# Documentation for RISC-V CPU Assignment

董海辰 518030910417

2020 年 1 月 4 日

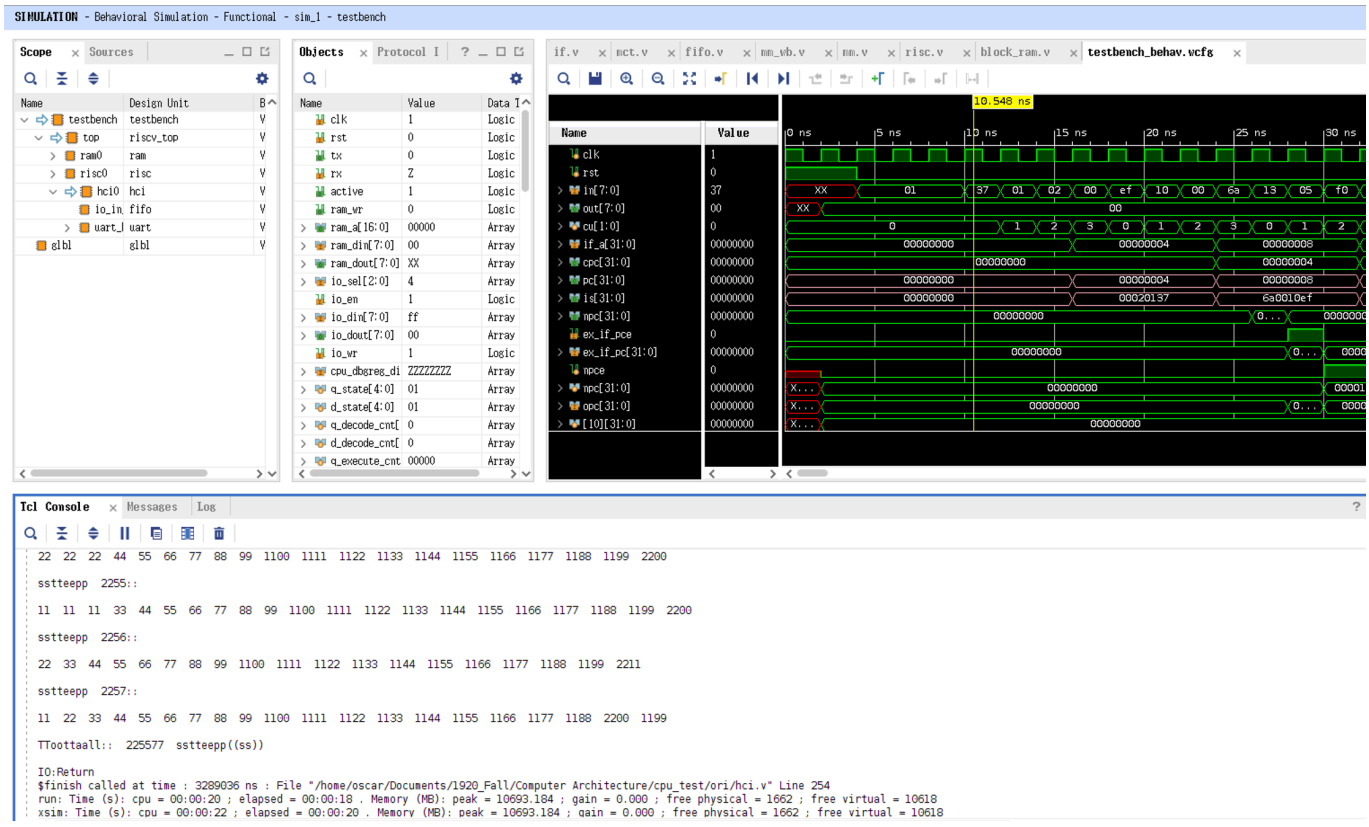
## 1 Overall

The processor implemented for this assignment is 5-stage pipelined, with instruction cache and branch target buffer.

