Digital System Design Final Project: Pipelined MIPS

姚鈞嚴 (B04901032), 傅子興 (B04901015), 何吉瑞 (B04507009)
Group 3
Department of Electronic Engineering

Abstract—**This report describes our pipelined MIPS design. Features such as data path implementation, hazard handling, branch prediction, memory design (including register file and cache), multiplication and division will be introduced. We accomplish both baseline and extension synthesis and achieve 1.90 μm^2·s in baseline, 0.45 μm^2·s in branch prediction, 120 μm^2·s in level 2 cache, and 0.344 μm^2·s in multiplication and division.**

Keywords—latch cells; data buffer; data propagating; machine-learning-based branch prediction

I. Introduction

Pipelined architecture features in high throughput. Nevertheless, if we implement MIPS by this approach, we face many relating hazards such as data hazard, jump hazard, load hazard, and branch hazard. Moreover, compared to MIPS in textbook [1], the requirements for the final project need to support the additional class of jump operations (e.g. *j*, *jal*, *jr*, and *jalr*). Thus, when designing MIPS, we divide it into several stages. First, we implement it without hazard handling. Then, we introduce data hazard and jump hazard. Finally, we complete load hazard and branch hazard. Next, extensions merely modify the chip passing the baseline.

This report is organized as follows. Section II presents the memory optimization, Section III presents the architecture optimization, and Section IV presents the extension optimization. Section V presents our synthesis settings and the corresponding results. Finally, Section VI concludes the report.

II. Memory Optimization

Memory, especially cache, has played an important role of the processor. Cache loads partial data from slow memory, or DRAM, typically, outside the chip to reduce access time, and hence, the higher the hit rate, the better the overall performance. However, its area is comparatively large. Fig. 1 (a) shows that before we implement hazard handling, the area of all memory blocks (register file, instruction cache and data cache) occupy over 90% of the MIPS. The two issues above are both the most sensitive part in terms of area × simulation time value (AT value), implying that optimizing both area and performance is our first policy. There are two different caches required in this system: instruction cache (I cache) and data cache (D cache). The former stores the instruction machine code and the other stores data that MIPS accesses.

A. Bit Cells

From [2], we know that the memory blocks in general processors is made up of SRAM. Though SRAM features in high density and sufficiently fast access, the synthesis tool of this project does not support the library. This makes us to search for other approaches, and latch cells turn out to be the solution. The area of a latch is half of the D flip-flop, showing good potential, while it is hard to control latches since its writable window is half of the clock cycle in comparison to its edge-triggered counterpart.

Fig. 2 presents our solution to this problem. We first observe the lower part. In fact, this is the circuit of clock gating with some modifications. Its primary function is to generate a time pulse to allow new data written into the latch cell. Then, we observe the upper part. We find that latch cell needs a data buffer to provide it a stable new data. We explain the mechanism by the following instance. Suppose that a new value needs to write a specific cell. When the clock rise-triggers, data buffer obtains the value. Then, after propagation delay of the rise-edged gated clock, the cell becomes transparent and the value in it is refreshed. Finally, after the gated clock signal turns off, the state in the cell keeps stable. For reset configuration, since latch usually does not have a reset-relating port, we can still reset it with the aid of the data buffer and the appropriate gated clock signals.

The result on reducing area is marvelous, especially I cache and D cache with more bits of tags and other flags like valid. I cache benefits from 38% reduction [Table I], D cache benefits from 40% reduction [Table II], and the entire MIPS benefits from 35.8% [Fig. 1 (b)]. However, there exists tradeoff such as lack of stability and less EDA tool support, making the post-synthesis simulation cycle very large (about 6ns). In short, this method offer an alternative way to reach similar AT value but features in small area.

B. Instruction Cache

I cache stores machine codes of a series of process, transferred from assembly code with an assembler. When the program counter (PC) inputs an assigned address, I cache outputs the corresponding instruction code. Fig. 3 shows the interface of our I cache. When I cache identifies the address from proc_addr, it either directly gives out the data (to proc_rdata) or stalls MIPS (proc_stall) and then requires data from slow memory. We adopt the direct-mapping method (8 blocks × 4 words). A block includes 4 words (total 128 bits), a valid bit and a tag (25 bits). Besides, we implement an additional memory data buffer (128 bits) to store the data from memory when mem_ready = 1.

