

프로젝트 결과 보고서

miniMIPS 기반
XOR / Shift / Rotate
연산 속도 개선을 위한 CPU

12191529 장준영

12211472 김현민

12211785 송상윤

12236553 안효리

2025.12.03

목차

1. 주제 선정(3)
2. 하드웨어 구성(4)
3. 컨트롤 시그널 구성(7)
4. Control State Machine(8)
5. 명령어 구성(9)
6. 구현(18)
7. 성능 비교(19)

전체 코드

https://github.com/inha-kimhyunmin/COA_Project



프로젝트 결과 보고서

1 주제 선정

주제 선정 과정	암호화, 난수 생성 과정에서 XOR, Shift, Rotate 연산이 반복적으로 사용되는 것을 발견하고, 암호화, 난수 생성 과정에서 쓰이는 알고리즘의 성능 개선을 이를 수 있는 CPU를 설계하기로 결정하였다.
타겟 알고리즘	<p>ChaCha20</p> <ul style="list-style-type: none">- 스트림 암호. 키와 nonce를 입력받아 긴 난수 스트림을 생성하고 평문과 XOR하여 암호화한다.- 덧셈, XOR, 회전(rotate) 연산이 많이 쓰인다. <p>Xorshift32</p> <ul style="list-style-type: none">- 매우 가벼운 의사난수 생성기(PRNG). 보안성은 없지만 빠른 속도를 보장한다.- XOR + shift 만 사용된다. <p>Xoshiro128</p> <ul style="list-style-type: none">- 고품질 PRNG. xorshift보다 비트 품질이 좋고 빠른 속도를 가진다.- XOR + shift + rotate 를 조합한 연산이 쓰인다. <p>SHA-256</p> <ul style="list-style-type: none">- 해시 알고리즘. 임의 길이 입력을 고정 길이 해시로 압축한다.- 비선형 함수(Ch, Maj), rotate, shift 연산이 많이 쓰인다.
타겟 어플리케이션	<p>1. VPN</p> <ul style="list-style-type: none">- 터널링 기술을 통해 네트워크 트래픽을 암호화하고, 무결성과 기밀성을 보장하며, 안전한 세션 키 교환을 수행해 도청과 변조를 방지한다.- ChaCha20 또는 AES(대칭 암호), SHA 계열(해시·무결성) 알고리즘 사용, 그리고 보안 난수원이 만든 CSPRNG 난수를 사용한다. <p>2. 난수 생성 이용 프로그램</p> <ul style="list-style-type: none">- 확률적 모델 · Monte-Carlo · 물리 시뮬레이션을 수행한다.- Xorshift32, Xoshiro128, MT19937 등 빠른 PRNG가 사용된다. <p>3. 보안 / 인증 시스템</p> <ul style="list-style-type: none">- 신원 확인, 권한 부여 등의 역할을 수행하며 데이터 무결성, 기밀성을 보장하는 시스템이다.- SHA-256 해시를 이용한 인증 작업, 데이터 무결성 검증을 진행한다.

2 하드웨어 구성

기본 구조	MiniMIPS의 기본 구조를 사용하였다.
Register 구성	<p>MiniMIPS의 레지스터 구조를 그대로 사용하였다. 총 32개의 레지스터를 사용하며, 각 레지스터는 32비트값을 저장한다. 0번 레지스터는 zero로 사용하고, 31번 레지스터는 return address로 사용하였다.</p>
ALU 구성	<p>X 입력과 Y 입력 총 2개의 입력이 있으며, 연산 유닛은 Adder, Logic Unit, Shifter로 구성하였다.</p> <p>Adder는 ADD, SUB 연산을 수행하며 연산 결과와 연산 결과가 zero인지, 연산에서 overflow가 발생 여부를 출력한다.</p>

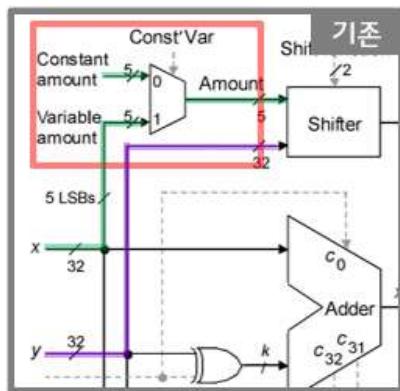
Set Less Than 연산에서는 ALU 연산 결과의 MSB만 추출하여 연산 결과 선택 MUX 입력으로 연결된다.

Logic Unit은 AND, OR, XOR, NOR 연산을 수행한다.

Shifter는 Barrel Shifter 구조를 사용하여 여러 비트를 한 번에 이동할 수 있도록 설계하였으며, Rotate 연산 또한 가능하도록 구현하였다.

Adder, Logic Unit, Shifter에 들어가는 컨트롤 신호는 ALUFUNC이며 2비트 신호로 구성하였으며 ALUFUNC 컨트롤 신호는 3개의 유닛에 전부 입력되어 하나의 컨트롤 신호로 여러 유닛의 동작을 조절한다.

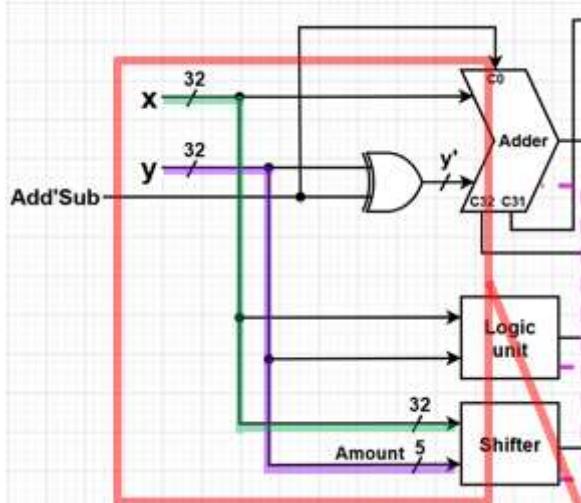
각 Unit의 결과는 MUX 입력으로 들어가며, FnType 컨트롤 신호를 통해 어떤 연산 결과를 출력할 것인지를 결정한다.



기존 MiniMIPS의 ALU의 Shifter에서는 레지스터 값으로 연산을 하는 명령어(Shift Variable)과 Instruction의 sh부분 값으로 연산을 하는 명령어가 둘 다 존재하였다.

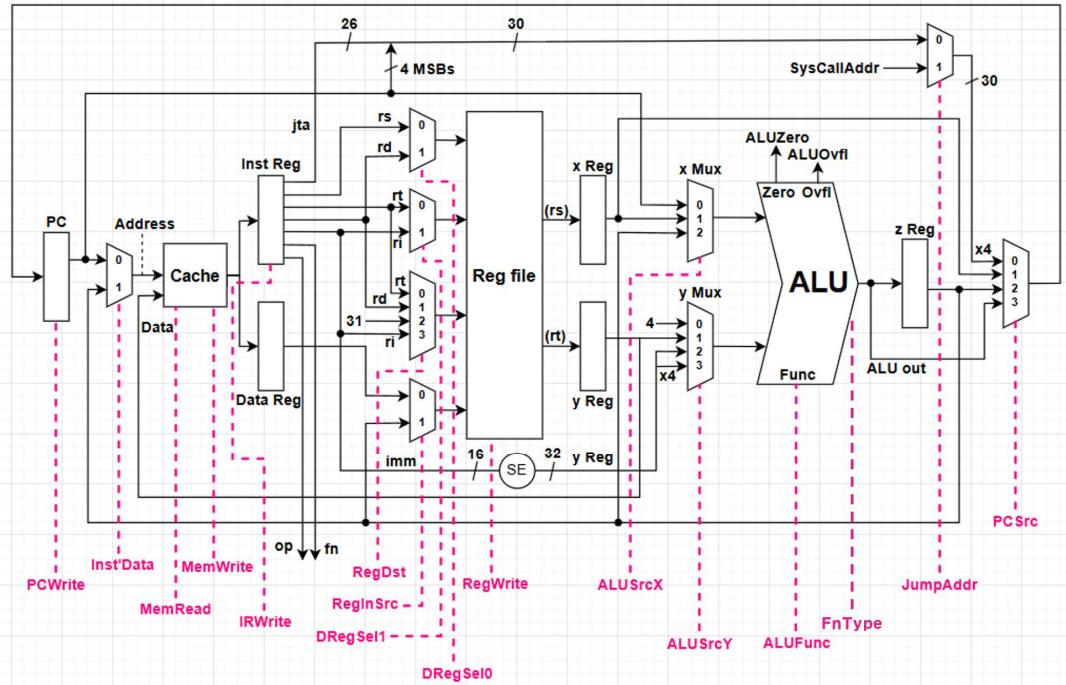
명령어 구성에서 Shift Variable 명령어를 제외하였으므로, Constant amount와 Variable amount를 선택하는 MUX를 제거하고, Y MUX를 통해 넘어오는 값(imm, rt)을 Shift amount로 사용하기로 결정하여 X MUX를 통해 넘어오는 입력을 Shift하는 변수로 결정하였다.

