Digital System Design Final Project Report

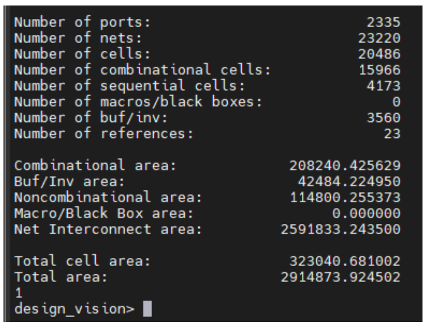
RISC-V G5

B07901187 宋家齊 B07505026 田昀曜 B07611033 林書宇

Baseline

A. Pass Baseline Gate Level Simulation

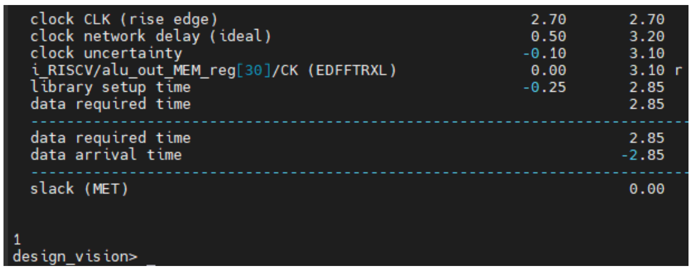
- sdc cycle time = 2.7ns, slack = 0
- area = 323040 um^2



```
Number of ports:                          2335
Number of nets:                          23220
Number of cells:                         20486
Number of combinational cells:           15966
Number of sequential cells:               4173
Number of macros/black boxes:                0
Number of buf/inv:                        3560
Number of references:                       23

Combinational area:              208240.425629
Buf/Inv area:                     42484.224950
Noncombinational area:           114800.255373
Macro/Black Box area:                 0.000000
Net Interconnect area:          2591833.243500

Total cell area:                 323040.681002
Total area:                     2914873.924502
1
design_vision>
```

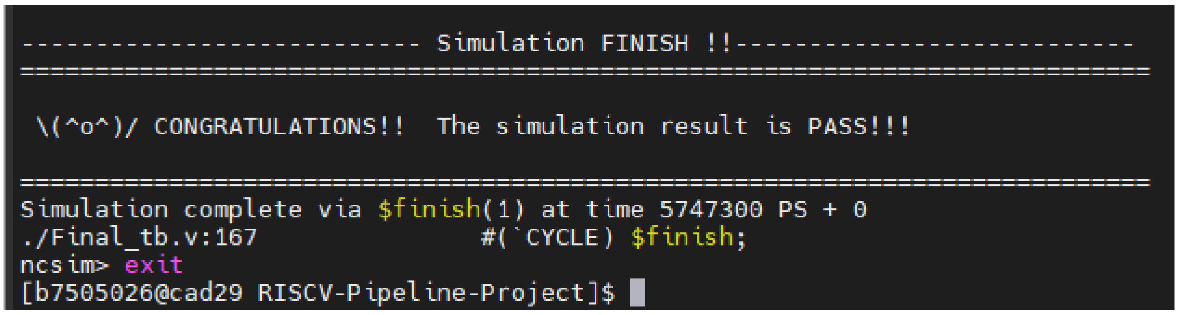


```
  clock CLK (rise edge)                                  2.70       2.70
  clock network delay (ideal)                            0.50       3.20
  clock uncertainty                                     -0.10       3.10
  i_RISCV/alu_out_MEM_reg[30]/CK (EDFFTRXL)              0.00       3.10 r
  library setup time                                    -0.25       2.85
  data required time                                                2.85
  ------------------------------------------------------------------------
  data required time                                                2.85
  data arrival time                                                -2.85
  ------------------------------------------------------------------------
  slack (MET)                                                       0.00


1
design_vision>
```

B. AT

- AT = 323040*5747 = 1,856,510,880 um^2*ns ≈ 1.85*10^9
- tb cycle time = 2.6ns
- 2210 cycles (haszard)



```
--------------------------- Simulation FINISH !!---------------------------
=============================================================================

 \(^o^)/ CONGRATULATIONS!!  The simulation result is PASS!!!

=============================================================================
Simulation complete via $finish(1) at time 5747300 PS + 0
./Final_tb.v:167               #(`CYCLE) $finish;
ncsim> exit
[b7505026@cad29 RISCV-Pipeline-Project]$
```

C. Overview of Baseline circuit

- Icache
  - 2way, 32word.
  - Read only. Without proc_write to cache, write-back to memory, no dirty bit.
- Dcache
  - 2way, 32word.
  - Without write-back to memory, no dirty bit.

