5-Stage Pipelined RISC-V Processor

Cheng Lin, Lee Shu Kai, Chan Di Yu, Chen

Department of Electrical Engineering, National Taiwan University

{b05901002, b05901027, b05901058}@sntu.edu.tw

Abstract

In this report, we propose a few novel designs on top of the RISCV pipeline structure, with the aim to optimize the synthesis area and the critical path, while speeding up the execution cycles. We will closely examine our improved designs on Forward Unit, Caches, and Stall Unit. Furthermore, we will discuss our implementation on extensions; namely, branch prediction and compression. Various experiments will be performed to highlight the improvements each design brings, in terms of cycle time saved or cycles saved. Last but not least, we will provide an in-depth analysis regarding the testing data and the conditions of the experiment. The final result of baseline AT is 8.2E +12 when the length of Fibonacci series is 40.

Introduction

RISCV is a relatively new design, architected to support similar sets of instructions under lower timing and area criterions. With its simplified design, modularized function blocks, and lofty purpose of become a royalty-free, universal ISA, RISCV has become increasingly popular. Hence, optimizing its pipeline structure is definitely necessary for adapting these ideal designs to reality. For the next part, we will discuss our various designs, in hope of optimization.

Materials and Methods

1. Design on Baseline Model

Five stages are introduced to build the CPU: IF, ID, EX, MEM, WB. Each of them serves its own purpose. IF stage is getting the instruction from the instruction cache; ID stage is analyzing the instruction, generating control signal, immediate and data for register; EX stage performs the arithmetic and output the result; In MEM stage, when **sw** or **lw** instruction is used, the data cache will write or read. Otherwise, this stage only transfers the result from ALU to WB stage; and WB stage chooses which source (ALU or memory) is going to store in register.

Below are some modules we designed to deal with hazards:

(1). JumpBranch

These five stage can support R-type, I-type (except JALR), SW, LW. However, to support jump and branch operation, the module JumpBranch, which handles jump/branch operations, must be included.

Execution principle of the **JumpBranch** module:

(i). The module can tackle the instructions that may change the PC.

(ii). The module situates in ID stage to reduce flushing. (If it's designed in EX stage, it'll flush IF and ID stage. If it's designed in ID stage, only IF stage will be flushed.)

(iii). Extra adder and subtractor (checking equivalence) is place in ID stage.

(iv). When jump or branch occurred, PC must stall and IF will be flush.

(2). Forwarding Unit

Because of the pipeline design, the data used in EX stage from register could be the "invalid" while the R-type or **lw** instruction has updated the data in the EX or MEM stage. So, the forward unit must be introduced. Execution principle of the **Forwarding Unit** module**:**

(i). Forwarding data sources come from EXMEM and MEMWB two pipeline.

(ii). If both of the data are needed to forward, MEMWB data has higher priority.

(iii). If the forwarding register is r0, there's no need for forwarding.

However, for a situation: **lw** followed by a R-type instruction which requiring the data from **lw**, forwarding unit may fail to forward data on time because **lw** get updated data in MEM stage while R-type instruction in EX stage needs the newest data. To solved this situation, bubble (NOP) must be inserted in the MEM stage.

(3). Stall Unit

Stall Unit is a module that tackles the cache stall from both instruction cache and data cache.Given that many instructions or cache may cause stall, flush and bubble, we integrate a module called StallUnit_Normal that can control the next state of each stages: STALL, FLUSH or NORMAL.

StallUnit_Normal module:

(i). Each pipeline and PC has three state controlled by two bit:

a. STALL: Output the value that output in previous clock edge.

b. FLUSH: Output the control signal that won't read or write from memory and won't write the data back to register. It's cause bubble in stage after the pipeline.

c. NORMAL: Output the data that comes from input of the pipeline.

(ii). When Cache_stall == 1, meaning instruction cache or data cache stall, all pipeline is stalled.

(iii). When JumpBranch_stall == 1, meaning jump or branch in ID stage needs forwarding data from EX stage, to reduce the critical path, next state for PC and IFID will be STALL, next state for IDEX will be FLUSH and others will be NORMAL.

