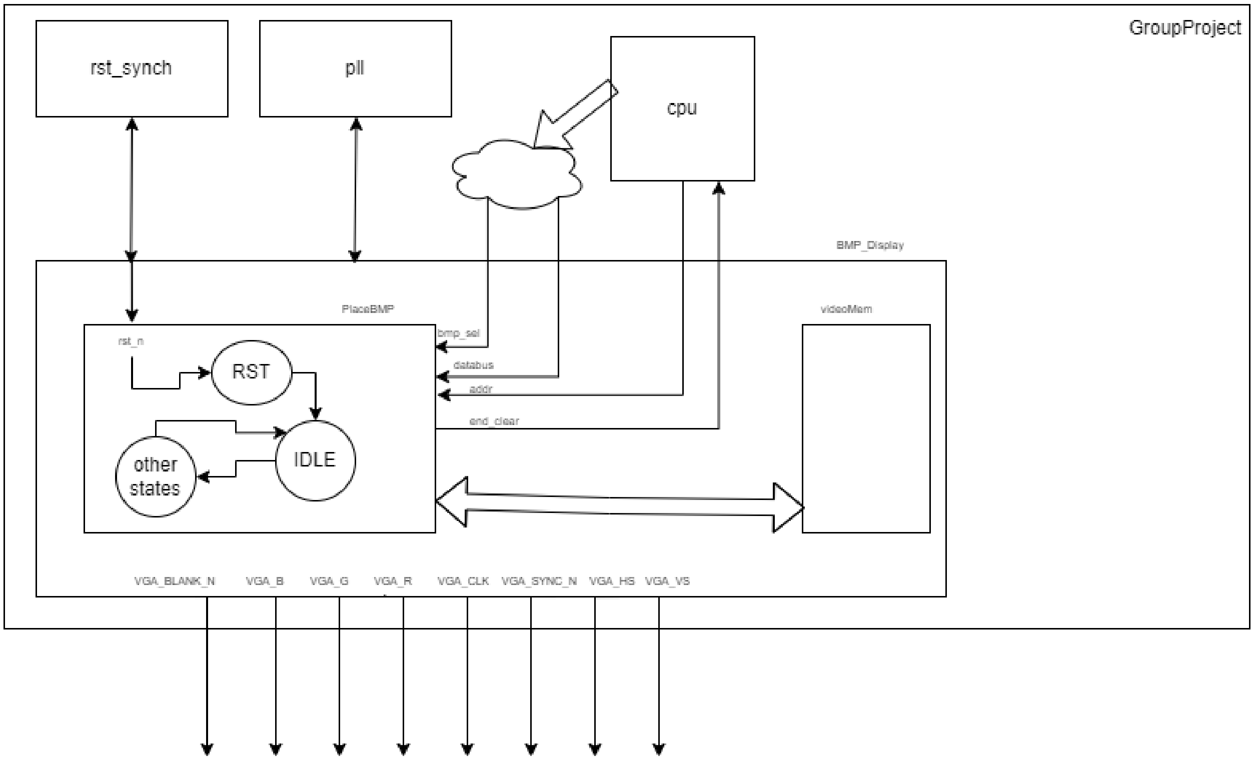
In addition we had several other memory locations that were mapped outside of the mmaps_reg module due to proximity with other functionalities. They were all mapped in the top level module GroupProject.

Memory Address in Hex	Description
0xC001	Reads 10 bit value from Switches.
0xC004	Transmit Buffer (IOR/W = 0), Read Buffer (IORW/N = 1)
0xC005	Status Register (IOR/W = 1)
0xC006	DB (low) Baud rate division buffer low byte
0xC007	DB (low) Baud rate division buffer high byte
0xC008	Encodes the X coordinates of an image being placed on a VGA screen.
0xC009	Encodes the Y coordinates of an image being placed on a VGA screen.
0xC00A	Write to VGA screen signal.
0xC00B	Branch Stats write signal.

2.5 VGA Module

We had some small changes to the VGA module which included adding a hard async reset that would completely clear the screen when the reset mapped key was pressed on the FPGA. This was ultimately done by adding an extra state to the Place_BMP state machine wherein the rst signal would trigger it, and would continue being in, till it had cleared all the videoMem addresses and set them to hex 0 (which is the color black). The idea was that at a collision, the player could start with a clean slate and replay the game.

