Team 24: RISC-V CPU (Pipelined with Data Cache)

Table of Contents

Processor

- Single Cycle
  - F1 Program (Samuel)
  - Program Counter & Instruction Memory (Qidong)
  - Control Unit (Samuel, Bharathaan)
  - ALU (Bharathaan)
  - Data Memory (Chenglin)
- Pipeline
  - Programs For Pipelined CPUs (Samuel)
- Data Cache
  - Data Cache (Chenglin, Qidong)
- Others
  - Result Verification (Samuel)
- Top-Level
  - Top Level File (Bharathaan)

Individual Statements

- Personal Statement: Chenglin
- Personal Statement: Qidong
- Personal Statement: Bharathaan
- Personal Statement: Samuel

Quick Start

`Entrypoint.sh` is a shell written to easily load different sets of instructions & data into the processor. Three modes are available: `f1`, `ref`, and `comp`. Each mode runs the F1 program, the reference program, or the component specific testbench respectively. See Table 1 below for a list of operations you can run with the entrypoint.sh shell.

Table 1: Entrypoint.sh Usage

Command	What It Does	Remarks
`source entrypoint.sh f1`	Runs F1 Lights program.	

Command	What It Does	Remarks
`source entrypoint.sh f1 debug`	Runs F1 Lights debug program, which shows special state when LFSR subroutine is running.	
`source entrypoint.sh ref gaussian`	Runs reference program, loading gaussian waveform into memory.	
`source entrypoint.sh ref noisy`	Runs reference program, loading noisy waveform into memory.	
`source entrypoint.sh ref sine`	Runs reference program, loading sine waveform into memory.	
`source entrypoint.sh ref triangle`	Runs reference program, loading triangle waveform into memory.	
`source entrypoint.sh comp alu`	Runs ALU component testbench.	Only works in v2.0-pipeline release.
`source entrypoint.sh comp ctrl`	Runs Control Unit component testbench.	Only works in v2.0-pipeline release.
`source entrypoint.sh comp mem`	Runs Data Memory component testbench.	Only works in v2.0-pipeline release.
`source entrypoint.sh comp pc`	Runs Program Counter component testbench.	Only works in v2.0-pipeline release.

If `entrypoint.sh` does not work, you should configure the `debug.sh` script to execute what you want. Note that the instruction memory loads from `program/instr.hex` and the data memory loads from `program/data.mem`. These were dynamically created by `entrypoint.sh`, which means you would need to copy the data over and rename before running.

Overview

Versions (See Release)

- Single Cycle: `v1.0-single-cycle`
- Pipeline: `v2.0-pipeline`
- Data Cache (merged to main): `v3.0-data-cache`

Evidence Of Working Processor

See the following embedded videos, for the F1 program and the 4 waveforms PDF programs.