ALU 구성 - 기존 MIPS와 의 차이점



변경된 ALU 구조이다. Shift Amount 입력은 Y 입력으로부터만 들어가도록 구성하였다. 이 구조에 맞게 Shift 명령은 X 입력으로 들어온 값을 Y 입력으로 들어온 값만큼 Shift 연산을 진행하는 것으로 결정하였다.

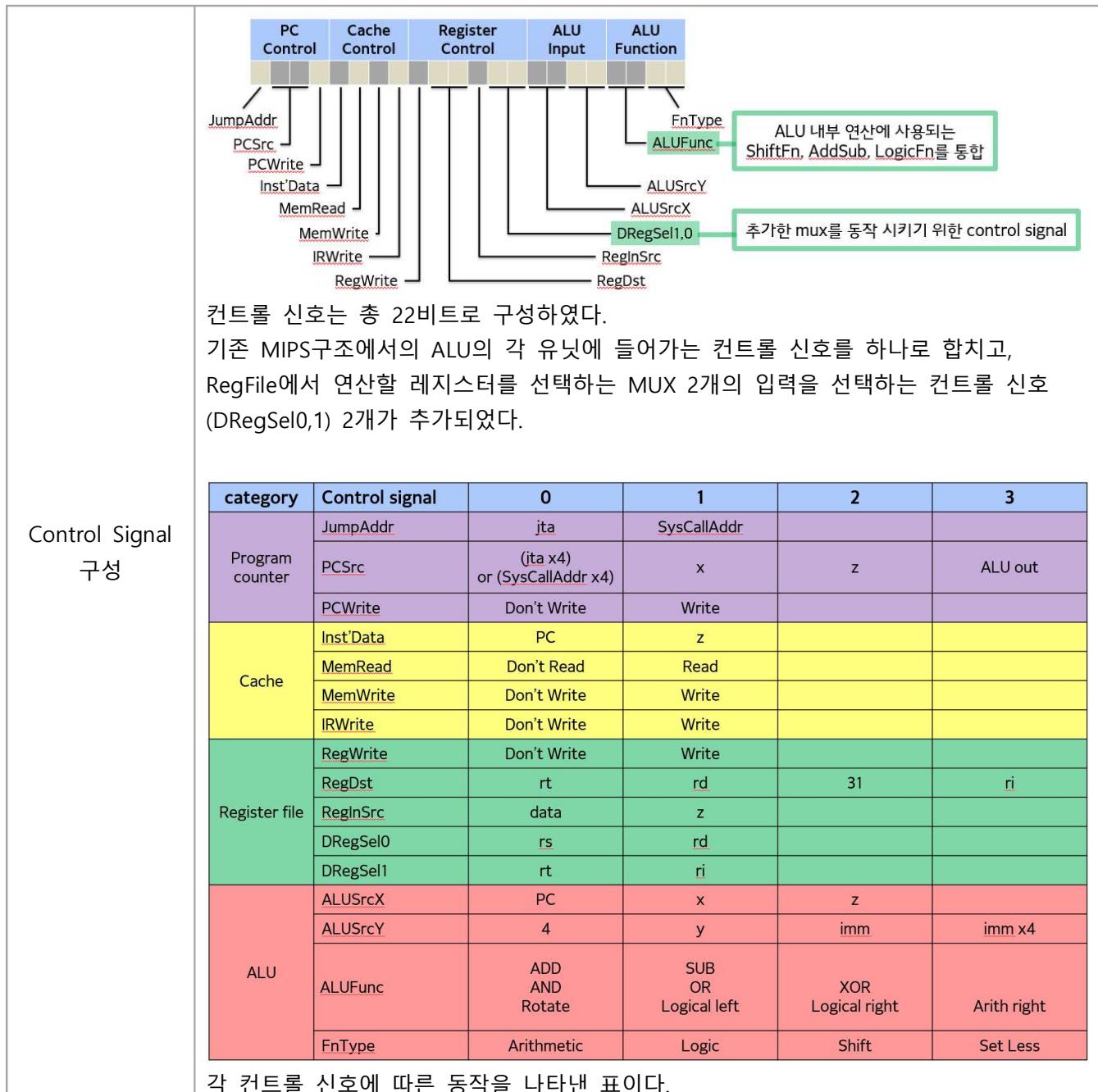
전체 하드웨어 구성



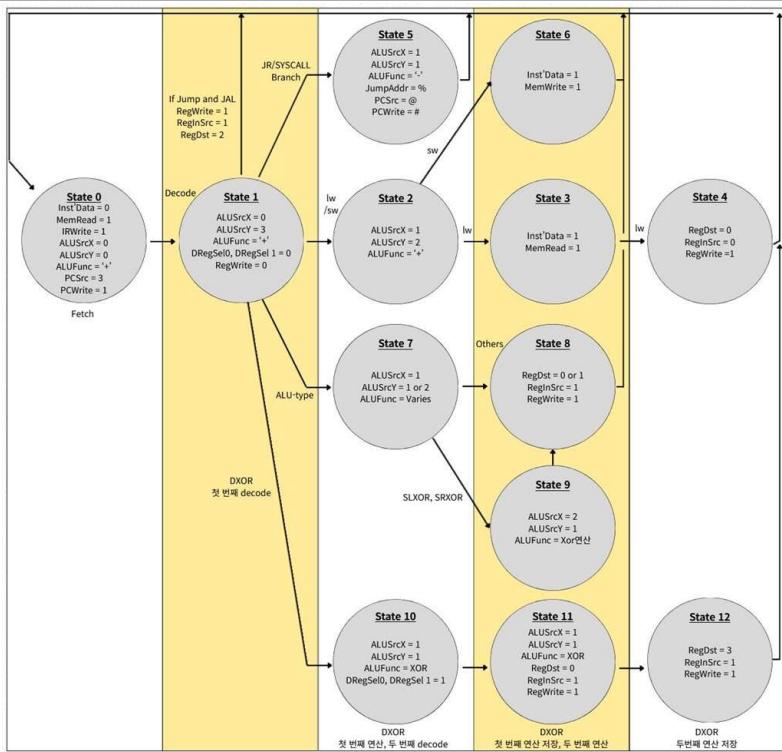
전체 하드웨어 구성도이다.

먼저, D-type 명령어를 실행하기 위해 RegFile에서 연산할 레지스터를 선택하는 MUX 2개를 추가하였으며 이 2개 MUX의 입력을 선택하는 컨트롤 신호 DRegSel0, 1을 추가하였다. 또한, D-type 명령어에서 ri에 저장하는 과정이 존재하므로 Destination Register를 선택하는 MUX에 ri 입력을 추가하였다.
다음으로, 추가한 명령어인 SLXO, SRXO 동작 시 ALU 연산 결과를 다시 연산하는 과정이 존재하므로, 이를 위해 Z Reg에서 X mux로 이어지는 Datapath를 추가하였다.