We also encoded an end_clear signal which was memory mapped so the game ASM could keep track of it in a forever loop. Once the end clear jumped to high, the loop would break and normal execution would continue. We had some concerns with this being very time consuming because of the width of the videoMem but ultimately on testing it was shown that this process took no more than 2 seconds.



3. Software

Our software can be broken down into 3 parts. Processor firmware testing of Appended ISA instructions and assembler directives (2.1), game code and firmware (2.2), and game code testing (2.3).

3.1 Thorough Testing of Appended Instructions and Assembler Directives in ASM

We have rigorously tested our appended instructions, including all immediate instructions in our ISA, miscellaneous instructions such as ANDN and UMUL/SMUL, the MOVC instruction, and assembler directives PUSH and POP. We used several detailed ASM files to test for instruction correctness by testing operation correctness and edge cases, which was compiled using our modified Perl assembler.

The test cases are detailed in the succeeding subsections.

3.1.1 Immediate instructions

For the immediate instructions we ensured that the correct data value is selected at the source mux enabled in our id stage.

This is done by,

1) Testing the correctness of basic operations to see if the initial implementation of the instructions is correctly calculated and passed through the processor.

2) Testing the immediate instructions remains correct if the immediate use is at its minimum and maximum value. In most cases this was a 4 bit value, so 0 to 15 in decimal.

3) Testing for correct behavior when register values overflow, particularly with ADDI and SUBI

4) Testing to see if the zero and overflow flags are raised when a zero, or overflow is seen. This was tested by using a sequence of B eq instructions to test the zero flag or a B ov instruction to see if the overflow flag is raised.

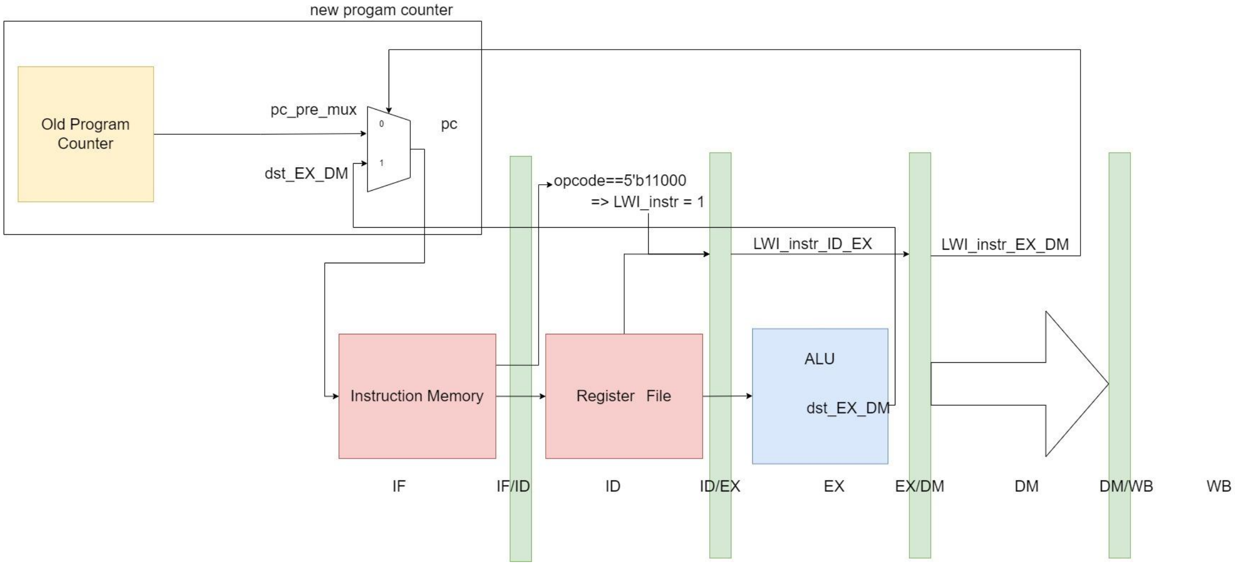
3.1.2 MOVC instruction

Our MOVC instruction tests test for correctness by looking at two types of tests,

1) The first checks if a single isolated MOVC instruction flows correctly through the pipeline. Two types of instructions, register loads before any main execution, and String Memory instructions at the bottom of the memory instruction were written.

