Preface

All of our work is in the branch pipeline_top. To run the pdf you need to run vbuddy_ref_test.sh in rtl_pdf for the pdf prorgram, and the same script in rtl_f1 for the F1 program. Also, all our personal statements are in main in markdown_files directory.

Our team are all very passionate about computer architecture, and we saw this assignment as an opportunity to do something great. We set ourselves incredibly ambitious goals, which were only achievable by working effectively as a team. For this reason, we have chosen to spend more time highlighting the results of our work in this document, with shorter individual statements to declare contributions.

Introduction

Our processor is a 5-stage pipelined RISC-V CPU with a 3-level memory hierarchy, hazard detection, and branch prediction, which provides a full implementation of all base user-level RV32I instructions. The memory hierarchy comprises separate 2-way set-associative L1 data and instruction caches, which share a joint 4-way set-associative L2 cache, and joint main memory.

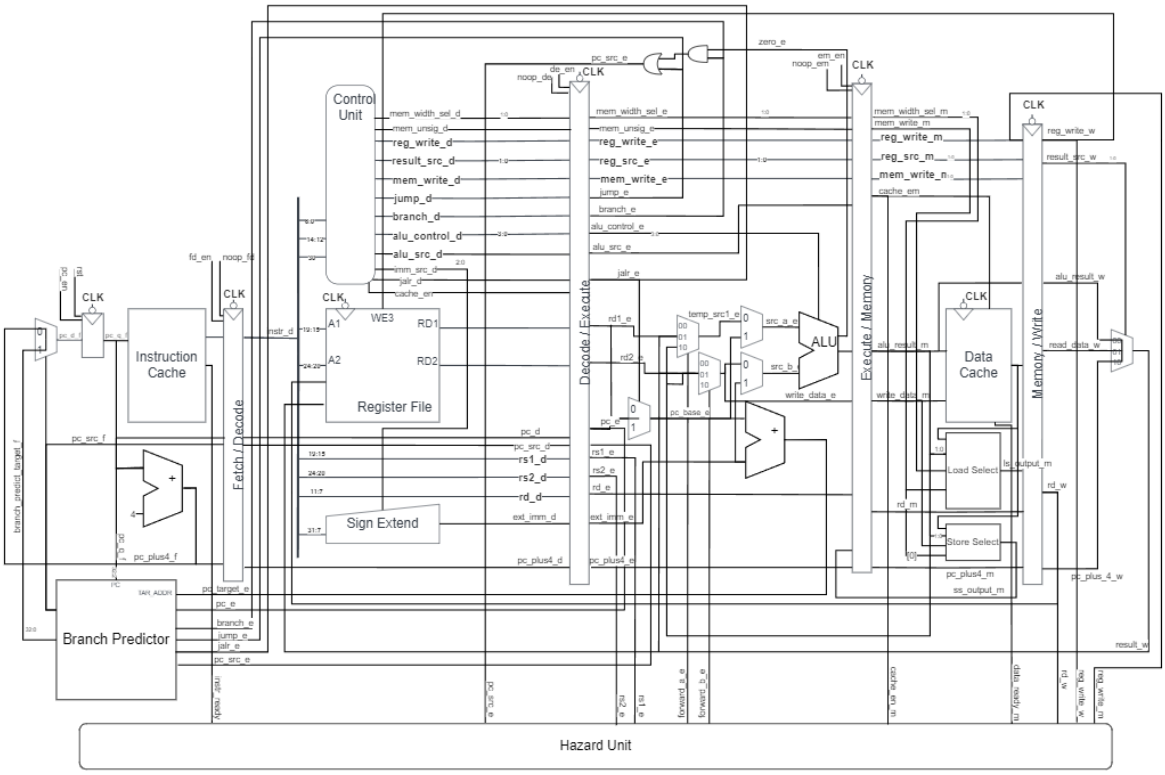
Alongside the CPU SystemVerilog source code, our team produced a full cycle-accurate functional model of the pipelined CPU in C++. This was used to aid verification of the register-transfer level (RTL) implementation. Additionally, we developed a constrained random verification tool to generate random valid RV32I machine code for enhanced testing.

Our team's goal throughout the project was to maximise the performance of our CPU as if it were to be implemented in silicon. Sometimes this conflicts with optimising the performance of the CPU in instructions per cycle (IPC).

A good example of this is our implementation of multi-level caching, which hugely reduces IPC, but results in a dramatic performance increase when accounting for realistic memory timings. In our team's RTL, we introduce artificial delays to simulate real-life behaviour. An example of this is that the main memory access time is 100 cycles, while for the L2 cache it is just 10 cycles. We took particular care when implementing new features to ensure that they did not greatly increase the delay of the critical path, so that a high theoretical clock speed could be maintained.