3 Control Signal 구성



4 Control State Machine



Control State Machine을 나타낸 그림이다.

Control State Machine 구성

State 0에서는 Fetch과정을 진행하며, Fetch 과정에서는 PC값의 주소로 Cache에 접근해서 해당 주소에서 Instruction을 IR에 저장하며, PC+4를 진행해서 PC에 쓴다. State 1에서는 Decode과정을 진행하며, Decode 과정에서는 opcode와 function code를 해석해서 명령어를 판별하며, Branch 명령을 대비한 주소를 미리 연산한다. 만약 opcode와 function code를 해석해서 명령어가 Jump명령이면 PC값과 Instruction jta 부분을 연산하여 PC값에 저장하며, Jump and Link 명령이면 State 0에서 연산한 PC+4를 31번 레지스터에 저장하고, Jump 명령을 실행한다.

State 2, 3, 4, 6은 Load Word, Store Word 명령을 위한 State이며 기존 MiniMIPS와 동일하게 동작한다.

State 5는 Jump Register, System Call 명령을 위한 State이며 기존 MiniMIPS와 동일하게 동작한다.

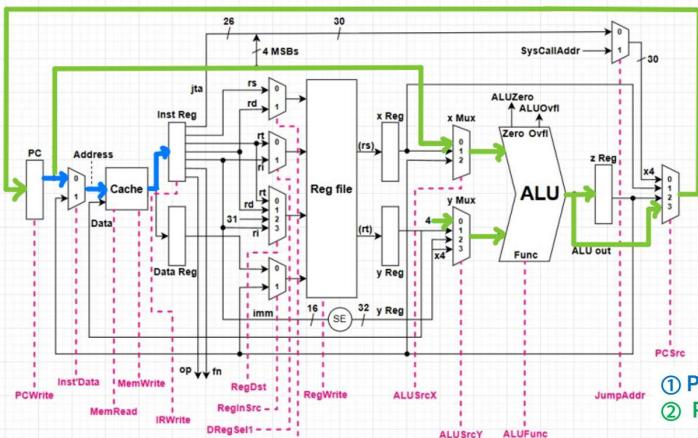
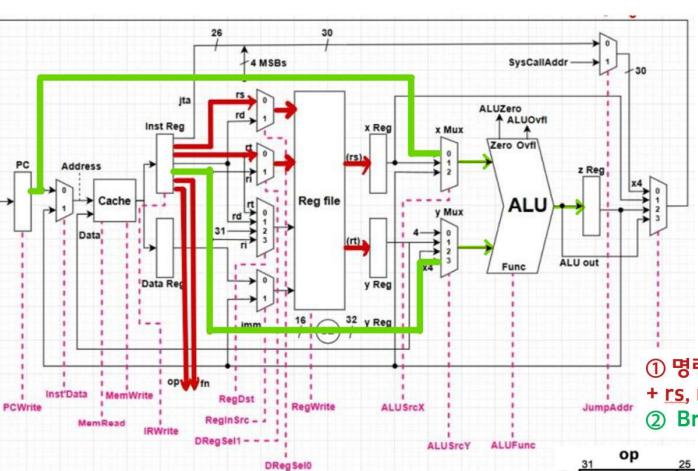
State 7, 8은 ALU연산을 진행하는 명령을 위한 State이며, 기존 MiniMIPS와 동일하게 동작한다.

State 9은 새로운 명령어 SLXO, SRXO를 위한 State이며 State7에서 Shift 연산 결과와 rt레지스터의 값을 XOR하여 Z Register에 저장한다.

State 10, 11, 12는 새로운 명령어 DXOR 명령어를 위한 State이며, State 10에서는 첫 번째 XOR 연산을 진행하여 Z Register에 저장하며 두 번째 연산하는 값들을 X Reg, Y Reg에 저장한다. State 11에서는 Z Register에 저장된 값을 RegFile에 저장하며, 두 번째 XOR 연산을 진행하여 Z Register에 저장한다. 마지막으로 State 12에서는 Z Register에 저장된 값을 RegFile에 저장한다.

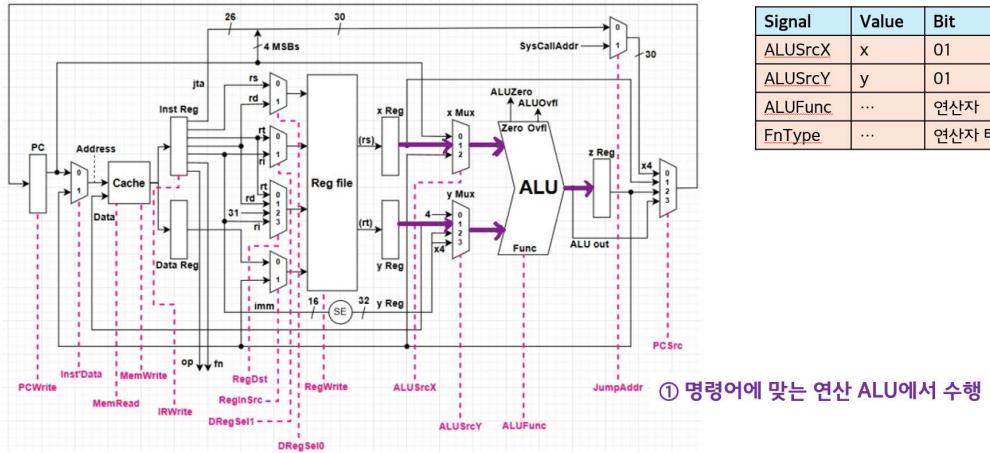
5 Instruction 구성

		Type	Instruction	Usage	opcode	funct
구현한 전체 Instruction	R	ADD	add rd, rs, rt	000000	100000	
		SUB	sub rd, rs, rt	000000	100001	
		AND	and rd, rs, rt	000000	100100	
		OR	or rd, rs, rt	000000	100101	
		XOR	xor rd, rs, rt	000000	100110	
		NOR	nor rd, rs, rt	000000	100111	
		ROT	rot rd, rs, rt	000000	000000	
		SLL	sll rd, rs, rt	000000	000001	
		SRL	srl rd, rs, rt	000000	000010	
		SRA	sra rd, rs, rt	000000	000011	
		JR	jr rs	000000	001000	
		SYSCALL	syscall	000000	001100	
		SLXO	slxo rd, rs, rt, sh	000000	101001	
		SRXO	srxo rd, rs, rt, sh	000000	101010	
		SLT	slt rd, rs, rt	000000	111000	
	D	DXOR	dxor r1, r2, ra, rb	000000	110010	
새로운 Instruction	I	ADDI	addi rt, rs, imm	001000	-	
		SUBI	subi rt, rs, imm	001001	-	
		ANDI	andi rt, rs, imm	001100	-	
		ORI	ori rt, rs, imm	001101	-	
		XORI	xori rt, rs, imm	001110	-	
		LW	lw rt, imm(rs)	100011	-	
		SW	sw rt, imm(rs)	101011	-	
		BEQ	beq rs, rt, L	000100	-	
		BNE	bne rs, rt, L	000101	-	
	J	SLTI	slti rt, rs, imm	001011	-	
	J	J	j jta	000010	-	
	JAL	JAL	jal jta	000011	-	
새로운 Instruction	Type	Instruction	Usage	opcode	funct	
	R	ROT	rot rs, rt, rd	000000	000000	
		SLXO	slxo rs, rt, rd, sh	000000	101001	
		SRXO	srxo rs, rt, rd, sh	000000	101010	
	D	DXOR	dxor rs, rt, ra, rb	000000	110010	
XOR, Shift, Rotate 연산 최적화 CPU를 구성하기 위해 새로운 Instruction 4개를 추가하였다. 먼저 ROT는 rt 레지스터의 값만큼 rs 레지스터의 값을 왼쪽으로 비트 회전을 하여 rd 레지스터에 저장하는 연산이다.						