2) The second checks if two MOVC instructions in a row work in the pipeline with similar sub-cases as before.

3) The last checks if our attempt to beat the deadlock condition is caused by reading three MOVCs instructions in a row.



The block diagram above shows how the MOVC pipeline works. Familiar blocks which do not affect the pipeline are not elaborated on and are assumed to contain their normal functionality.

3.1.3 Miscellaneous instructions

Miscellaneous instructions include all instructions already provided with Eric's given processor and any non-immediate instructions such as ANDN and MUL. ANDN and MUL are our primary focus because we added these instructions and the given instructions are assumed to be correct in legacy.

1) The first runs the given processor instructions to ensure any immediate instruction we added to the ISA did not break the other instructions. This was crucial in the main game_and_stats asm (the one that holds our game's execution loops) as it governed a big chunk of our register operations.
2) The next check is to see if our added instructions are correct for basic operations and flow correctly through the processor.
3) Test 3 will test if our added instructions work when the added instructions are intertwined with the given instructions.

3.1.4 Assembler Directives, PUSH and POP

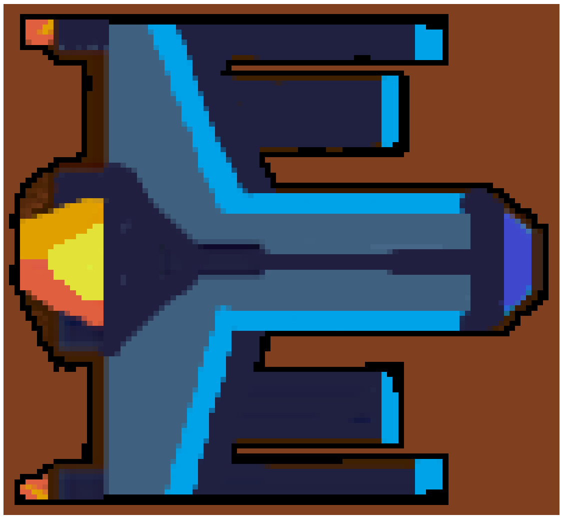
Our assembler directives are not within our ISA, so we will discuss them separately from the other appended instructions.

1. Our tests check if the amount of space allocated on the stack is correct by trying to fill the entire allocated space.
2. We test implementation using function calls and argument registers.

3. Our test will also check combinations of consecutive PUSH and POP instructions and if we are getting desired behavior.
4. We also test interleaved behavior, such as a series of PUSHes followed by some POPs, more PUSHes, and POPs. This is to check if there is any inconsistency in how the stack pointer moves up and down the stack.

3.2 Game Code

The game we coded is a spaceship-meteor dodging game that is displayed on a VGA monitor. Images of the spaceship and meteor are mapped to external memory addresses on the FPGA, and loaded onto the VGA monitor at locations specified in our game code. The spaceship image is player controlled. The player can determine whether the spaceship moves up or down by pressing two memory-mapped keys on the FPGA. If the player wants to move upwards, they press KEY[1], downwards, they press KEY[2].