(iv). When HU_stall == 1, meaning **lw** followed by a R-type instruction which requiring the data from **lw,** it uses same control signal in 3.

(v). When JumpBranch_flush == 1 meaning jump or branch in ID stage doesn't need forwarding data from previous instructions to determine whether jump or not, next state for IFID will be FLUSH and others will be NORMAL.

situation\ pipeline	PC	IFID	IDEX	EXMEM	MEMWB
JumpBranch_stall or HU_stall	STALL	STALL	FLUSH	NORMAL	NORMAL
JumpBranch_flush	NORMAL	FLUSH	NORMAL	NORMAL	NORMAL
Cache_stall	STALL	STALL	STALL	STALL	STALL

Table1. The Control States of Pipeline Registers of StallUnit_Normal

2. Improvements

(1) . Stall Unit: Because of the high cost of cache stall (at least 10 cycles cost), we try to utilize the cache stall cycles.

There are a few scenarios we can exploit:

(i). When **lw** followed by a R-type instruction which requiring the data from **lw,** if this R-type instruction causes the cache to stall, **lw** can be done first to save 1 cycle.

(ii). **sw** instruction causing data cache stall followed by an instruction named $I_{example}$ causing instruction stall. $I_{example}$ causes instruction cache stall, and **sw** can be done first, which save at most 20 cycles (two caches need write back and allocate simultaneously).

StallUnit_special module is introduced:

(i). When Icache_stall == 1 meaning instruction cache is stall, next state of PC will be STALL, next state of IFID will be FLUSH and others will be NORMAL.

(ii). When Dcache_stall == 1 meaning data cache is stall, next state of EXMEM and MEMWB will be STALL, next state of other pipeline and PC will be determined by different situation.

b-1. For the next state of IDEX: If (EX stage is bubble is satisfied) and (MEM isn't **lw** or data from MEM isn't forwarding to ID is satisfied), next state of IDEX is NORMAL. Else, next state is STALL.

b-2. For the next state of IFID and PC: If the condition for IDEX is NORMAL is satisfied or ID is bubble is satisfied, next state of them is NORMAL. Else next state is STALL.

b-3. When Icache_stall == 1 & JumpBranch_flush == 1, the data instruction cache will be overwritten, thus we need to tackle this special case.

	PC	IF/ID	ID/EX	EX/MEM	MEM/WB
JumpBranch_stall or HU_stall	STALL	STALL	FLUSH	NORMAL	NORMAL
JumpBranch_flush	NORMAL	FLUSH	NORMAL	NORMAL	NORMAL
Icache_stall	STALL	FLUSH	NORMAL	NORMAL	NORMAL
Dcache_stall	b-2	b-2	b-1	STALL	STALL
JumpBranch_flush & Icache_stall	STALL	STALL	FLUSH	NORMAL	NORMAL

Table2. Shows the control state of pipeline or PC of StallUnit_Special

(2) Cache

The original cache module is separated into 3 parts valid, tag, and data, which causes when memory or processor needs to modify the block, it'll need three mux to choose the valid, tag, data in the same block. The memory is a negative-edge-triggered module sending signal at different phase of other modules. So adding a buffer at signal **mem_ready** makes the cache have a whole cycle to store the data from memory. For instruction cache, there's no way that processor will write data into it. So, simplifying the instruction cache into read-only cache can reduce the area.

3. Extensions

(1)Branch Prediction

(i)Description

It can be noticed that for each Jump/Branch operation, a bubble (FLUSH) must be inserted into the IF stage, since the next, incorrect PC address is already extracted from the PC module. In other words, a cycle must be wasted in order to maintain correctness.

For J-type operations, that seems to be inevitable; however, for B-type operations, we can try to predict whether the next branch will return the taken address or not. For our design, we implement two types of Branch Prediction Unit: 2-bit FSM and 2-Level Adaptive Learner.

(ii)Structure

To implement branch prediction, a few modifications to the original design must be made. First of all, Fig x. Is the design for our Branch Prediction Unit (BPU), which calculates the two possible addresses of branch operations, and then chooses one of the addresses based on the state of the Finite State Machines (FSM).

