

An Implementation of A RV32IC Processor Core.*

Jedidiah B. Mortey, Soumya Biswas, Wei-Tang Wang[†]

June 18, 2025

Abstract

This project presents the design and implementation of a processor core based on the RV32 integer base instruction (RV32I) and extended to support compressed instructions (C). The processor includes a basic five-stage pipline with additional logic for data forwarding and hazard detection and a UART module for writing programs into the instruction memory. Backend flow was also carried out using Cadence Genus and Innovus for Synthesis and PnR respectively, with simulations run in QuestaSim and Power Analysis done with Primetime.

Keywords: *RV32I, UART, Cadence*

I. DESIGN

RISC-V is a relatively new instruction-set architecture (ISA) that was originally designed to support architecture research and education. Waterman et al. (2016). As the architecture was designed in a modular way, this project presents an implementation of the base 32-bit instruction set extended to support compressed instructions (RV32IC).

I.1. Module Summary

The following is a summary of all the modules that make up the CPU.

CPU: includes pipeline registers and all the other modules organized to execute instructions as loaded in program memory. A block diagram of the CPU is shown in figure I.

Fetch Stage: evaluates the program counter for the next clock cycle. The program counter from this module serves as the address of the instruction in the program memory to be executed next. Control hazards are also detected and handled in this stage and in the case of a control hazard, wrongly loaded instructions are squashed.

Decode stage: includes the register file, control module and the pc resolver. This stage of the pipeline provides operands and control signals required for executing instructions passed in from the fetch stage. Branch resolution is done here with the pc resolver.

Execute Stage: is the main compute stage. It includes the ALU and other logic to carry out arithmetic and logic operations. The results of this stage is passed to the mem stage and the forward mux to be used where dependencies exist.

Mem Stage: includes the data memory module and two truncation modules to reorganize data before storage on writes and after reading on loads. A block diagram for how this is done is shown in figure II.

Forward Mux: handles forwarding of data to resolve data hazards. Data from later stages (EX, MEM, WB) in the pipeline are forwarded where subsequent instructions are dependent on data from prior instructions before it is written back to the register file. This prevents stalls and improves pipeline efficiency

Hazard Detection Unit: includes logic to check for conditions of data hazards. It works with the forward mux to resolve write after read hazards and stalls the pipeline where for-

warding does not resolve the hazard.

Gshare Predictor: includes the branch prediction buffer and a global history predictor which helps to improve the accuracy of prediction.

UART: receives program instructions serially and repackages them to be stored in the instruction memory for execution. Inputs are "io rx" for receiving serial data an "io data valid" to enable recording of the serial data. It outputs an 8 bit data packet, four of which make up a line in program memory, and operates at baud rate of 921600. Its parent module is the CPU.