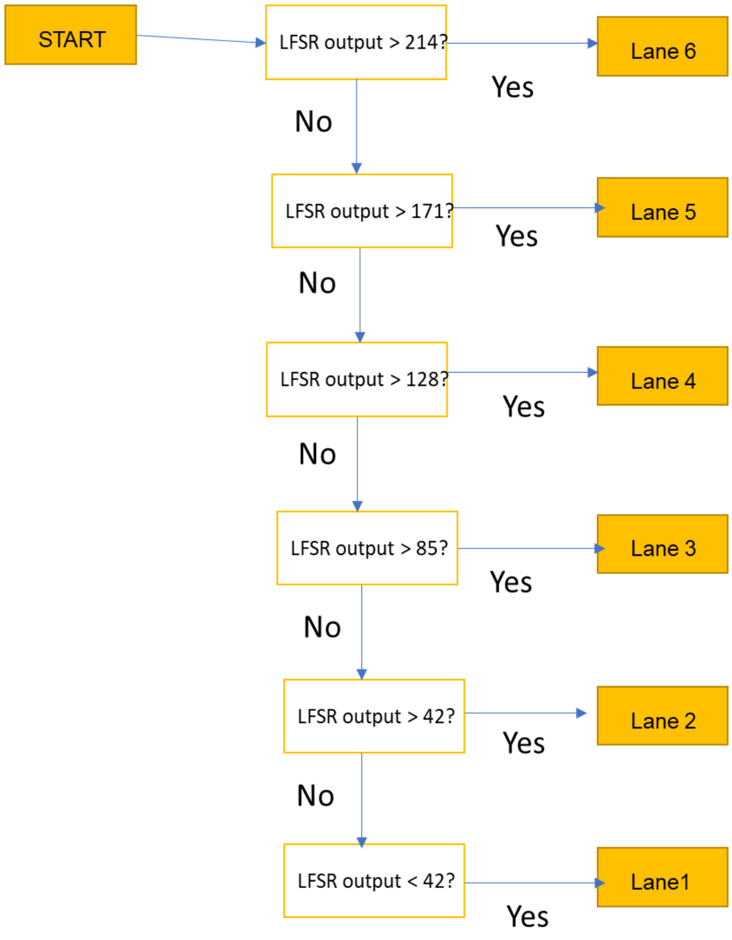


Spaceship -> 100 pixels by 100 pixels

Asteroid -> 60 by 60 pixels

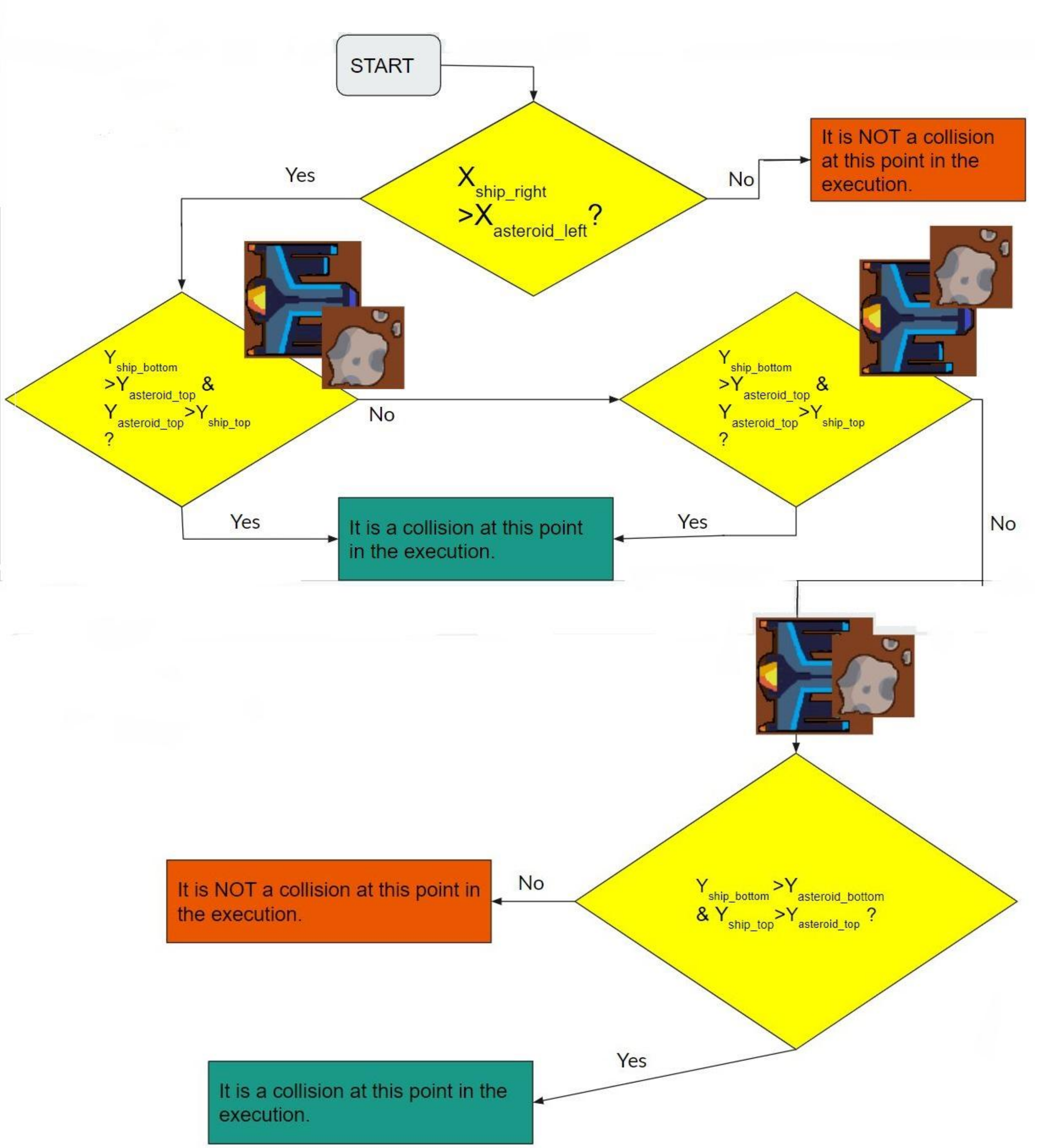
Unlike the spaceship, the meteors are sent across the screen one at a time within 6 possible lanes. The lanes represent different y-coordinate locations on the right side of the VGA monitor. The size of the meteor image determines the lane location. The meteor is of size 60x60, so each lane is designed to hold an image of this size. We decided to use 80 pixels for lane size. This allowed for the meteor to spawn in the middle of a lane by adding 10 pixels to the meteor's starting location without affecting other meteor spawns. The locations of the meteor spawns are seen in our demonstration section.

