RISC-V RV32I Processor Coursework

Personal Statement of Contributions

Jacob Alexandrou

Overview

Sign Extension Unit

Link to module

I made the sign extension unit for lab 4 and its structure did not need to be changed for the implementation of the full single cycle CPU. It is quite a simple module; only having to select bits from the input instruction and concatenate. This is done differently for each instruction type:

Instruction Formats	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Register/register	funct7							rs2					rs1					funct3			rd					opcode						
Immediate	imm[11:0]												rs1					funct3			rd					opcode						
Upper Immediate	imm[31:12]																				rd					opcode						
Store	imm[11:5]							rs2					rs1					funct3			imm[4:0]					opcode						
Branch	[12]	imm[10:5]						rs2					rs1					funct3			imm[4:1]				[11]	opcode						
Jump	[20]	imm[10:1]										[11]	imm[19:12]								rd					opcode						

In the module, the type of instruction is determined by the ImmSrc control signal. I chose to set the value of ImmSrc for each instruction type like so:

ImmSrc	Instruction type
000	Immediate

ImmSrc	Instruction type
001	Store
010	Branch
011	Jump
100	Upper Immediate

and then created an enum for this in the code:

```
typedef enum bit[2:0]   {Imm, Store, Branch, Jump, UppImm}   Instr_type;
```

Then I simply used a case statement and defined the correct bit selection for each instruction type.

Instruction Memory

Relevant commits:

- Added InstrMem

The instruction memory is a ROM with a 32-bit input and output. It takes the program counter as input and outputs the corresponding instruction.

There are a few design decisions to note here:

Firstly, the address used for the instruction in the ROM is the program counter shifted right by 2 bits:

```
    instr = rom_array[{2'b0, PC[31:2]}];
```

This is to account for the byte offset and the fact that the PC increments by 4.

Secondly, instruction hex code was provided to us as 8-bit(byte) data. Although it may not reflect how a real RISC-V processor functions; for simplicity, and due to the fact that there is no need to select individual bytes of the instructions in this coursework, I decided to store instructions in the ROM as 32-bit words.

To help with this I modified the `format_hex.sh` shell script provided by changing the line:

```
od -v -An -t x1 "$1.bin" | tr -s '\n' | awk '{$1=$1};1' > "$1.hex"
```

to :

```
od -v -An -t x4 "$1.bin" | awk '{$1=$1};1' > "$1.hex"
```

This meant the script would output 32-bit big-endian data instead of 8-bit little-endian data so I could load the output hex files straight into the instruction memory.

Data Memory

Relevant commits:

- Added DataMem
- Updated cpu.sv to include data memory
- Modified data memory and control signals for byte addressing
- Corrected DataMem

When I first added data memory to the CPU I created a memory file very similar to the instruction memory, except it was a RAM instead of ROM and it was possible to write to the memory when the write enable (we) control signal was high. This was capable of running the load and store word instructions.

Later on in the design process when the load and store byte instructions were required I heavily modified the design. I will discuss this up-to-date version.

The first change I made to the memory was to change the data size in the ROM to 8-bit, therefore meaning that individual bytes could be easily accessed and stored when using the input address directly.

See the code for reading and writing bytes below:

Writing:

```
    else if (we && ByteOp) begin
        ram_array[Address] <= WriteData[7:0];
    end
```

Reading:

```
    if (ByteOp) begin
            ReadData = {24'b0, ram_array[Address]};
    end
```

Note: For writing bytes to memory we always select the bottom 8-bits of the write data since the store byte instruction always stores the least significant byte in the register to memory.

You will notice in both the code snippets that the boolean `ByteOp` is present. This is a control signal I added to distinguish between word and byte operations in the memory. `ByteOp` is set to high for the byte instructions including `SB` and `LBU` and is low for all others.

The final feature of the memory to note is how words are written and stored following the restructure. This can be seen clearly from the code:

Writing:

```
    always_ff @(posedge clk) begin
        if (we && !ByteOp) begin
            ram_array[{Address[31:2], 2'b0}]   <= WriteData[31:24];
// Big endian storage
            ram_array[{Address[31:2], 2'b0}+1] <= WriteData[23:16];
            ram_array[{Address[31:2], 2'b0}+2] <= WriteData[15:8];
            ram_array[{Address[31:2], 2'b0}+3] <= WriteData[7:0];
        end
```

When writing the input address is taken and the bottom two bits are replaced with zeros to ensure we have the base word address. Following this, bytes are selected from the `WriteData` from most significant to least and are stored in the base word address and the next 3 higher addresses respectively. This means the word is stored in memory in the big-endian format.