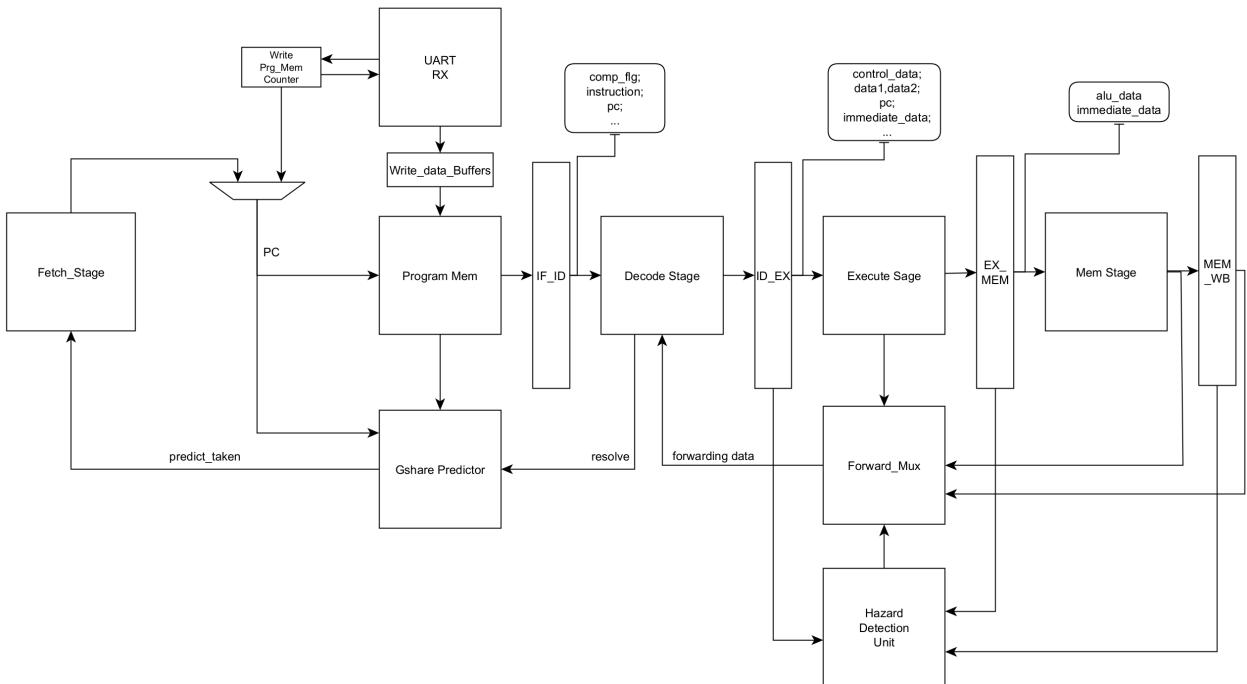


Figure I: Architectural diagram of the CPU.

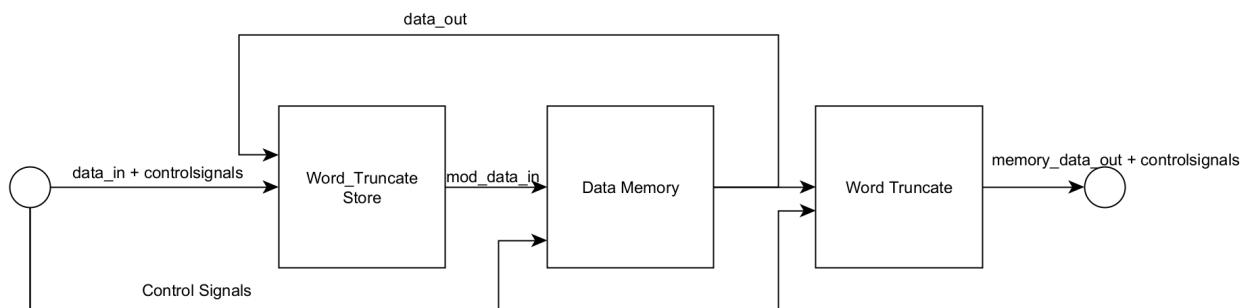


Figure II: Architectural diagram of the Mem stage.

I.2. Synthesis, Place and Route, and Simulation

After the CPU was confirmed to operate as designed, the RTL was synthesized with Cadence Genus, PnR was then done in Cadence Innovus. The final layout can be seen in fig A1. Table I below show the operating metrics of the final design. Post layout simulations were successful.

Area(um^2)	Frequency(MHz)	Power(mW)
2000x2000	100	31.7

Table I: Post-PnR characteristics of the CPU.

II. DESIGN RATIONALE

II.1. Branch Prediction

In our implementation of branch prediction, we chose to implement branch resolution in the Decode stage (both branch decision and target address calculation), compared to other groups that have implemented branch resolution in the Execute stage. The advantage of this approach is, single clock cycle penalty on a wrong prediction, instead of two cycles penalty. However, with this design choice, it comes with more hardware resource requirements. Due to the fact that an additional Arithmetic Logic Unit (ALU) is required in the Decode stage, in contrast to the other approach, which the ALU in the Execute stage can be utilized and no additional hardware costs would be needed.



