1D Project Final Report

Electronic Game Prototype

Team Number: <*3-6*>

Yeh Swee Khim, Lin Cheng, Jia Yunze, Xie Han Keong, Jisa Mariam Zachariah

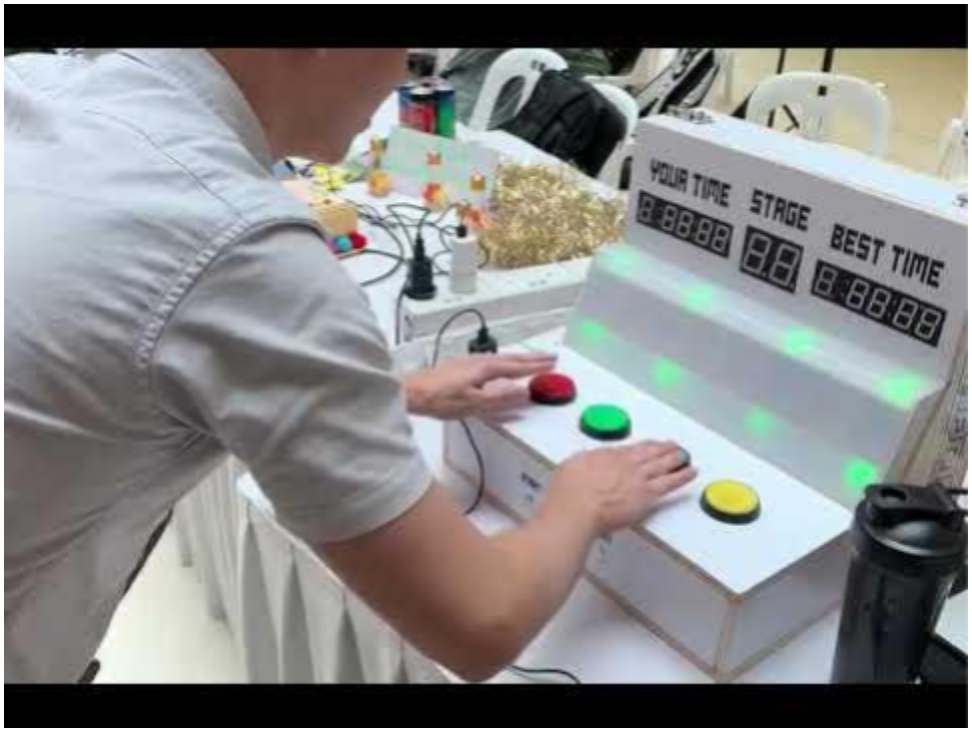
---

Table of Contents

TAP MANIA

What is Tap Mania?

Our game is called *Tap Mania*. It is a skill-based and time-based game which tests the player's reaction time and hand-eye coordination. The aim of our game is to press the same combination of colours as shown on the user interface as quickly as possible. The best time achieved will be displayed next to the timer, so that the next player can see the timing to beat. (not implemented due to technical difficulties)



Watch our Top Player during the Exhibition play Tap Mania here!

Game Design

Tap Mania was designed to maximise repeatability and fun. Our goal was to create a very simple game that is intuitive to play, yet challenging to master, so that it can be engaging and fun for people of all ages. Above all, we hope to create a spirit of competition amongst friends and others so that they will play again to improve their timing and compete with one another.

Minimalistic User Interface

Tap Mania has a minimalistic user interface. There are four colored buttons: Red, Green, Blue, and Yellow. The LED array also has the same four colors. This is intuitive enough for the player to understand how to play the game in an instant.

Figure 1: User interface of *Tap Mania*

The backlight is situated above and below the LED array. It serves a passive role to give the player feedback about the game. When the player passes a stage, the backlight will flash green momentarily to reward the player. When the player presses a wrong button, the backlight will flash red to show that the player has made a mistake.

The displays for ‘YOUR TIME’, ‘STAGE’, and ‘BEST TIME’ are self-explanatory. However, the juxtaposition of ‘YOUR TIME’ against ‘BEST TIME’ is intentional–it presents the player with a challenge to achieve a better time than the best time.

Open Arena Concept

Tap Mania is based on an **open arena** concept. When a player starts playing *Tap Mania*, ‘YOUR TIME’ will run until the player has completed a series of 20 stages. After doing so, if the player’s time is better than the best time, then the player has successfully beaten the previous record. ‘BEST TIME’ will be updated to the time that the player has achieved, setting a new record for other players to beat. This is the idea behind open arena–that all players are invited to battle for the best time.

Spirit of Competition

Tap Mania is designed to be a very competitive game. It encourages players to play again to improve their timing so that they can get the best time or beat their friends. In fact, the more a player practices, the better his timing will be. Combined with the fact that a game should not take more than 2 minutes to complete, *Tap Mania* is designed to be very repeatable as well.

Source of Inspiration

We were inspired by an old-school game called Audition. In Audition, the objective is to press a given sequence of arrow keys in order to make a virtual character perform the best dance moves and earn the highest score. This game was fun largely due to the fact that it is skill-based and time-based, and we wanted to emulate this in order to engage players more, encourage repeatability, and promote a spirit of competition between friends.

Figure 2: User interface of *Audition*

Design Rationale

We felt that this idea was suitable within the scope of this project. This was due to a few reasons. We thought that if we used colours instead of arrows we can simply use an array of LEDs, which we felt was not too difficult to construct. Also, we wrote a program for the game and found that its requirements align with the objectives of the project to utilise more than two ALU functions (ADD, SHL, SHR, CMPEQ, etc.). Moreover, due to the simplicity of the game, we expected that the game would be manageable within the constraints of the project such as budget and time frame (i.e. for development, building, and debugging). Lastly, the difficulty of the game could be scaled in various ways: the delay penalty for pressing the wrong button, the length of the LED sequence that lights up and the variation of the LED sequence can all be increased to make the game more difficult.

User Manual

(1) Press the ‘START’ button to begin playing *Tap Mania*.

(2) When the LEDs are lit up, press the buttons with the corresponding colors in order from left to right.

(3) If the correct button was pressed, the LED will be switched off.

(4) When a stage has been completed, the backlight will flash green.

(5) If the wrong button is pressed, the backlight will flash red and there will be a short delay as a penalty.

(6) As you progress through the game, the number of LEDs lit up will increase from four to six to eight. It will be more difficult to press the correct buttons quickly without pressing the wrong button.

(7) [Not Implemented] When you complete the last stage, ‘YOUR TIME’ will stop running and start blinking. If you achieved a better time than the best time, ‘BEST TIME’ will be updated to your time and blink as well.

(8) At any time during the game, you can press the ‘RESET’ button to reset the game back to an idle state.

Hardware Construction

The construction of our game can be split into three stages: Preliminary Blueprints, Assembly of Main Frame, and Fixing of Electrical Components.

Preliminary Blueprints

We used AutoCAD to visualise the prototype for our game, and devised a plan to buy and cut the wood needed to construct our prototype.

Figure 3: AutoCAD draft of the prototype

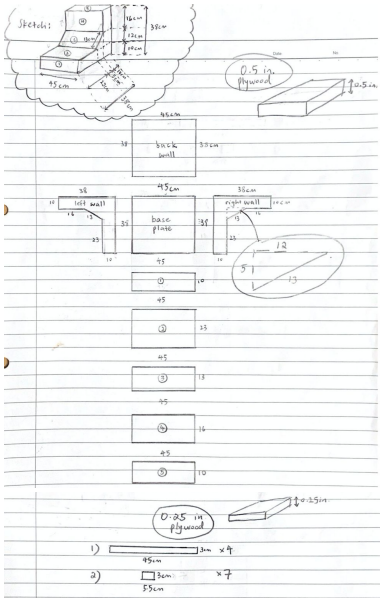


Figure 4: Wood plan

Assembly of Main Frame

With these blueprints, we were able to purchase the exact amount of wood needed for the main frame. We also purchased other components required such as the arcade buttons, LEDs, seven segment displays and the frosted acrylic sheet. With everything, we proceeded to assemble the main frame in SUTD's FabLAB.

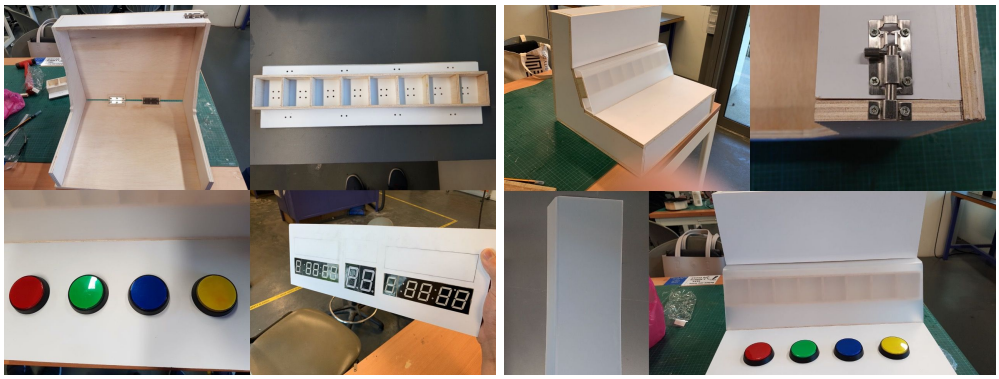


Figure 5: Collage of various snapshots during the construction of the main frame

Fixing of Electrical Components

Once the main frame was assembled, we focused on constructing the LED array. We first attached a 330Ω resistor to each LED's ground. Then, we followed a common anode design and connected all the grounds together with wires. After that, to each LED's anode, we connected a wire which was joined at the other end to a male jumper wire, and connected all these male jumper wires to a 2x20 female pin header serving as an adapter to the MOJO. Lastly, all exposed wires were insulated and reinforced with hot glue. After the LED array was completed, we also connected the backlight and buttons in a similar fashion.