CPU overview

The diagram below gives a high level overview of the CPU. Our top-level design is based upon that of Harris and Harris. (4)



Entire CPU diagram

Standard components

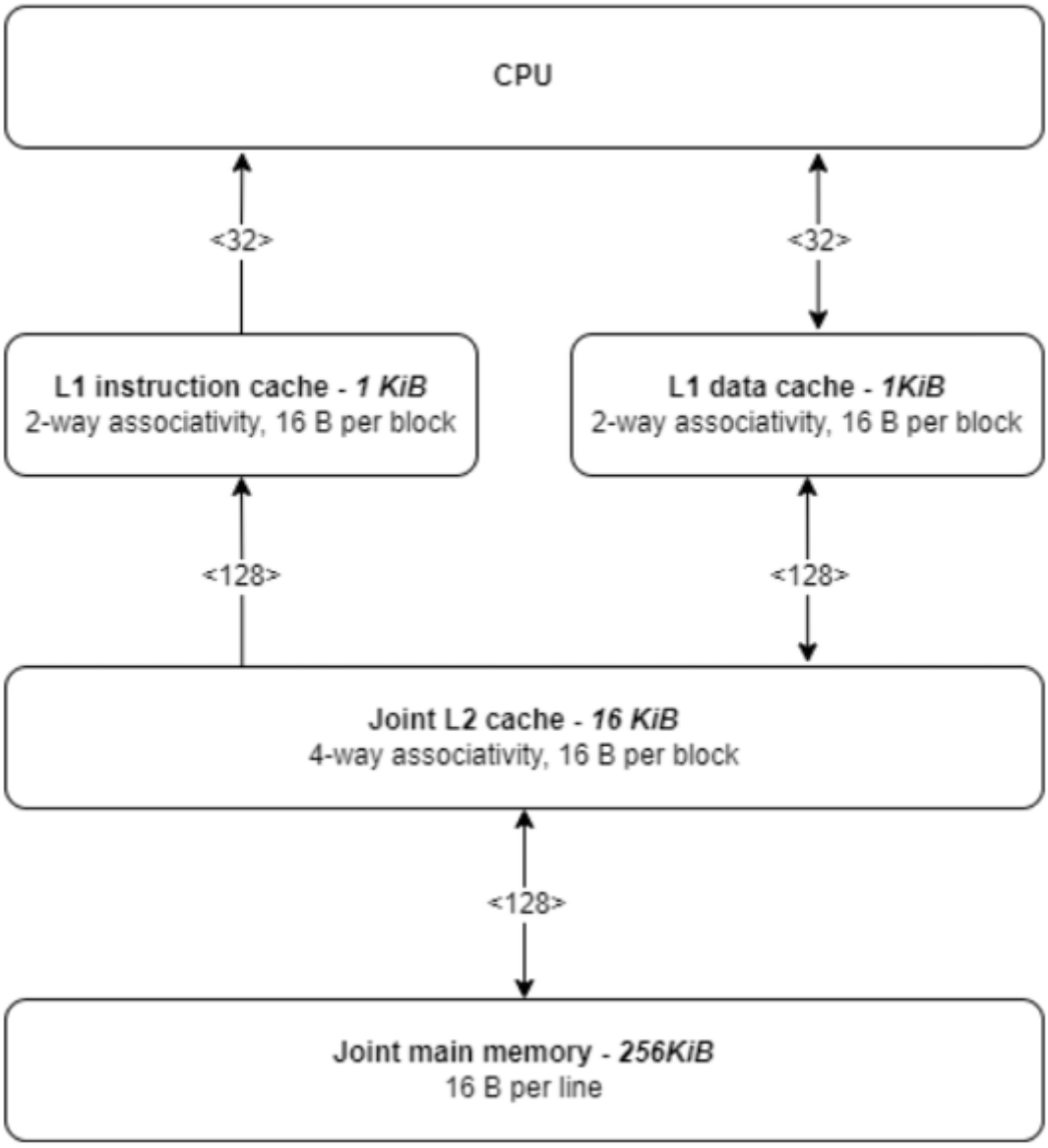
The arithmetic logic unit (ALU) supports all RV32I base user-level instructions. We define these instructions as all those instructions described in Chapter 2 of the RISC-V User-Level Instruction Set Manual (2), excluding FENCE, environment and CSR instructions. Our team decided not to implement these additional instructions, because they require significant additional hardware while providing little extra functionality for a single-core processor.

The register file is quite standard, with the exception of the ability to externally write values directly to s1, and read values directly from a0. This change was made to support Vbuddy's access to the CPU data for the reference programs.

The control unit and sign-extend unit both use straightforward implementations. They support all the same instructions as the ALU as outlined above.

Memory hierarchy

To maximize real-world performance, the CPU has separate L1 instruction and data caches, supported by a joint L2 cache and main memory. Joining the two memories gives our CPU a Von-Neumann architecture, like all modern high-performance processors.



Memory hierarchy diagram

All levels of our memory hierarchy store data in blocks of 4 words (16 bytes). This enables the processor to exploit the spatial and temporal locality of both the instructions and the data. The ports between the CPU and both L1 caches are 32-bits wide, enabling single-cycle SW and LW instructions, but reducing the performance of SB and SH instructions. We felt that this was a worthwhile tradeoff given that modern programs mostly use SW and LW instructions to reduce the number of memory accesses (which improves both speed and power-efficiency).