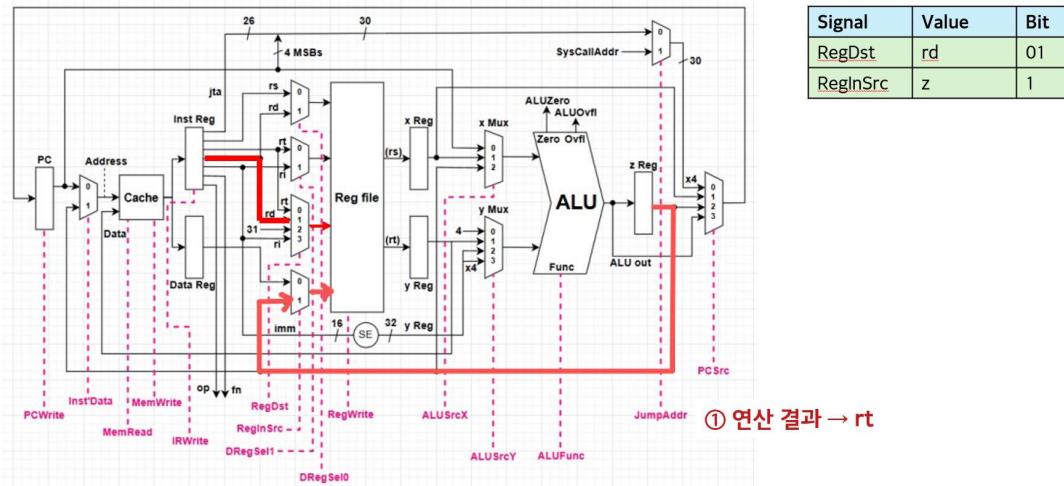
	<p>SLXO, SRXO는 rs 레지스터의 값을 sh값만큼 shift를 시키고 이 값을 rt 레지스터의 값과 XOR 연산을 하여 rd 레지스터에 저장하는 연산이다.</p> <p>DXOR은 rs 레지스터의 값과 rt 레지스터의 값을 XOR 연산을 하여 rt 레지스터에 저장하고, ra 레지스터의 값과 rb 레지스터의 값을 XOR 연산을 하여 rb 레지스터에 저장하는 연산이다.</p>																																																																		
새로운 Instruction 선정 이유	<p>XOR, Shift, Rotate 연산을 활용하는 알고리즘에서는 Rotate 연산과 Shift한 값을 XOR하는 연산, 여러 번 반복되는 XOR 연산이 있음을 확인하였다. 기존 MiniMIPS 구조에서는 ROT 명령어가 존재하지 않으므로 이를 추가하였으며, Xorshift 명령어에서 사용되는 “X를 Shift한 값을 X와 XOR하여 X에 저장” 연산에서의 성능 개선을 위해 SLXO, SRXO 명령어를 추가하였다.</p> <p>마지막으로, Xoshiro, ChaCha20 알고리즘에서는 반복적인 XOR 연산이 있음을 확인하였고, 정해진 32비트 Instruction 길이 안에서 2개의 XOR 연산을 진행하는 명령어를 구현하여 반복적인 XOR 연산에서의 성능을 개선하였다.</p>																																																																		
전체 Instruction의 Datapath	<p>공통 과정(Fetch, Decode)</p> <p>1. Fetch</p>  <table border="1" data-bbox="1135 875 1421 1224"> <thead> <tr> <th>Signal</th> <th>Value</th> <th>Bit</th> </tr> </thead> <tbody> <tr> <td>PCSrc</td> <td>z</td> <td>10</td> </tr> <tr> <td>PCWrite</td> <td>Write</td> <td>1</td> </tr> <tr> <td>Inst'Data</td> <td>PC</td> <td>0</td> </tr> <tr> <td>MemRead</td> <td>Read</td> <td>1</td> </tr> <tr> <td>MemWrite</td> <td>Don't Write</td> <td>0</td> </tr> <tr> <td>IRWrite</td> <td>Write</td> <td>1</td> </tr> <tr> <td>RegWrite</td> <td>Don't Write</td> <td>00</td> </tr> <tr> <td>ALUSrcX</td> <td>PC</td> <td>00</td> </tr> <tr> <td>ALUSrcY</td> <td>4</td> <td>00</td> </tr> <tr> <td>ALUFunc</td> <td>ADD</td> <td>00</td> </tr> <tr> <td>FnType</td> <td>Arithmetic</td> <td>00</td> </tr> </tbody> </table> <p>① PC에 있는 주소 → 메모리 접근 → IR 저장 ② PC ← PC+4</p> <p>2. Decode</p>  <table border="1" data-bbox="1119 1381 1421 1684"> <thead> <tr> <th>Signal</th> <th>Value</th> <th>Bit</th> </tr> </thead> <tbody> <tr> <td>PCWrite</td> <td>Don't Write</td> <td>0</td> </tr> <tr> <td>MemRead</td> <td>Don't Read</td> <td>0</td> </tr> <tr> <td>IRWrite</td> <td>Don't Write</td> <td>0</td> </tr> <tr> <td>DRegSel0</td> <td>rs</td> <td>0</td> </tr> <tr> <td>DRegSel1</td> <td>rt</td> <td>0</td> </tr> <tr> <td>ALUSrcX</td> <td>PC</td> <td>00</td> </tr> <tr> <td>ALUSrcY</td> <td>imm x4</td> <td>11</td> </tr> <tr> <td>ALUFunc</td> <td>ADD</td> <td>00</td> </tr> <tr> <td>FnType</td> <td>Arithmetic</td> <td>00</td> </tr> </tbody> </table> <p>① 명령어에서 읽은 op, fn을 통해 명령어 판별 + rs, rt → Reg file ② Branch 명령어를 대비한 주소 미리 계산</p>	Signal	Value	Bit	PCSrc	z	10	PCWrite	Write	1	Inst'Data	PC	0	MemRead	Read	1	MemWrite	Don't Write	0	IRWrite	Write	1	RegWrite	Don't Write	00	ALUSrcX	PC	00	ALUSrcY	4	00	ALUFunc	ADD	00	FnType	Arithmetic	00	Signal	Value	Bit	PCWrite	Don't Write	0	MemRead	Don't Read	0	IRWrite	Don't Write	0	DRegSel0	rs	0	DRegSel1	rt	0	ALUSrcX	PC	00	ALUSrcY	imm x4	11	ALUFunc	ADD	00	FnType	Arithmetic	00
Signal	Value	Bit																																																																	
PCSrc	z	10																																																																	
PCWrite	Write	1																																																																	
Inst'Data	PC	0																																																																	
MemRead	Read	1																																																																	
MemWrite	Don't Write	0																																																																	
IRWrite	Write	1																																																																	
RegWrite	Don't Write	00																																																																	
ALUSrcX	PC	00																																																																	
ALUSrcY	4	00																																																																	
ALUFunc	ADD	00																																																																	
FnType	Arithmetic	00																																																																	
Signal	Value	Bit																																																																	
PCWrite	Don't Write	0																																																																	
MemRead	Don't Read	0																																																																	
IRWrite	Don't Write	0																																																																	
DRegSel0	rs	0																																																																	
DRegSel1	rt	0																																																																	
ALUSrcX	PC	00																																																																	
ALUSrcY	imm x4	11																																																																	
ALUFunc	ADD	00																																																																	
FnType	Arithmetic	00																																																																	