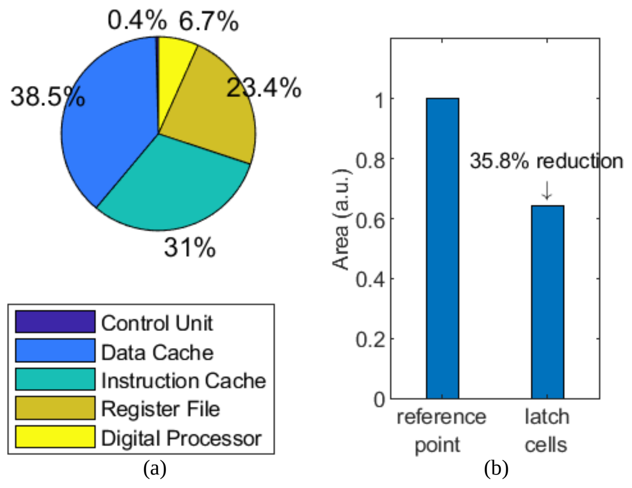


Fig. 1. Area metric without hazard handling. (a) Distribution of blocks. (b) Area reduction with latch-based memory.

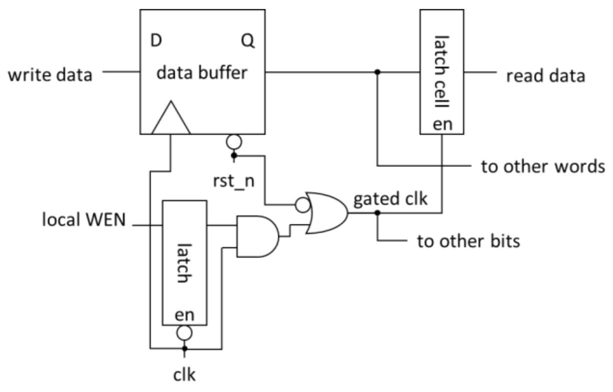


Fig. 2. Schematic of latch-based memory.

To reduce area, we disable all write functionality. We reduce all writing-relating ports, such as proc_write, proc_wdata, and set mem_write and mem_wdata to constant zero in order to fit slow memory interface. Fig. 4 shows the schematic of the finite state machine (FSM) in I cache. When the read access hits, I cache directly gives out the data without stalling the system; otherwise, it jumps to READ and SAVE state to require data from memory and stalls the system concurrently. The separation of two states is to reduce the long path from memory to proc_rdata and to avoid compressing too much process in only half of a cycle as the slow memory passes data at negative edge.

We further go through several structures. For the comparison and conclusion of our experiment, please refer to Table I.

1) *Two-State:* There are only two states in I cache, IDLE and READ. The advantage is that the operation is the simplest and minimize the number of cycles. However, there is only half cycle to store data when the mem_ready triggers, making it a bottleneck and hard to reduce the simulation cycle time for post-synthesis simulation.

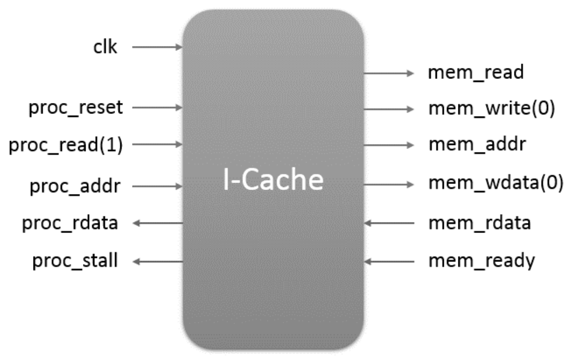


Fig. 3. Instruction cache interface.

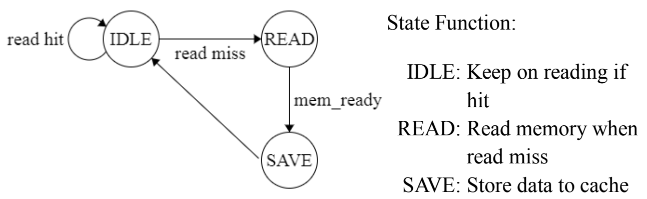

read hit
IDLE
read miss
READ
mem_ready
SAVE
State Function:
IDLE: Keep on reading if hit
READ: Read memory when read miss
SAVE: Store data to cache

Fig. 4. The finite state machine of instruction cache.