The L1 instruction cache and L1 data cache uses 2-way set associativity and each have a capacity of 1 KiB. We chose to use a low set-associativity to reduce the hit time of the L1 caches, which is the most critical element of their performance. This justifies our assignment of single-cycle hit times to both. We use a least-recently used (LRU) replacement policy to capitalize on temporal locality.

The joint L2 cache has 4-way set associativity with a capacity of 16 KiB. We increased the set-associativity for our lower-level cache to optimise its hit rate, even if this decreases the hit time. Since the miss penalty in L2 is so high, this is a worthwhile tradeoff. We assigned an L2 access time of 10 cycles, and use a pseudo-LRU replacement policy. The L2 cache handles one L1 request at a time, alternating between them to reduce stalls. This design is inspired by an Intel Core i7 processor, which is well described in Hennessy and Patterson's Computer Architecture: A Quantative Approach (3).

The main memory has a capacity of 256 KiB, and has a 16-byte interface with L2. We assigned main memory a realistic access penalty of 100 cycles.

Pipeline and hazards

The CPU uses a classical 5-stage pipeline with the following stages: fetch, decode, execute, memory, and write. Hazarding logic is required for producer-consumer forwarding from the memory and write stages. Great care is required in the case that the producer is a LOAD instruction immediately succeeded by its consumer. This requires stalling the pipeline for one cycle to allow time for a memory access.

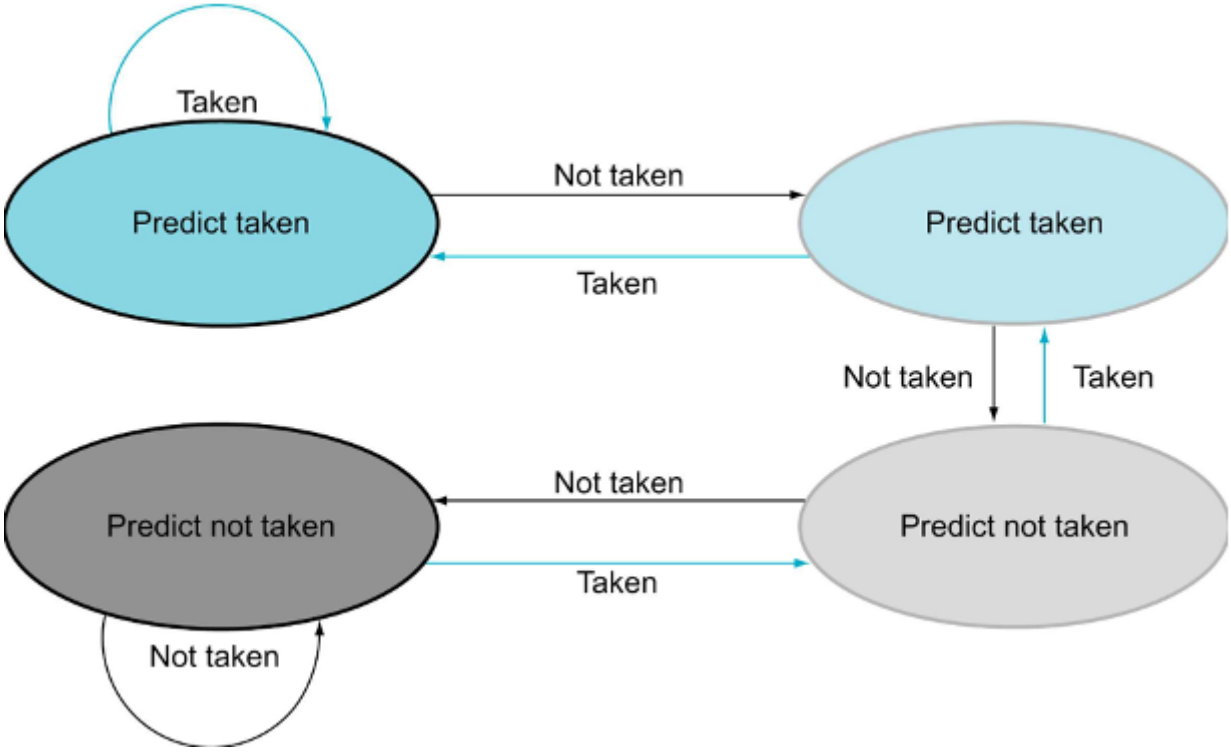
An additional data hazard is caused by our implementation of SB and SH instructions. In order to correctly execute these instructions without corrupting nearby data, the pipeline stalls for a cycle while the correct word is loaded, then we pass the word through a store select unit for multiplexing, and forward this result back into the memory stage for the next cycle to be stored. We carefully analysed the delay of the critical paths of the processor to determine whether this was feasible, and found that the critical path was the logic for branching. Using just 3 multiplexers in the store select unit, we manage not to exeeced the delay of the critical path in the memory stage.

In addition to these data hazards, the hazard unit accounts for control hazards caused by mispredicted branches. On a mispredicted branch, the instructions in the fetch and decode stages are flushed, and the program counter is updated to the correct value.

Branch prediction

Each mispredicted branch instruction has a delay penalty of 2 clock cycles, due to the need to flush the instructions in the fetch and decode stages. After analysing the reference program, we found that it branched much more often than not, and so decided to implement a branch predictor to improve the performance of our CPU.