The meteor spawn lane is determined by an 8-bit LFSR. The generated LFSR output is compared to all lane starting positions. The first lane location of the LFSR is larger than will be the lane the meteor spawns in. The LFSR is a top-level hardware block that’s memory mapped with a seed of 0x88 hex that is incremented by 1 every 65,536 clock cycles to avoid pseudo-randomness. The seed in itself is arbitrary and can be changed to no effect on the rest of the implementation.



The speed of the meteor is coded by reading a memory-mapped switch on the FPGA that enables or disables the branch prediction unit within the processor. When the player enables the switch, the speed of the processor is increased due to increased branch prediction and flow efficiency. In effect, it allows the processor to exploit predetermined branch prediction outcomes derived from the BTB unit and skip a total of 2 cycles (IF, ID) and skip straight to the EX stage of the pipeline relevant to the branch in question. This allows the game code to execute instructions faster, increasing the speed of the spaceship and meteor. The player disables this switch to go back to the inefficient, or slow, mode of the game.

The game ends when a player collides with a meteor. The collision is determined by the x,y outline coordinates of the ship and meteor. If the ship x coordinates are greater than the meteor x coordinates, and if there is any overlap in the two's y coordinates, a collision is detected and the game ends. Every cycle in the RUN stage of the ASM, we first check if there is a collision, before checking for a new key press from the user.


START
Yes
$X_{ship\_right} > X_{asteroid\_left}$?
No
It is NOT a collision at this point in the execution.
$Y_{ship\_bottom} > Y_{asteroid\_top}$ & $Y_{asteroid\_top} > Y_{ship\_top}$?
No
$Y_{ship\_bottom} > Y_{asteroid\_top}$ & $Y_{asteroid\_top} > Y_{ship\_top}$?
Yes
It is a collision at this point in the execution.
Yes
No
It is NOT a collision at this point in the execution.
No
$Y_{ship\_bottom} > Y_{asteroid\_bottom}$ & $Y_{ship\_top} > Y_{asteroid\_top}$?
Yes
It is a collision at this point in the execution.

At the end of the game, string assembler directives are used to display branch prediction statistics within a host PC terminal. These are executed with the help of our MOVC instructions. Our code keeps track of, and displays, the total number of branches taken, the number of branch hits, and the number of branch mispredictions generated.