R-Type ADD, SUB, AND, OR, XOR, NOR, ROT, SLL, SRL, SRA, SLT

3. Execute

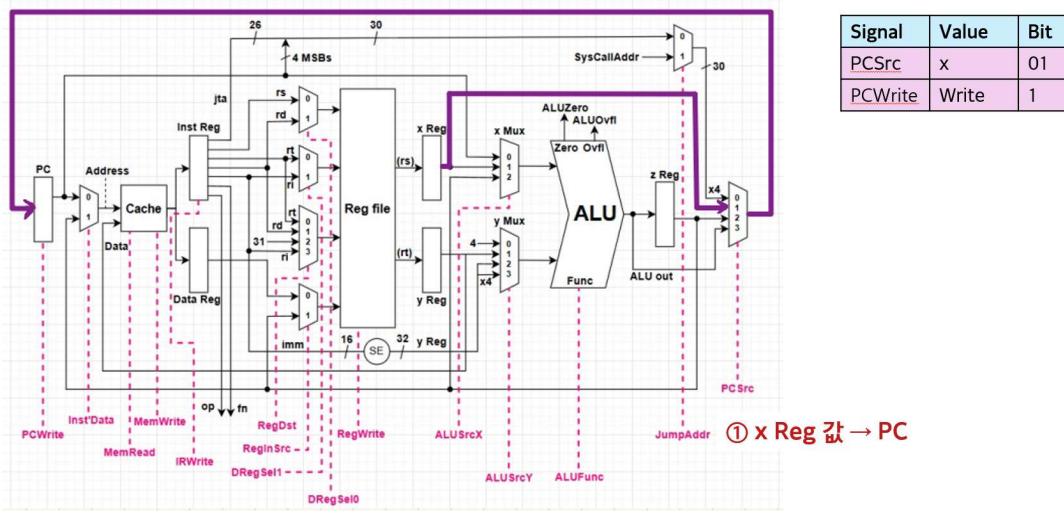


4. Write Back



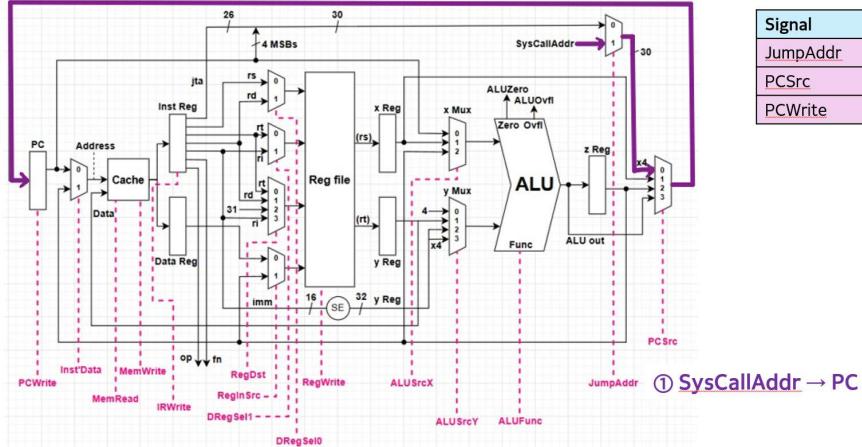
R-Type JR

3. Execute



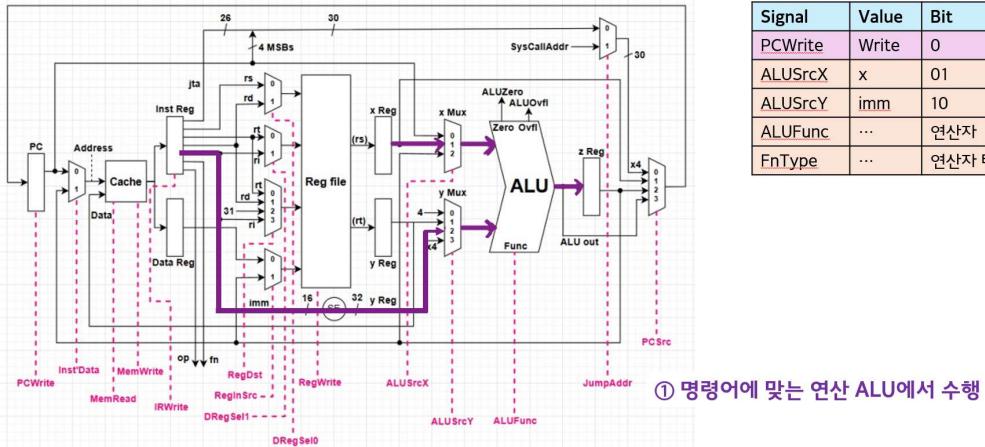
R-Type SYSCALL

3. Execute

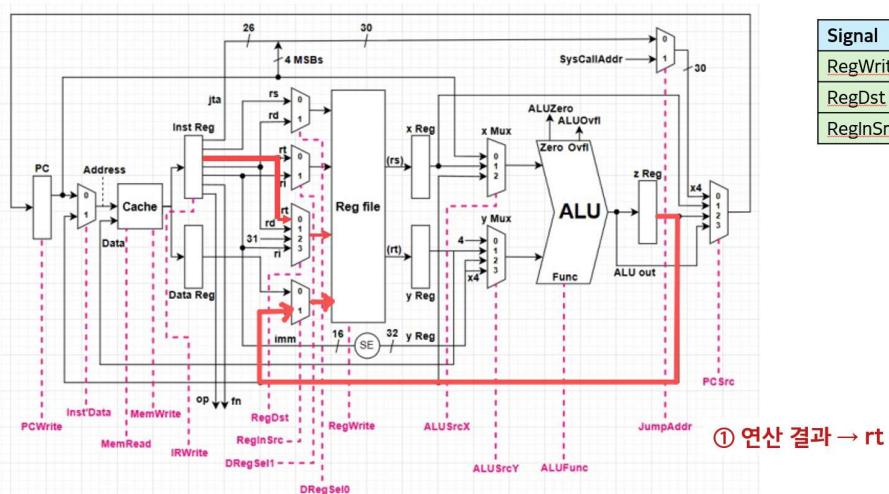


I-Type ADDI, SUBI, ANDI, ORI, XORI, SLTI

3. Execute

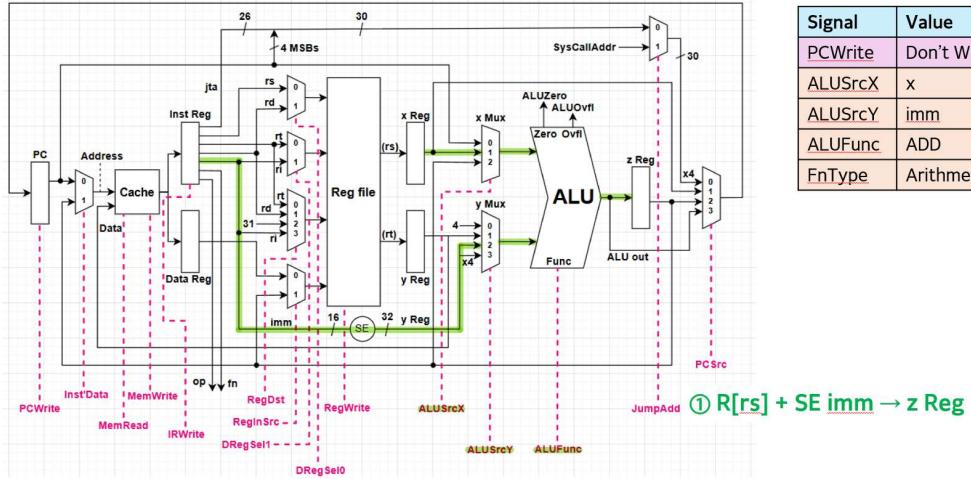