We need to modify the JumpBranch module as well. Because a prediction may be wrong, we need to pass both possible addresses to the ID stage, evaluate the equivalence, and confirm if the prediction is wrong. If a “miss” signal is detected, i.e., the prediction is incorrect, the next PC address will be changed to the correct address branch should return. Also, the JumpBranch module will notify the Stall Unit to flush the ID stage at the next cycle, just as the original branch operation would.

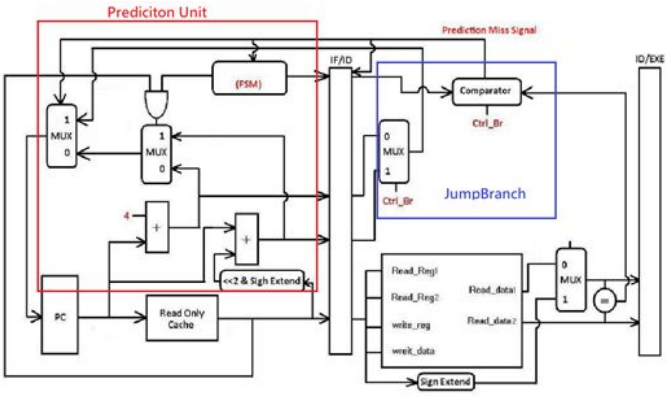

Fig1.

(iii)Finite State Machines (FSM)

(a)2-bit FSM (2bit)

This is a simple FSM for modeling branch decisions. As shown in Fig 2., there are 4 states in total (from upper-left to bottom-left, clock-wise): Strongly Taken (ST), Weakly Taken (WT), Weakly Not Taken (WNT), and Strongly Not Taken (SNT). Each state will output corresponding decisions: taken or not taken. States will transit to other states one cycle after a branch prediction is performed, given a hit/miss signal, which indicates correct/incorrect predictions.

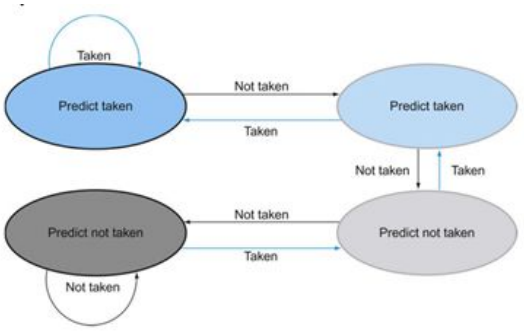

Fig2.

(b) 2-Level Adaptive Learner (2AL)

This design is implemented on top of the 2-bit FSM, where the branch history is also taken into account. This design is based on the concept that repetitive patterns often occur in programs, and if the BPU can remember past patterns, it can achieve more accurate predictions.

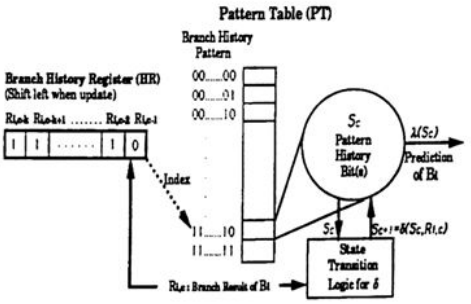

Fig3.

As shown in Fig 3., we keep track of the last k branches (k=2 in our case), mark the correct result of each branch (taken or not), and use them as indexes to recognize states. After the corresponding state is identified, its decision will be outputted. If a prediction is incorrect, 2AL will also change the last state accessed using the FSM provided. 2AL is designed with a flexibility of choosing various FSMs, and for our case, we implement 2-bit FSM for state transition.

(2). Compression

(i).Description

In the following passage, we'll use the abbreviation "Ins." for "Instruction".

Compressed Ins. set aims to reduce common instructions into shorter words; 16-bit instructions are hence designed, with each mapped to a 32-bit Ins. The compression shorten the Ins. using the methods such as reducing the length of Imm, limiting the range of register(maybe only one register in the range) that a certain Ins. could use, equalizing RS and RD…,etc.