D. General design

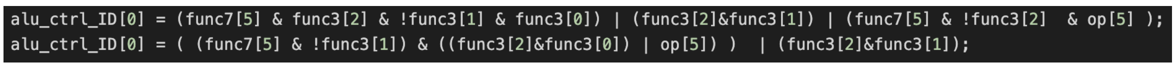
- 5 stages pipeline, IF, IF, EX, MEM and WB.
- With forwarding units, a load-use hazard detection unit.
- Branch and jump determination at ID stage.
  - If the *rs* register at EX stage is needed for branch or jump, we have to let this *rs_data* be propagated, and forward it from MEM stage. Otherwise, connecting it from alu_output to PC adder will make it a critical path. There's no way to handle this problem, we will waste two cycles if encountering this situation with branch taken or jump.

E. **Design technique** (not all are used)

- Block input
  1. Icache_ren: Since reset ends at negedge, we can't read Icache at the next posedge, or we'll have only a half cycle to prepare the data. Therefore, we delay the Icache_ren for a cycle.
  2. mem_ready, mem_rdata: Since mem_ready is set to high at negedge, mem_rdata is sent at the same time. We have to delay mem_ready in order to leave a whole cycle time for mem_rdata to be stored at cache.
- Memory prefetch
  - Speed up by 44 cycles. Our approach may not be perfect, because we didn't specify PC+4 to fetch the data. At *Compare_tag state,* we set mem_read to high, meaning that the memory is always preparing for the data. After some cycles, if instruction needed at PC now is missing, we enter the *Allocate state.* Since we have set mem_read to high before, we don't need to wait five cycles to fetch the instruction. To make an analogy, it's like someone is going to the library to borrow a book. But he is told which one to borrow on the way to the library.
- Dcache reset
  - Speed up by 24 cycles. Instead of resetting the Dcache data to all zeros, we reset them to valid, and all-zero data. Since D_mem are all zeros, if we need the address whose tags happen to be zero, we don't need to allocate this data from memory.
- K-map reduce
  1. We use 'type' as FSM to determine the instruction type, and alu_ctrl to decide the alu operation. We don't need to list type and alo_ctrl in order; instead, we can balance the bit 1 in each column.
  2. The logic level does matter. The following pass different sdc cycle times.

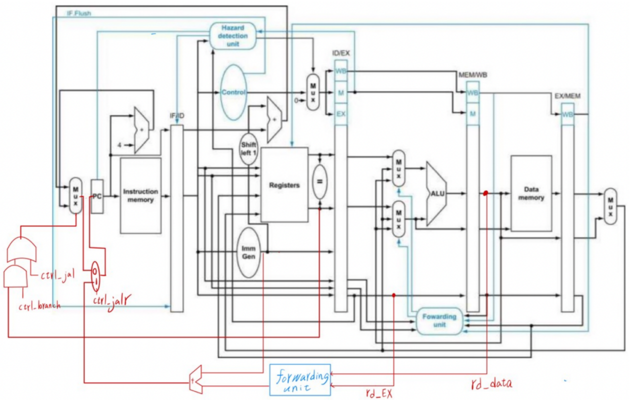
```
alu_ctrl
 3'd0   0000 +
 3'd1   0001 -
 3'd7   0111 &
 3'd3   0011 |
 3'd2   0010 ^
 3'd4   0100 <
 3'd8   1000 <<
 3'd14  1110 >>
 3'd15  1111 >>>
```

```
Type
3'd0 000 R
3'd2 010 I
3'd1 001 S
3'd3 011 B
3'd6 110 J
```



```
alu_ctrl_ID[0] = (func7[5] & func3[2] & !func3[1] & func3[0]) | (func3[2]&func3[1]) | (func7[5] & !func3[2]  & op[5] );
alu_ctrl_ID[0] = ( (func7[5] & !func3[1]) & ((func3[2]&func3[0]) | op[5]) )  | (func3[2]&func3[1]);
```

- Write register
  1. Sequential circuit: By intuition, we use sequential circuit to store the value into registers. But this would require several muxes to be placed at the MEM stage, which may become a critical path.
  2. Combinational circuit: We place this mux at the WB stage, just like the circuit on the textbook. In this way, we have to add registers for saving to avoid latch. These two approaches should have the same area; however, the combinational one is larger.



F. Review

- We spent about two days passing the Baseline. Basically, we just simply build the circuit on the textbook. There are two problems that took us most of the time. First, the forwarding unit on the textbook is not enough; since branch and jump is determined at the ID stage, we have to add one more forwarding unit for them. Second, we can’t forward the data from the end of the stage, for example alu_output, which will make it a critical path.

G. Remark

- In the above version. We blocked only mem_ready and mem_rdata and didn’t apply techniques of memory prefetch and Dcache reset.