Figure III: Simulation waveform of a branch instruction

As can be observed from the figure above:

- In the Fetch stage, instruction is decoded to determine whether it's a branch or not. If

it is, its prediction is simultaneously being read from the prediction buffer and program counter is changed. In this case, predicted to be "not taken"

- The following clock cycle, branch is resolved. In this case, the branch is resolved to be "taken".
- When such wrong prediction happens, insert a NOP to the Decode stage to flush the wrong instruction, and fetch from the correct PC address to continue execution.

Another point worth mentioning in our design is the size of the branch prediction buffer. We made it to be 256 addresses big, with the intention of a 1-to-1 mapping with program memory, which is 256 words big. This means that every branch instruction has a dedicated prediction buffer space, which will take full advantage of its own past history. In contrast, a smaller branch prediction buffer will act as a "cache", and mapping techniques of program memory has to be decided; This will naturally lead to several program memory addresses mapping to the same prediction buffer address, and the histories of these instructions will be mixed together. So in essence, it is simply a tradeoff between hardware resource costs and accuracy.

When it comes to the GSHARE predictor, benefits of this design choice are less intuitive. We are able to use more bits in the address index, 8 bits to be exact, to select from the Pattern History Table. Which means our group utilized more recorded instances of global history than the other groups to make our branch prediction. Whether this approach can consistently produce higher prediction accuracy is highly dependent on the programs being run, and further comprehensive tests are needed to arrive at some conclusion.

II.2. Compressed Instruction

In our implementation for decoding compressed instruction, we start with a simple check in the instruction fetch stage when we are reading the program memory by looking at the last two bits of the instruction(standard instructions always end with bits 11 while compressed instructions can have 00, 01 or 10 as the last two bits). Depending on the result, we increment the program counter by 4 for standard instructions and 2 for compressed instructions. Since our program memory size is 256x32, reading a compressed instruction can create an offset

(the program counter does not remain a multiple of 4). We deal with this by reading two instructions in the case of an offset and concatenating the top 16 bits of the first instruction and lower 16 bits of the second instruction in the memory.

In contrast to the design choices of other groups, we do not use any predecoding logic to convert our compressed instruction to a standard instruction type. Instead, we directly decode our compressed instruction. In the instruction decode stage, we receive a 32 bit instruction from the memory in either case. We recheck the condition for compressed instruction in our modified control block. If the condition is satisfied, we check the conditions for the different compressed instructions and assign the appropriate values to the control signals accordingly for further execution. Subsequently, we also need to decode the source and destination register addresses from the compressed instruction, as the assignment of these addresses is more complex compared to the standard instructions. We also formulated a new function to derive the immediate extension for compressed instruction types. We add a compflg signal to indicate whether an instruction is compressed or not which is used by some other blocks for branch prediction and ALU in the execute stage to check for some particular cases of compressed instructions. These design choices were made to execute some compressed jumping and branching instructions properly. In case of a standard instruction, the logic in the control block remains the same.

Avoiding predecoding logic in the fetch or decode stage enables us to have a shorter critical path in these stages, giving us more slack in our current implementation. Modifying our control block decreased our worst negative slack for setup from 4.3ns to 3.584ns while the slack for hold remained about the same. In both cases, the critical path is found to be correlated to the decode stage, so we can conclude that our modifications affected the critical path.

However, decoding a compressed instruction directly was harder to implement as it required changes in multiple blocks in our design. Using predecoding logic instead would have been easier to adapt into our pre-existing design for standard instructions even if it came with a trade off of a worse slack.

References

Waterman, A., Lee, Y., Patterson, D., and Asanović, K. (2016). The risc-v instruction set manual. Version 2.1.