Cache miss costs lots of cycles, so we tried to reduce the miss rate. With the help of the compressed Ins. set, the total number of blocks needed for a certain function is reduced, therefore the miss rate could be lower.

(ii).Structure

Implementation of compressed Ins. does not require many changes, we simply need to add to the IF stage a MUX to determine the next PC (PC+2 or PC+4), and a Compression Unit to decode the compressed Ins.. The Compression Unit contains a 16-bit register and a Decompressor.

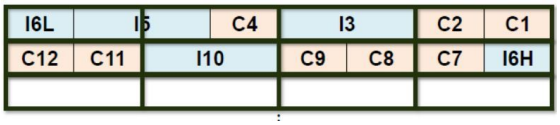

Fig5.

(iii). Implementation

Extracting 16-bit Ins. won't be difficult because it's easy to tell a 16-bit Ins. from a 32-bit Ins. But the inter-word Ins.(e.g. I5 in the figure above) does cause problem since the RISCV can't read this kind of Ins. in one cycle. Our approach is to store the most significant 16 bits in the data read from ICACHE every time. Using the figure above as an example, in the first cycle, we extract C4 and store the right part of I5. In the second cycle, extract I5 using the stored right part and the left part that is read from ICACHE, store I6L at the same time. In the third cycle, combining the stored I6L and the I6H read from ICACHE, store C7 at the same time.

(iv).Jump/Branch Problem

Another problem occurs when the target address of a jump/branch operation is an inter-word Ins.. In this situation, the stored instruction is not valid, so the method mentioned above wouldn't work. We devised two approaches to solve this problem: 4 bytes and 6 bytes, which are named after the number of bytes that RISCV would read from ICACHE in one cycle. Note that the ICACHE in the 6 bytes design could only read the 6 bytes such like I3+C4, reading the 6 bytes such like C2+I3 is not available. Also, it's not available for reading the inter-block 6 bytes such like I5(left side)+I6.

Use the figure above as an example, assume that the PC just jumped to I5. For 4 bytes, in the first cycle, the reading and storing of the right part of I5 would cause the RISCV to stall. Then the situation is back to the one mentioned in "3.Implementation". For 6 bytes, in the first cycle, extract I5, the stored Ins. is still not valid. In the second cycle, we couldn't read the complete I6 due to the limitation on our ICACHE, so we could only store I6L and make the RISCV stall. Then the situation is back to the one mentioned in "3.Implementation".

That doesn't seem to make a difference, but consider the figure below. For 6 bytes, in the second cycle, we can extract C6 directly, and the situation is back to the one mentioned in "3.Implementation". 4 bytes spends one more cycle then 6 bytes in this example. If the 32-bit and 16-bit Ins. occur uniformly, 6 bytes doesn't need to stall at about 53% of the situations that 4 bytes needs to stall.

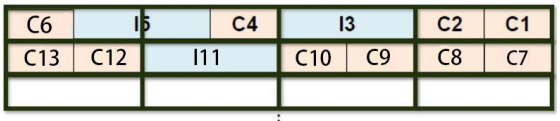

Fig6.

Experiment Setting

1. TestBenches

(1). hasHazard

This instruction set performs the Fibonacci series calculation and bubble sort on the Fibonacci series from largest to smallest which implies worst case bubble sort. To examine the calculation result, it'll store the result ($r8) into memory address 0x3FC. For Fibonacci series, examination will execute after a number from the sequence is calculated. For bubble sort, examination will execute after all bubble sort is done.

(2). BrPred

This test bench is relatively simple; "not branch" (*n_br*), "inter branch" (*inter_br*), "branch" (*br*) operations will be performed consecutively throughout this program, each with an iteration of *a*, *b*, and *c* times separately. (*a*, *b*, *c*, are parameters that can be adjusted). Due to error of the test bench given, we can ignore the parameter *b* for the sake of this experiment. By adjusting *a* and *c*, we will examine the execution cycles of each design.

