RISC-V PROCESSOR

Team 8 - Diversity 8

Team Information

Name	CID	Github
Shashank Vinoo*	02600927	shashank-vinoo
Ryan George	02581757	ryangeorge285
Archit Bhansali	02566199	Arkit28
Rishabh Rastogi	02555762	RishabhR-06

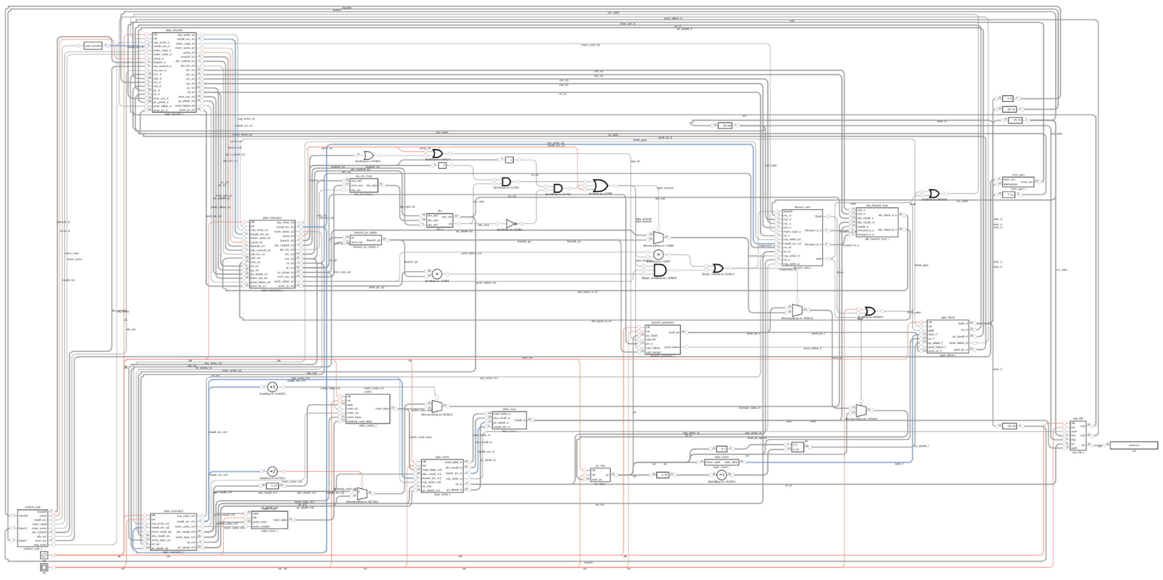
Link to repo

Introduction

All four of us are very passionate about computer architecture and have spent a considerable amount of time working on this project. In this project we complete a RISC-V(IM) processor that features:

1. Deep pipelining with 7 stages
2. Cache
3. Branch prediction
4. Multiply extension (Including divide instructions)

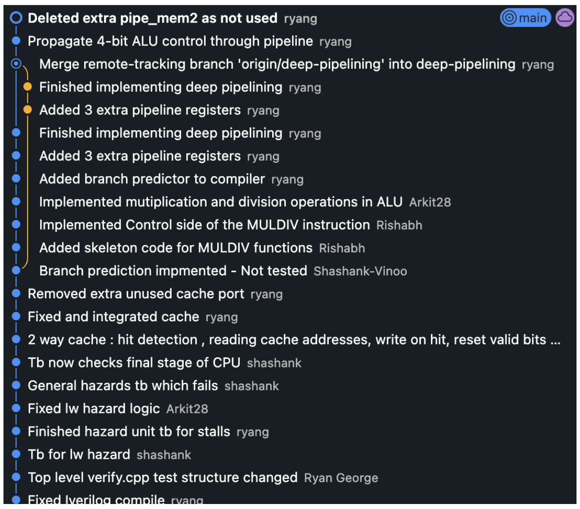
CPU Architecture



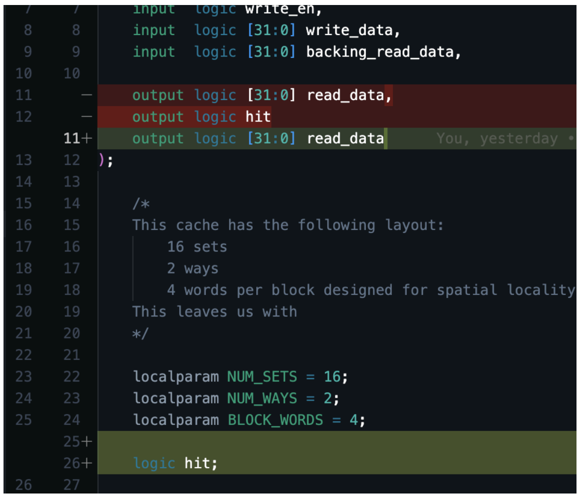
Design

Development Process

The overall development process we followed was iterative, working on a prototype and then improving it.
We used git extensively to source control our work and it worked very effectively for us. Whenever we needed to do any hazardous work, we used to branch out to prevent any need for reset commits.
Sometimes commits broke the CPU, and using git switch in detached mode really helped us see what was happening before along with git diff to get a line by line visual comparison of our code.



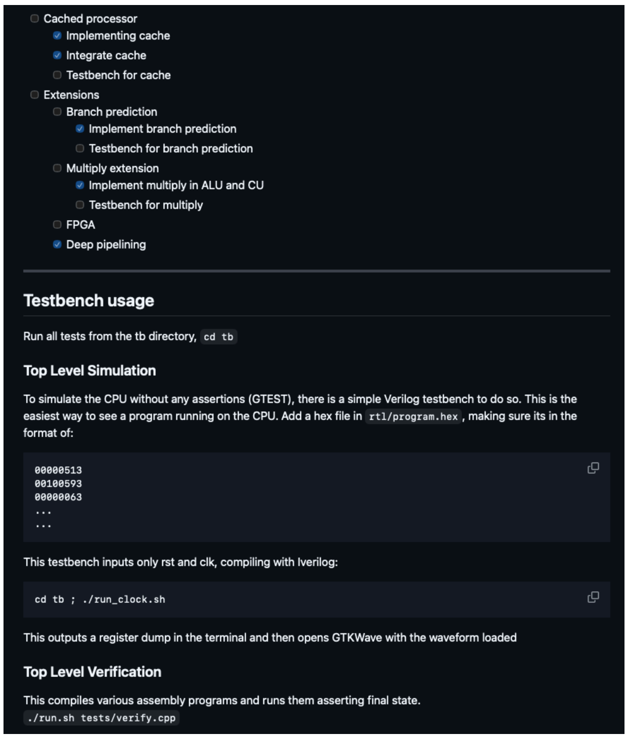
```
Deleted extra pipe_mem2 as not used ryang                                    main
Propagate 4-bit ALU control through pipeline ryang
  Merge remote-tracking branch 'origin/deep-pipelining' into deep-pipelining ryang
  Finished implementing deep pipelining ryang
  Added 3 extra pipeline registers ryang
  Finished implementing deep pipelining ryang
  Added 3 extra pipeline registers ryang
  Added branch predictor to compiler ryang
  Implemented mutiplication and division operations in ALU Arkit28
  Implemented Control side of the MULDIV instruction Rishabh
  Added skeleton code for MULDIV functions Rishabh
  Branch prediction impmented - Not tested Shashank-Vinoo
Removed extra unused cache port ryang
Fixed and integrated cache ryang
2 way cache : hit detection , reading cache addresses, write on hit, reset valid bits ...
Tb now checks final stage of CPU shashank
General hazards tb which fails shashank
Fixed lw hazard logic Arkit28
Finished hazard unit tb for stalls ryang
Tb for lw hazard shashank
Top level verify.cpp test structure changed Ryan George
Fixed Iverilog compile ryang
```



```
    input  logic write_en,
    input  logic [31:0] write_data,
    input  logic [31:0] backing_read_data,

-   output logic [31:0] read_data,
-   output logic hit
+   output logic [31:0] read_data      You, yesterday •
);

    /*
    This cache has the following layout:
        16 sets
        2 ways
        4 words per block designed for spatial locality
    This leaves us with
    */

    localparam NUM_SETS = 16;
    localparam NUM_WAYS = 2;
    localparam BLOCK_WORDS = 4;
+
+   logic hit;