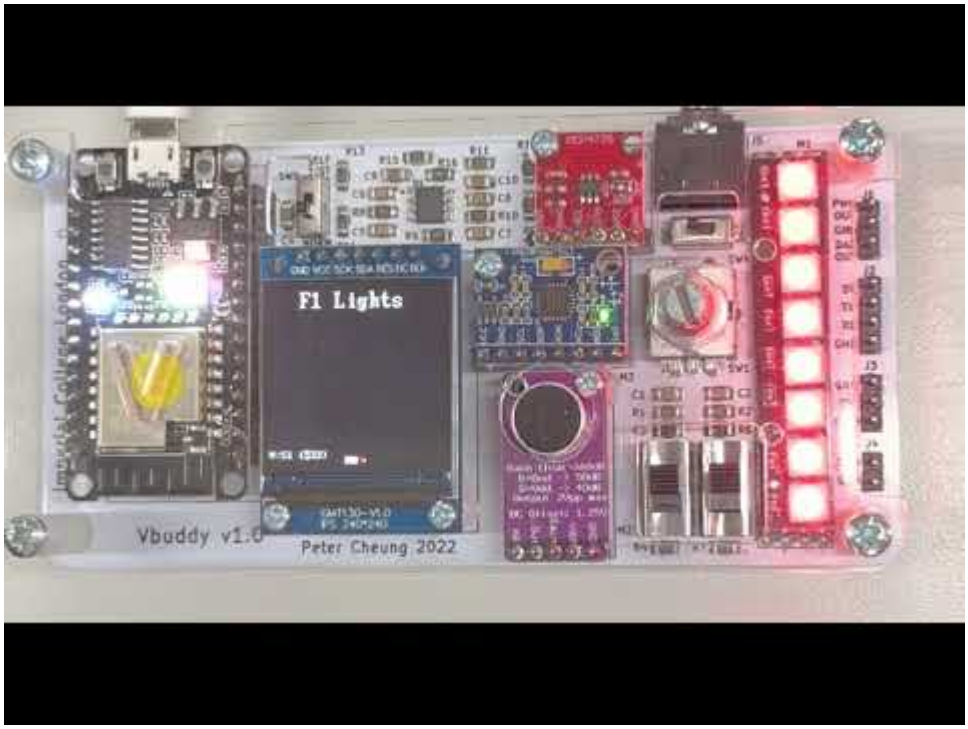

Figure 1: F1 Lights

Figure 2: Reference Program (Gaussian)

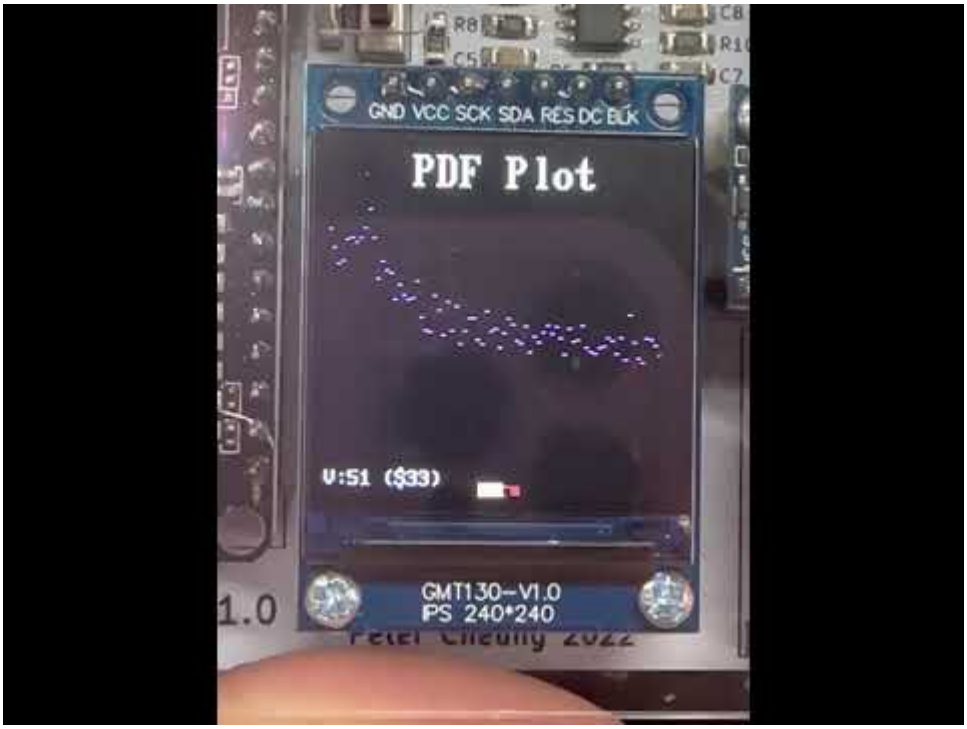

Figure 3: Reference Program (Noisy)

Figure 4: Reference Program (Sine)

Figure 5: Reference Program (Triangle)

Contribution Table

Note: o = Main Contributor; v = Co-Author.