(3). Compression & Decompression & Compression Shift

The behaviors of the three testbenches are all the same: the number CCCC is multiplied by the number 8763 (both are in HEX representations). The difference is that "Compression" contains 16-bit and 32-bit instructions, while "Decompression" contains only 32-bit instructions. Since the destinations of jump and branch is always beginning of a word in "Compression", we can't test the difference of the performances of 4 bytes and 6 bytes, so we shift "Compression" by half word and become "Compression Shift". The destinations of jump and branch is always the centre of a word in "Compression Shift".

Results and Discussions

1. Baseline Model

For the module in baseline experiment, there are two variables that can compare on the CHIP. One, cache is whether the read-only direct map (DM) or read-only 2-way cache. Two, whether special design Stall Unit or

normal Stall Unit has greater performance. To compare the difference, we use Exp1 as the control group, Exp2,3,4 as the experimental group.

	Exp1 Baseline	Exp2 Icache_2way	Exp3 Dcache_2way	Exp4 StallUnit_Special
Icache	DM	2way	DM	DM
Dcache	DM	DM	2way	DM
StallUnit	Normal	Normal	Normal	Special

Table 2. Baselie experiment control and experimental group

We perform the experiment in the RTL level, “I_mem_hasHazard” is used as a testing data. By changing the Fibonacci series length, we compare the time cycle among 4 different settings.

Cycle count proportion	Exp2 Icache_2way	Exp3 Dcache_2way	Exp4 StallUnit_Special
16	1.68%	0.00%	-1.05%
20	1.19%	0.00%	-0.74%
24	0.88%	0.00%	-0.55%
28	0.68%	0.00%	-0.42%
32	0.49%	-4.69%	-0.93%
36	0.39%	-1.86%	-0.72%
40	0.31%	1.18%	-0.58%
44	0.25%	4.04%	-0.47%
47	0.21%	5.68%	-0.41%

Table3. Change in cycle count compared to baseline model (in %)

Cycle count difference to Exp1	Exp2 Icache_2way	Exp3 Dcache_2way	Exp4 StallUnit_Special
16	40	0	-25
20	40	0	-25
24	40	0	-25
28	40	0	-25
32	40	-380	-75
36	40	-190	-74
40	40	152	-75
44	40	646	-76
47	40	1064	-77

Table 4. Change in cycle count compared to baseline model

Observing the experimental group, we can conclude that Dcache_2way has least cycle at Fibonacci series length = 32 or 36, StallUnit_Special has least cycle at other Fibonacci series length

By comparing in the same experiment with different Fibonacci series length, we can see for Icache_2way the difference keeps 40 cycles along different series length. While for Dcache_2way the difference keeps 0 until length = 28, drops at length = 32 and gradually increase the difference when length increases.

The following explanation is based on the test data "I_mem_hasHazard".

(1). Exp2: The reason that difference of # cycle keep constant between Exp2 and Exp1 is that for some instruction 2-way cache uses the same block while DM has more block which can avoid the miss(Table5)

(2). Exp3: To explain the significant cycle counts drop on length = 32, we can observe that to examine the answer of 32nd number of series the **sw** address must access 0xff, 0x1c frequently, which makes the DM cache stall for every time while 2-way cache utilize the advantage 2 data in one block avoiding a series of stall. It's same situation when it comes to examine the 32nd number of bubble sort.(Table6)

The increasing of cycle count along with series length is caused by the poor replacement policy of 2-way cache which is FIFO. This policy may wash out the data that access frequently causing lots of stall cost.

(3). Exp4: For StallUnit_Special, the main time margin comes from the instruction and data cache stall at the same time.Take one instruction sequence for example, the instruction cache stall caused by instruction 0x10 overlaps the data cache stall caused by the instruction **sw** in 0x08 .(Fig7&8)

PC_addr (byte)	Word	sub-block	**DM block**	DM opration	**2way block**	2way opration
64	16	4	**4**	allocate	**0**	allocate
192	48	12	**4**	allocate	**0**	allocate
8	2	0	**0**	HIT	**0**	writeback & allocate