4. Write Back

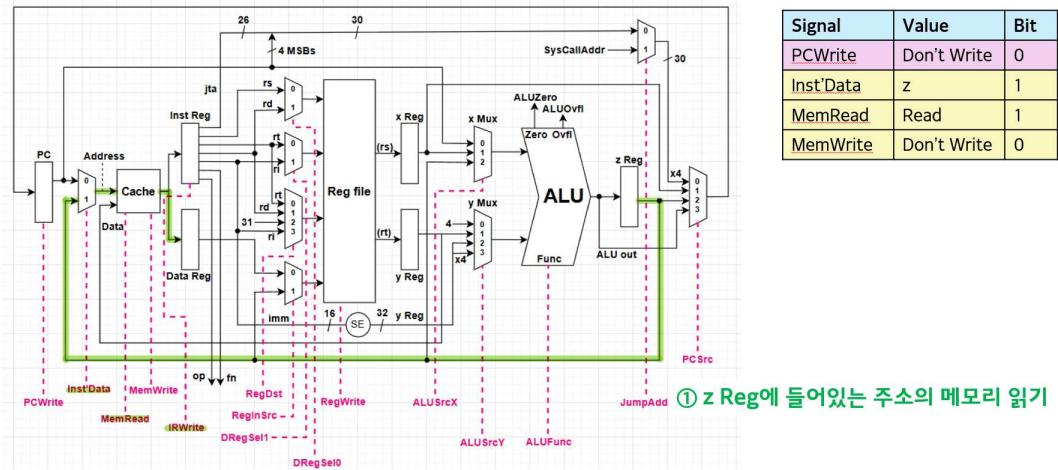


I-Type LW

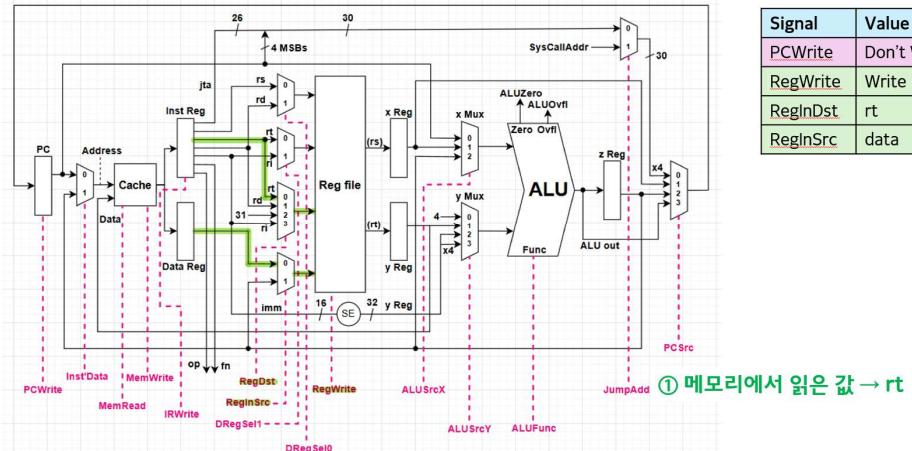
3. Execute



4. Memory Access



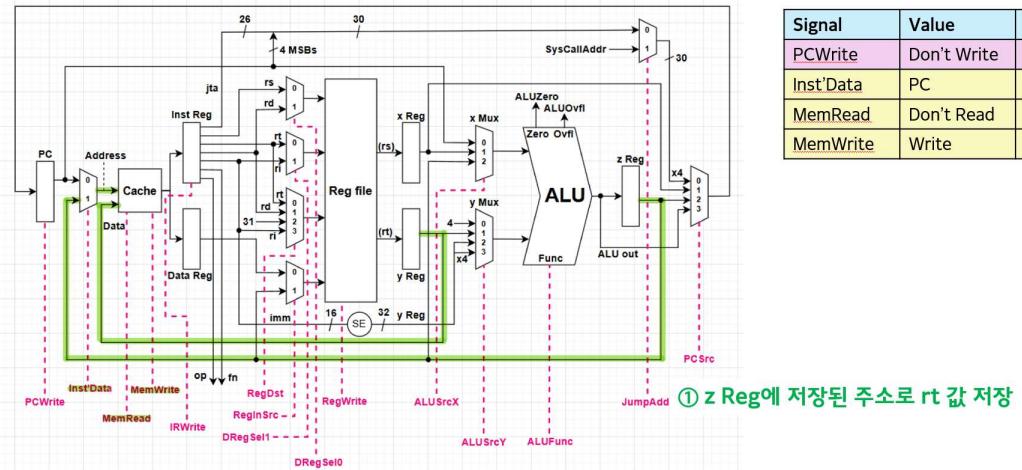
5. Write Back



I-Type SW

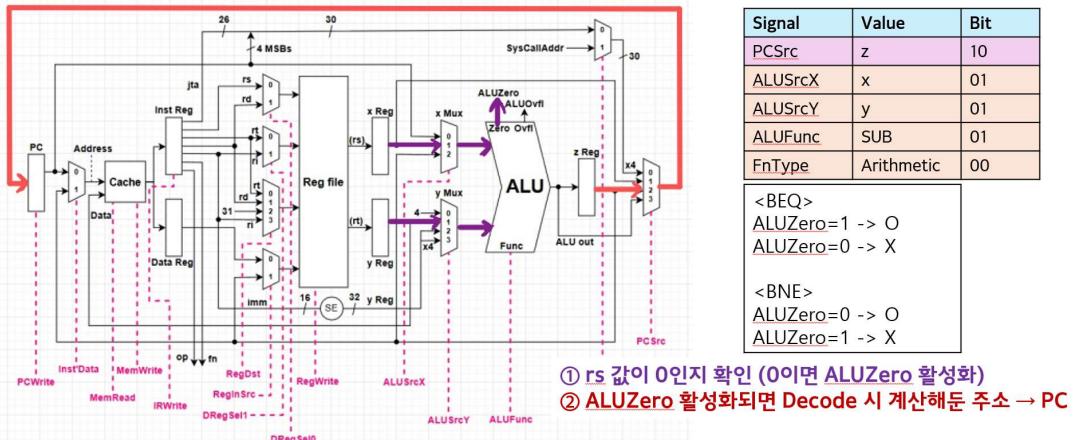
3. Execute 과정은 LW와 동일

4. Memory Access



I-Type BEQ,BNE

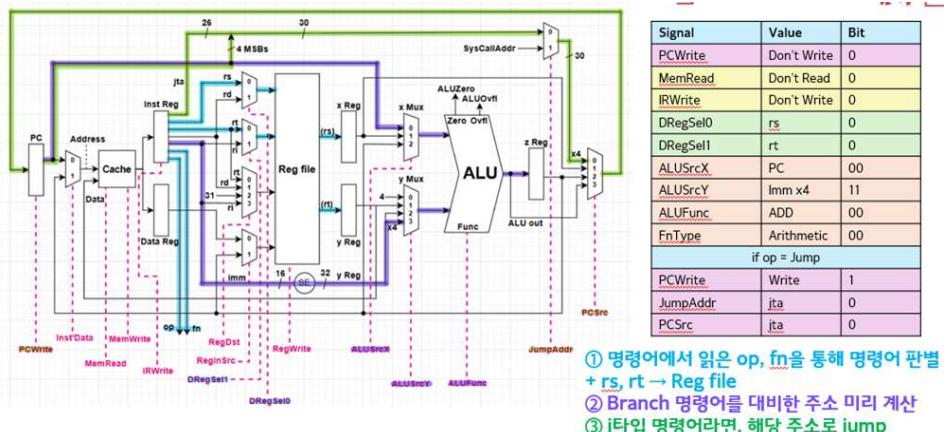
3. Execute



J-Type

(Fetch 과정은 동일)