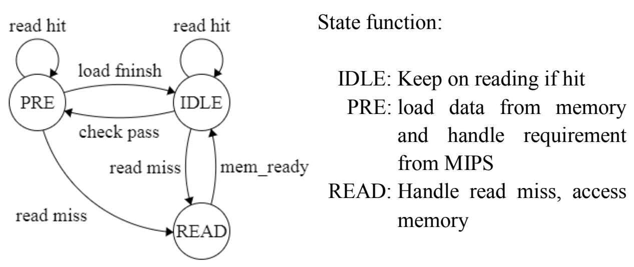

read hit
read hit
load fninsh
PRE
IDLE
check pass
read miss
mem_ready
read miss
READ
State function:
IDLE: Keep on reading if hit
PRE: load data from memory and handle requirement from MIPS
READ: Handle read miss, access memory

Fig. 5. The finite state machine of instruction cache with pre-fetch mechanism.

TABLE I. COMPARISON METRICS OF INSTRUCTION CACHE

	Period (ns)	*Cycle #*	*Total time (ns)*	*Area (μm^2)*	*AT value ($\mu m^2 \cdot s$)*
Two-State	4.6	1997	9183.9	300940.5	2.76
Two-way	4.3	1997	8584.95	302123.6	2.59
Prefetch	4.8	1985	9525.6	315039.1	3.00
Latch-cell	6.1	2107.5	12855.75	260462.6	3.35

a. Testbench: hasHazard.

2) *Two-Way Set Associative:* It is another common type of address mapping. To keep the same number of words, there are total 4 sets, each with 2 elements which stores its own valid, tag and 4 words. We apply least-recently used as the replacement policy. The post-synthesis simulation time is almost the same as using direct mapping, but there is larger area because of longer tags and the hardware of replacement.

3) *Pre-Fetch:* Pre-fetching is an idea that we can load instructions from memory and handle continuous read access from MIPS simultaneously. The reason of pre-fetching tends to be a good strategy to I cache since

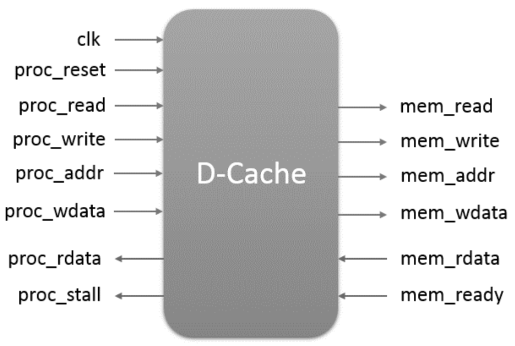
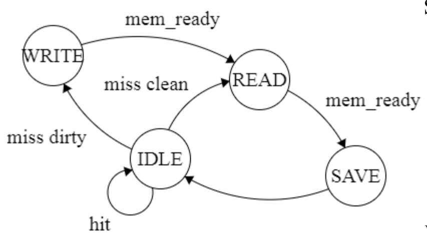


Fig. 6. Data cache interface.



State function:

IDLE:	Keep on reading and writing if hit
READ:	Read memory
SAVE:	Store data to cache
WRITE:	Write memory

Fig. 7. The finite state machine of data cache.

TABLE II. COMPARISION METRICS OF DATA CACHE

	Period (ns)	*Cycle #*	*Total time (ns)*	*Area (μm^2)*	*AT value ($\mu m^2 \cdot s$)*
Three-State	4.4	2149	9453.4	296688.5	2.8
Two-way	3.9	3992.5	15570.75	303602.1	4.73
Latch-cell	6.1	2104.5	12837.45	254329.9	3.26

[a] Testbench: hasHazard.

MIPS often requires instructions from cache in order, and the used instruction is seldom used after executed. As a result, keeping updating data in I cache may be a good way to reduce miss rate and stalling cycles. We use a preload control unit to handle this issue and only pre-load next 4 words only if the index of the block array is 0 now (the first word) to save much loading time, and no *beq*, *j*, *jal* commands appear in the current reading block to avoid wasted cycles. For the FSM, please refer to Fig. 5. However, due to area overhead, extra time cost for decision and actual existence of loop instructions, we do not adopt this structure at last.

C. Data Cache

D cache is used to store and load data, controlled by *lw*, *sw* command. As there is only a 32-word register file inside MIPS, in order not to exceed the storage, MIPS have to communicate with D cache. Fig. 6 shows the interface of our D cache. D cache needs to handle both read and write access, and is controlled by proc_read and proc_write. The reading process is similar to I Cache (Section II-B), so we put emphasis on write access. We also adopt the direct-mapping method (8 blocks × 4 words) while there are additional 8 bits to indicate which block is dirty as we use write back as our writing strategy. Besides, memory data buffer (128 bits) also exists in D cache.