Table5. Explanation of Exp2

sw_addr	**DM block**	**DM tag**	**DM opration**	**2way block**	**2-way tag**	**DM opration**
ff	7	f	writeback & allocate	3	7	writeback & allocate
1c	7	0	writeback & allocate	3	0	writeback & allocate
ff	7	f	writeback & allocate	3	7	HIT
1d	7	0	writeback & allocate	3	0	HIT
ff	7	f	writeback & allocate	3	7	HIT
1e	7	0	writeback & allocate	3	0	HIT
ff	7	f	writeback & allocate	3	7	HIT
1f	7	0	writeback & allocate	3	0	HIT

Table 6. Explanation of Exp3

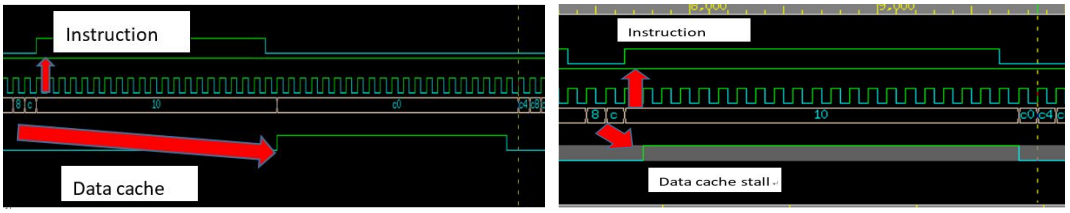

Fig7 & 8 Baseline Stall and StallUnit_Special

2. Branch Prediction

(1). number of cycles (on BrPred)

we compare the execution cycles of different models on test bench BrPred while modifying the parameter *a* and *c* (*b* is always 10) such that a, b, and c sum to 100. Based on the result of Table 7., it is clear that BPU does improve the execution time. Furthermore, we can observe that as *a* and *c* alternate in a linear fashion, all the cycles seem to also decrease in proportion. This is due the simple and iterative nature of BrPred: n less *n_br* operations and n more *br* operations implies n times the difference in execution times.

a/c	Baseline	2bit	2AL
80/10	662	638	642
60/30	642	598	602
40/50	622	558	562
20/70	602	518	522

Table 7. # of cycles of different models

a/c	2bit	2AL
80/10	3.63%	3.02%
60/30	6.86%	6.24%
40/50	10.30%	9.65%
20/70	13.97%	13.30%

Table 8. decrease in # of cycles (compared to baseline)

By Table 8., it can be noticed that 2AL always performs a little worse than 2bit. The reason for this phenomenon is also because of BrPred: BrPred contains such simple operations that it could be easily learned without the help of branch history. In this case, using branch history for indexing states may actually be slowing 2AL down, since it requires extra time to transit to other states, before learning the new pattern.

However, simple test benches like BrPred cannot represent the power of 2AL, nor are they realistic since practical programs may contain more random jumps/ branches. To observe this issue, we also test our models on the hasHazard test bench.

(2) number of cycles (on hasHazard)

Table 9. presents the execution cycles saved by each model on hasHazard, compared to the baseline model, while adjusting nb, which stands for the length of the Fibonacci sequence to be bubble-sorted. It is obvious that 2AL's performance is far better than 2bit. As $nb = 45$, 2AL can save up to 1019 cycles. In terms of percentage in decrease, Fig. 9. indicates that 2AL can save up to 6% of execution time, while 2bit almost saved none.

nb	2bit	2AL
5	-3	-1
10	-7	39
15	-7	104
20	-7	194
25	-7	309
30	-7	449
35	-7	614
40	-7	804
45	-7	1019

Table 9. # of reduced cycles (compared to baseline) on hasHazard

The reason to this is also because of the feature of hasHazard. The test bench haaHazard generates *nb* numbers in increasing order, then bubble-sort them in decreasing order. Because this is the worst case of bubble sort, the pattern (not taken, taken) keeps recurring. It's easy for 2bit to stuck back and forth between WT state and WNT state when encountering such patterns. However, 2AL can utilize branch history and learn this pattern in a few iterations. Therefore, 2AL can save a lot of time.