Our branch predictor consists of a 16-entry branch history table (BHT) which contains the addresses of branch instructions, and their corresponding target addresses. Each entry contains two bits which are used as a counter to predict whether or not each branch should be taken. The BHT is direct-mapped based on the address of the branch instruction. We do not store JALR instructions in the BHT, as their taken-ness is less predictable.



Branch prediction for 1 address FSM [1]

Once a branch is recorded into the BHT, it is assigned a weakly taken state (light blue in the diagram above). If the program counter matches the address of an instruction whose address is already recorded in the table, then the BHT outputs a branch prediction. If the state of the recorded address in the table is strongly taken or weakly taken, the branch is taken. Otherwise (ie. when the state is strongly not taken or weakly not taken) we do not take the branch. The state is then updated based on whether the result was successful or not, as indicated by the diagram.

Verification

Our team believes that the reference programs are not sufficient to verify the functionality of a RISC-V processor implementing the full RV32I base use-level instruction set. In order to aid verification, we created a cycle-accurate model of the pipelined (non-cached) CPU in C++, which is around 1500 lines long. This tracks the architectural state of the CPU (program counter and registers), and the values given by the CPU model can be compared against those given by the RTL each cycle to help with verification and debugging. We used this model to fully verify our model without caching and branch prediction, but did not have time to update the model with these extra features.

Alongside the CPU model, we produced a constrained random verification tool, which produces a machine code file with random valid RV32I instructions. Constrained random verification allows for the testing of rare edge cases, which an engineer might have missed.

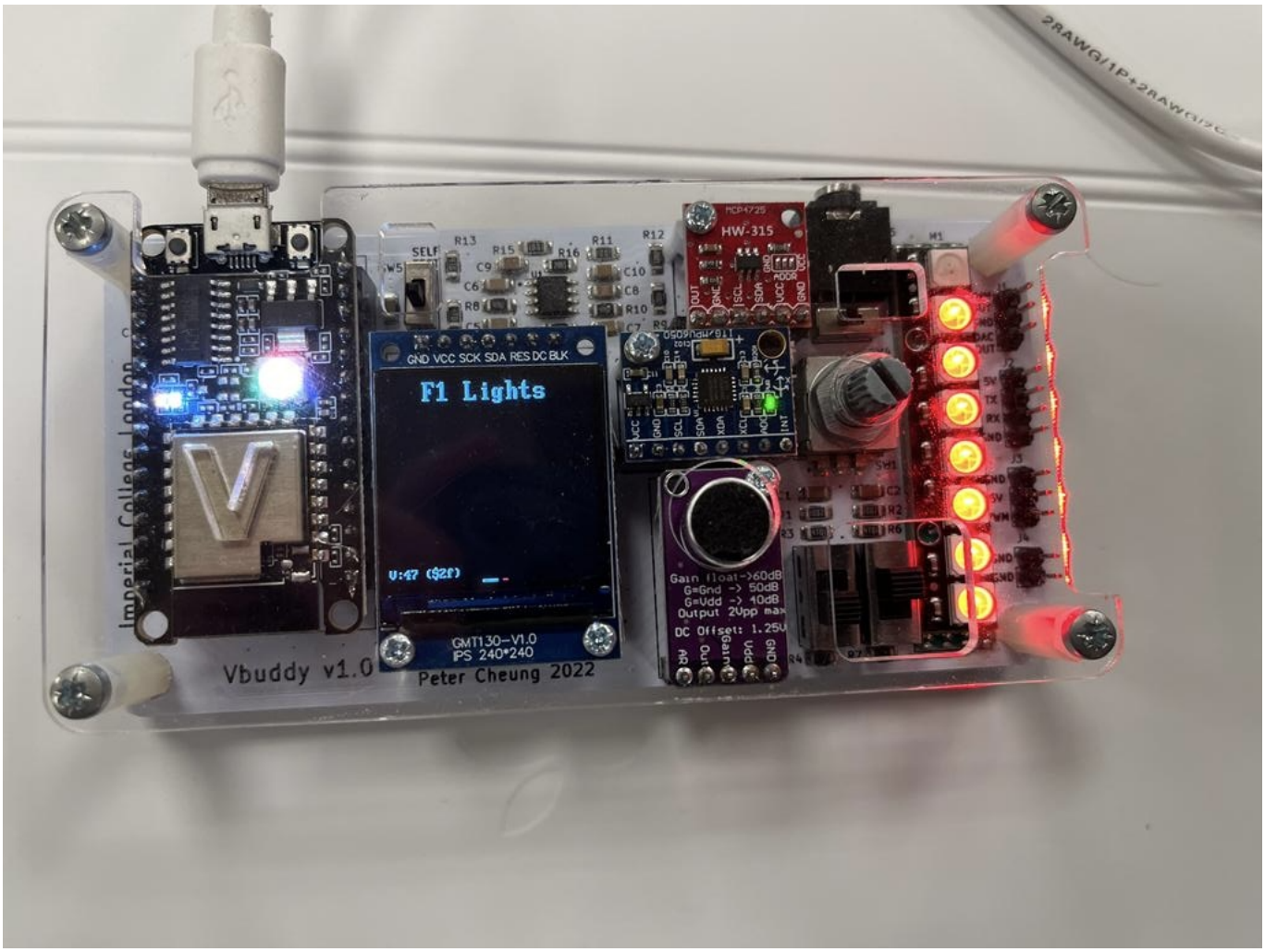
Silicon implementation and timings considerations