H. Improvement after report

- After referencing others' presentations, we found that there’s some techniques we had used before, but something went wrong then. However, that technique works on the current version. It's only slightly different from the above one.
- AT = 312026*5635 = 1,758,266,510 um^2*ns ≈ 1.75*10^9
- sdc cycle time = 2.6 ns
- tb cycle time = 2.6 ns
- run time = 2067 cycles

Compression

A. AT

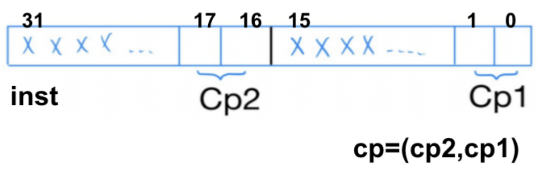
- A*T = (313457 - 294020)*1728.3 = 541,747,733 um^2*ns ≈ 5.8*10^8
- area = 294020 um^2
- sdc cycle time = 3ns
- tb cycle time = 4.2ns

B. Advantage and disadvantage of RVC

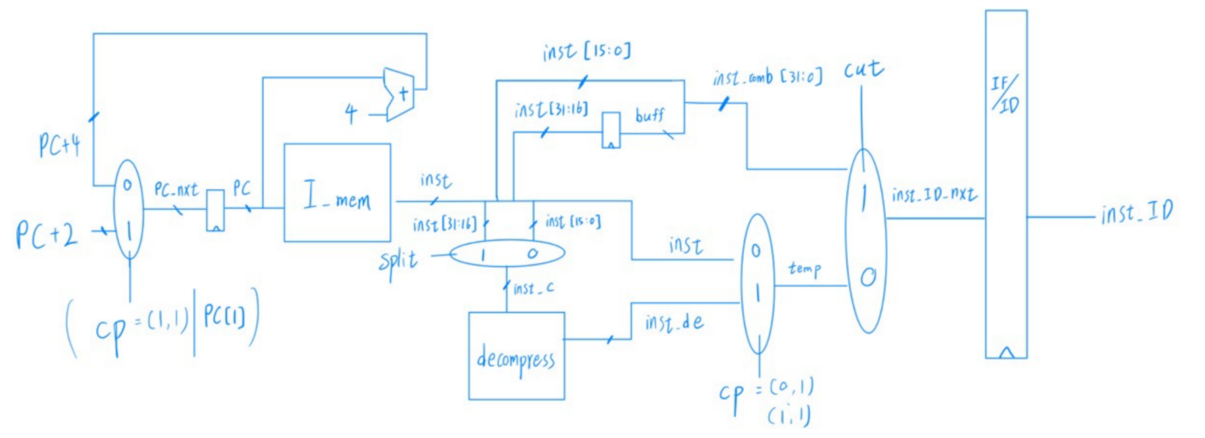
1. Less run time: RVC uses less run time because it has less instruction length, allowing Icache to store more RVC instructions than store 32 bit ones. This implies that RVC will have less miss rate, decreasing run time.
2. More area usage: A decompressor used to decompress a RVC instruction will have some area cost.
3. Complexity: Using a decoder to implement RVC will increase complexity, every stage has to make some modification according to it.

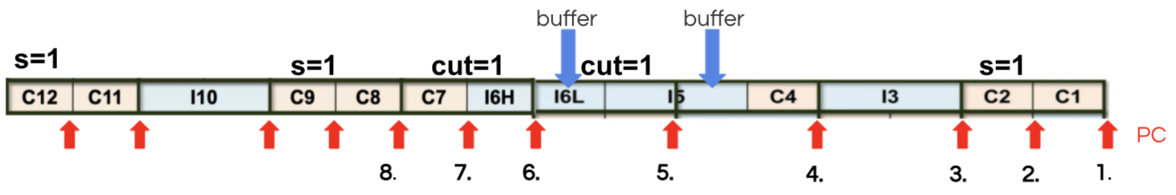
C. Design

- Preliminary
  1. CP (compare): Given a 32 bits instruction and determine the {16,17} and {1,0} bits. If {1,0} are ‘11’, then this is a 32 bits instruction, assigning cp=0; otherwise, this is a 16 bits instruction, assigning cp=1. If cp=0, then cp of the next 16 bits must be X, don’t care, because it may be immediate of 32 bits instruction.
  2. cut: When cp2=0, we need to buffer the upper 16 bits and set cut_nxt = 1. If cut =1, we have to take the buffered 16 bits and concatenate it with the current lower 16 bits.
  3. split: Read upper 16 bits RVC.



- PC increment and address alignment
  - PC never stalls, no cycle wasted.
  - Always PC+4, except when cp=(1,1) or (1,x) & cut or PC[1].
- Example (figure below)
  1. cp=(1,1), read lower 16 bits, PC+2.
  2. cp=(1,1) & s=1, read upper 16 bits, PC+2.
  3. cp=(x,0), read 32 bits instruction, PC+4.
  4. cp=(0,1), read lower 16 bits, buffer upper 16 bits, PC+4.
  5. cp=(0,x) & cut=1, concatenate buffered 16 bits with lower 16 bits. read this 32 bits instruction. PC+2.
  6. cp=(0,x) & cut=1, concatenate buffered 16 bits with lower 16 bits. read this 32 bits instruction. PC+2.
  7. cp=(0,x), read upper 16 bits, PC+2.
  8. cp=(1,1), read lower 16 bits, PC+2.