Appendix

A. Additional figures

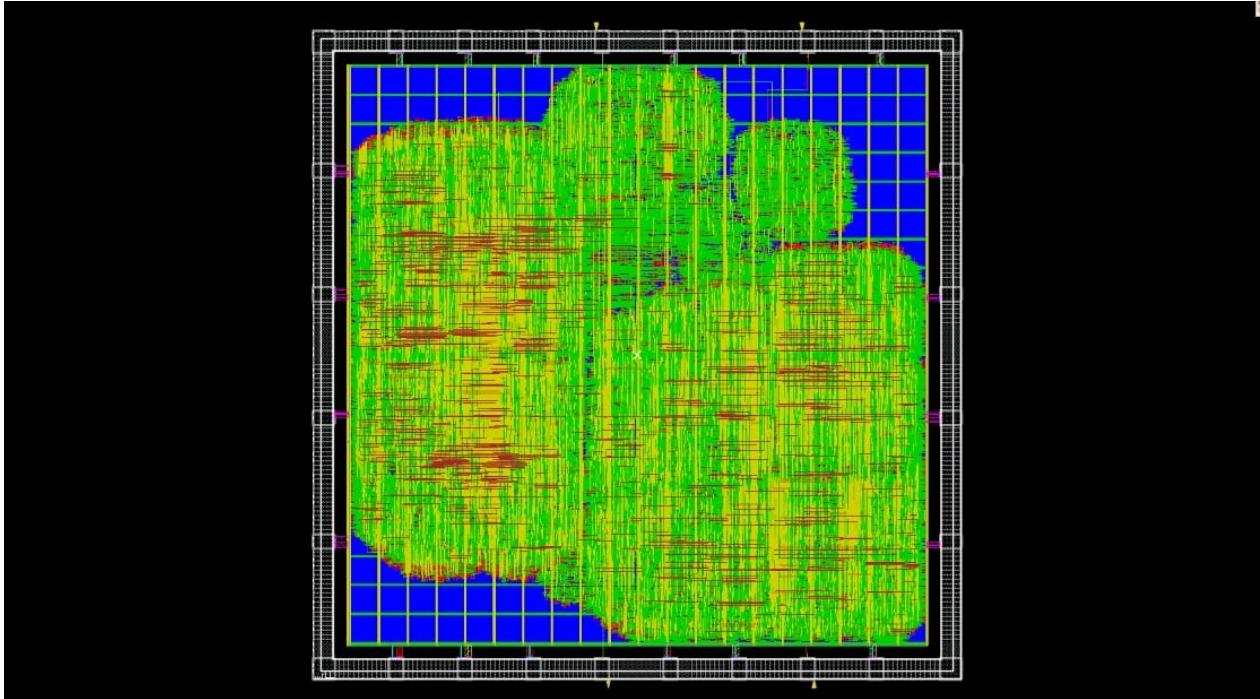


Figure A1: Final layout of the designed CPU.

```

Power-specific unit information :
  Voltage Units = 1 V
  Capacitance Units = 1 pF
  Time Units = 1 ns
  Dynamic Power Units = 1 W
  Leakage Power Units = 1 W

Attributes
-----
  i - Including register clock pin internal power
  u - User defined power group

Power Group      Internal Power   Switching Power   Leakage Power   Total Power  (%) Attrs
-----          (W)             (W)              (W)           (W)        (%)
clock network    0.0179 6.560e-03 5.225e-06  0.0241 (75.94%) i
register         2.058e-05 3.001e-06 1.212e-04 1.529e-04 ( 0.48%)
combinational    1.982e-04 1.475e-04 7.127e-03 7.464e-03 (23.52%)
sequential        0.0000 0.0000 0.0000 0.0000 ( 0.00%)
memory           0.0000 0.0000 0.0000 0.0000 ( 0.00%)
io pad            1.848e-05 1.369e-06 1.787e-15 1.985e-05 ( 0.06%)
black_box         0.0000 0.0000 0.0000 0.0000 ( 0.00%)

Net Switching Power = 6.712e-03 (21.15%)
Cell Internal Power = 0.0178 (56.01%)
Cell Leakage Power = 7.253e-03 (22.85%)
-----
Total Power      = 0.0317 (100.00%)

```

Figure A2: Power report.

Design Timing Summary

Setup	Hold	Pulse Width
Worst Negative Slack (WNS): 3.584 ns	Worst Hold Slack (WHS): 0.044 ns	Worst Pulse Width Slack (WPWS): 3.000 ns
Total Negative Slack (TNS): 0.000 ns	Total Hold Slack (THS): 0.000 ns	Total Pulse Width Negative Slack (TPWS): 0.000 ns
Number of Failing Endpoints: 0	Number of Failing Endpoints: 0	Number of Failing Endpoints: 0
Total Number of Endpoints: 81482	Total Number of Endpoints: 81466	Total Number of Endpoints: 48765

All user specified timing constraints are met.