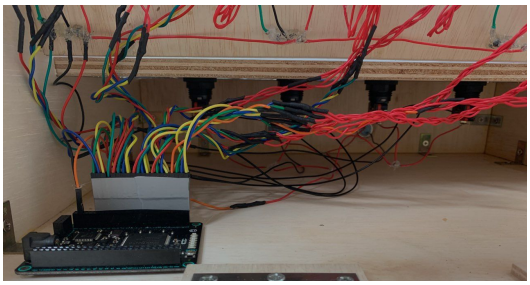


Figure 6: A snapshot of how the LEDs and buttons were connected to the MOJO

About Our Seven-Segment Displays

We faced many issues with our seven-segment displays. This began when we purchased our 1.2-inch and 1.8-inch displays but did not foresee that they would not be able to be powered by MOJO's 3.3V supply. Due to their size, each segment actually consists of two LEDs in series, which entails twice the voltage drop. Since we had seven-segment displays consisting of red LEDs (which require 2V per LED in series) and displays consisting of blue LEDs (which require 3V per LED in series), this meant that we actually needed to supply an external power source of at least 6V.

We tried our best to figure out how to create a circuit which would allow the MOJO's 3.3V to switch on an external 6V battery source on both the high side and the low side. This is to say that there would be a switch allowing current to flow in from the battery to a digit (high-side switching), and also a switch allowing current to flow out from a particular segment to ground (low-side switching). We did this according to how MOJO's IO Shield was configured, and the reason for this is to enable multiplexing of a seven-segment display (common anode), which reduces the number of pins that we would need to use on the MOJO per display.

After carrying out some research, we found the following circuit for high-side switching in the accepted answer to a thread on an electronics forum page (Pefhany, 2014). We tested the circuit on breadboard and found that it worked, and so we decided to replicate it on a prototyping board.

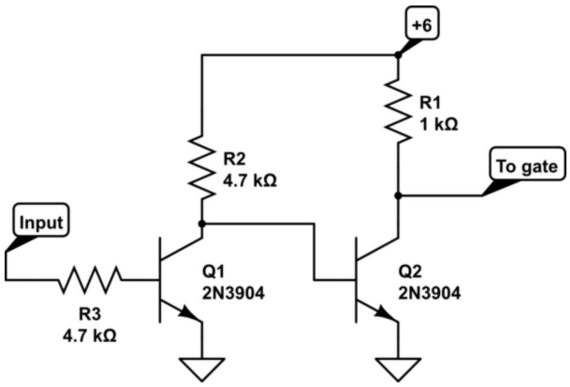


Figure 7: The circuit for high-side switching that we tried to use

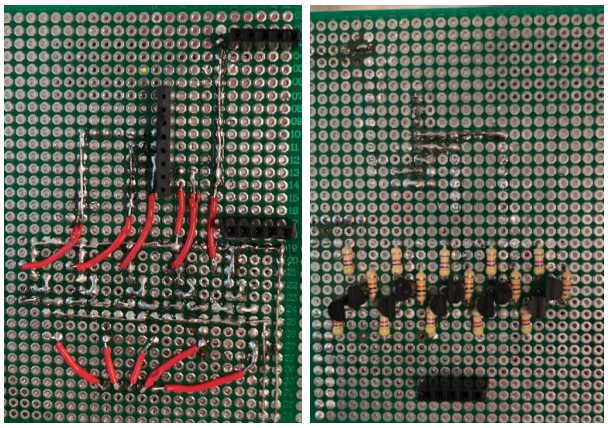


Figure 8: Front (left) and back (right) view of our implementation of the circuit

However, luck was not on our side. Even though the circuit worked as intended, we ran into a stumbling block. We were unable to find a circuit for low-side switching that was compatible with this circuit. We were stuck, running very short on time, and had spent all our resources on this circuit. There was very little hope to continue.

Fortunately, Prof. Natalie was in the Digital Systems Lab when we were about to give up. She stayed behind and gave us a crash course about how transistors worked. Eventually, she showed us how to configure a circuit that is able to support both high-side and low-side switching using half the number of transistors compared to the original circuit we tried to use.

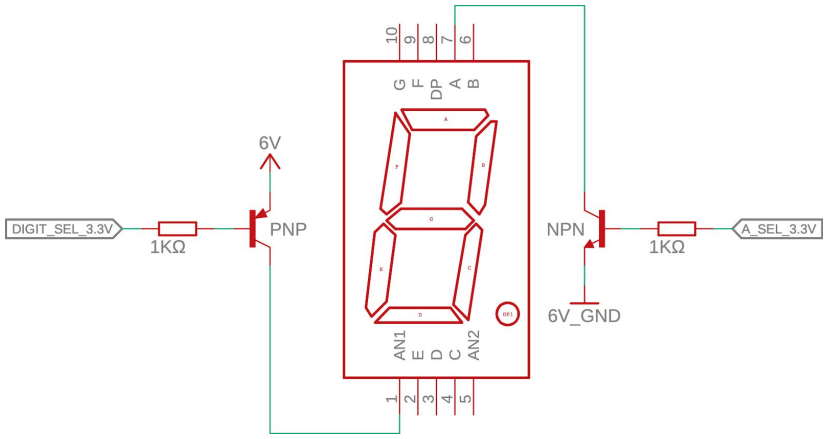


Figure 9: A simplified circuit which supports both high-side and low-side switching

Unfortunately, this knowledge came too late. We had no more resources left, there was no more time to go out to purchase any components, and no time to solder the circuit onto another prototyping board like we did before. However, we did not give up. We resorted to using breadboard and jumper wires to connect the circuit together, since the exhibition was the next day and the circuit only needed to last for that day.

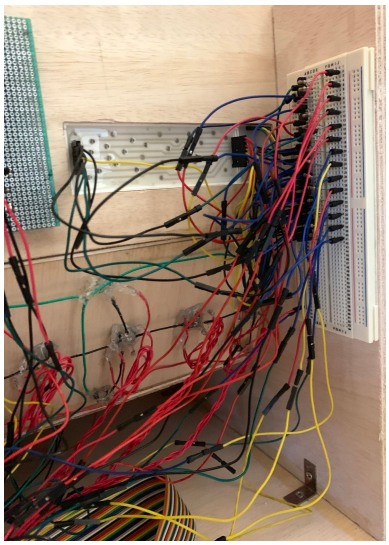


Figure 10: Our implementation of Prof. Natalie's circuit on breadboard

Nonetheless, even though we tried our best to get the seven-segment displays to work, we most probably made a mistake during the wiring or the mapping of pins inside our MOJO custom user constraints file. The Lucid code we wrote for the seven-segment display which was tested previously and worked on MOJO's IO shield simply did not work on our seven-segment displays. We had done everything we could, but there was absolutely no more time left to debug the issues with our seven-segment displays. At the end, as disappointing as it seemed, we had no other choice but to leave the displays unpowered, serving only as an aesthetic finish to the user interface.

Exhibition Summary

During the exhibition, we decided not to let the problem with seven-segment displays deter *Tap Mania* from being as competitive as we envisioned it to be. We used an online stopwatch as a timer for the game and designed an excel sheet to record the best times of people who came by our booth and played our game. As the day went by and more people came to know about our game, we found that there was a tendency for players to come back again to beat their previous time, especially if a friend had achieved a better time than them. There were even some players who were so competitive that they played non-stop just to perfect their skill at the game and achieve the very best time. The top player achieved a timing of merely 36 seconds! At the end, not only were we amazed at his skill, we were also very happy to see that *Tap Mania* was successful in being a fun, competitive, and repeatable game.

Figure 11: Jisa explaining the rules of *Tap Mania* to a player

Figure 12: Spreadsheet of all players who recorded their time

Budget

Components	Cost / S$
Wood	67
Acrylic	5
Seven-Segment Displays	35
Buttons	55
LEDs	9
Miscellaneous (Electrical)	53
Stickers	6
Total:	230

Figure 13: Table of components purchased and total cost incurred

References

Pefhany, S. (2014). Stack Exchange - Electrical Engineering. Retrieved from https://electronics.stackexchange.com/questions/99488/convert-microcontroller-3-3v-output-to-6v/99493#99493

Agus, N. (2019). Beta Datapath. In 50.002 Computational Structures. Retrieved from https://github.com/natalieagus/50.002-Summary-Notes/blob/master/12b_BetaDataPath.pdf

Appendix

𝒴 Architecture

To implement our game, we designed a von Neumann computer based on the Beta architecture we learnt in the 50.002 Computation Structures module (Agus, 2019). We decided to name it 𝒴 (because G for Game). The main differences are that data memory is not writable, PC is incremented by 1 only, and we introduced an I/O unit which will be connected to the buttons, LED array, backlight, and seven-segment displays. The I/O unit has its own set of registers that can be read and written using GET and SET instructions.