3.3 Testing of Finished Branch Prediction Game

There are four final parts to test for our project: dynamic branch prediction unit, LFSR, Gameplay, and SPART databus display

Branch Prediction Testing:

1. Testing reads from the Branch Target Buffer (BTB) by preloading many branches into written ASM files and sent to a CPU simulation in Modelsim using $readmemh
2. Testing if a branch's PC is correctly written to a BTB cache when the branch is predicted to be taken.
3. Make sure that the Branch Predictor takes a branch after a miss if the strong bit is high

LFSR Testing:

1. Provide the LFSR with multiple different seeds to see if the output of the LFSR is random
2. Use isolated testing game code to confirm if LFSR is receiving seed from assembly file
3. Check if the LFSR seed is being changed every time the memory-mapped timer overflows

Game Movement Testing:

1. Display the generated images onto the VGA monitor using the game code.
2. Check if the spaceship moves when a player presses the up/down keys.
3. Run the game to see if meteors spawn randomly and move across the screen.
4. Have the meteor and spaceship collide to determine if the game ends correctly.

Spart Databus Display Testing:

1. Check if SPART is correctly sending data over the VGA display with miniproject1 testbench.
2. Test if SPART sends branch prediction statistics to the PC terminal at the end of the game.
3. Check if data is still displayed if the branch predictor is turned on and off throughout the game.

4. Engineering Standards Employed in Design

Our design made use of engineering standards for design compatibility. These were:

- IEEE 1364 –2001 Verilog
- IEEE 1800-2009 systemVerilog
- De-facto UART standard
- De-facto VGA monitor standard

Our design made use of both Verilog and SystemVerilog IEEE standards for designing our processor and branch predictor. Using this standard makes the code universally understandable by anyone in the industry who follows the IEEE engineering standards as described above.

Our team also used a de-facto baud rate of 115200 for a UART two USB cable connection from our FPGA to a host PC. Using a standard baud rate for our UART allows for any device within the computer market/industry to use our UART connection without fail.

Lastly, our team employed the de-facto VGA monitor standard because VGA output connectors are a standard output for all PCs. Also, all DE1-SOC FPGAs have a VGA output connector, so anyone can display our spaceship-meteor game.

5. Potential Societal Impacts of Design

Our game showcases how a CPU can improve efficiency by implementing dynamic branch prediction. Our game code contains image movement subroutines that are connected through branches. When branch prediction is disabled, the speed of the game is slow due to all game branch instructions flowing through the CPU pipeline every time the spaceship or meteor moves or collides with each other. But when branch prediction is activated, the game's speed dramatically increases because the branch predictor determines which branch will be taken without the need to flow through the CPU.

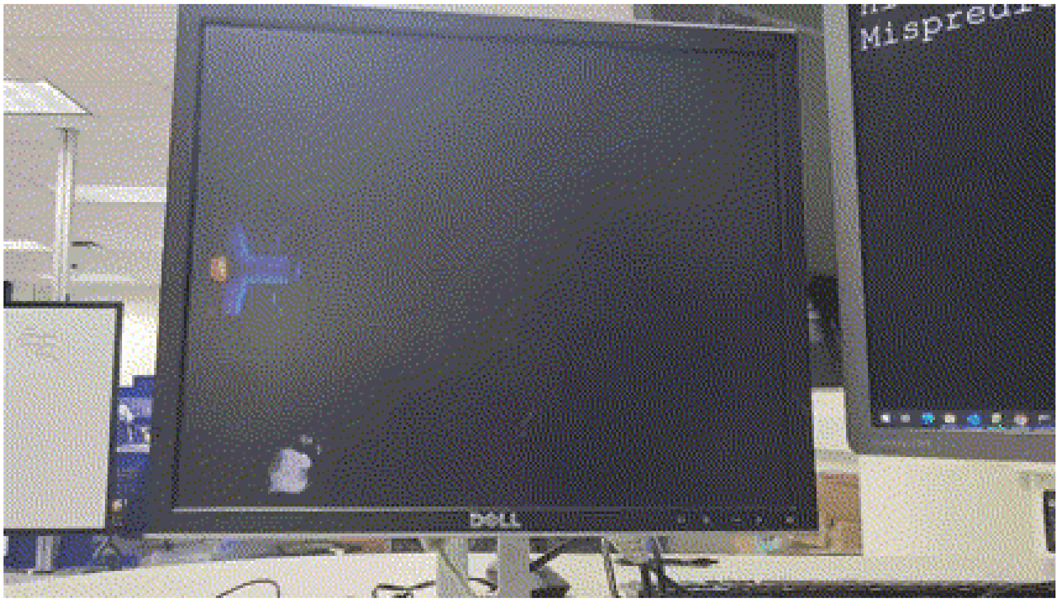
If a branch predictor makes our game notably faster, it would be able to improve any CPU instruction pipeline speed. CPUs would be able to solve complex equations faster in any field of research, increase computer graphics, or allow for more computer applications to run simultaneously.