Fig. 7 shows the schematic of the FSM in D cache. When the read or write access hits, D cache directly gives out or writes the data without stalling the system; otherwise, it stalls the processor, jumps to READ and SAVE state to require data from memory if the assigned block is miss and clean, and jumps to WRITE state if the assigned block is miss and dirty. The reason of separating READ and SAVE states is the same as accounted in I cache.

We further go through several structures. For the comparison and conclusion of our experiment, please refer to Table II. In terms of two-way set associative structure, since the issue is the same as I cache, we omit the discussion. Thus, we only analyze the reduction of states. The minimal states can be only three in D cache, which include IDLE, READ and

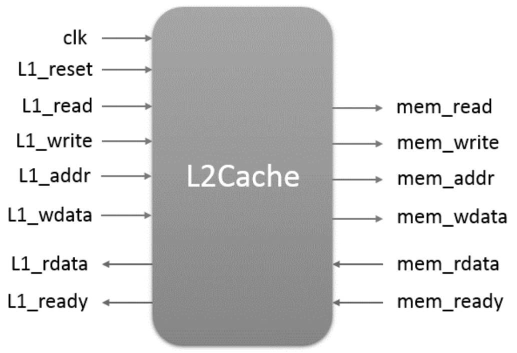
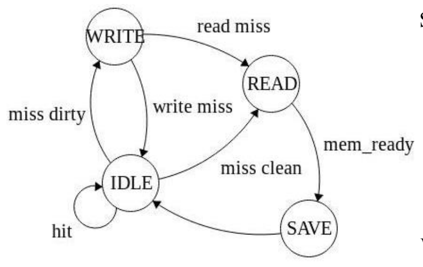


Fig. 8. Level 2 cache interface.



State function:

IDLE:	Keep on reading and writing if hit
READ:	Read memory
SAVE:	Store data to cache
WRITE:	Write memory

Fig. 9. The finite state machine of level 2 cache.

TABLE III. COMPARISION METRICS OF LEVEL 2 CACHE

Required Comparison Metrics		
Avg. mem. access time[b] (cycles)	*Total exe. Time (ns)*	*Post-syn sim. period (ns)*
1.0432	129857.24	4.09

Detailed Post-Synthesis Simulation Result					
	Period (ns)	*Cycle #*	*Total time (ns)*	*Area (μm^2)*	*AT value ($\mu m^2 \cdot s$)*
Reg cell	4.09	31750	129857.24	925727.99	120
Latch cell[c]	4.9	31672.5	155195.25	924699.4	144

[a] Testbench: L2Cache.

[b] Avg. mem. access time = HT1 + MR1×(HT2+MR2×MP2), HT1 = 1, MR1 = 4%, HT2 = 1, MR2 = 2%, MP2 = 4

[c] Latch cells are used only in register file

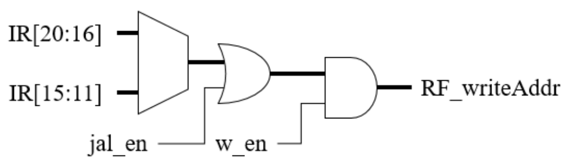


Fig. 10. Write configuration of the register file.

WRITE. The advantage is the same as the case in I cache. However, the negative-edge-triggered mem_ready forces all the relating operations to finish in a half cycle, and as a result, dominates the bottleneck in post-synthesis simulation.

D. Level 2 Cache

Level 2 cache (L2 cache) acts as a memory to level 1 cache (L1 cache, including I cache and D cache), and acts as a cache to slow memory. The existence of L2 cache may shorten the access time for L1 cache, and thus reduce the miss penalty for L1 cache. The interface of L2 cache is shown in Fig. 8. We only support L2 cache to D cache. In RTL code design, we use a wrapper to combine D cache and L2 cache, and make it act effectively as the original D cache. We adopt the direct-mapping method (64 blocks × 4 words). A block includes 4 words (total 128 bits), a valid bit, a dirty bit and a tag (22 bits). Besides, we implement an additional memory data buffer (128 bits) to store the data from memory when mem_ready = 1.