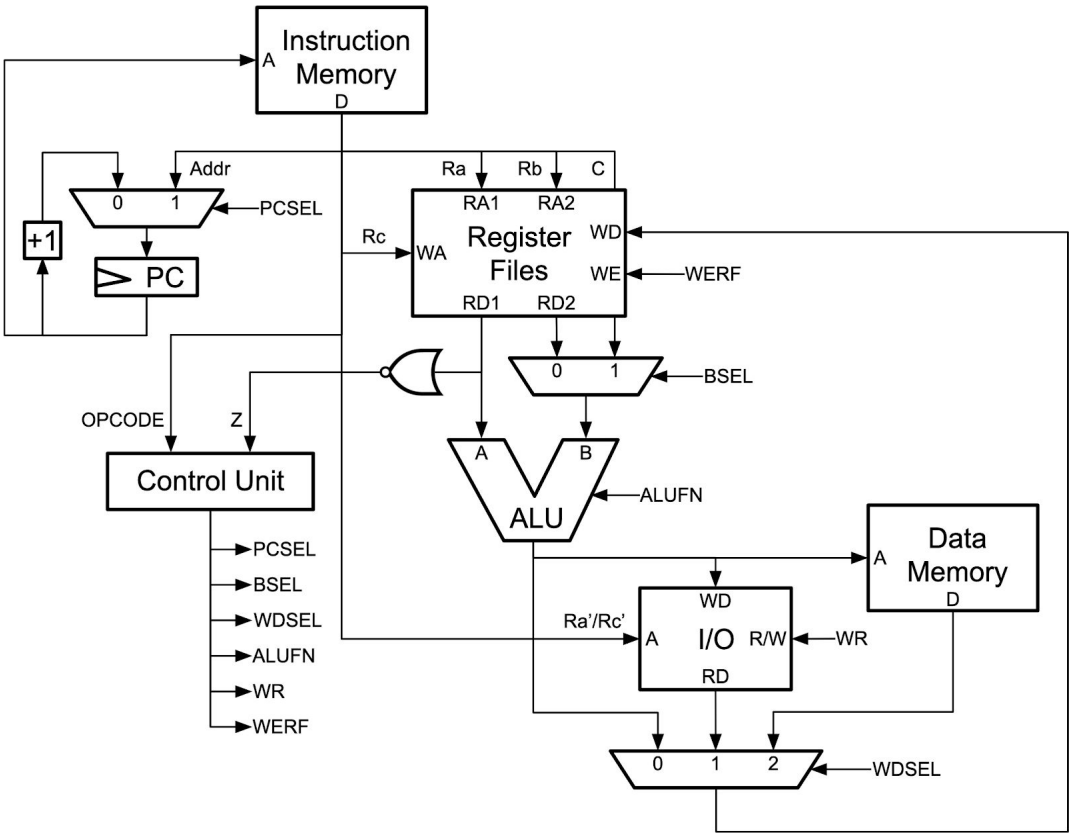


Figure 14: The 𝒴 architecture

I/O Files

The addresses used for the I/O registers are shown below.

R0	R1	R2	R3	R4	R5	R6	R7
`button`	`led1`		`led2`		`timer/timer_state`		`stage`
R8	R9	R10	R11	R12	R13	R14	R15
	`best_time`		`delay/delay_state`				

Figure 15: 𝒴 I/O registers assignments

Register Files

Since the I/O unit already contains registers, only 8 registers are needed for 𝑌. This means that 3 bits are needed for addressing. The first 3 registers (R0, R1, R2) are unassigned, and the last register is assigned permanently to 0. The specific roles of `input`, `next`, `offset`, and `ans` are discussed in Tap Mania Program.

R0	R1	R2	R3	R4	R5	R6	R7
			`input`	`next`	`offset`	`ans`	`0`

Figure 16: 𝑌 register file assignments

ALU Functions

𝑌's ALU supports the following functions as shown below according to their ALUFN codes.

↓ 5:3 2:0 →	000	001	010	011	100	101	110	111
000	ADD	SUB						
001								
010							XOR	
011	AND		ASEL				OR	
100	SHL	SHR		SRA				
101								
110				CMPEQ		CMPLT	CMPLE	
111								

Figure 17: 𝑌 ALU functions

Instruction Set

𝑌 supports the following operations as shown below according to their opcodes. Unassigned opcodes default to NOP (No Operation). They will be further discussed in the following pages.

↓ 5:3 2:0 →	000	001	010	011	100	101	110	111
000	ADD	SUB			CMPEQ	CMPLT	CMPLE	
001					XOR		AND	
010	ASEL		OR		SHL	SHR	SRA	
011								
100	ADDC	SUBC			CMPEQC	CMPLTC	CMPLEC	
101					XORC		ANDC	
110	ASEL		ORC		SHLC	SHRC	SRAC	
111	LD	GET	SET			BEQ	BNE	

Figure 18: 𝑌 instruction set

OP Instruction Class

The OP (Operate) instruction class is responsible for the set of functions associated with the ALU, which includes functions from adder, boolean, shifter, and compare units. In an OP instruction, three register addresses must be provided: Ra and Rb–whose values will be the operands, and Rc–where the result will be stored in. The format and register language for the OP class is shown below.

0xxxxx	Rc[3]	Ra[3]	Rb[3]	*unused*[17]

```
OP(Ra,Rb,Rc): Reg[Rc] ← Reg[Ra] op Reg[Rb]
```

The datapath for the OP class is as follows:

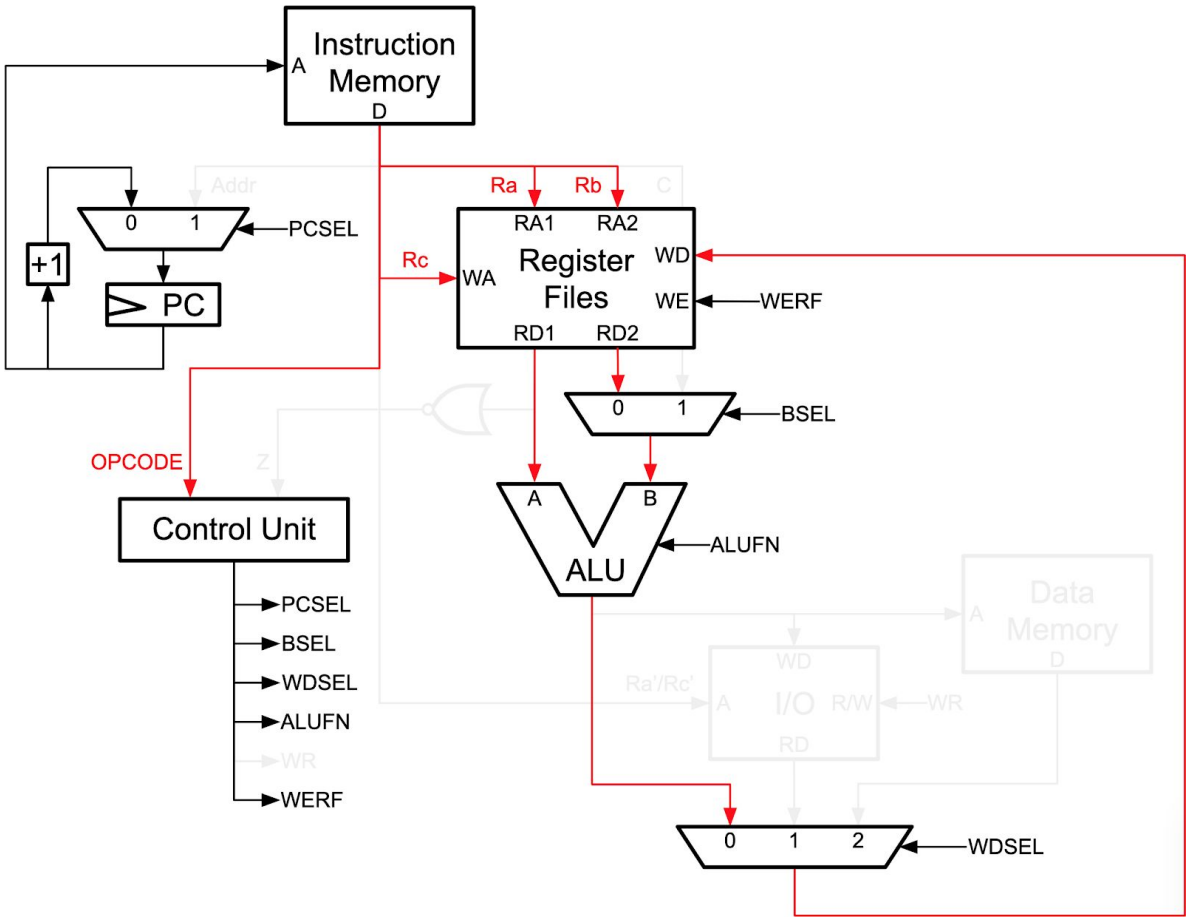


Figure 19: OP instruction class datapath

OPC Instruction Class

The OPC (Constant) instruction class is an alternative set of instructions to utilise the ALU functions. In an OPC instruction, two register addresses must be provided: Ra–whose value will be the first operand, and Rc–where the result will be stored in. The other operand is a signed constant C which can be up to 16 bits long. The instruction format and register language for the OPC class is shown below.