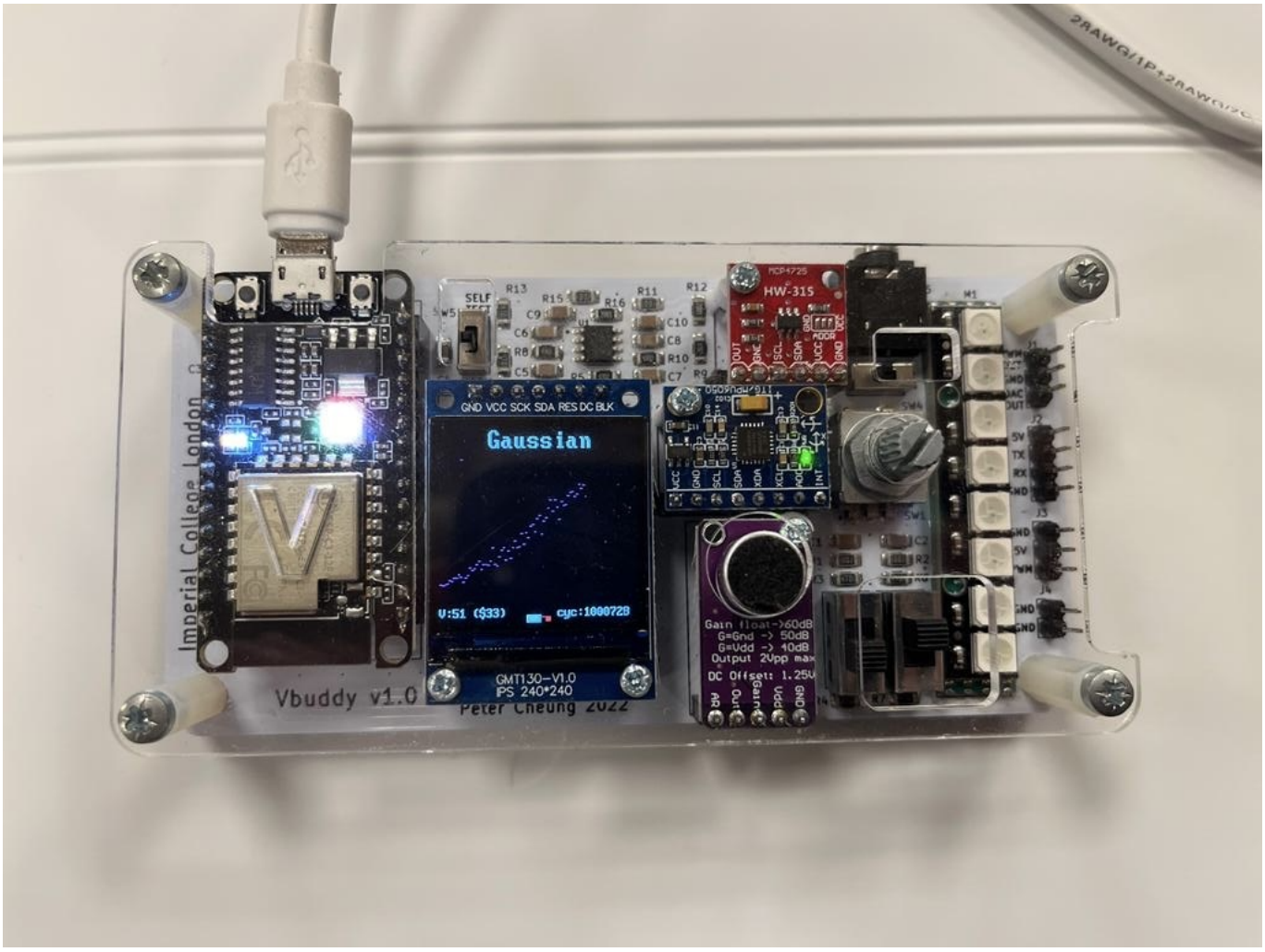
Even though the CPU will never be physically produced, our team has been mindful of how the implementation would work outside of simulations. A lot of decisions on how to structure the pipeline were based upon minimizing the critical path. By doing so, the CPU can be clocked at a much higher rate, thus enabling the execution of more instructions per second. We have based these considerations on the timings shown in the table below (1).