Reading:

```
    ReadData = {ram_array[{Address[31:2], 2'b0}],
                ram_array[{Address[31:2], 2'b0}+1],
                ram_array[{Address[31:2], 2'b0}+2],
                ram_array[{Address[31:2], 2'b0}+3]};
```

Reading works in a similar way; we start by reading the byte in the base word address, then concatenate this with the bytes in the next 3 higher addresses. The output `ReadData` is then the required full 32-bit word.

Jump Instructions

Relevant commits:

- Added control signals and logic for JAL and JALR
- Created test program jumps.s and fixed errors in CPU

In order to implement the two jump instructions, `JAL` and `JALR`, I had to add a few features to the CPU:

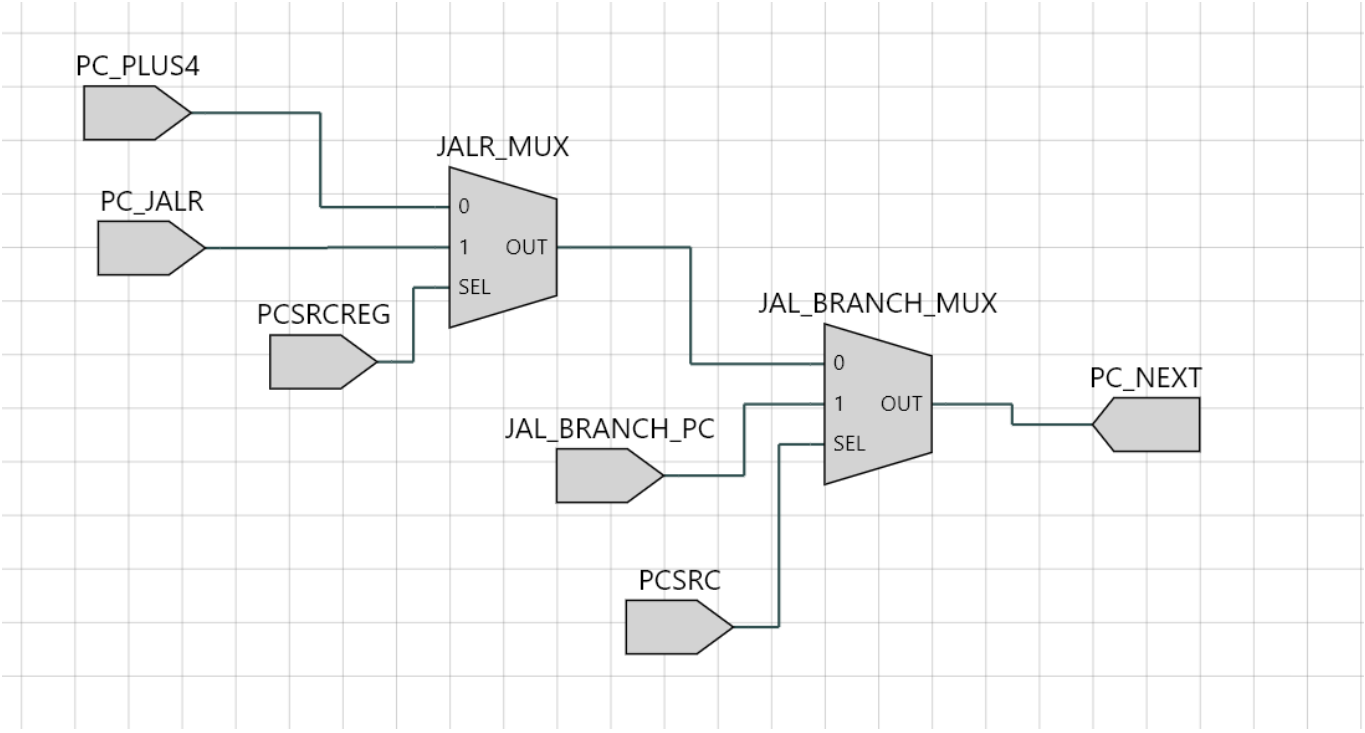
1. Calculate the program counter values for each jump instruction correctly.
2. Add control logic to select the next value of the program counter for the `JAL` and `JALR` instructions and integrate this with branch instructions.
3. Add an output and control logic to store `PC+4` in the register file when a jump instruction occurs.