- Decompressor
  - Simply just map the 16 instructions to 32 bit one. Notice that the rd' has two positions, while rs1', rs'2 has fixed positions.

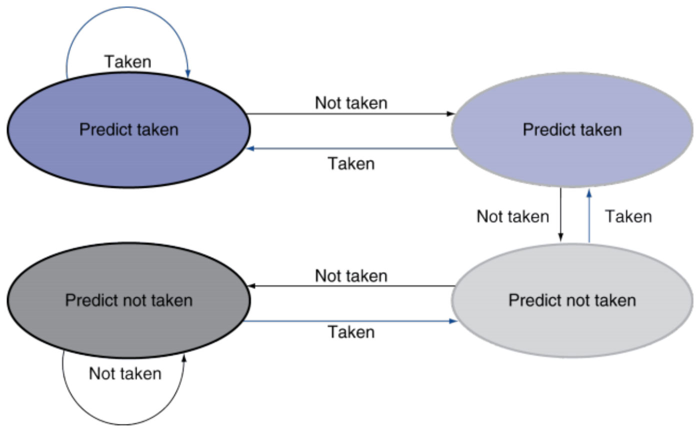
D. Review

- Implementing RVC is kind of interesting. The PC increment and address alignment issue is worthy of thinking. In the online course, TA wants us to handle this problem, and it brings me a sense of accomplishment after solving this. The decoding part is tedious, however. I think that no further improvement can be done on this extension. Finally, due to its extension and being open source, RISC-V is taking over the world now. It's a good chance to have an initial look at this powerful ISA. Thanks to the Professor and TA for giving us the chance to implement this project.

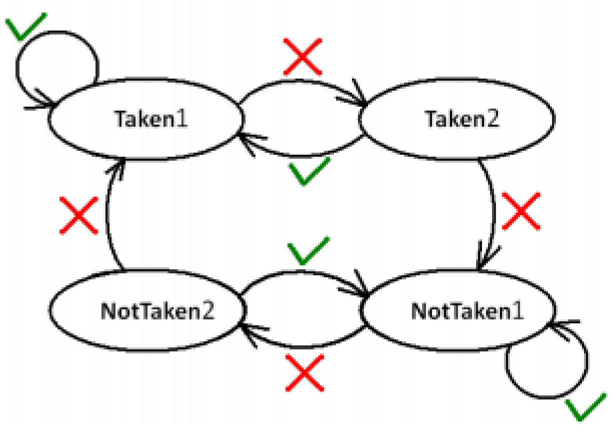
Branch Prediction

A. Predictors

- 2-bit predictor v1
  - FSM:



  - Features:
    It'll take two consecutive wrong predictions to transition from strongly T/NT state to weakly NT/T state, and take one wrong prediction to transition from weakly t/nt state to weakly nt/t state. There's a serious problem with this predictor. If the branch pattern is T→NT→T→NT… we'll transition between weakly T and weakly NT state, which means every prediction is wrong, wasting many cycles.
- 2-bit predictor v2
  - FSM



  - Features: It'll take two consecutive wrong predictions to transition from T/NT state to NT/T state. No strong or weak state like predictor v1.

B. Predictors

- Mechanism:
  - Record previous two branch history as follow:

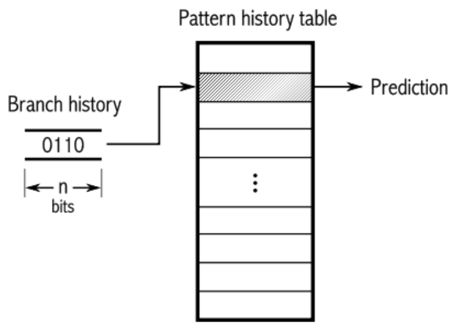
    T: taken. NT: not taken

    NT / NT (i.e. 00)

    NT / T (i.e. 01)

    T / NT (i.e. 10)

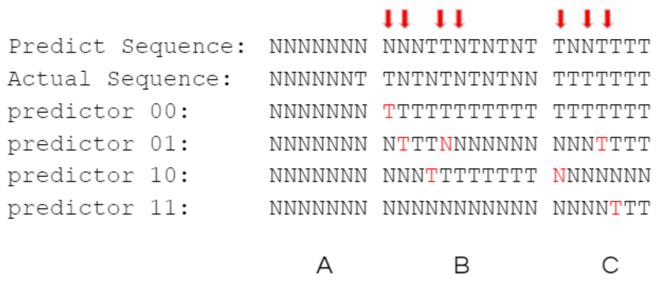
    T / T (i.e. 11)