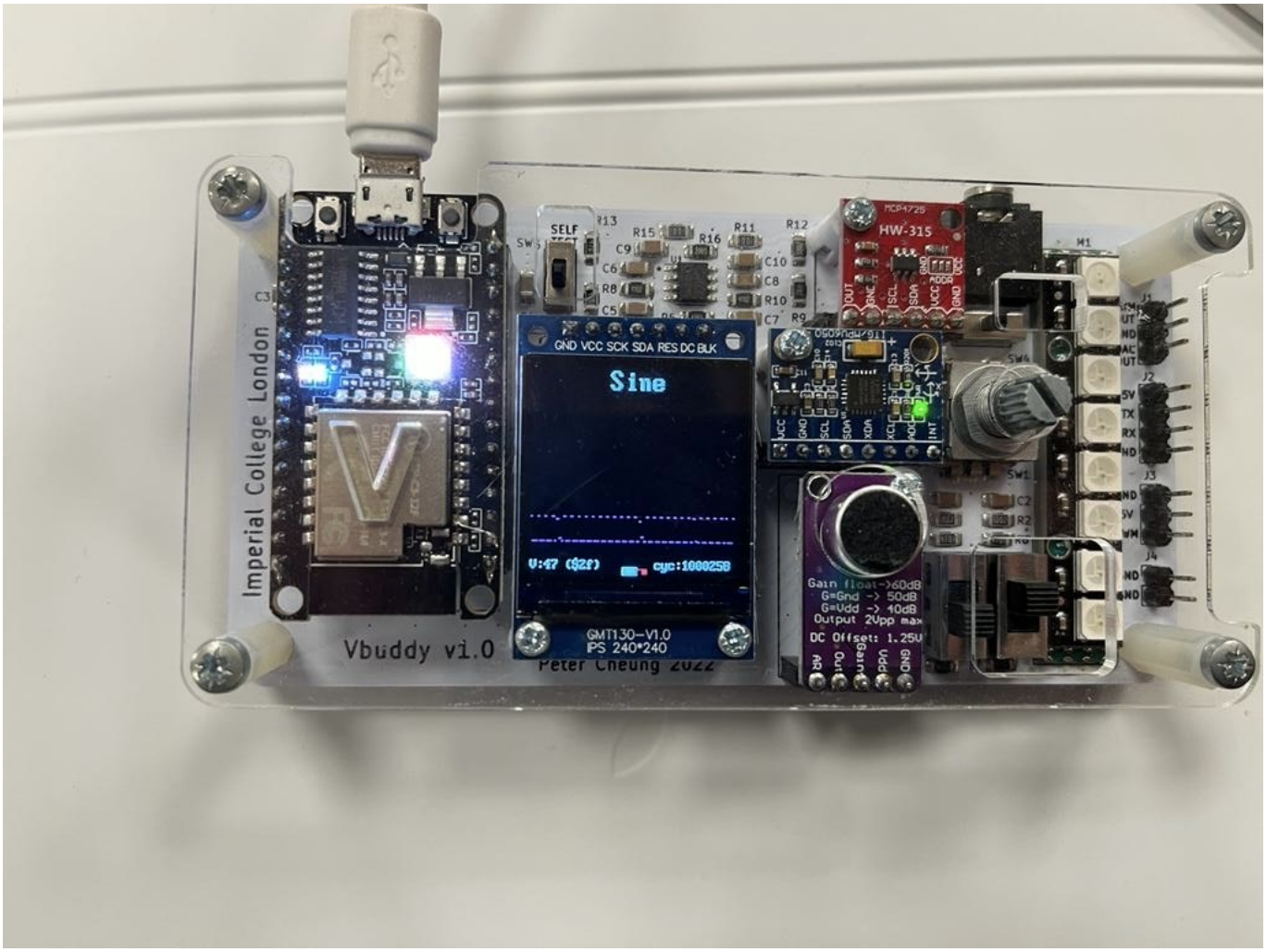
Element	Parameter	Delay (ps)
Register clk-to-Q	t_{pcq}	40
Register setup	t_{setup}	50
Multiplexer	t_{mux}	30
AND-OR gate	$t_{AND\text{-}OR}$	20
ALU	t_{ALU}	120
Decoder (control unit)	t_{dec}	25
Extend unit	t_{ext}	35
Memory read	t_{mem}	200
Register file read	t_{RFread}	100
Register file setup	$t_{RFsetup}$	60