To calculate the program counter value for `JAL` I had to ensure that the control signal `ImmSrc` was set to `3'b011` in order to select and sign extend the correct 20-bit immediate for the J-type instruction. This

immediate could then be added to the program counter in the same way as for branch instructions.

`JALR` is different in the sense that the program counter value comes from the ALU output. The instruction uses the regular I-type extension and the immediate is added to the `RD1` output of the register file.

I designed the logic to select the next program counter value as two cascading multiplexers:



(These were added as code in the `PC_Next.sv` module)

As above, two control signals are use to select `PC_Next`:

`PcSrcReg` is a new control signal I added that is only high for the `JALR` instruction.

`PcSrc` was the existing control signal used for branch instructions and also needs to be high for the `JAL` instruction. I implemented the following logic in the `ALUDecoder.sv` module for this:

```
    always_comb begin
        casez({Jlink, func3})
            4'b1???: PCSrc_o = 1;
            4'b0001: begin   // for bne, branch if alu output not zero
                if(branch && !zero) begin
                    PCSrc_o = 1;
                end
            end
            default: PCSrc_o = 0;
        endcase
    end
```

By using a casez statement I could include don't-cares in the case statement which is what the `?` are. `Jlink` is high only for the `JAL` instruction and when this is the case, `PCSrc` is always high. I have only included logic for the `BNE` branch instruction above; however the design is made to be easily scalable for

the other branch instructions by including different values of `func3` in the case statement and adding logic for them.

The last step to fully implement the jump instructions was adding the ability to write `PC+4` to the register file.

To do this I firstly had to add another output to the `PC_Next` component which always output the value of the current program counter plus 4.

I then created another cascading multiplexer at the write data input of the register file. The code implementation of this can be seen here:

```
    .wd3 (StorePC ? PC_Plus4 : (ResultSrc ? ReadData : ALUout)),
```

When the control signal `StorePC` is high, which is only when there is a `JAL` or `JALR` instruction, `PC+4` is written to the RegFile. In all other instructions, the write data is dependent on the `ResultSrc` control signal, selecting better then ALU or data memory output.

At this point, the jump instruction were fully implemented so I wrote a quick test program called `jumps.s` which featured a subroutine and used this to debug and confirm correct functionality.

Link to program **here**.

F1 Program

Relevant commits:

- Added left shift to ALU
- Updated control unit to include slli instruction
- Created and tested simple f1 program
- Corrected simple f1 program
- Modified f1.s to run on pipelined CPU

SLLI Instruction

Before creating and testing an f1 starting light program I knew I would need to implement the shift left logical immediate instruction (SLLI) instruction. This can be used to shift the contents a register left by a number of bits specified in the immediate.

The implementation was quite simple; firstly adding a new ALU function which would shift `ALUop1` by `ALUop2`:

```
    LSHIFT: ALUout = ALUop1 << ALUop2;
```

Then adding some control unit logic to set the correct value of the `ALUctrl` signal which can be seen in this commit

Program

At this point, all the instructions required for f1 program were implemented so I wrote a simple version which after some fixes looked like this:

```
.text
main:
    jal     ra, init           # execute init subroutine
loop:
    jal     ra, reset          # execute reset subroutine
    jal     ra, shift          # execute shift subroutine
    j       loop               # loop forever
init:
    addi    t1, zero, 0xFF     # load t1 with 255
    ret
reset:
    addi    a0, zero, 0x0      # a0 used for output
    addi    a1, zero, 0x1      # set a1 to 1
    ret
shift:
    addi    a0, a1, 0          # load a0 with a1
    slli    a1, a1, 1          # shift a1 left by 1 bit
    addi    a1, a1, 1          # increment a1 by 1
    bne     a1, t1, shift      # if a1 !=255, branch to shift
    addi    a0, a1, 0          # load a0 with a1
    ret
```

I also made a simple testbench to produce a `.vcd` waveform output but did not include any Vbuddy functionality. When testing the program initially it was not functioning as intended and after examining the waveform I realised I had somehow made it this far without making register x0 unwritable.