1xxxxx	Rc[3]	Ra[3]	xxxx	C[16]

```
OPC(Ra,C,Rc): Reg[Rc] ← Reg[Ra] op SEXT(C)
```

The datapath for the OPC class is as follows:

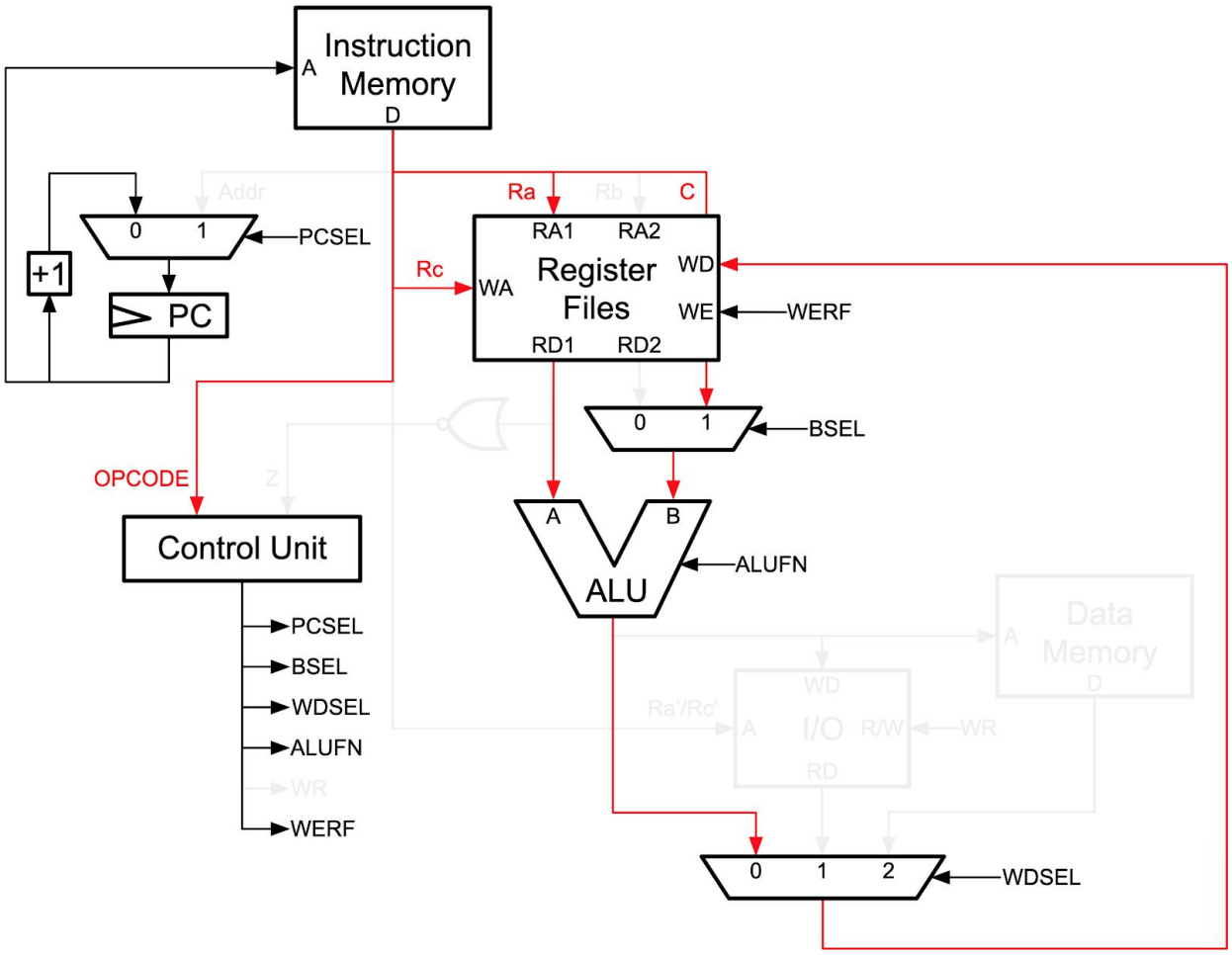


Figure 20: OPC instruction class datapath

LD Instruction

The LD (Load) instruction will load a value from memory into a register. Since there is no need to write to memory for our game, we have decided to make the data memory read-only in the $\mathcal{Y}$. Thus, there is no need for the corresponding ST (Store) operation. In a LD instruction, two register addresses must be provided: Rc–where the retrieved value will be stored, and Ra–whose value will be added to another provided signed constant C to calculate which address to lookup in memory. The instruction format and register language for LD is shown below.

111000	Rc[3]	Ra[3]	xxxxxxxxxxxx	C[8]

`LD(Ra,C,Rc): Reg[Rc] ← Mem[Reg[Ra]+SEXT(C)]`

The datapath for the LD instruction is as follows:

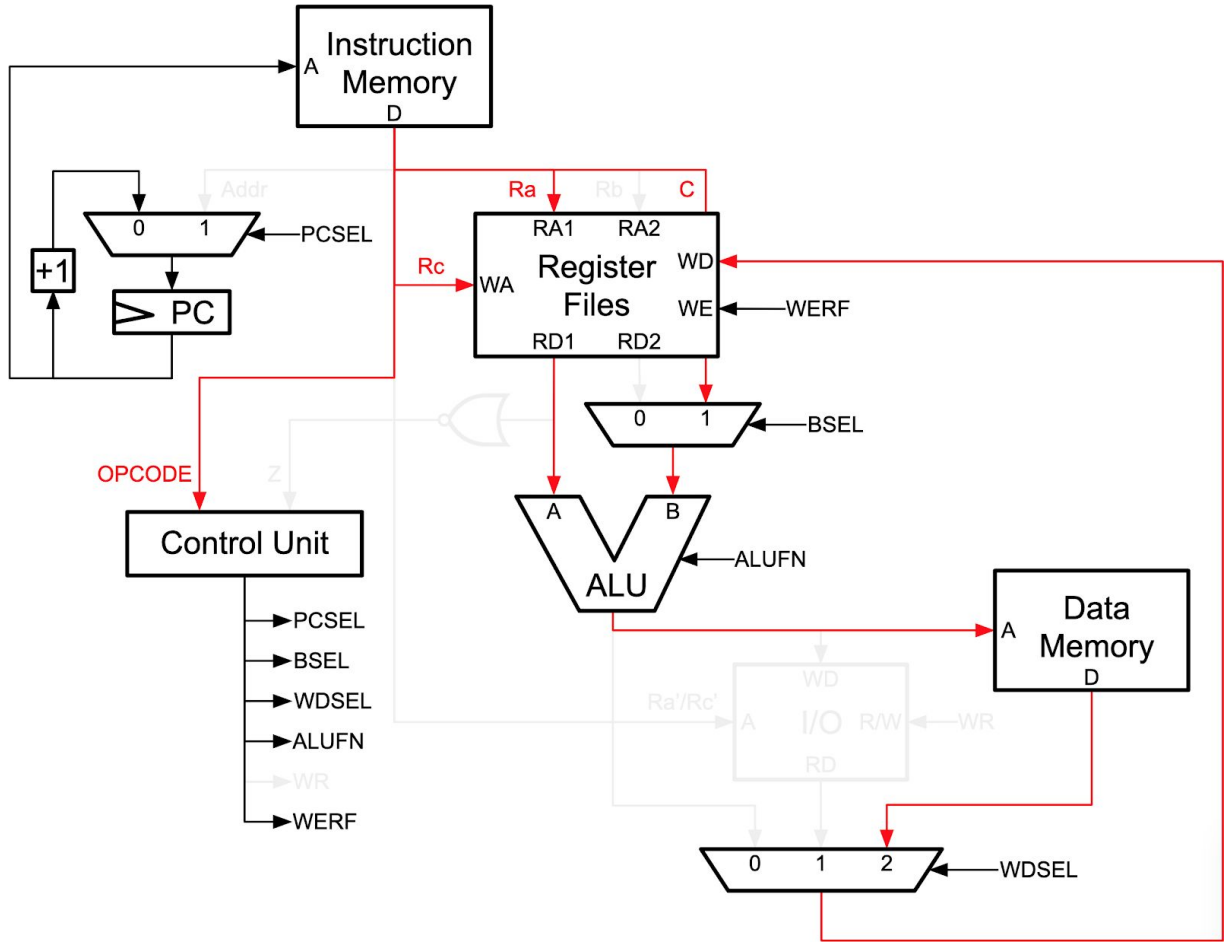


Figure 21: LD instruction datapath

BEQ/BNE Instruction

The BEQ/BNE (Branch if Equal/Branch if Not Equal) instructions serve as branching points in a program. It allows the program to jump to another instruction in the instruction memory if a value in a certain register is 1 or 0. If not, the next instruction at PC+1 will be executed as per usual. In a BEQ/BNE instruction, only one register address needs to be provided: Ra–whose value will be checked for branching. Another 8-bit value should also be provided: Addr (address)–where the program will branch to. The instruction format and register language for BEQ/BNE is shown below.