Figure A3: FPGA timing report w/ 50MHz clock

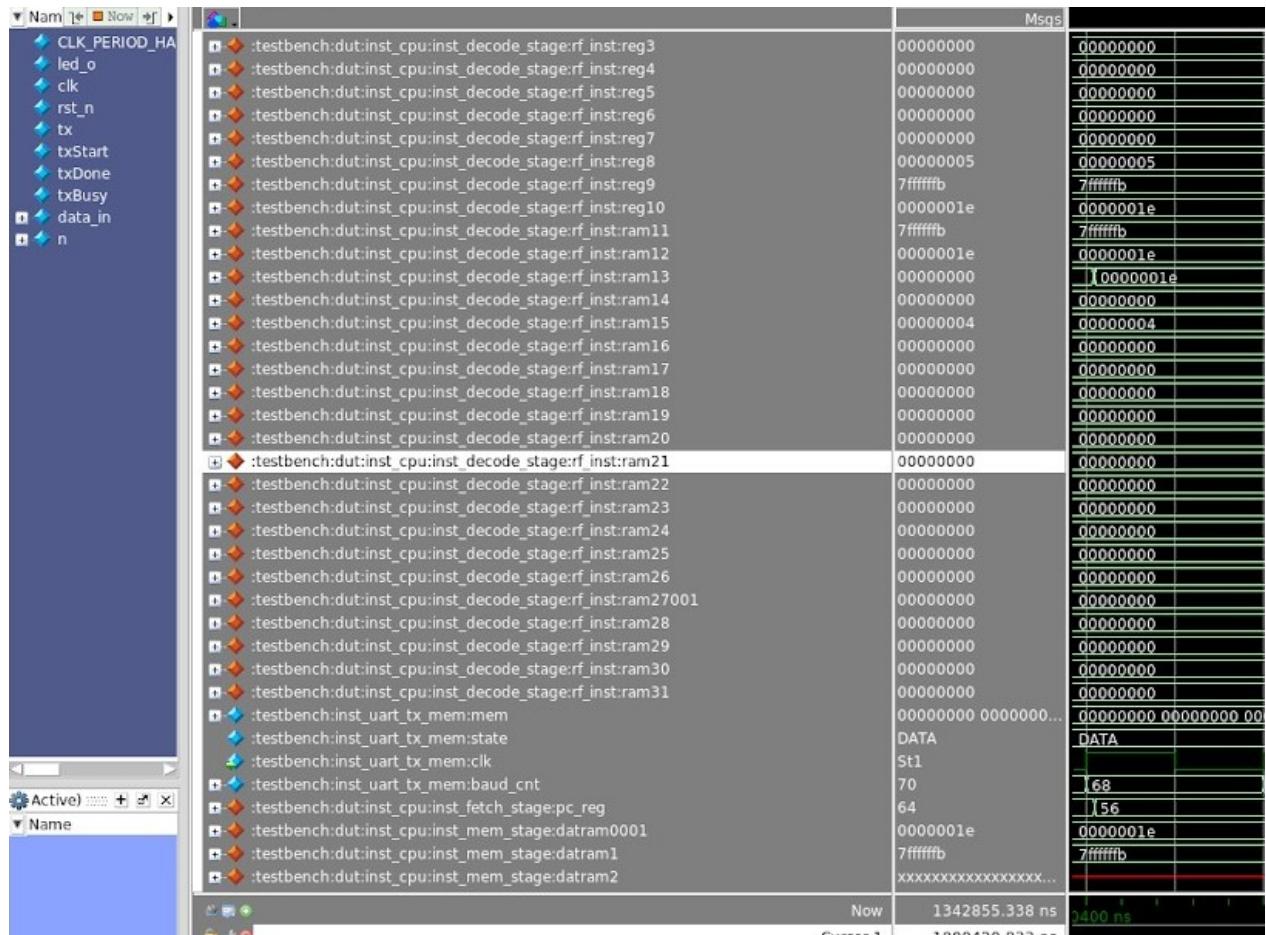


Figure A4: Post-layout