## 2 Results

### 2.1 Simulation results

The final version passed given tests all but the ones with unacceptable simulation time (some with manual code editing for input, and with sleeps commented).



| Testcase | After ICache | After BTB | CPI |
|---|---|---|---|
| expr | 28342 | 20936 | 1.325 |
| gcd | 5074 | 4482 | 1.621 |
| pi (with c=280) | 3858914 | 2766346 | 1.322 |
| qsort (with only 1 output) | 6613410 | 5298224 | 2.108 |
| bulgarian | 4080246 | 3289036 | 1.929 |
| basicopt1 | | 1759930 | 1.650 |
| hanoi | | 629190 | 2.480 |
| lvalue2 | | 142 | 1.315 |
| magic | | 3332266 | 2.648 |
| manyarguments | | 234 | 1.393 |
| multiarray | | 13650 | 1.684 |
| queens | | 3019462 | 2.653 |
| statement_test | | 7012 | 1.816 |
| superloop | | 1232050 | 1.226 |
| tak (with a=14) | | 732816 | 2.907 |

## 2.2 FPGA Testing

| Testcase | 100 MHz | 200 MHz |
|---|---|---|
| pi | 1.676014 | 0.846720 |
| qsort | 13.516065 | 6.773527 |
| bulgarian | 3.320367 | 1.694799 |
| heart | 449.361421 | 270.917225 |

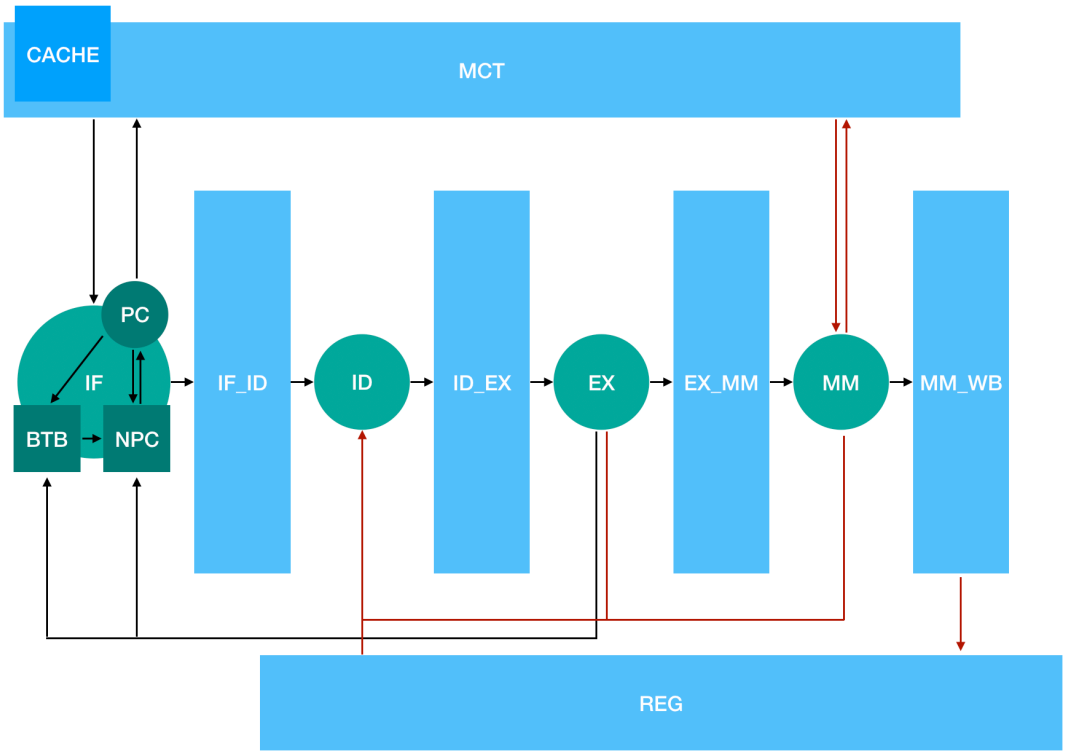
```
(base) ➜  riscv git:(master) ✗ sudo ./run_test_fpga.sh pi
RAM size: 130c
13 06 05 00 13 05 00 00 93 f6 15 00 63 84 06 00
33 05 c5 00 93 d5 15 00 13 16 16 00 e3 96 05 fe
67 80 00 00 63 40 05 06 63 c6 05 06 13 86 05 00
93 05 05 00 13 05 f0 ff 63 0c 06 02 93 06 10 00
input file not found, skipping
initialized UART port on: /dev/ttyUSB1
55 41 52 54
55 41 52 54
uploading RAM: 130c blks:5
blk:0 ofs:0000
blk:1 ofs:0400
blk:2 ofs:0800
blk:3 ofs:0c00
blk:4 ofs:1000
RAM uploaded
Enter r to run, q to quit, p to get cpu PC(demo)
r
CPU start
31415926535897932384626433832795288419716939937510582097494459230781640628620899862803482534211706798214808651328236647093844695558223172
35940812848111745028412701938521155596446229489549303819644288109756659334461284756482337867831652712019914564856692346348610454326648213
93607260249141273724587066063155881748815209209628292549171536436789259036113305305488204665213841469519415116094330572736575959195309218
61738193261179310511854807446237996274956735188575272489122793818301194912983367336244065664386021394946395224737190702179869437027705392
71762931767523846748184676694513200056812714526356082778577134275778960917363717872146844090122495343014654958537105792279689258923542019
95611212902196864344181598136297747713099605187072113499999983729780499510597317328160963185

CPU returned with running time: 0.846720
```