1111xx	xxx	Ra[3]	Addr[8]	*unused*[12]

```
BEQ(Ra, Addr):     If (Reg[Ra]==1) PC ← Addr

BNE(Ra, Addr):     If (Reg[Ra]==0) PC ← Addr
```

The datapath for the BEQ/BNE instruction is as follows:

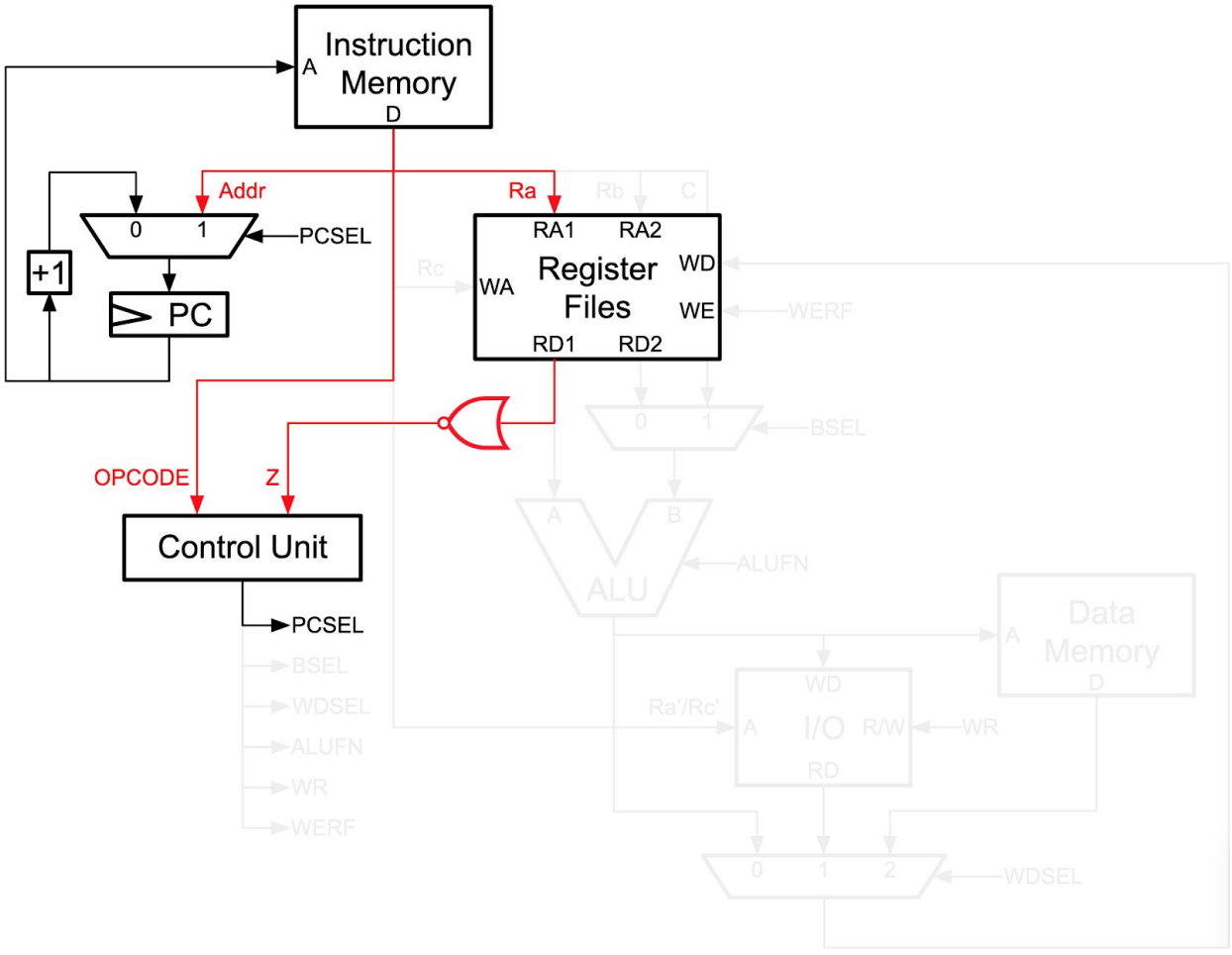


Figure 22: BEQ/BNE instruction datapath

GET Instruction

The GET instruction loads a value from the I/O registers into the register files. This enables subsequent ALU operations to be conducted on these values. In a GET instruction, two register addresses must be provided: Rc–where the retrieved value will be stored, and Ra'–the I/O register whose value will be retrieved. The instruction format and register language for GET is shown below.

111001	Rc[3]	xxx	Ra'[4]	*unused*[16]

```
GET(Ra',Rc): Reg[Rc] ← IO[Ra']
```

The datapath for the GET instruction is as follows:

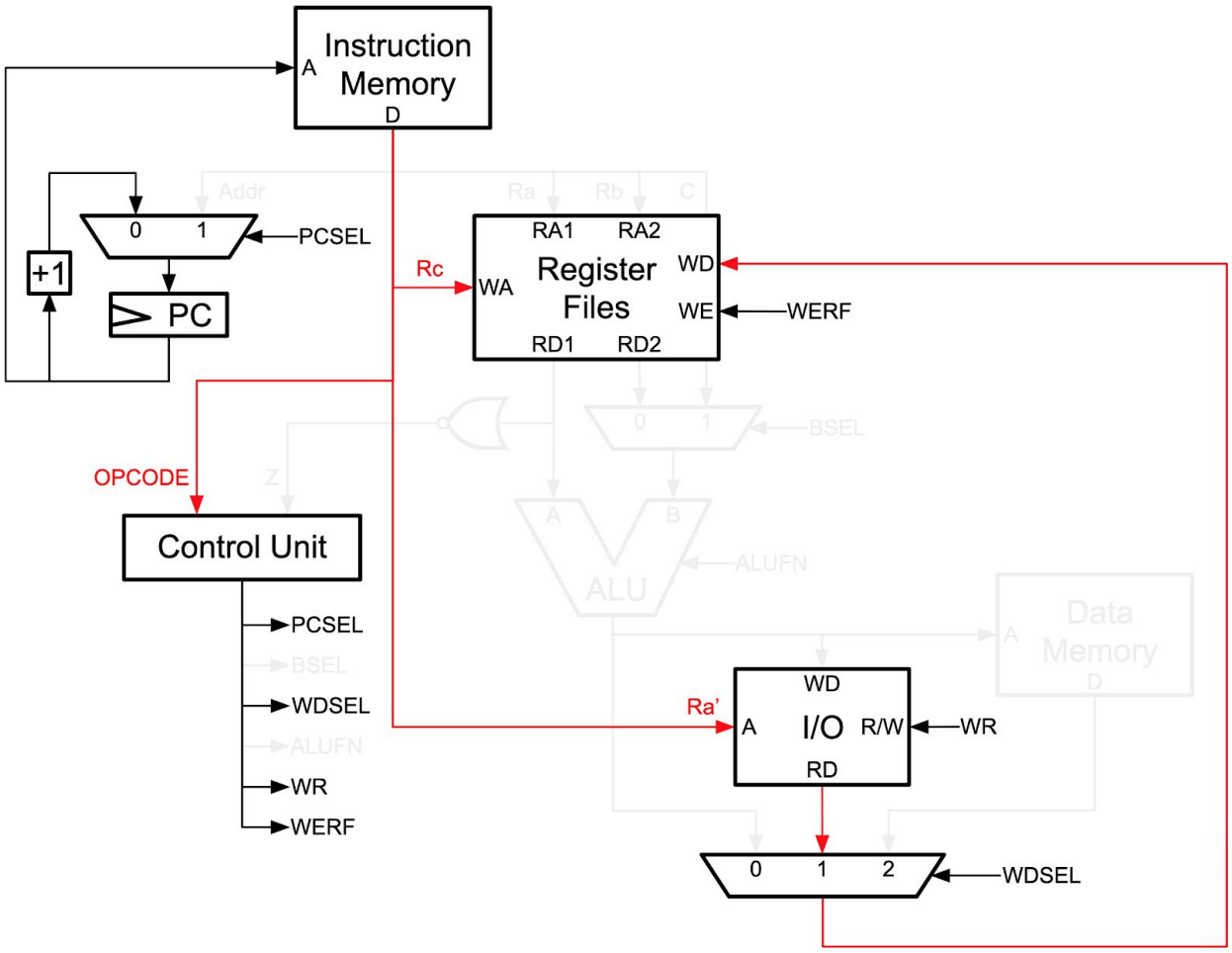


Figure 23: GET instruction datapath

SET Instruction

The SET instruction loads a value from the register files into the I/O registers. This allows the outputs (i.e. 7-segment displays, LED array) to be updated to a configuration. In a SET instruction, two register addresses must be provided: Rc'–the I/O register where the value will be stored, and Ra–whose value will be added to another signed constant C to calculate the value to store. The instruction format and register language for SET is shown below.