Table 2: Contribution List

Task	Files	Chenglin	Qidong	Bharathaan	Samuel
Single Cycle	-----	-----	-----	-----	-----
Repo Setup	`.gitignore`, `.gitkeep`, ...				o
Entry Script	`debug.sh`, `entrypoint.sh`, ...			v	o
F1 Program	`f1_branch.s`, `f1_jump.s`, ...				o
Program Counter & Instruction Memory	`pc_reg.sv`, `instruction_memory.sv`, `pc_reg_tb.cpp`	v	o		
Control Unit	`control_unit.sv`, `extend.sv`, `control_top_tb.cpp`			v	o
ALU	`alu.sv`, `reg_file.sv`, `alu_tb.cpp`			o	
Data Memory	`data_memory.sv`, `data_memory_tb.cpp`	o	v		

Task	Files	Chenglin	Qidong	Bharathaan	Samuel
Top-Level Debugging	rtl.sv, ref_tb.cpp, f1_tb.cpp			o	v
Pipeline	-----	-----	-----	-----	-----
Pipeline Programs	f1_pipeline.s, f1_pipeline_debug.s, pdf_pipeline.s			v	o
Fetch Stage Registers	instruction_memory_pip.sv		o		
Decode Stage Registers	decode_register.sv				o
Execute Stage Registers	memory_stage_register.sv			o	
Memory Stage Registers	memory_writeback_register.sv	o			
Top-Level Debugging	rtl.sv, ref_tb.cpp, f1_tb.cpp			o	v
Data Cache	-----	-----	-----	-----	-----
Data Cache (1-way)		o	o		
Data Cache (2-way, incomplete)		o	o		
Top-Level Debugging	rtl.sv, ref_tb.cpp, f1_tb.cpp	v	v	o	

Specifications

Table 3: Implemented Instructions

Type	Instructions
R	add, xor
B	bne
I	addi, slli, lb, lw, lbu

Type	Instructions
S	`sb`, `sw`
U	`lui`
J	`jal`, `jalr`

Table 4: General Specifications

Property	Value
Instruction Memory Size	2^12 bits
Instruction Width	32-bit
Data Memory Size	2^17 bits
Data Witdh	8-bit
Data Cache Size	128 bytes
Data Cache Sets	8
Data Cache Ways	1
Data Cache Block Size	4 * 32-bit

For more detailed detailed specifications, see our Specification Sheet.

File Structure

Ideal Final File Structure

```
root
├── docs/
│   ├── img/
│   ├── ipynb/
│   └── *.md
├── program/
│   ├── f1/
│   ├── ref/
│   ├── *.sh
│   └── *.mk
├── source/
│   ├── alu/
│   ├── ctrl/
│   ├── mem/
│   ├── pc/
│   └── rtl.sv
├── testbench/
│   ├── alu/
│   ├── ctrl/
│   ├── mem/
```

```
│   ├── pc/
│   └── rtl/
├── .gitignore
├── entrypoint.sh
├── entrypoint.cfg
├── debug.sh
├── debug.cfg
├── vbuddy.cfg
└── README.md
```

Directories

1. `docs`: directory that holds information about the project and its source files.
2. `source`: directory that holds all the .sv design files.
3. `testbench`: directory that holds the testbench for each component and overall design.
4. `program`: directory that holds the .s source files for program and data to be loaded into the processor.

Notable Files

1. `README.md`: overall readme document.
2. `.gitignore`: standard file to ignore verilator obj_dir, .vcd, and other os files.
3. `entrypoint.sh`: shell script that builds and runs the design, with easy debugging features.
4. `entrypoint.config`: configuration file for verilator arguments, such as suppressing warnings and others.
5. `debug.sh`: fall back for entrypoint.sh.
6. `source/rtl.sv`: top level file for our design.