This was a quick fix, with line 19 of `RegFile.sv` being changed to:

```
if (we3 && ad3 !==0)
```

Now the register would not write if the write address was zero.

After this fix, the program worked well, and I handed over to my teammate James to modify the program and add a testbench for Vbuddy.

I also modified my program with `NOP` instructions to test if it would run on the pipelined CPU.

Reference Program

Relevant commits:

- Added control signals for lui
- Updated ALU for lui instruction
- Fixed Rtype control signals
- Updated pdf testbench so pdf.s now runs
- Made testbench to run reference program on Vbuddy
- Tested reference program on pipelined CPU
- Modified testbench so reference program runs on vbuddy

Features Added

When provided with the reference program I created a simple testbench and tried running it on our CPU. Unfortunately the output pdf waveform was not produced. The first error I noticed was that I has not yet implemented the `LUI` instruction required for some executions of the `LI` pseudo instruction.

To implement this I had to add logic inside the control unit to set the value of `ALUctrl` to a new value (6) for the `LUI` instruction. I then created a new function in `ALU.sv` corresponding to this value of `ALUctrl` called `PASSOP2` which simply passed through `ALUop2` to the output. This could be used with the sign extension for U-type instructions to execute the `LUI` instruction successfully.

The second bug I found during debugging was with the `MemWrite` control signal for R-type instructions. It was set to high and needed to be low.

Testing

Due to the fact the program took ~150,000 clock cycles to execute, this could not all be run on Vbuddy. I therefore created a testbench that would only plot the output data on Vbuddy once the `display:` subroutine was executed in the program.

I did this by firstly adding a new line to the reference program in `loop3:` just before the value of a0 is set to one of the pdf values:

```
LI      a0, -1              #   a0 = -1, this is used to control a bool in
the tesbench
```

As per the comment, I used this to control a boolean in the testbench called `StoreNextVal`. This boolean is set to high when `a0 == -1` and it tells me that in the next clock cycle, the value of a0 will be one of the pdf values. I could then use this to store all the pdf values in an array, and output them later to command line and to Vbuddy.

The code for storing the values looks like this:

```
    if (StoreNextVal) {
            if (ValuesStored <= 255) {
              Pdf_Values[ValuesStored] = top->a0;
              ValuesStored++;
```

```
                StoreNextVal = 0;
            }
            else break;
        }
```

Notice `StoreNextVal` is set back to zero every time a pdf value is stored. The size of the array is also limited to 255 as this is the number of pdf values the program produces.

The output section of the testbench is as follows:

```
for (int i = 1; i <= 240; i++){    //limit to 240 here since this is the
resolution of vbuddy display in x direction
      std::cout << "X: " << i << " Y: " << Pdf_Values[i] << std::endl;
      vbdPlot(Pdf_Values[i], 10, 190);  //Scaled the display slighty so
the top and bottom values can be seen more clearly
    }
```

The values stored in the array are simply output to Vbuddy, resulting in very fast execution of the program.

Pipelining

For the pipelined CPU I added many `NOP` instructions to the reference program to avoid hazards. This meant that the program now took roughly 3 times as many clock cycles to run.

In addition to this the testbench had to be modified slightly so that the program would run correctly on Vbuddy.

Since there were now two `NOP` instruction between the cycle the `StoreNextVal` boolean was set and the moment `a0` was set to the pdf value, `a0 = -1` was stored to the array twice before the desired value of a0. In hindsight I should have altered this but instead I increased the size of the array by 3 times and added to the output section:

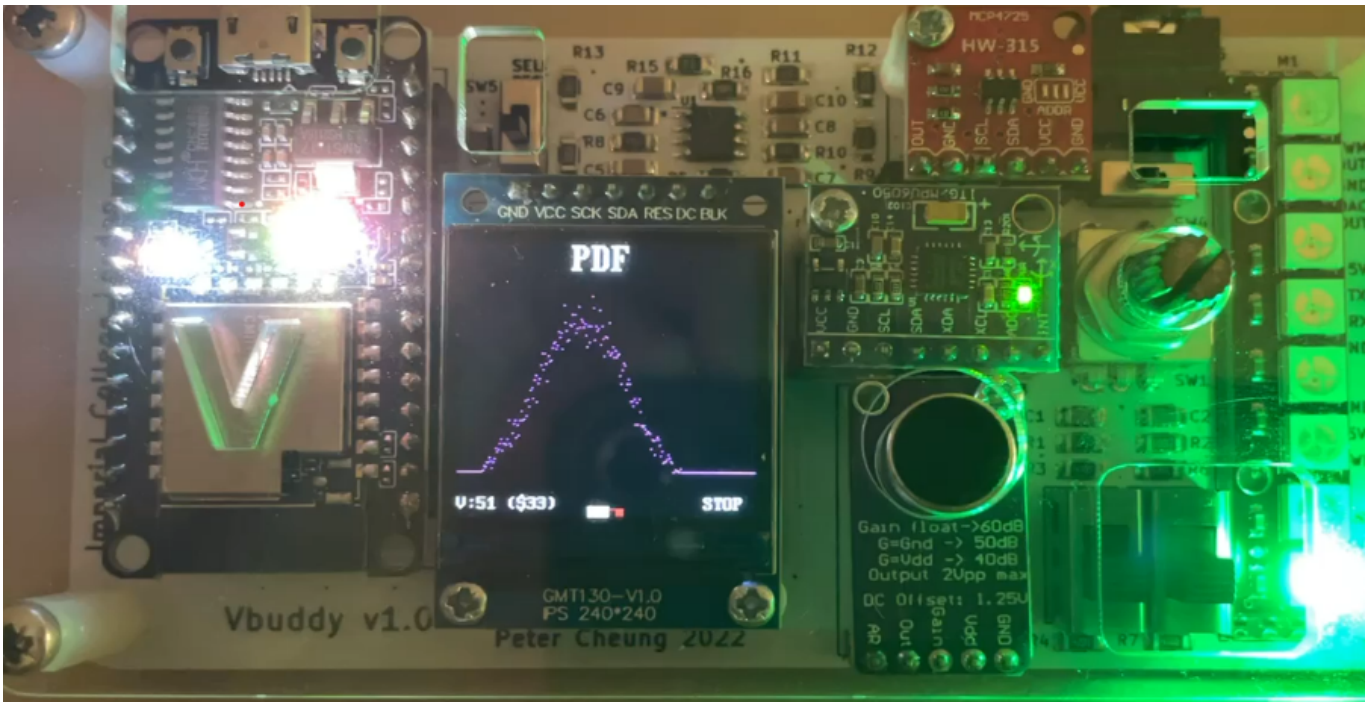
```
    if(Pdf_Values[i] != -1)
```

So the value would only be output to Vbuddy if it was not equal to -1.