111010	xxx	Ra[3]	Rc'[4]	C[16]

```
SET(Ra,C,Rc'): IO[Rc'] ← Reg[Ra]+SEXT(C)
```

The datapath for the SET instruction is as follows:

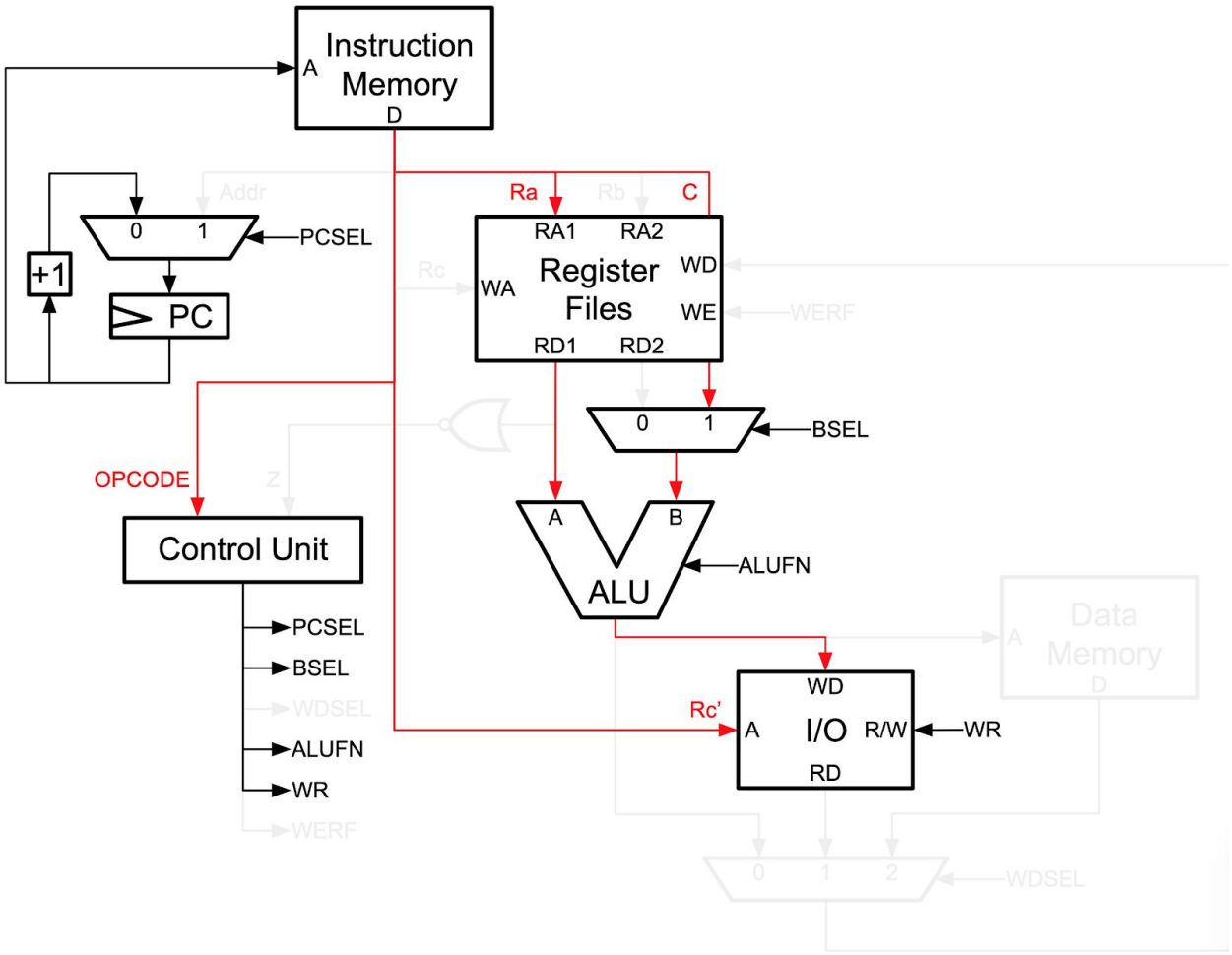


Figure 24: SET instruction datapath

Control Logic

The following is the truth table for the control unit showing the signals used for the different instructions.

Signal	OP	OPC	LD	BEQ	BNE	GET	SET
PCSEL	0	0	0	Z ? 1 : 0	Z ? 0 : 1	0	0
BSEL	0	1	1	–	–	–	1
WDSEL	0	0	2	–	–	1	–
ALUFN	F(OP)	F(OPC)	“+”	–	–	–	“+”
WR	0	0	0	0	0	0	1
WERF	1	1	1	0	0	1	0

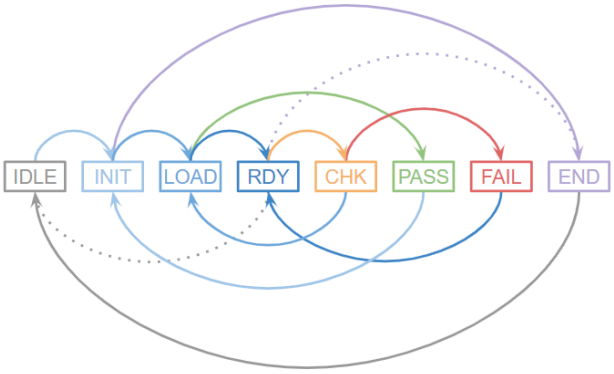
Figure 25: $\mathcal{Y}$ control unit truth table

Tap Mania Program

To implement *Tap Mania*, we wrote a program that runs on *Y*. The program is based on an algorithm that we designed to load a sequence of expected inputs into the LED array, check for the user's input, and update the LED array and backlight accordingly depending on whether the player's input was correct or not. In our game, we used one-hot encoding to represent four colours of LEDs. Red is 1000, Green is 0100, Blue is 0010, and Yellow is 0001. The following will explain more about the program.

State Transition Diagram

The program that we wrote can be generally classified into eight main internal states. The following diagram and table illustrates the sequence of events that will happen throughout the course of the game. The next few sections will attempt to illustrate exactly what happens in each state.

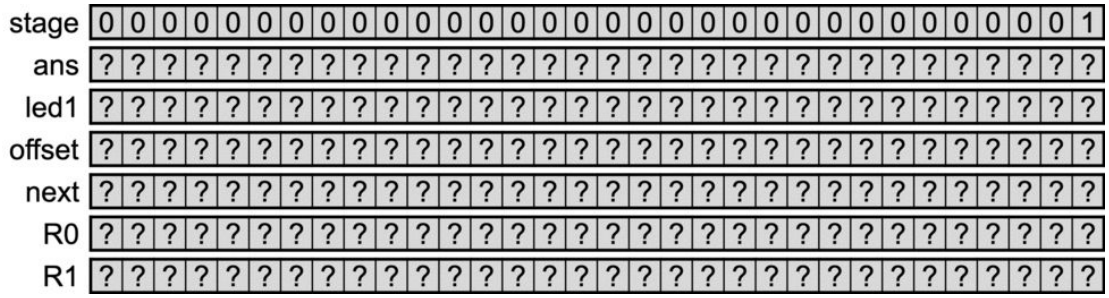


STATE	NEXT STATE	CONDITION
IDLE	INIT	Start button pressed
INIT	LOAD	Next stage is initialised
INIT	END	No more stages left
LOAD	RDY	Next color is loaded
LOAD	PASS	No more colors left
RDY	CHK	Color was pressed
RDY	IDLE	Reset button pressed
RDY	END	Timeout reached
CHK	LOAD	Color pressed was correct
CHK	FAIL	Color pressed was incorrect
PASS	INIT	Backlight flashed green
FAIL	RDY	Backlight flashed red
END	IDLE	Player time displayed

Figure 26: *Tap Mania* state transition diagram

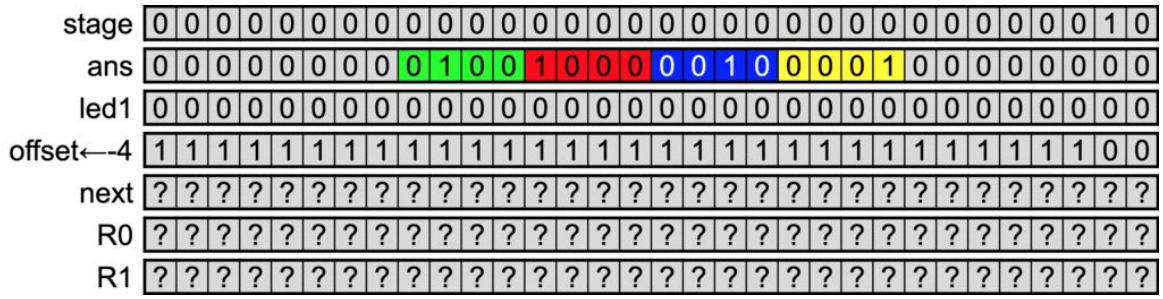
INIT State

Suppose the player has just completed stage 1. The game will be in INIT state. In INIT state, `led1` (the LED array) is reset to 0 and `stage` is incremented. In stage 2, the sequence is XXGRBYXX. Its one-hot encoded bitstring is loaded from ROM into `ans`, and `offset` is initialised to -4. State goes to LOAD.