  - The predictors we choose are 1-bit predictor and 2-bit predictor v2
- Performance of Pattern-History-Table:

  For example, Pattern-History-Table with 1-bit predictor:

  (N stands for branch not taken, T stands for branch taken)



```
Predict Sequence:  NNNNNNN NNNTTNTNTNT TNNTTTT
Actual Sequence:   NNNNNNT TNTNTNTNTNN TTTTTTT
predictor 00:      NNNNNNN TTTTTTTTTTT TTTTTTT
predictor 01:      NNNNNNN NTTTNNNNNNN NNNTTTT
predictor 10:      NNNNNNN NNNTTTTTTTT NNNNNNN
predictor 11:      NNNNNNN NNNNNNNNNNN NNNNTTT

                      A          B         C
```

  A,B,C stands for three different types of sequences in the test pattern; the actual sequence of the first appearance of the first arrow is “NNT” (NN corresponds to predictor 00). However, the prediction predictor00 will take at that point is branch not taken, it’ll then update to branch taken.

C. Analysis of test pattern:

- There are a total of 3 parts of sequence, respectively a, b, c.
  a. All not taken.
  b. NT / T/ NT / T / NT…
  c. All taken.
- Five cases of comparisons:
  1. All not taken
     a. All correct
     b. Predict wrong every two branch instructions.
     c. All wrong.
  2. 2-bit predictor v1
     a. All correct
     b. All wrong.
     c. Predict wrong for the first three branch instructions, all correct thereafter.

3. 2-bit predictor v2
    a. All correct
    b. Predict wrong every two branch instructions.
    c. Predict wrong for the first three branch instructions, all correct thereafter.
4. BHT 1-bit
    a. All correct
    b. Predict wrong for the first three branch instructions, all correct thereafter.
    c. Predict wrong for the first three branch instructions, all correct thereafter.
5. BHT 2-bit(v2)
    a. All correct
    b. Predict wrong for the first four branch instructions, all correct thereafter.
    c. Predict wrong for the first four branch instructions, all correct thereafter.

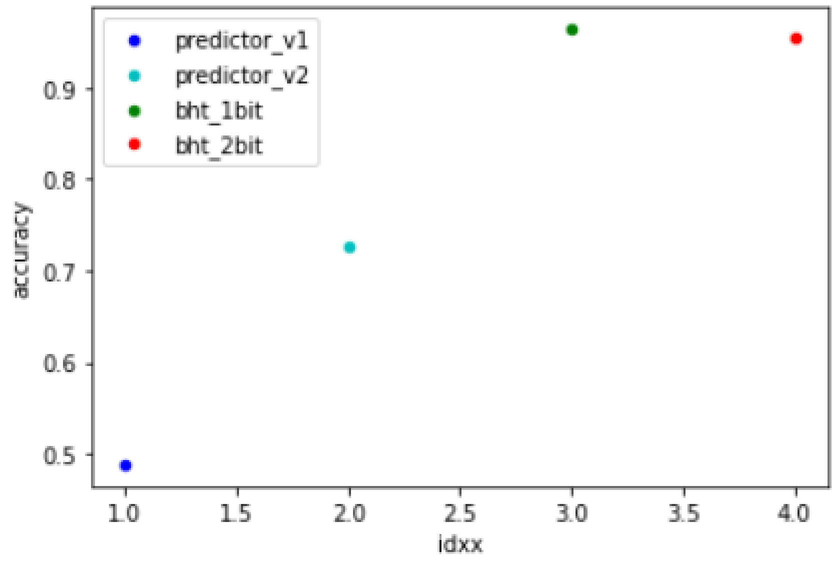
D. Analysis of Branch predictor on Baseline (original test case):

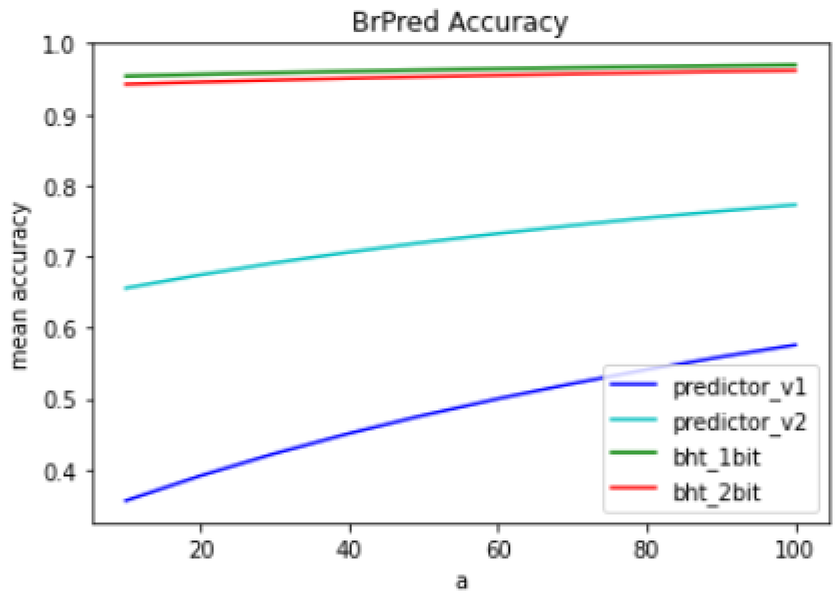
1. Without prediction: 2001 cycles
2. 2-bit predictor v1: 2015 cycles
3. 2-bit predictor v2: 1985 cycles
4. BHT 1-bit predictor: 1907 cycles
5. BHT 2-bit predictor v2: 1907 cycles