Fig. 9 shows the schematic of the FSM in L2 cache. Most of the design is similar to D cache except that when L2 cache finishes write back, the state transfers to READ if L1_read = 1 while it transfers to IDLE if L1_write = 1 since the data from L1 cache are 128 bits. Another difference is that when we want to stall L1 cache, we pull down L1_ready to fit the original function of mem_ready in L1 cache.

The comparison metrics required from the project and other detailed post-synthesis simulation results are shown in Table III.

III. Architecture Optimization

For a pipelined MIPS, there are total five stages, including instruction fetch (IF), instruction decode (ID), execution (EX), memory (MEM) and write back (WB). Fundamental large blocks such as I cache, register file (RF), arithmetic logic unit (ALU), D cache are located at the same stage as those in the textbook [1]. In this section, we introduce how we arrange the data paths and how we deal with hazards.

A. Register File Write Access Modification

In terms of writing data into RF, we have to consider the relating control signals such as RegDst, Jal, and RegWrite. For a typical design, the RegWrite signal enables the write access while the others determine the address. Thus, the address path includes these MUXes controlled by the two signals. To avoid propagating RegWrite from ID stage to WB stage, we handle these signals in the circuits as shown in Fig. 10. The first MUX for RegDst still exists. For Jal and RegWrite signals, the address result is 31 (5'b11111) if both Jal and RegWrite are true, and is 0 (5'b00000) if RegWrite is false since $zero (number 0) is always a constant zero, implying that RF is not written. Thus, we use OR gate for Jal and AND gate for RegWrite instead of MUXes to reduce the area.

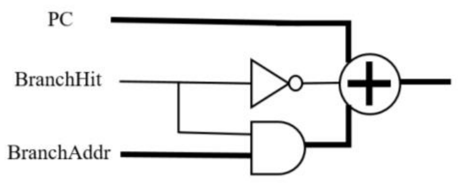


Fig. 11. Merging the branch address with the adder for the program counter.

B. Bubble Reduction and Data Propagation

Directly storing the jump address into RF (*jal* and *jalr*) seems efficient, but requires a bubble to avoid conflict with WB stage. Instead, we propagate it to ALU, that is, we additionally implement a MUX for the first input of ALU. This condition occurs when the command is *jal* and *jalr*. Then, a meaningless operation, logical left shift by zero bits, executes in ALU. Thus, the value holds and correctly stores into RF after the data paths of the remaining stages. Moreover, for *mfhi* or *mflo* commands, we apply the same approach to pass the value from register Hi or Lo, respectively.

As for *beq*, we handle it in ID stage. Conventionally, we use an adder in ID stage to calculate the branch address and pass the result to the MUX that chooses the next values of the program counter (PC) [1]. However, since there is also an adder in IF stage, we merge them so as to reduce area. Fig. 11 shows the concept. Here BranchAddr denotes the branch-extended immediate. In regular cases, BranchHit is false, and the result is sum of current PC address and 4. If BranchHit is true, owing to the mechanism of pipelining, the result should be the sum of current PC value and BranchAddr. Note that this optimization cannot be applied to branch prediction since without prediction, the difference between the PC value in the ID stage and that in the IF stage is always 4, when we sense *beq* in ID stage.

C. Arithmetic Logic Unit Optimazation

Based on the design, our ALU supports 10 functions except for multiplication and division, both of which belong to extension. Table IV lists the control codes and corresponding functions. We divide the functions into four classes (shift, logic, simple arithmetic, and multiplication/ division) via the most significant 2 bits. In terms of multiplication and division, please refer to Section IV-B.

1) *Shift:* Shift type operations include *sll*, *srl* and *sra*. We expect that directly typing register-transfer level (RTL) codes may lead to large area (three shifters and a MUX), so we implement Funnel Shifter [3]. Initially, we need to generate a bus with 63 bits [Table V]. Then, we use a large MUX to obtain the final shifted value [Fig. 12]. Note that if we want to obtain a right-shifted output, the number of shifts (Y[10:6]) needs to change

TABLE IV. CONTROL CODES OF ALU

ALUCtrl	*function*	*Relating Instructions*
0000	shift X left for Y[10:6] bits	*sll, jal, jalr, nop, mfhi, mflo*
0010	arithmetic shift X right for Y[10:6] bits	*srl*
0011	logical shift X right for Y[10:6] bits	*sra*
0100	X & Y	*and, andi*
0101	X | Y	*or, ori*
0110	X ^ Y	*xor, xori*
0111	~(X | Y)	*nor*
1000	X+Y	*add, addi, lw, sw*
1001	X−Y	*sub*
1011	(X<Y)?1:0	*slt, slti*
1100	multiplication[a]	*mult*
1101	division[a]	*div*
default	output 0	

a. For multiplication and division, the relating circuit is in new modules rather than the original ALU block.