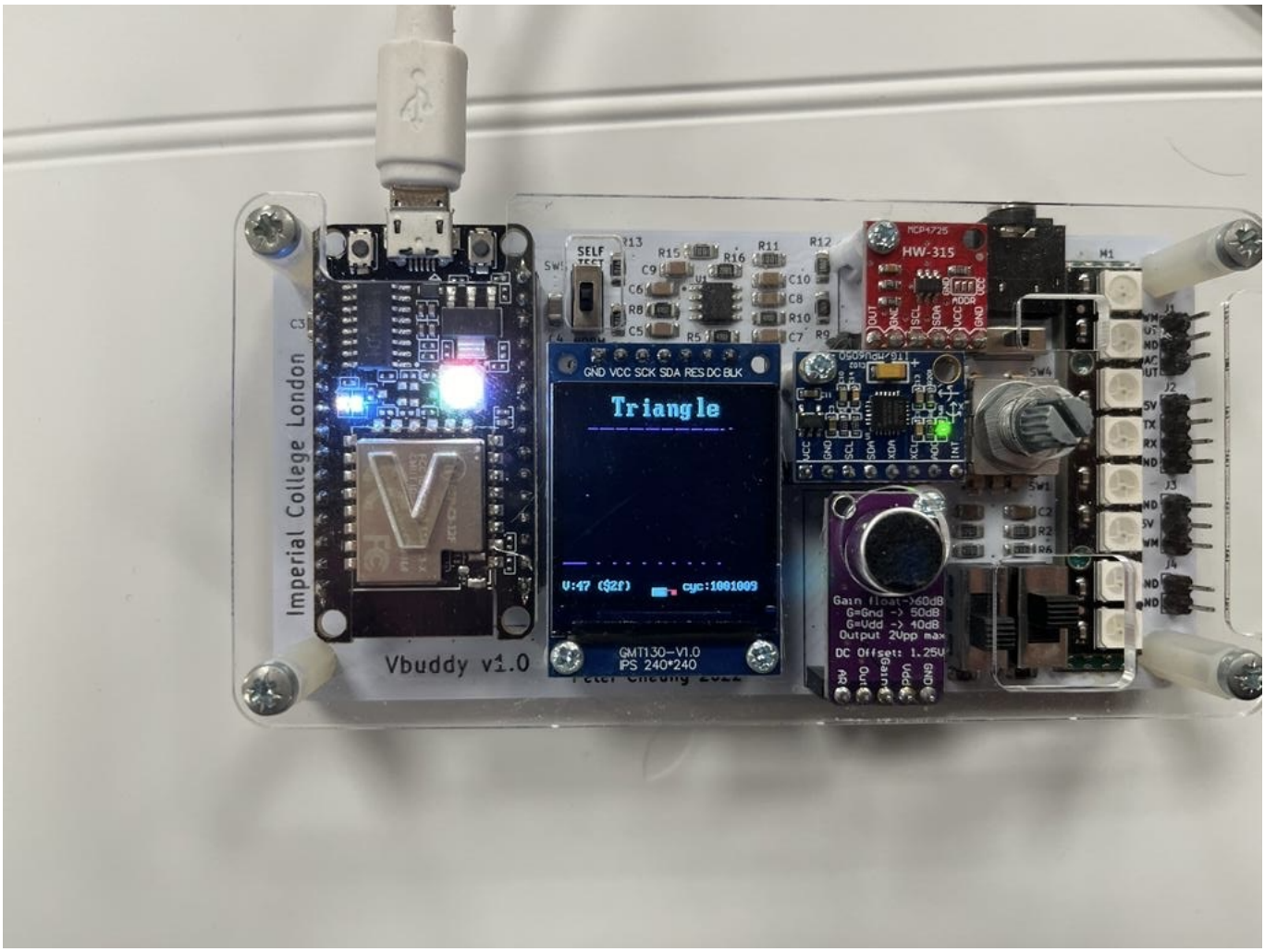
Delays for different components

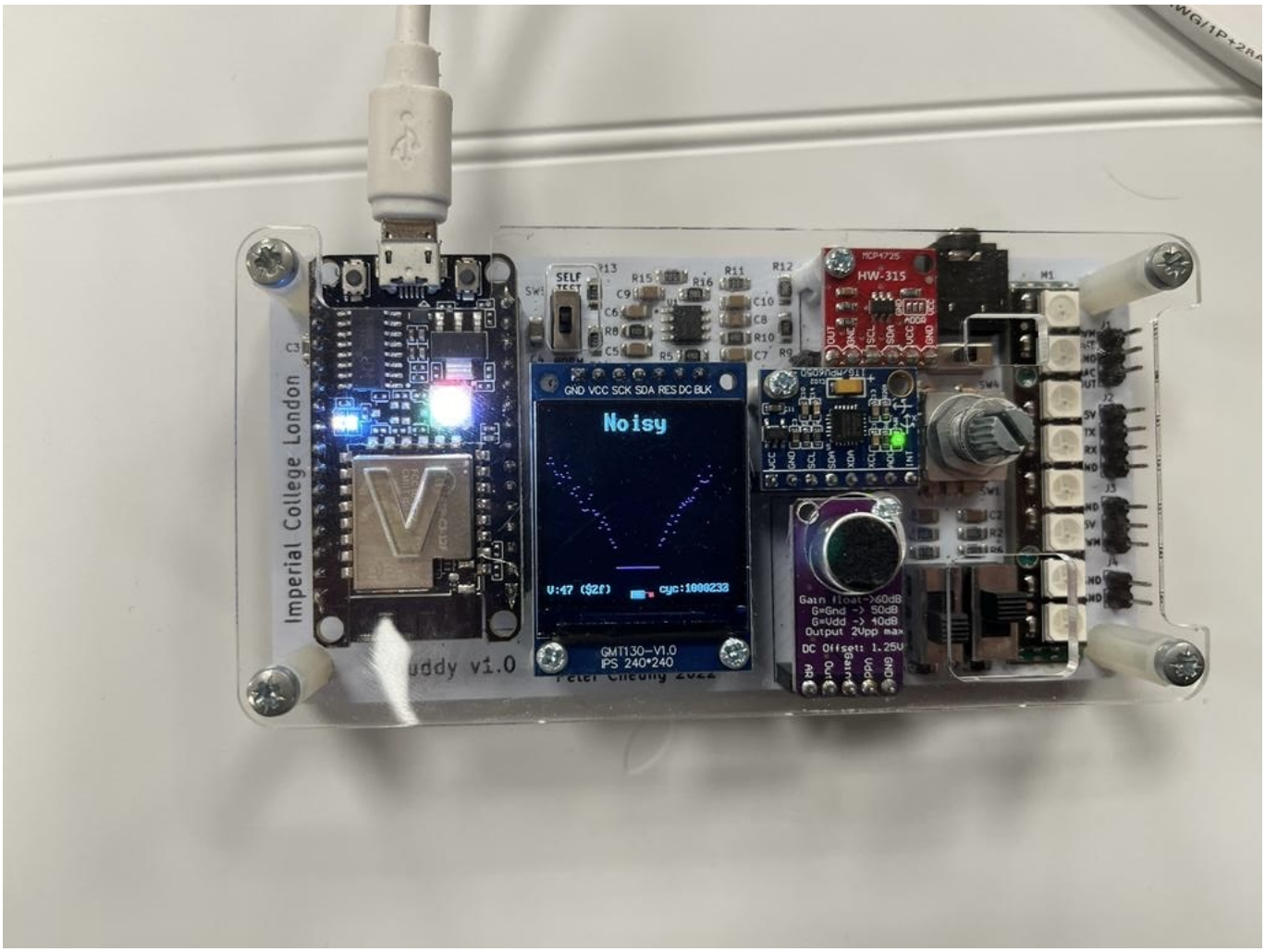
Reference programs


F1 Lights
GND VCC SCK SDA RES DC BLK
GMT130-V1.0
IPS 240*240
Vbuddy v1.0
Peter Cheung 2022
Imperial College London
HW-315
Gain float->60dB
G=Gnd -> 50dB
G=Vdd -> 40dB
Output 2Vpp max
DC Offset: 1.25V


Gaussian
GND VCC SCK SDA RES DC BLK
GMT130-V1.0
IPS 240*240
Vbuddy v1.0
Imperial College London
HW-315
Gain float->60dB
G=Gnd -> 50dB
G=Vdd -> 40dB
Output 2Vpp max
DC Offset: 1.25V


Imperial College London
Vbuddy v1.0
Peter Cheung 2022
GND VCC SCK SDA RES DC BLK
Sine
GMT130-V1.0
IPS 240*240
HW-315
Gain float->60dB
G=Gnd -> 50dB
G=Vdd -> 10dB
Output 2Vpp max
DC Offset: 1.25V


Imperial College London
Vbuddy v1.0
GND VCC SCK SDA RES DC BLK
Triangle
V:47 ($2f)
cyc:1001009
GMT130-V1.0
IPS 240*240
HW-315
SELF TEST
Gain float->60dB
G=Gnd -> 50dB
G=Vdd -> 10dB
Output 2Vpp max
DC Offset: 1.25V



F1 lights:

The program for the Formula 1 starting lights consists of 2 main states: idle and counting. The state on start-up is idle, which generates a seed for the random delay based on when the button is pressed. It uses a loop, which continuously increases (and resets to add an upper constraint) a counter until a button on Vbuddy is pressed. When this happens, the state is changed and the starting lights sequence begins. Eight lights turn on one after another with a fixed delay, which was calibrated to be 1s, but depends on the speed of the computer running the simulation. When all the lights are turned on, the random delay seed is increased by 0x10 (to add a lower constraint on the random wait time), which is then used in a counter for a random delay. Then all the lights are turned off and the program goes to idle state ready to start again and generate a new seed.