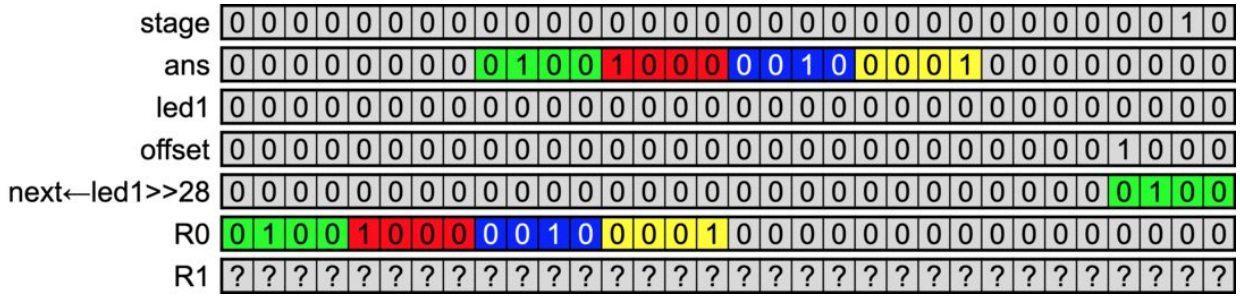


LOAD State

In LOAD state, the idea is to find the next expected input. If the next input in the sequence is blank, we carry on finding until we find an expected input that is nonzero. First, `offset` is incremented by 4. Then, `ans` is shifted left by `offset` and stored in `R0`. Following that, `R0` is shifted right by 28 bits and stored in `next`. If next is 0 (as we see here), the process is repeated. Once again, `offset` is incremented by 4, `ans` is shifted left by `offset` and stored in `R0`, and `R0` is shifted right by 28 bits and stored in `next`. `next` is still 0, so this whole process is repeated once more, and we eventually see that the next `next` is G.

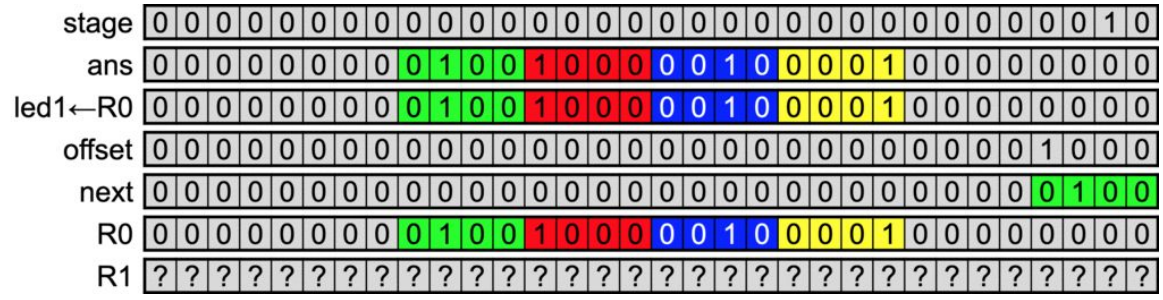


Once we have found a nonzero `next`, we shift `R0` back by `offset`. `R0` now contains the rest of the incomplete sequence, so we set `led1` to the contents of `R0`. State now goes to RDY.



RDY State

In RDY state, the game is suspended in a loop waiting for the player's input. When the player presses a button (suppose that the player pressed the G button), its one-hot encoded bitstring gets stored in `R0`. State goes to CHK.

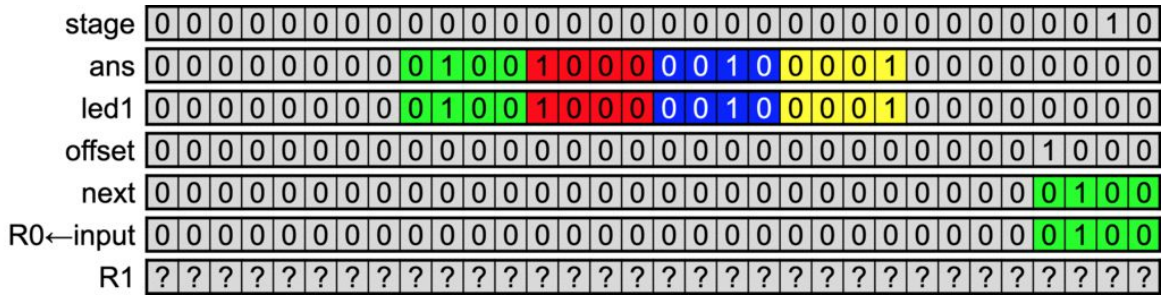


CHK State

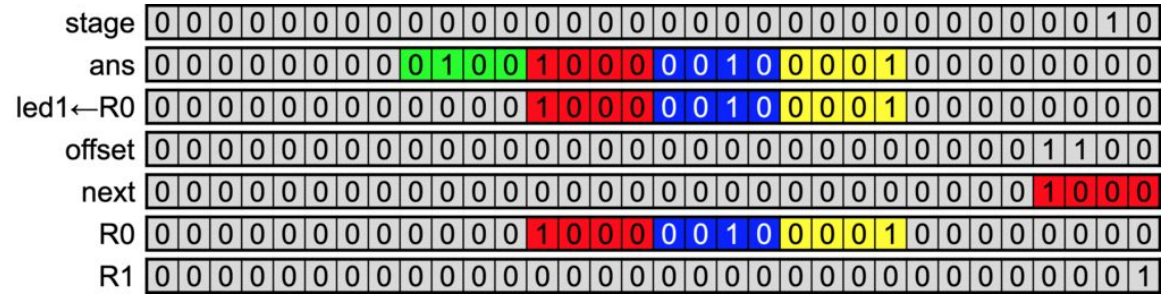
In CHK state, the program checks if the value stored in `next`, which contains the next expected input, is equal to the player's input stored in `R0`, storing the truth value in `R1`.

CHK State: LOAD Case

If they are equal, the player's response is correct, and state goes back to LOAD: `offset` is incremented by 4, `ans` is shifted left by `offset` and stored in `R0`, and `R0` is shifted right by 28 bits and stored in `next`. The next `next` is R, which is nonzero, so `R0` is shifted back right by `offset`, then `led1` is set to the resulting value in `R0`, and state goes back to RDY.

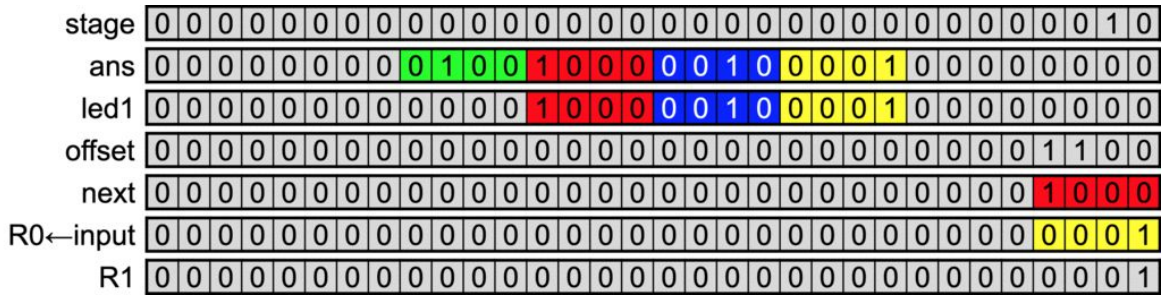


Then, in RDY state, the game is suspended in a loop again waiting for the player's input. Now suppose the player pressed the Y button. The input gets stored in `R0` and state is switched once more to CHK.

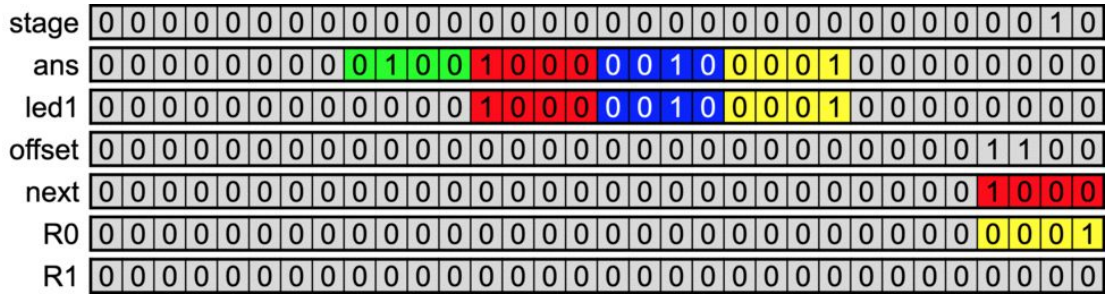


CHK State: FAIL Case

In CHK state, the program checks once again if the input stored in `R0` is equal to `next`. This time, unlike before, they are not equal. State goes to FAIL instead, where the backlight will flash red and the program will be delayed for a short time. After that, state goes back to RDY.