TABLE V. BUS VALUE FOR FUNNEL SHIFTER

Function	*Bus Value*
sll	Bus = {X, 31'b0}
srl	Bus = {31'b0, X}
sra	Bus = {{31{X[31]}}, X}

TABLE VI. MUX OUTPUT OF BOOLEAN FUNCTION UNIT

X[i]	*Y[i]*	*and*	*or*	*xor*	*nor*
0	0	0	0	0	1
0	1	0	1	1	0
1	0	0	1	1	0
1	1	1	1	0	0

2) to one's complement. For instance, if we want to shift left by 31 bits (5'b11111), we choose Bus[31:0] while if we want to shift right by 0 bits (5'b00000), we still choose Bus[31:0]. Note that 5'b00000 is the one's complement for 5'b11111.
3) *Logic:* Logic type operations include *and*, *or*, *xor*, *nor* and so on. We expect that directly typing RTL codes may lead to large area overhead, so we implement Boolean Function Unit [3]. In the original case, we have to compute AND, OR, XOR, NOR first and choose the correct logic function with a MUX, while in our proposed case, two inputs directly control the MUX to choose the truth value we compute first via the least significant 2 bits of the control codes [Table VI]. We therefore reduce the area.
4) *Arithmetic:* Arithmetic type operations include *add*, *sub*, *slt* and so on. We expect that directly typing RTL codes may lead to large area overhead (adder, subtractor and comparator), so we implement adder-subtractor [3]. If the function is slt, the ALU executes subtraction and sense the compare flag. The practical implementation is shown in Fig. 13.

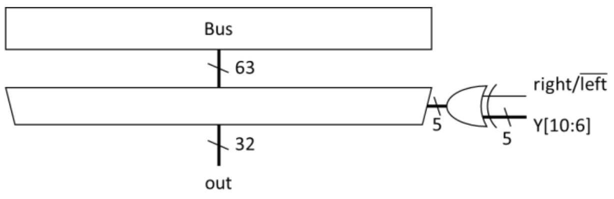


Fig. 12. Output stage of Funnel Shifter.

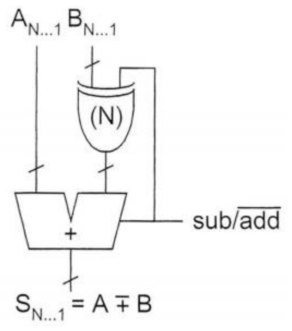


Fig. 13. Adder-subtractor.

D. Hazard Handling

There are several types of hazard as follows, all of which relate to the data refreshing of RF. In general, if the hazard happens in EX stage, we need to pipeline the hazard control. If the hazard happens in ID stage, we directly control it.

1) *Data Hazard:* This may happen if we sense the write address of RF in EX stage is the same as the read address of RF in ID stage. Hence, we forward the pipelined data. More specifically, we link the data from different stages (original EX, MEM, WB, and pipelined WB) together to the two input ports of ALU with 4-to-1 MUXes. In terms of pipelined WB, we have to consider the case that we sense the hazard between ID stage and WB stage, and hence, we need to pipeline the data in WB stage.
2) *Jump Hazard:* This may happen when we sense *jr* or *jalr* in ID stage. We also apply a MUX with the data input from original ID, EX, MEM and WB stages with the control signal named ForwardJump.
3) *Load Word Hazard:* This may happen when the *lw* command executes in the EX stage and the command in ID stage needs the data from D cache. Hence, we need to generate a bubble in EX stage and stall the process in both IF and ID stage, and the hazard belongs to other types.
4) *Branch hazard:* This may happen when we sense *beq* in ID stage. We directly forward the data to the comparator, which is similar to jump hazard. However, due to long critical path, we generate a bubble in EX stage when EX stage and ID stage access the same relating address of RF. In addition, if we sense *lw* in EX stage, we generate two bubbles in EX stage since

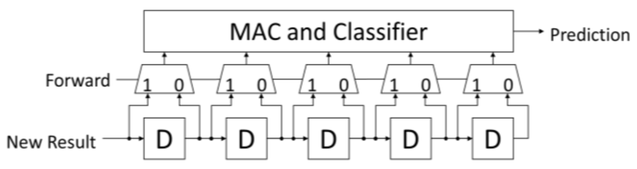


Fig. 14. Finite impulse response filter with classifier.