simulation results of test7.S

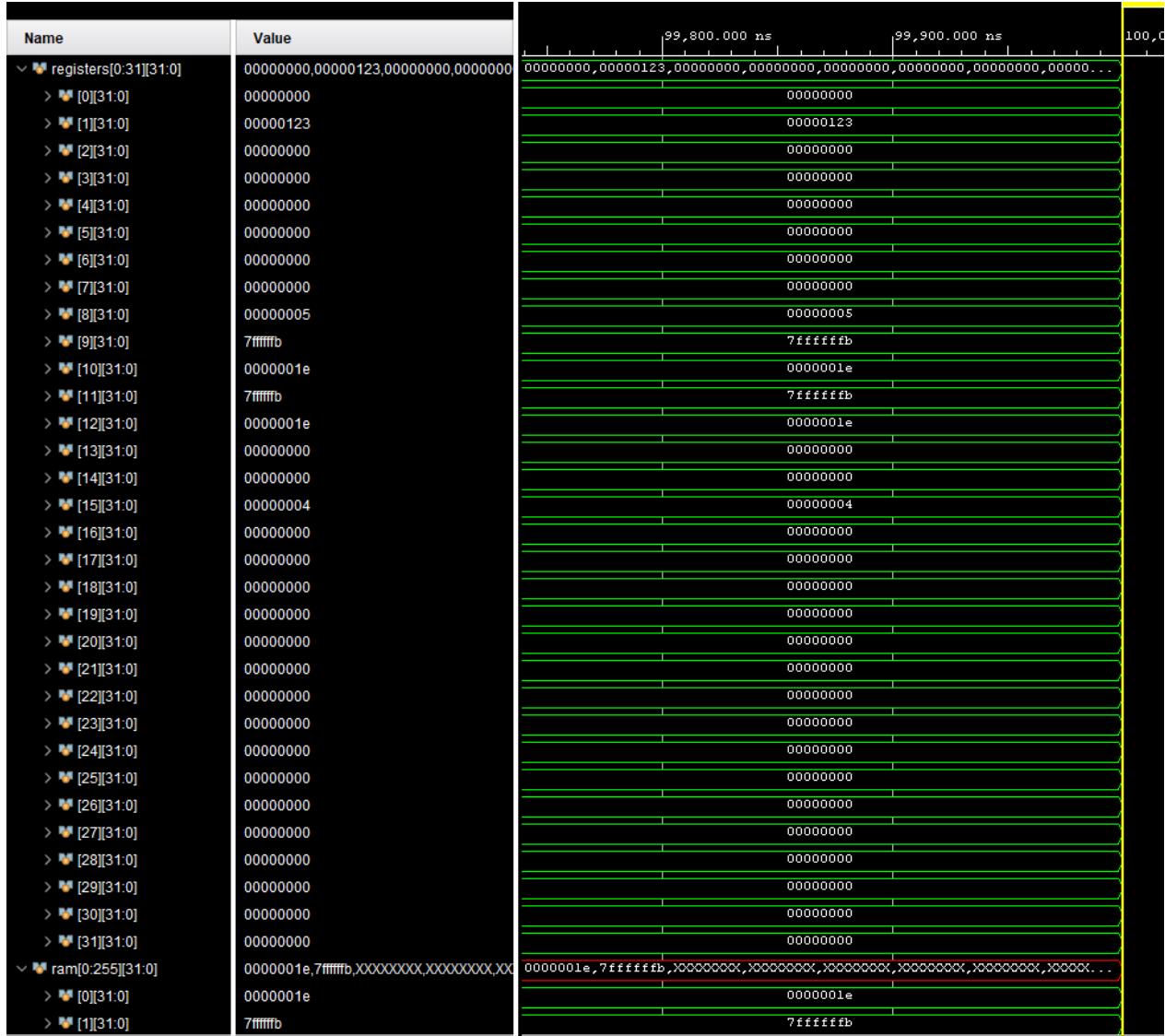


Figure A5: RTL simulation results of test7.S