Our branch prediction design can also have positive environmental impacts because of energy efficiency and less E-waste. A more efficient CPU means it is more energy-efficient. Less energy is consumed in the CPU if branch prediction improves the CPU pipeline flow. And less energy means less energy consumption is required to power the computer the branch predicting CPU is within. Also, E-waste is decreased from the increased efficiency because the CPUs will not have to perform at maximum production to run anymore.

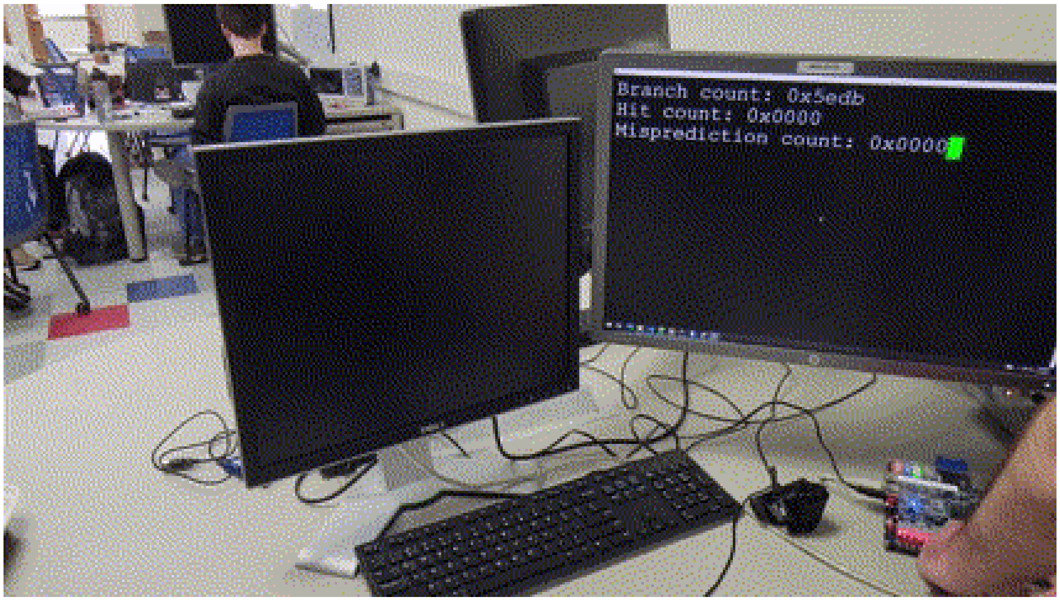
6. Final Application Demonstration

No branch prediction:

When the player has branch prediction disabled, the spaceship and meteor slow down. This is due to no access to and from the branch target buffer.



The statistics are not updated upon collision, so they are not shown.



```
Branch count: 0x5edb
Hit count: 0x0000
Misprediction count: 0x0000
```

Turning on branch predictor:

When the player enables branch prediction, the spaceship and meteor speed up to showcase the increased processor efficiency. To the right, three statistics are displayed. These are branch count, branch hit count, and branch misprediction count. These statistics are updated upon collision.

7. Contributions of Individuals

Names	Tasks
Brody Skaleski	● Appended ISA with immediates ● ISA immediate verification ● Coded Right to left meteor movement ● Implemented meteor spawn randomness ● Merged ship movement and meteor movement into final game code
Josh Bowers	● Coded spaceship movement ● Coded meteor lane spawn points ● Implemented the assembler ● Made push/pop into an assembler directive ● Coded LFSR ● Added STRING assembler macro
Anish Munimadugu	● Implemented Branch Target Buffer within our CPU ● memory mapped all peripherals ● Wrote code to display branch prediction statistics at the end of the game to a terminal using SPART ● implemented branch predication enable/disable
Ayan Deep Hazra	● Implemented MOVC and helped with ISA immediates ● Generated game image hex files and displayed them ● Extended ISA to 17-bits ● PUSH/POP and MOVC verification ● Implemented end condition of the game when a collision occurs ● Reset condition in VGA