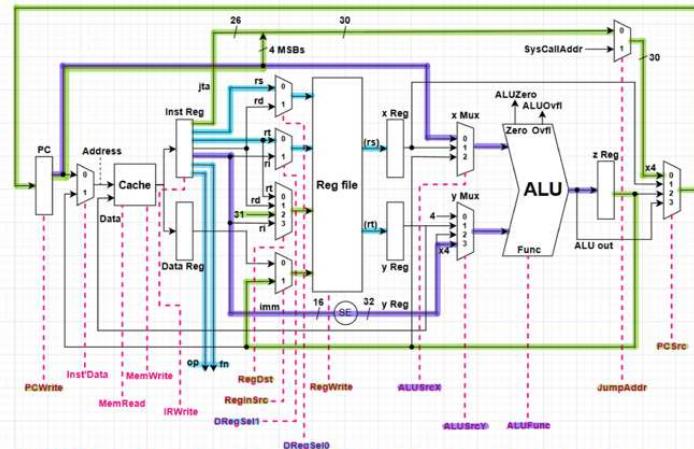
2. Decode+ Execute



J-Type JAL

(Fetch 과정은 동일)

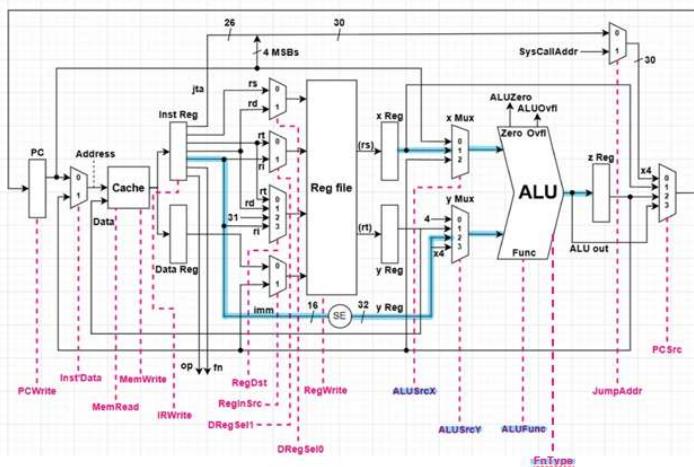
2. Decode + Execute



- ① 명령어에서 읽은 op, fn을 통해 명령어 판별 + rs, rt → Reg file
- ② Branch 명령어를 대비한 주소 미리 계산
- ③ j타입 명령어라면, 해당 주소로 jump + link를 위한 PC+4 값 31번 레지스터에 저장

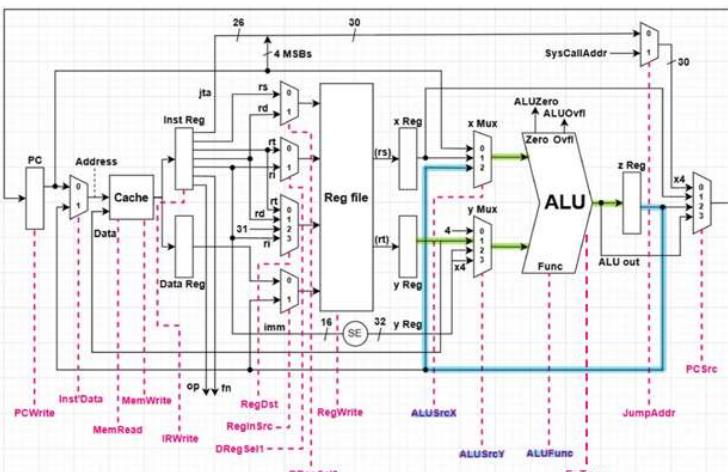
SRXO, SLXO

3. Execute



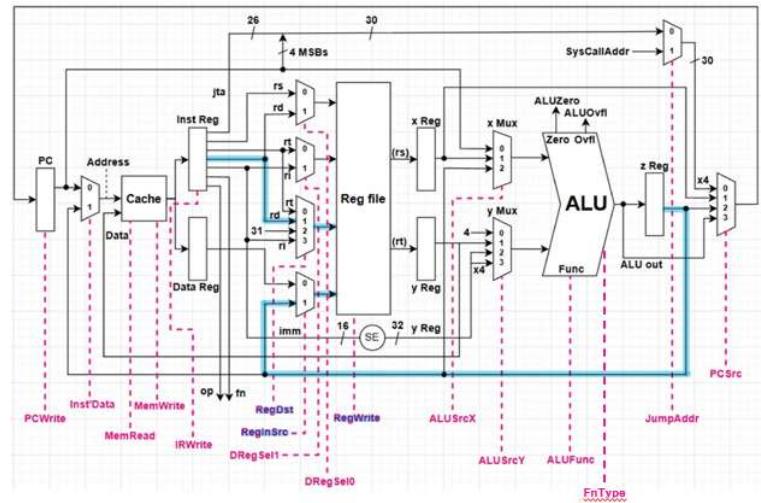
① SE imm의 하위 5 bit 만큼 rs shift

4. Execute 2



① shift한 data → x Mux
② shift한 data ⊕ rt

5. Write Back

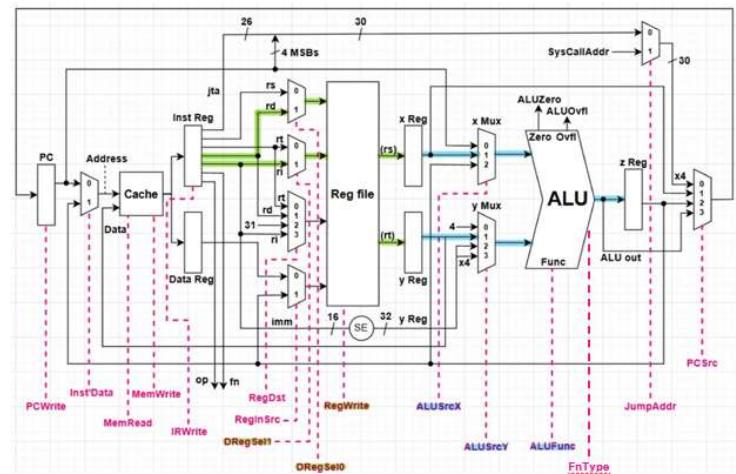


Signal	Value	Bit
RegWrite	Write	1
RegDst	rd	01
RegInSrc	z	01

① 연산 결과 → rd

D-Type DXOR

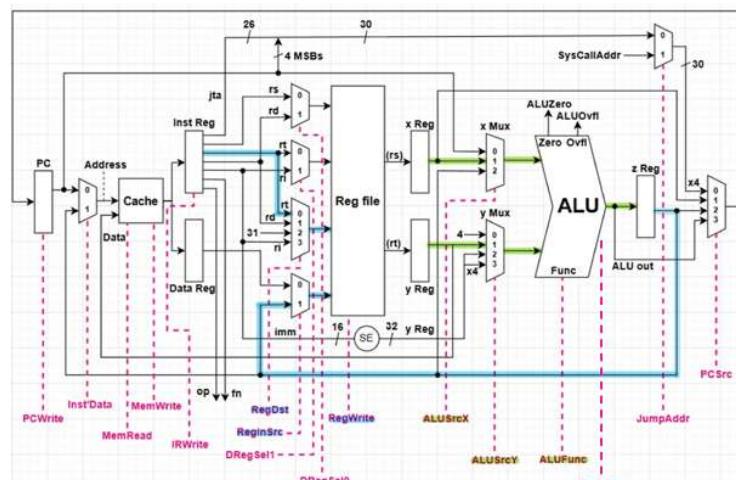
3. Execute 1 + Decode 2



Signal	Value	Bit
RegWrite	Don't Write	0
DRegSel0	rd	1
DRegSel1	ri	1
ALUSrcX	x	01
ALUSrcY	y	01
ALUFunc	XOR	10
FnType	Logic	01