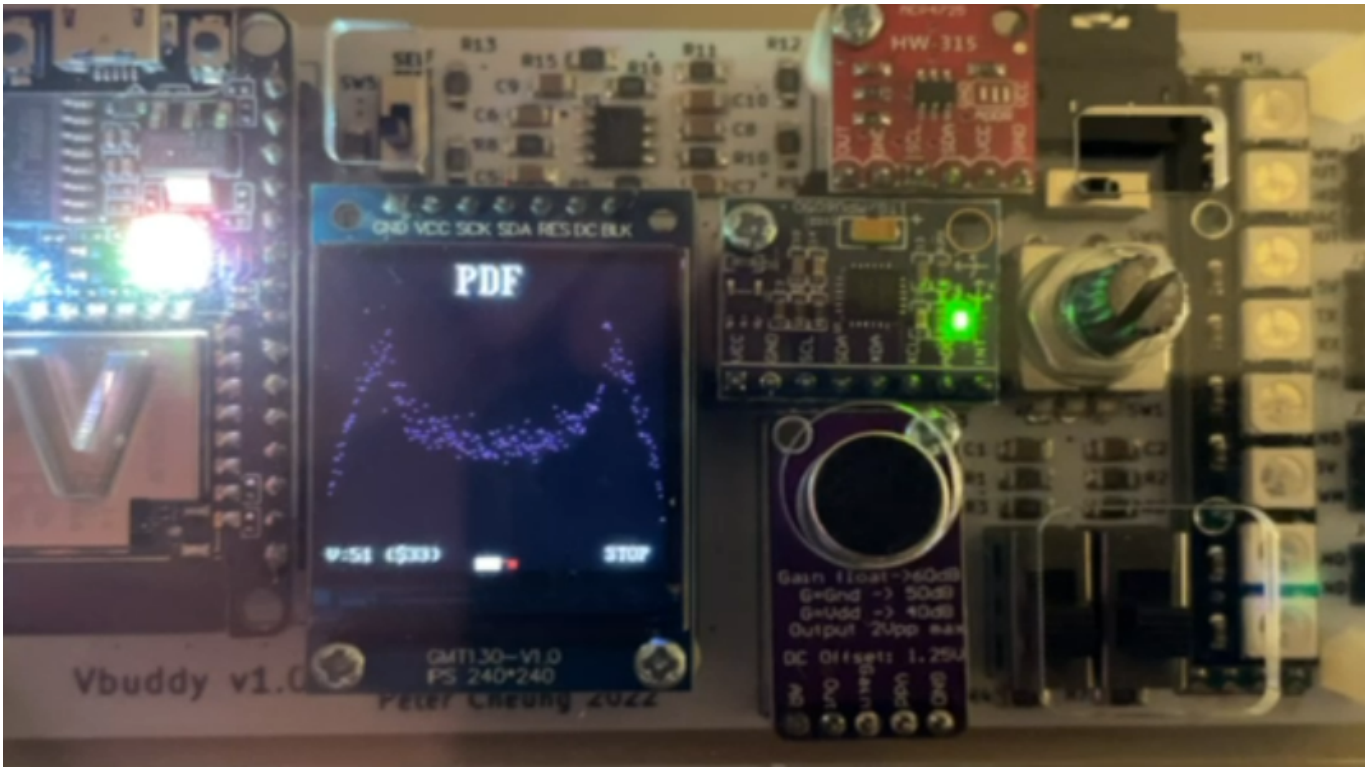
Results

Here are examples of Vbuddy displaying the result graphs:

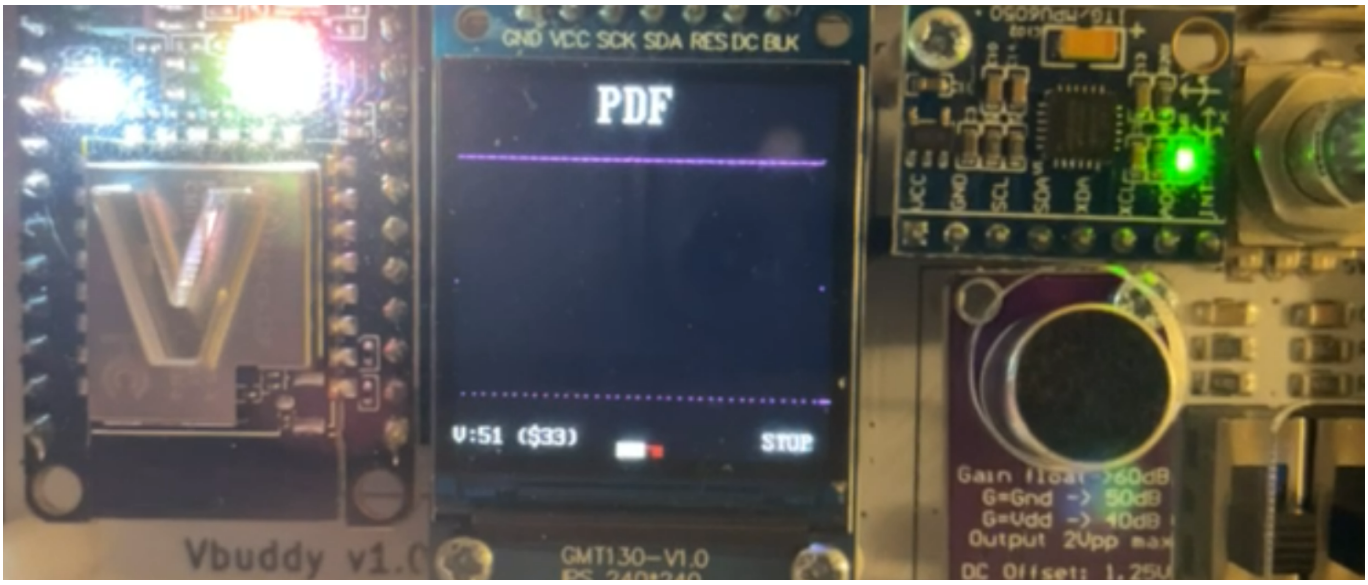
Gaussian:



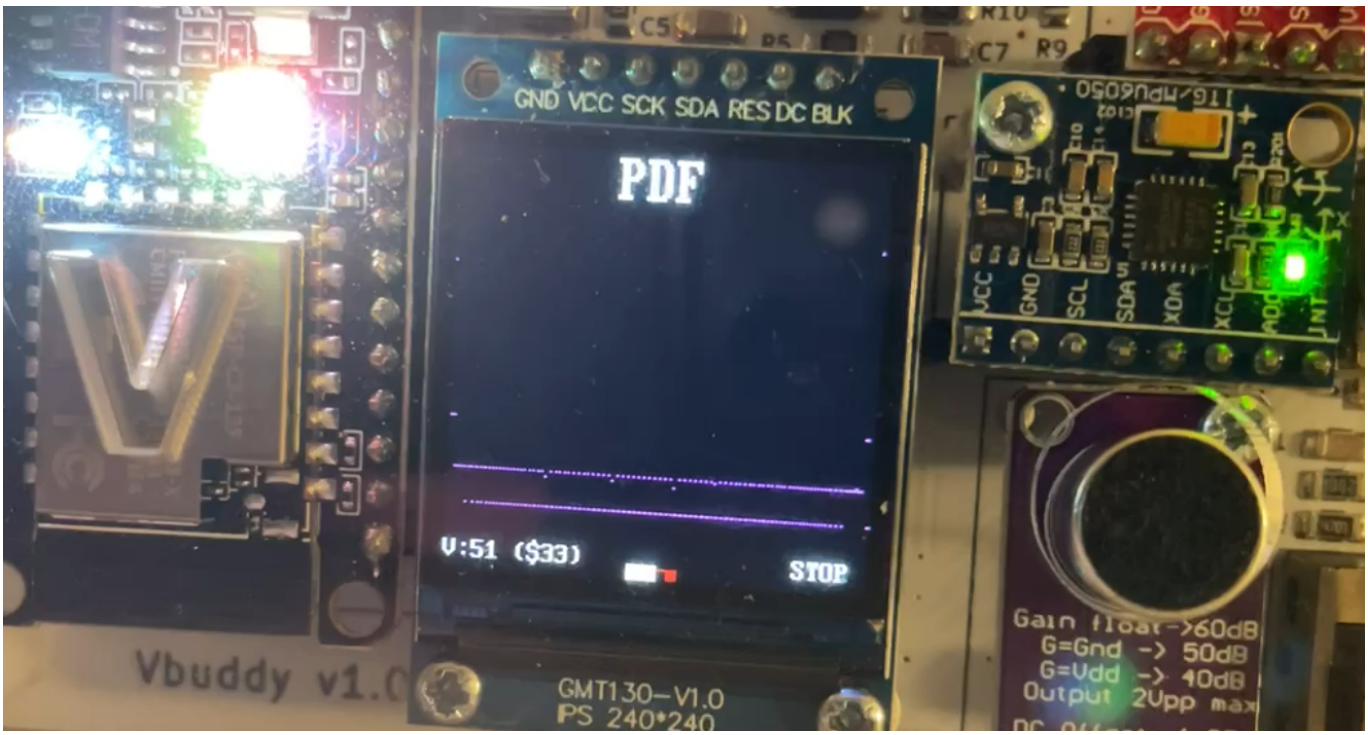
Noisy:



Triangle:



Sine:



Additional Comments

In addition to the contributions listed above, I also did some general bug fixing and verification of the CPU. For example:

- Updated CPU so addi, bne and lw instructions are fully working
- Verified basic counter program runs on pipelined CPU

To conclude, I enjoyed the time I spent working on the project and I feel satisfied that I learnt a good amount about RISC-V, SystemVerilog and about hardware design in general. I would have liked to have worked on the data cache and possibly other features if there was more time; perhaps I will revisit these in the future. One thing I would do differently if I had the chance is to take the time to properly plan the changes I made at the beginning of the project, because when I started doing this for later changes I

found I was more successful, spent less time debugging and it made the whole process more efficient and enjoyable.