```

Left: Screenshot of git commits *Right: Git diff in VSCode*

We also used the github README to quickly organise our work and task, and provide example instructions on how to run the verification tools:



- [ ] Cached processor
  - [x] Implementing cache
  - [x] Integrate cache
  - [ ] Testbench for cache
- [ ] Extensions
  - [ ] Branch prediction
    - [x] Implement branch prediction
    - [ ] Testbench for branch prediction
  - [ ] Multiply extension
    - [x] Implement multiply in ALU and CU
    - [ ] Testbench for multiply
  - [ ] FPGA
  - [x] Deep pipelining

Testbench usage

Run all tests from the tb directory, `cd tb`

Top Level Simulation

To simulate the CPU without any assertions (GTEST), there is a simple Verilog testbench to do so. This is the easiest way to see a program running on the CPU. Add a hex file in `rtl/program.hex`, making sure its in the format of:

```
00000513
00100593
00000063
...
...
```

This testbench inputs only rst and clk, compiling with Iverilog:

```
cd tb ; ./run_clock.sh
```

This outputs a register dump in the terminal and then opens GTKWave with the waveform loaded

Top Level Verification

This compiles various assembly programs and runs them asserting final state.
`./run.sh tests/verify.cpp`

Overall, our group was linked together by git and it enabled us to effectively carry out this project. It also helped us solidify our skills in git, which is something every engineer needs to know.

Pipelining and Hazards

Our CPU uses a 5 stage pipelining: fetch, decode, execute, memory and write. To implement the pipelined architecture we created four register modules corresponding to fetch->decode, decode->execute etc. The benefit of pipelining is that it allows for greater clock speeds because multiple instructions progress through the pipeline per clock cycle, as compared to a simpler architecture without pipelining which must wait for one instruction to be completed before loading the next one.

The most frequent type of hazard encountered was the Data Hazard, where an instruction requires data that has not yet been written back by a preceding instruction. Our solution prioritized forwarding over stalling to maintain high throughput. Forwarding logic was implemented to detect when the result of an instruction currently in the Execute (E), Memory (M), or Write Back (W) stage is needed as an operand for an instruction in the Decode (D) or Execute (E) stage. In such cases, the result is immediately routed from the output of the source stage back to the input of the stage requiring the data, bypassing the register file write cycle. Stalling was reserved only for the load-use hazard, where an instruction immediately follows a Load instruction that provides its operand. Since the data is only retrieved from memory during the M stage, a single clock cycle stall is necessary to ensure the correct data is available before the dependent instruction enters the E stage.

Control Hazards arise from branch, jump, and call instructions, as they change the Program Counter (PC) and thus invalidate instructions already fetched into the pipeline. To minimize the performance penalty, we employed a simple yet effective strategy. For unconditional jumps and calls, the target address is calculated early in the Decode (D) stage. For conditional branches, we implemented a basic Not-Taken static branch prediction scheme. The pipeline optimistically fetches the next sequential instruction. If the branch is determined to be *taken* in the Execute (E) stage, the misfetched instructions in the F/D and D/E registers are flushed (converted to NOPs), and the PC is redirected to the target address. This flush incurs a penalty of only one wasted clock cycle.

Cache

Our processor uses a small 2-way set-associative cache with 16 sets to reduce the number of accesses to backing memory. In our design, each block stores only a single 32-bit word, which keeps the hardware simple but still helps the pipeline run more smoothly. The address is split into a 4-bit set index and a tag, and both ways in the selected set are checked for a valid tag match. On a hit, the data is returned immediately, and writes update the cached word directly. On a miss, the cache loads the value from backing memory and replaces an entry using a single-bit LRU scheme, or fills an invalid way if one is available.

Even though it’s a small cache, building it gave us hands-on experience with tag matching, replacement policies, and the interaction between memory hierarchy and pipeline performance.