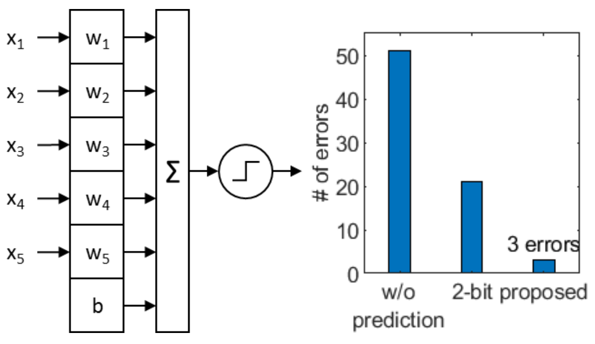


Fig. 15. (a) MAC and classifier. (b) Performance of different types of branch prediction architectures (based on the testbench for branch prediction).

we expect that the path from D cache to PC through the branch comparator is long.

IV. Extension Optimization

There are three optional extensions in this project and we accomplish all. The following describes how we optimize these extensions. However, for L2 cache, please refer to Section II-D, and for mfhi and mflo commands, please refer to Section III-B.

A. Branch Prediction

After observing the branch result from all of the testbenches, we discover that simply implementing a 2-bit prediction unit does not effectively enhance the performance. As a result, we look for other implementations. Actually, if we record the branch result each time, we can construct a time signal, that is, we can exploit a finite impulse response (FIR) filter to generate the prediction [Fig. 14]. Whenever we obtain a new branch result in ID stage, we shift the delay registers. Then, when the MIPS sense the new branch command in the IF stage, we obtain the MAC and classifier result to determine the prediction. An exception happens if there are continuously two branch commands and the one in the ID stage predicts correctly. Since the delay registers does not shift yet, we need to choose the forward values.

In terms of machine learning, we can regard branch prediction as a classification problem, so we implement the network as shown in Fig. 15 (a). The weights and the bias are trained in advance. However, to reduce complexity of the hardware, quantization is essential and we finally choose 4 bits. We first compute the MAC result, and activate it with a step function, or in reality, by extracting the most significant bit. Since five of the inputs are only 1 bit, the multiplication is nothing but an AND function, which is a good news to MAC. Accumulation is also supposed to be simple due to quantization. Fig. 15 (b) shows the result of our proposed implementation. Compared to original chip and the one with 2-bit prediction, ours only outputs three errors, an obvious reduction.

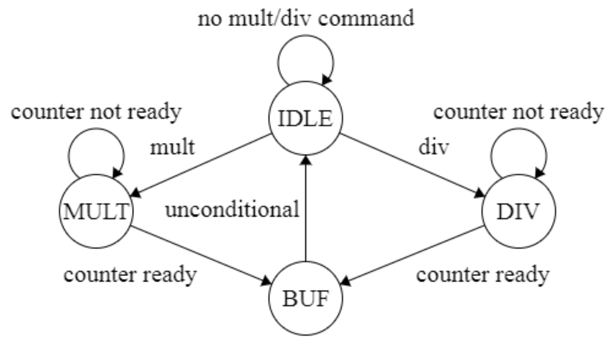


Fig. 16. Finite state machine of the multiplier-divider.

B. Multiplication and Division

For simplicity, we propose a general FSM [Fig. 16] to control both the multiplier-divider and the original processor. When either multiplication or division commands (mult, div, respectively) occurs in the IDLE state, the state machine stalls the processor just like what caches do when they miss the data. The state then changes to MULT (or DIV). After the allocated cycles, the state becomes BUF and returns to IDLE in the next cycle. With some shift operations, we use 8 cycles to compute multiplication and 16 cycles to compute division, and the registers, Hi and Lo, store the temporary result.

1) *Multiplier:* Fig. 17 shows the multiplier block. Since there are 8 cycles and the data is 32 bits wide, we take four product results (I, J, K, L) and shift-add them together in each cycle. Fig. 18 shows the top view. Note that the shift in the temporary is constant since we execute multiplication from the most significant bits.
2) *Divider:* In the divider cell, we need to determine whether the difference is positive or not to select the correct remainder [Fig. 19]. This remainder is then sent to next stage iteratively to obtain the result of division. By similar mechanisim in multiplication, we build a divider block to calculate 2 iterations because we use 16 clock cycles here.