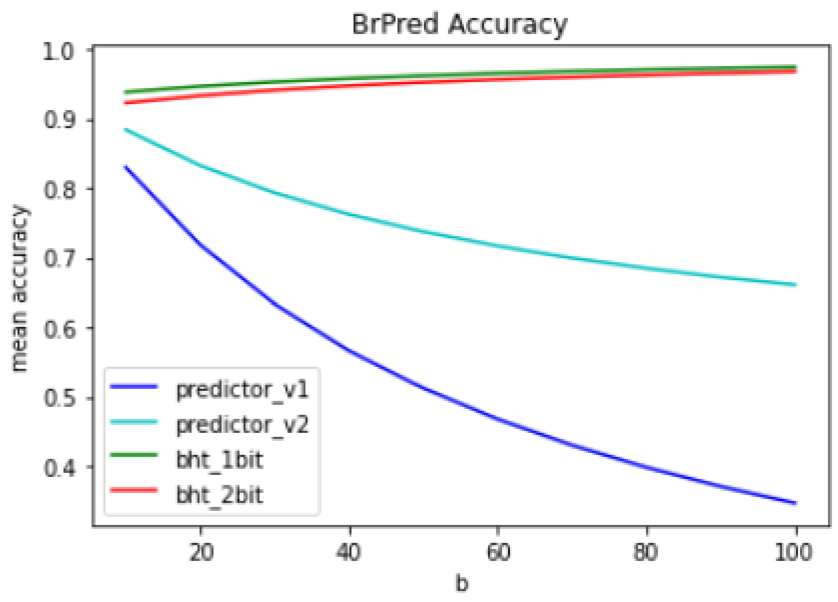
E. Comparison of different types of test pattern

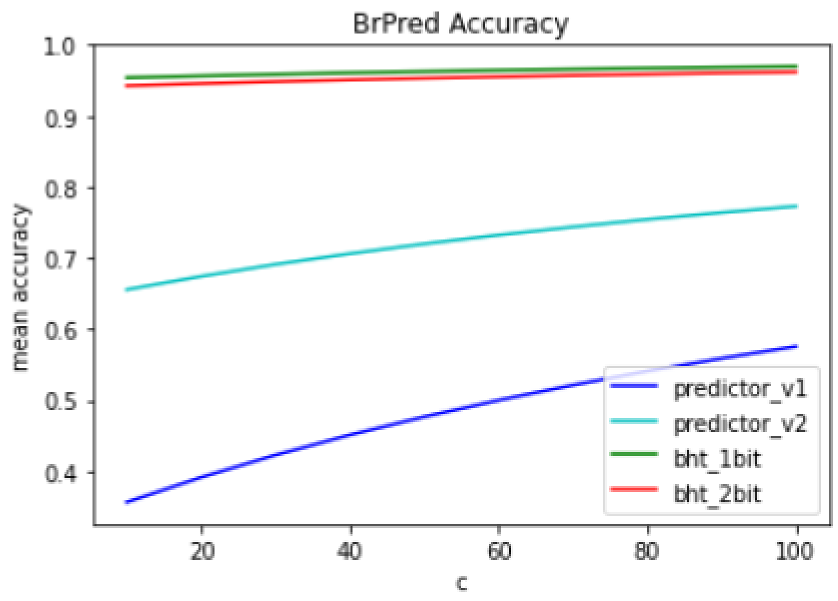
i. Settings: different types of predictors, three different branch sequences and different combination of numbers of each sequence
ii. Three parameters are set to values from 10, 20, 30, …, 100. And use the combinations to generate different testbenches.
iii. There are a total of 1000 testbenches for each predictors.
iv. "a" stands for the first branching sequence, which is all not taken
"b" stands for the second branching sequence, which is NT / T/ NT / T / NT…
"c" stands for the third branching sequence, which is all taken.
v. Inference: With respect to three different branch sequences, the predictor will have different performance, which is:
For a: the performance will grow as the number of a increases
For b: the performance will degrade as the number of b increases
For c: the performance will grow as the number of c increases

total average accuracy over 1000 testbenches:






BrPred Accuracy
mean accuracy
b
predictor_v1
predictor_v2
bht_1bit
bht_2bit


BrPred Accuracy
mean accuracy
c
predictor_v1
predictor_v2
bht_1bit
bht_2bit

2 Level Cache

B. L2 size comparizon

a. Setting

  i. Separate L1 cache to ICACHE and DCACHE
  ii. Both L1 ICACHE and DCACHE use 2-way
  iii. The size of each L1 cache is 32 words.
  iv. Compare L2 size: 64 words, 128 words, 256 words
  v. 10 possible value for nb
  vi. 7 possible value for incre
  vii. 70 combinations of testbench for each L2 size
  viii. There're total of 210 testbenches.

values of parameters in testbench: nb and incre

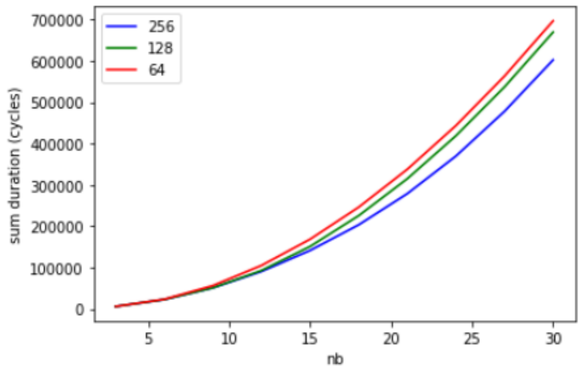
nb	3	6	9	12	15	18	21	24	27	30
incre	2	4	6	8	10	12	14			

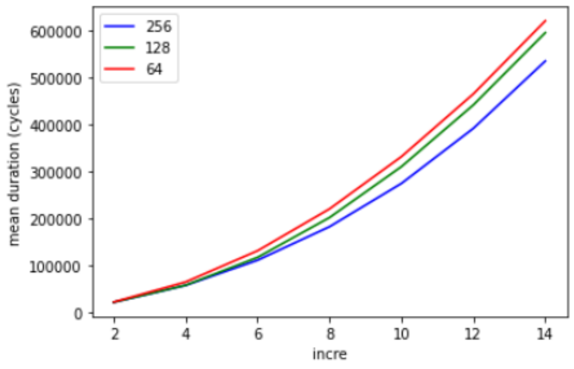
b. parameter meanings

  i. "nb"
    1. It means the index of the Fibonacci Series, the CHIP needs to calculate the target Fibonacci number.
  ii. "incre"
    1. It means how many numbers will be generated for each Fibonacci number in the series.
    2. The start value is the origin Fibonacci number, it will increment 1 for each step.
    3. The size of final array will be nb*(incre+1).
    4. In testbench, it will firstly calculate the target Fibonacci Number and then do the Bubble-Sort on the generated array.

c. compare duration cycles in different L2 size settings.

  i. results



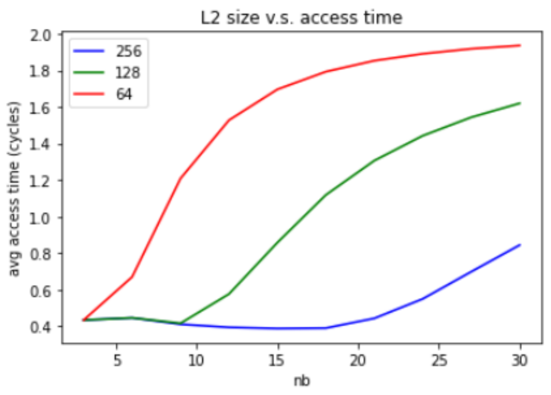


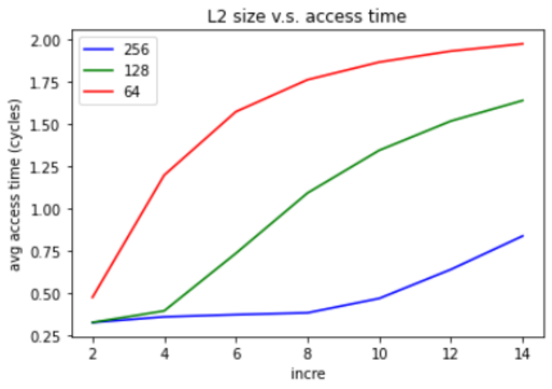
ii. inference

1. Duration grows up as nb and incre increases due to the increasing operations and the increasing accesses of cache.
2. 64 words L2 cache has the highest duration due to the smaller size. It easily has dirty access and then needs to write memory first then do the allocation. So the total stalls are larger than 128 words and 256 words L2 cache.
3. The difference of duration becomes larger due to the increasing nb or incre will have more operations.

d. compare average access time (cycles) in different L2 size settings.

i. results





ii. inference

1. Average access time grows up as nb and incre increases due to the increasing operations and the increasing accesses of cache.
2. The difference of access time is larger than the duration. Since the duration includes a lot of cycles of other instructions that are not related to memory access.
3. 256 words L2 cache has the smallest average access time due to the large capacity in the cache. But the trade off is the area of the design.