Branch Prediction

The predictor works like a memory of past behaviour. For each branch address it has seen before, it remembers two things, the last place that branch jumped to, and how often it has recently been taken versus not taken. It can be thought of as the CPU having a thought about each branch:

1) this one usually jumps back into the loop
2) this one usually just falls through

When the fetch stage encounters a branch it recognises, it uses that history to choose between two options:

- predict “taken” and start fetching from the remembered target address, or

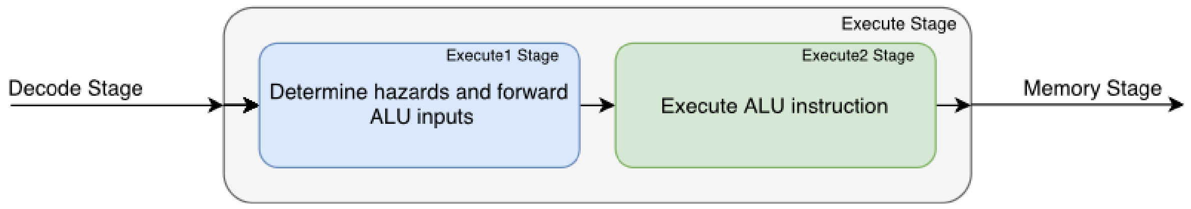
- predict “not taken” and just continue to the next sequential instruction.

We use a 2 bit FSM per branch so that the prediction doesn’t flip depending on a single unusual outcome. When the branch is eventually resolved later in the pipeline, the predictor updates its stored target and changes the confidence slightly toward “taken” or “not taken” based on what actually happened.

Even though this is a simple scheme compared to the predictors in modern high end CPUs, it still makes a clear difference: tight loops and frequently taken branches suffer fewer mispredictions, so the CPU spends more time doing useful work and less time recovering from wrong guesses.

Deep Pipelining

This processor implements deep pipelining, by splitting the execute stage into two. Intel in 2004 created a 31 stage pipeline called Pentium 4 “Prescott”, and although this had a large clock speed boost due to the critical path being shortened, it was plagued with stalls and other data hazards that came due to it. Typically, modern processors use about 10 stage pipelines as a sweet spot for both handling hazards and improved performance. We therefore for simplicity reasons split the execute stage to make a 7 stage deep pipelined processor.



Although this is a very basic implementation, it was great to learn about how it works and how companies now and in the past implemented it.

Verification

We didn’t trust the CPU just because it simply looked right in a waveform; rather we built a structured verification flow around the /tb directory to catch bugs early and repeatedly for future test benches.

At the lowest level, we wrote unit tests for individual modules (for example, the register file and branch predictor). These are C++ Unit Testbenches wrapped around the Verilator model, which drive the module with carefully chosen inputs and use assertions to check the outputs.

On top of that, we have architectural tests that run short RISC-V assembly programs on the full core and then check the final architectural state. From the tb/ folder, run.sh compiles a set of test programs, runs them on the Verilated CPU, and verifies that key register a0 and memory locations match the expected results. These programs cover core instruction groups (ALU ops, loads/stores, branches, jumps) as well as the extra features we added such as multiply/divide.

Finally, we use a simple clock driven testbench run_clock.sh to run longer programs and inspect behaviour in GTKWave. This was less about strict pass/fail and more about sanity checking the pipeline and memory interactions over time. For example, watching branches, cache accesses, and stalls line up with what we expect.

Together, the unit tests, architectural tests, and waveform inspection give us reasonable confidence that the implementation is doing what the RISC-V spec says, not just what we hoped it would do.

Improvements

- Improve the clarity and usefulness of Git commit messages to make it easier to identify specific commits.
- Implement a superscalar architecture to allow more parallelism and overall performance.
- Develop a more accurate and efficient cache system to improve memory access behavior.
- Utilise multiple branches within our Git commits to improve feature isolation and streamline development.
- Create test benches prior to hardware design to ensure correctness and reduce redesign.
- Implement memory-mapped I/O for streamlined hardware–software interaction.

.