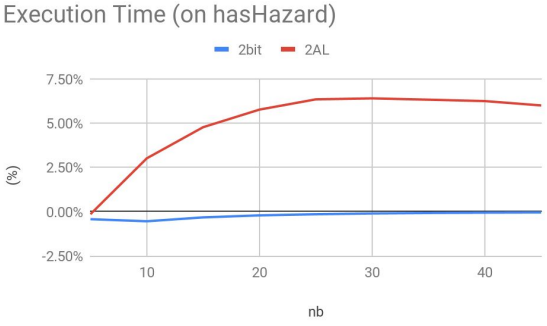

Fig9. Execution time saved on hasHazard

It is obvious then: 2bit is suitable for extremely simple programs, while programs with repetitive patterns are in favor of 2AL. In each cases, BPU performs as slow as the baseline model in the worst case, but it can provide adequate speed-up for most cases.

(3) area * number of cycles Analysis (on hasHazard)

Table 10. gives the synthesis area of models, synthesized with cycle time = 2.5 to compare with the baseline model. It is evident that 2bit is a lot smaller than 2AL in terms of area since 2AL requires recording multiple states, along branch history.

	Area	BPU area
Baseline	279409	x

2bit	279706	297
2AL	293638	14229

Table10. synthesis area and the BPU area

That being said, we will examine the (area*number of cycles) values of each models on hasHazard in order to see if using 2AL is more suitable than using 2bit under this case. Fig. 11. is the visualization of the result. We can observe that for nb >= 20, the (area*number of cycles) value is lower than others. Hence, it is indeed suitable for us to utilize 2AL as the BPU. (This is also true for most cases, since this test bench is more random than others)

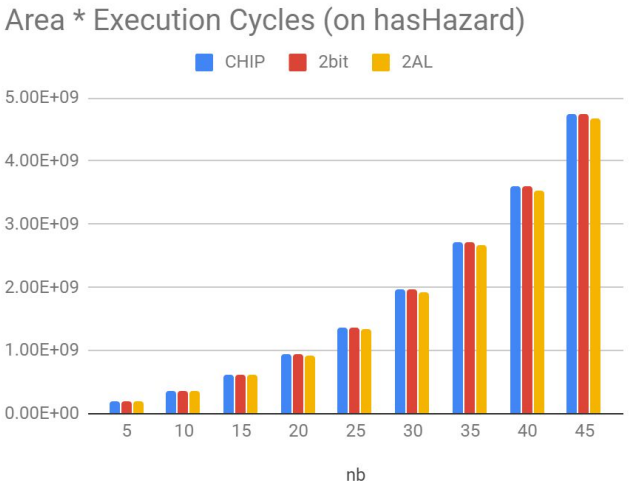

Fig10. Area*Execution Cycles (on hasHazard)

3. Compression

(1) Comparison (on Compression & Decompression)

	Baseline(on Decompression)	4 bytes(on Compression)	6 bytes(on Compression)
# of cycles	748.5	548.5	548.5
area (μm^2)	279409	292964.4	296994.9
# of cycle delta % (compare to Baseline)	0	-26%	-26%
area delta % (compare to Baseline)	0	4.80%	6.30%

Table11.

Table 11 shows the performances of using compression or not. The area of compression is slightly higher than Baseline. But notice that the # of cycles makes a great difference, this great difference results from the total number of blocks used in the testbench. Decompression uses 12 blocks while Compression uses only 8 blocks, it's obvious that the miss rate of running Decompression must be much higher than the miss rate of running Compression, therefore the # of cycles is much lower.

(2) Comparison (on Compression Shift)

	4 bytes(on Compression Shift)	6 bytes(on Compression Shift)
# of cycles	553.5	549.5
area - Baseline_area(μm^2)	13555.4	17535.9
A*# of cycles	7502913.9	9635977.05

Table12.

The table above shows the performances of the two designs. Look at the # of cycles first, 6 bytes stalls less cycles than 4 bytes, so it's reasonable that 6 bytes uses less cycles. But look at the A*# of cycles(A-T value), 6 bytes is much larger than 4 bytes. Also, the jump/branch problem could be avoided by using a better assembly compiler. So we'll use the design of 4 bytes.

(3) Area * number of cycles

Area: Area of compressed design - Area of baseline design

number of cycles: on the Compression test bench

A*T = 15537877.25 ((synthesized using cycle time = 2.5ns)