# 3 Structure

As normal 5-stage pipelined processors, this design mainly consists of 6 modules:

- Memory Controller
- Instruction Fetch
- Instruction Decode
- Execution
- Memory Access
- Write Back(with registers)

Also, there are modules used for connecting and control flows.




# 4 Detailed situations

## 4.1 Word fetching

Our memory is has only one port with one byte each cycle, so generally reading a word costs 6 cycles, because it takes one more cycle to handle and passing the initial address, and one more to receive the fourth byte.

I improved this into a 5-cycle word-reading by dropping the last cycle, and let IF or MEM get the last byte directly by the momery module. A 'cache-hit' signal is passed to indicate whether to use the byte from memory because MCT gives the whole word when cache hits.

Further more, technically speaking, MCT uses a static branch prediction strategy. As a result, it consumes only 4 cycles when fetching instruction in order, i.e., the new pc passed by IF is exactly 4 more than the old one.

## 4.2 Hazards

- Structure hazards The only structure hazard that will occur is for the memory port. When an instruction requires memory access, MCT will finish its current job and then do the access, and during this it will not accept new access requirements.
  Data access is always in priority than instruction access.
- Data hazards
  This is easy to handle in a design with single execution cycle. Trivial data forwarding can solve this problem. One spacial case is when using the value just loaded. I guarantee the correctness by adding 3 bubbles after a LOAD instruction so that the IF stage of the next instruction will be completed after this instruction entered the MEM stage, and will be stalled before ID until the memory operation completed.
  And MCT will not fetch the next instruction until the current data memory operation completed so that the IF of the third instruction will be the same cycle when the second one entered ID.

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| LOAD | IF | ID | EX | MM | - | - | - | WB | | |
| | | | | | IF | - | - | ID | EX | |
| | | | | | | | | IF | ID | |
| | | | | | | | | | IF | |

- Control hazards
  When the EX stage find the prediction that IF passed to it wrong, it will send an invalid signal to both IF and ID_EX.
  IF will mark the current instruction being fetched, because it is the last wrong instruction, and pass the justified pc to MCT after this fetch completed. In my design, I simply make the inst[1:0] be '10', equal to none of the types.
  ID_EX will stop passing wrong instructions to EX until it get the mark made by IF, indicating wrong instructions are all flushed.

## 4.3 Branch prediction

I implemented a branch target buffer direct-mapped with 32 entries. IF will treat pc+4 as the default static branch prediction result. And when EX passes the message that the prediction result is wrong, IF will inserted the correct prediction into the buffer.

This is a simple implementation, but works.

# 5 Summary and comments

- Sometimes bugs never occur until it runs fast, for example, with caches and predictions. I even found some extremely naive bugs like forwarding data writing to register 0 on the very last days of my work.
- I became confused when we run it on board with the negative time slack of -5ns. This shows the unreliability of judging it by CPI and the estimated maximal frequency. So what is a good design? The one with low CPI? Or the one that simply runs fast? idk:)