C. Cache type comparison

a. setting

i. The size of each L1 cache is 32 words.
ii. Fix the size of L2 cache to 128 words.
iii. L1 Icache and Dcache share the same L2.
iv. 10 possible value for nb
v. 7 possible value for incre
vi. We tried 5 different type settings.

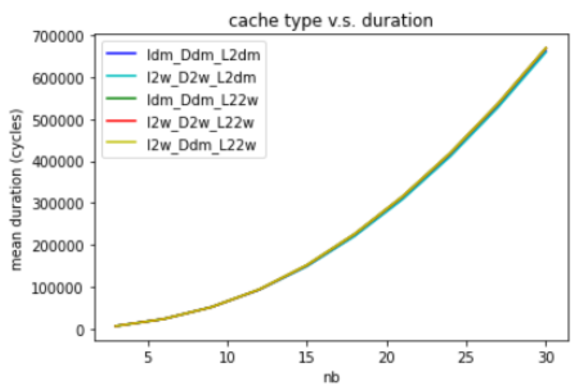
	1	2	3	4	5
I cache	2-way	dm	dm	2-way	2-way
D cache	2-way	dm	dm	2-way	dm
L2 cache	2-way	dm	2-way	dm	2-way

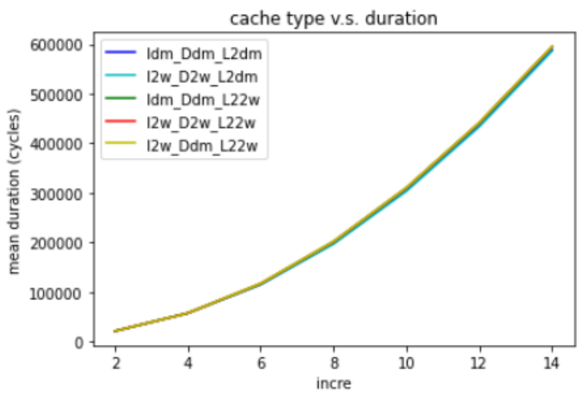
vii. 70 combinations of testbench for each type setting

viii. There're total of 350 testbenches.

b. compare duration (cycles) in different L2 size settings.

i. results



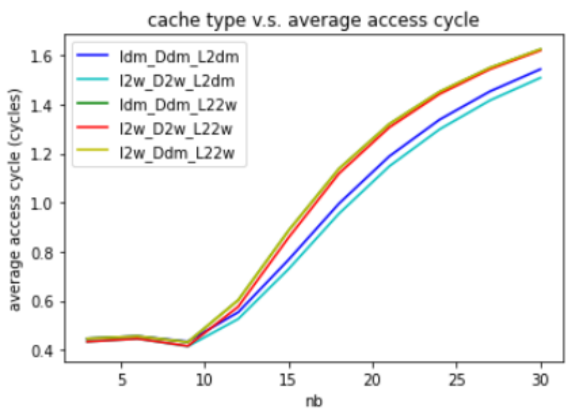


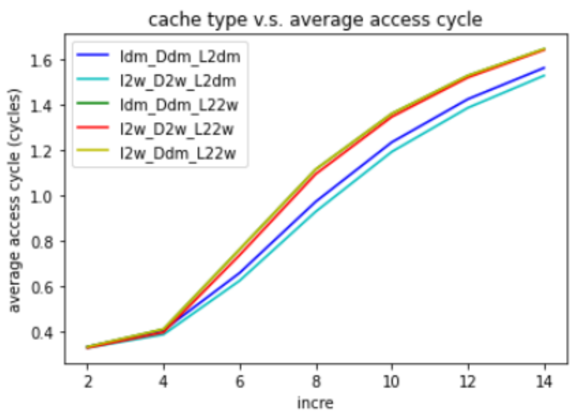
ii. inference

1. different types of cache settings have almost the same duration. We check the access address in waveform, and find that the access address is almost linearly accessing.

c. compare average access time (cycles) in different L2 size settings.

i. results





ii. inference

1. The setting of all dm (blue line) has smaller average access time than the setting of all 2-way (red line)

2. The best setting is L1 use 2-way and L2 use dm. (cyan line). It stalls only one cycle for each memory accessing on average.
3. Type of I cache only slightly affects the average access time. (yellow line and red line)