Future considerations

Although our project far exceeds the requirements of the highest end of the specification, there are a few features we wanted to implement but could not find time for. These are listed below.

- Exceptions: originally we wanted to add exception handling for the F1 starting lights program, but the need to add control and status registers, as well as the significant extra hardware required pushed this beyond the scope of our project.
- Memory-mapped input-output (IO): directly accessing the register contents of a CPU with a peripheral is extremely unrealistic. In reality, memory-mapped IO which bypasses the caches would be used to interface with Vbuddy, but we did not have time to add this feature

- Full functional model support: currently, the functional model is cycle-accurate for our pipelined implementation without caching and branch-prediction. Significant time would be required to bring this model up-to-date, which we considered unworthwhile. An easier approach would be to use the model as an instruction set-simulator, and only try to verify updates to registers in the RTL, which should happen with the same results in the same order, just during different clock cycles.

Sources

1. Patterson DA, Hennessy JL. Computer Organization and Design: The hardware software interface. Cambridge Massachussetts: Morgan Kaufmann; 2021.
2. Waterman, A., Asanovi, K. and SiFive Inc (2019) 'The RISC-V Instruction Set Manual, Volume I: Unprivileged ISA'. Berkeley: CS Division, EECS Department, University of California.
3. Hennessy, J.L. and Patterson, D.A. (2019) Computer Architecture: A quantitative approach. Cambridge, MA: Morgan Kaufmann Publishers.
4. Harris, S.L. and Harris, D.M. (2022) Digital Design and computer architecture: RISC-V edition. Cambridge, MA: Morgan Kaufmann.