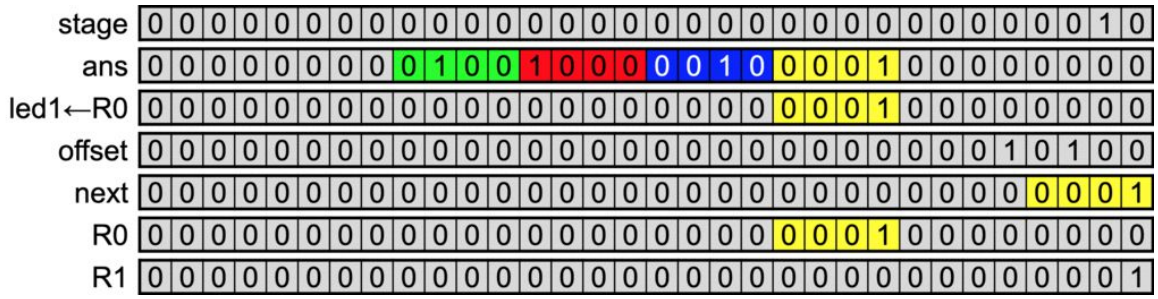


Suppose after this point that the player will press the correct sequence of buttons. Here, the program will continue to switch between the states in the same fashion described above until we reach the last button in the sequence (shown in 2x speed).



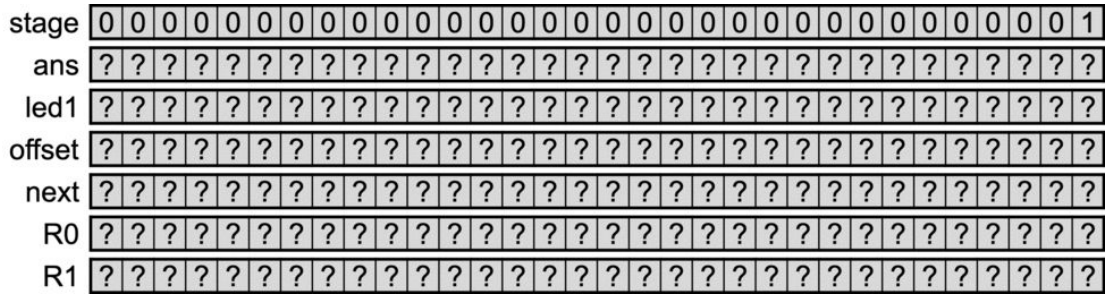
PASS State

In LOAD state, the program actually checks if `offset` has reached 32 or if `ans` shifted left by `offset` becomes zero. These cases are when there are no further expected inputs. State goes to PASS, which sets the backlight to green for a short while. After that, state goes back to INIT, restarting the whole cycle. In this example, we see the latter case occurring.



Overview

To wrap up this example, all of the above gifs have been combined and sped up to 4x speed to better show the changes happening in the registers across time.



Instructions

The instructions for our program for *Tap Mania* are as follows.

Addr.	State	Assembly Language	Description
0	RST	SET(R7, 0, timer_state)	Set timer to reset mode
1		SET(R7, 0, led1)	Set led array to off
2		SET(R7, 0, led2)	Set backlight to off
3	IDLE	GET(button, R0)	Get user input and store in R0
4		BEQ(R0, IDLE)	If no input, go back to IDLE
5		SHRC(R0, 5, R0)	Shift right to get start button bit
6		BEQ(R0, IDLE)	If start bit is 0 (not pressed), go back to IDLE
7		SET(R7, 1, stage)	Otherwise, set stage to 1
8		SET(R7, 1, timer_state)	Set timer to counting mode (go to INIT)
9	INIT	SET(R7, 0, led1)	Set led array to off
10		GET(stage, R0)	Get stage and store in R0
11		LD(R7, MaxStage, R1)	Load MaxStage into R1
12		CMPLT(R1, R0, R2)	Check if stage exceeded MaxStage
13		BNE(R2, END)	If true, go to END
14		LD(R0, StageOffset, R6)	Otherwise load next stage sequence into ans
15		ADDC(R0, 1, R0)	Increment temporary stage
16		SET(R0, 0, stage)	And set stage to that value
17		ADDC(R7, -4, R5)	Initialise offset to -4
18	LOAD	ADDC(R5, 4, R5)	Increment offset by 4
19		CMPLTC(R5, 32, R0)	Check if offset is less than 32
20		BEQ(R0, PASS)	If false (offset is 32), go to PASS
21		SHL(R6, R5, R0)	Shift ans left by offset and store in R0
22		BEQ(R0, PASS)	If this value is zero, go to PASS
23		SHRC(R0, 28, R4)	Shift this value right by 28 and store in next
24		BEQ(R4, LOAD)	If next is zero, go back to LOAD
25		SHR(R0, R5, R0)	Otherwise, shift value back right by offset
26		SET(R0, 0, led1)	Set led array to this value
27		LD(R7, MaxTime, R1)	Load MaxTime into R1 (go to RDY)
28	RDY	GET(timer, R0)	Get value of timer and store in R0
29		CMPLT(R0, R1, R2)	Check if the value is less than MaxTime
30		BEQ(R2, END)	If false then MaxTime exceeded. Go to END

31		GET(button, R3)	Get user input and store in R0
32		BEQ(R3, RDY)	If no input, go back to RDY
33		CMPLTC(R3, 16, R2)	Check if reset button or start button bit is 1
34		BNE(R2, CHK)	If false, go to CHK
35		ANDC(R3, 16, R0)	Otherwise, mask input to get reset bit
36		BNE(R0, RST)	If reset bit is 1 (reset pressed), go to RST
37		BEQ(R7, RDY)	Otherwise (start pressed), ignore. Go to RDY
38	CHK	AND(R3, R4, R2)	Check if user input and next are the same
39		BNE(R2, LOAD)	If same, go to LOAD (otherwise, go to FAIL)
40	FAIL	SET(R7, 1, delay_state)	Set delay to counting mode
41		SET(R7, 2, led2)	Set backlight to red
42		LD(R7, FailDelay, R1)	Load FailDelay into R1 (go to FAIL2)
43	FAIL2	GET(delay, R0)	Get value of delay and store in R0
44		CMPLT(R0, R1, R2)	Check if the value is less than FailDelay
45		BNE(R2, FAIL2)	If true then FailDelay not up. Go to FAIL2
46		SET(R7, 0, delay_state)	Otherwise, set delay to reset mode
47		SET(R7, 0, led2)	Set backlight to off
48		LD(R7, MaxTime, R1)	Load MaxTime into R1
49		BEQ(R7, RDY)	Go to RDY
50	PASS	SET(R7, 1, delay_state)	Set delay to counting mode
51		SET(R7, 0, led1)	Set led array to off
52		SET(R7, 1, led2)	Set backlight to green
53		LD(R7, PassDelay, R1)	Load PassDelay into R1 (go to PASS2)
54	PASS2	GET(delay, R0)	Get value of delay and store in R0
55		CMPLT(R0, R1, R2)	Check if the value is less than PassDelay
56		BNE(R2, PASS2)	If true then PassDelay not up. Go to PASS2
57		SET(R7, 0, delay_state)	Otherwise, set delay to reset mode
58		SET(R7, 0, led2)	Set backlight to off
59		BEQ(R7, INIT)	Go to INIT
60	END	SET(R7, 2, timer_state)	Set timer to pause mode
61		GET(timer, R0)	Get value of timer and store in R0
62		GET(best_time, R1)	Get value of best_time and store in R1
63		CMPLT(R0, R1, R2)	Check if timer value is less than best_time
64		BEQ(R2, HOLD)	If not true, then go to HOLD

65		SET(R0, 0, best_time)	Set best_time to timer value (Go to HOLD)
66	HOLD	SET(R7, 1, delay_state)	Set delay to counting mode
67		LD(R7, HoldDelay, R1)	Load HoldDelay into R1 (go to HOLD2)
68	HOLD2	GET(delay, R0)	Get value of delay and store in R0
69		CMPLT(R0, R1, R2)	Check if the value is less than HoldDelay
70		BNE(R2, HOLD2)	If true then HoldDelay not up. Go to HOLD2
71		SET(R7, 0, delay_state)	Otherwise, set delay to reset mode
72		BEQ(R7, RST)	Go to RST

GitHub Link

The GitHub link to our entire project is at the following address.

https://github.com/han-keong/tapmania/tree/master/1d