Fig. 20 shows how we implement the two funcitons together. Both blocks share the same inputs and their temporary results propagate to register Hi and Lo.

V. Synthesis Setting and Result

We use the required synopsys design constraints offered by the teaching assistance, and only modify the cycle for synthesis. The only constraint we add is that the tool supports *ungroup -all -flatten* except for L2 cache, which means that the generated gate level code is not classified to its original module. However, this constraint make it difficult for us to debug, so we only use t when we synthesize the final version

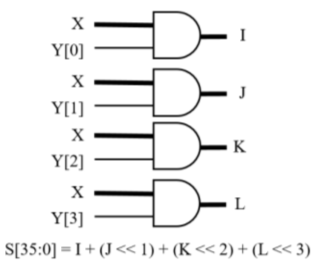


Fig. 17. The multiplier block.

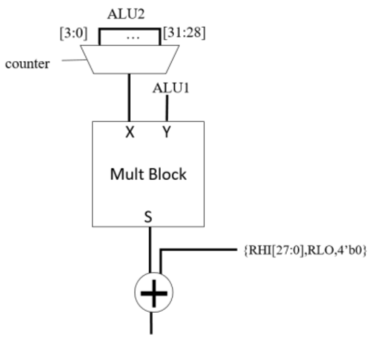


Fig. 18. Top view of the multiplier.

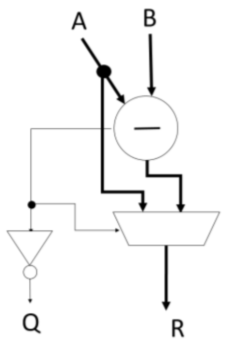


Fig. 19. The divider cell.

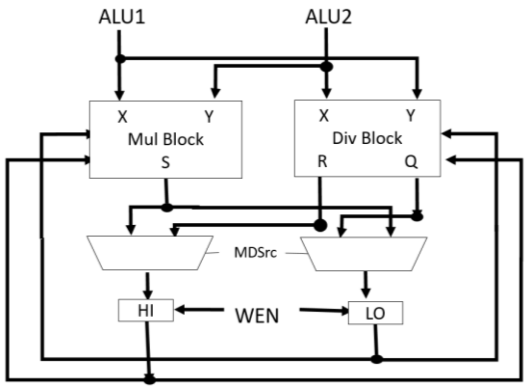


Fig. 20. The overall multiplier-divider.

of our design. For compile instruction, we use compile instead of compile_ultra, and does not give any other flag. The synthesis results are shown in Table VII.

TABLE VII. POST-SYNTHESIS SIMULATION RESULT

	.sdc cycle (ns)	***Sim. period (ns)***	***Sim. time (ns)***	***Area (μm^2)***	***AT value ($\mu m^2 \cdot s$)***
hasHazard	3	2.93	6628.22	287086.35	1.90
BrPred	3	2.83	1498.09	300207.25	0.450
L2Cache	3	4.09	129857.24	925727.99	120
Mult/Div	3	3.21	1073.87	320492.88	0.344

VI. CONCLUSION

To design a MIPS is a hard task. Plotting the architecture is necessary before we practically type the RTL code. After gradually accomplishing each stage that we define in Section I, we harvest the satisfying result. The steps that we minimize the AT value follow the sensitivity and thus we first focus on shrinking the area of memory, then compressing the timing of cache, and finally, modifying local blocks and data paths. However, due to clock delay, exploiting latch-based memory may increase the simulation time. In addition, for multiplication and division, we reduce the latency from 32 cycles to 8 and 16 cycles, respectively. Moreover, our MIPS is more than a normal MIPS since we combine various techniques, which we learn from different courses such as (advanced) integrated circuit design and machine learning, to this course. For instance, we achieve the prominent result of branch prediction via machine learning. If we consider the practical applications, we can store different weights and bias in memory to deal with different types of predictions, or even train the MIPS simultaneously when it executes, although it is out of the scope of the specification according to this project. According to our optimization techniques explained above, we expect our MIPS to be the comparable one among all of the groups.

REFERENCES

[1] D. Patterson and J. Hennessy, *Computer organization and design*, 5th ed. 2017.

[2] "Static random-access memory", *En.wikipedia.org*, 2018. [Online]. Available: https://en.wikipedia.org/wiki/Static_random-access_memory.

[3] N. Weste and D. Harris, *Integrated circuit design*, 4th ed. 2011.