① $rs \oplus rt$
② $rd, ri \rightarrow x\text{ Mux}, y\text{ Mux}$

4. Execute 2 + Write Back 1

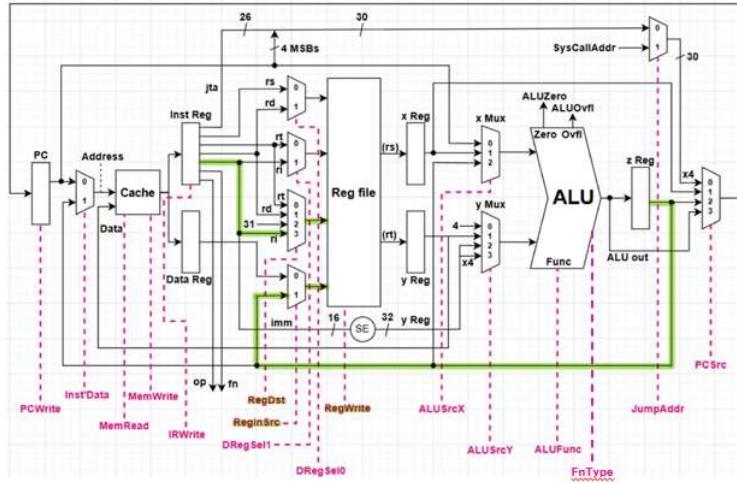


Signal	Value	Bit
RegWrite	Write	1
RegDst	rt	00
RegInSrc	z	1
ALUSrcX	rd	01
ALUSrcY	ri	01
ALUFunc	XOR	10
FnType	Logic	01

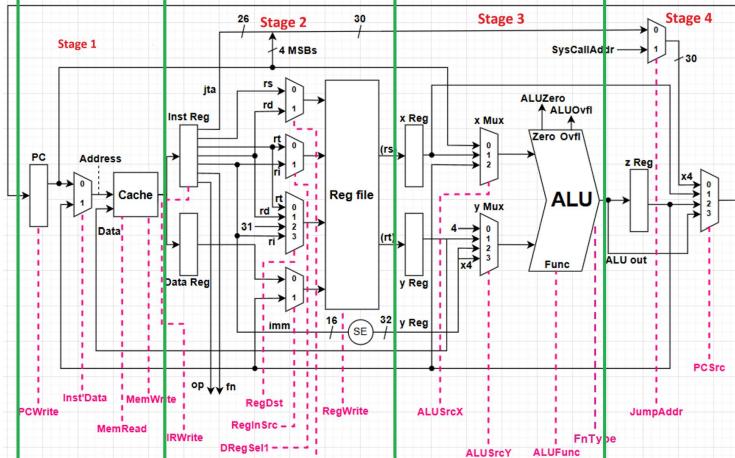
① $rd \oplus ri$
② rs, rt 연산 결과 → rt

5. Write Back 2

Signal	Value	Bit
RegWrite	Write	1
RegDst	ri	11
RegInSrc	z	1



① rd, ri 연산 결과 → ri

	<p>https://github.com/inha-kimhyunmin/COA_Project</p>  <p>(QR코드로 접속 가능)</p>
전체 코드	<p>ALU의 각 유닛과 하드웨어 전체 구조의 각 레지스터와 유닛들을 먼저 Verilog 코드로 구현 후, ALU의 각 유닛들을 합쳐서 ALU를 구성하였으며, Control State Machine에 맞게 명령어에 따라 State를 조정하고, 해당 State에 맞는 Control Signal을 출력해주는 Controller 유닛을 구성하였다.</p>
하드웨어 설계 과정	 <p>이후 전체 하드웨어 구조를 4개로 분할하여 각각 Stage의 하드웨어 유닛을 합쳐서 총 4개의 Stage를 구성하고, 이 4개의 Stage와 Controller 유닛을 합쳐서 전체 하드웨어 구조를 Verilog 코드로 완성하였다.</p>
기계어 생성 과정	<p>어셈블리어 표기 방식으로 코드를 작성 시 이를 32비트 Instruction으로 변환하는 어셈블리를 Python으로 작성하였다. 새로운 명령어인 SLXO, SRXO는 \$rd, \$rs, \$rt, \$sh 순으로 어셈블리어를 적도록 규정하였고 DXOR은 \$rs, \$rt, \$rd, \$ri 순으로 적도록 규정하였다.</p> <p>어셈블리어로 Xoshiro, Xorshift, ChaCha20, SHA-256 코드를 작성한 후, 어셈블리를 이용하여 어셈블리어 코드를 기계어로 변환하였다.</p>
테스트	<p>ALU의 각 유닛과 하드웨어 전체 구조의 각 레지스터와 유닛들을 TestBench를 작성한 다음 테스트를 진행하여 유닛의 무결성을 확인 후, 각각 Stage별로 TestBench를 작성하여 각 Stage가 정확하게 동작하는지를 확인하였다.</p> <p>이후 전체 구조의 TestBench를 작성하여 명령어에 맞게 CPU가 오류 없이 동작하는지 확인하였다. 명령어는 TestBench 작성 시 Cache Memory에 직접 입력하고, 명령어 시작 주소를 PC값 시작 값으로 지정하여 테스트를 진행하였다. 실행 결과가 명령어 실행시 예상 결과와 동일한지 확인하여 CPU 검증을 진행하였다.</p>

ChaCha20은 덧셈·XOR·로테이트 연산을 반복해 16개의 내부 상태를 갱신하고, 이를 통해 스트림 형태의 암호화용 바이트열을 생성하는 알고리즘이다. 대칭키 기반이며 빠르고 예측이 어렵다.

ChaCha20 알고리즘은 총 20 Rounds로 구성되며, 이는 10개의 Double-Rounds로 구성된다. 각 Double-Rounds는 8개의 Quarter-Rounds로 구성된다.

ChaCha20 알고리즘 의사코드를 C언어로 표현한 구문이다.

```
void chacha20_block(uint32_t state[16], uint32_t out[16]) {
    uint32_t x[16];
    memcpy(x, state, 16*4);

    for (int i = 0; i < 10; ++i) { // 10 double-rounds = 20 rounds
        // column round
        QR(x[0], x[4], x[8], x[12]);
        QR(x[1], x[5], x[9], x[13]);
        QR(x[2], x[6], x[10], x[14]);
        QR(x[3], x[7], x[11], x[15]);

        // diagonal round
        QR(x[0], x[5], x[10], x[15]);
        QR(x[1], x[6], x[11], x[12]);
        QR(x[2], x[7], x[8], x[13]);
        QR(x[3], x[4], x[9], x[14]);
    }

    for (int i=0; i<16; ++i) out[i] = x[i] + state[i]; // final add
}
```

ChaCha20

각 Quarter Round의 연산은 다음과 같다.

(<<<는 Rotate Left)

```
QR(a, b, c, d){
    a += b; d ^= a; d <<< 16
    c += d; b ^= c; b <<< 12
    a += b; d ^= a; d <<< 8
    c += d; b ^= c; b <<< 7
}
```

새로운 명령어 DXOR, ROT를 이용하여 최적화를 진행할 수 있는데, Quarter-Rounds 하나에서는 DXOR 명령어를 사용할 수 없다. 그러나 2개의 Quarter-Rounds의 연산을 순차적으로 진행하면 DXOR 명령어를 사용할 수 있다.

기존

```
x[0] += x[4]; x[12] ^= x[0] x[12] <<< 16 (첫 번째 Quarter_Rounds의 첫 번째 연산)
x[1] += x[5]; x[13] ^= x[0] x[13] <<< 16 (두 번째 Quarter_Rounds의 첫 번째 연산)
```

변경

```
x[0] += x[4]; x[1] += x[5]
x[12] ^= x[0]; x[13] ^= x[0] -> DXOR 명령어 사용 가능
x[12] <<< 16, x[13] <<< 16 -> ROT 명령어 사용 가능
```

ChaCha20 알고리즘에서 2개의 Quarter-Rounds에서 기존 CPU 구조를 사용했을 때는 ADD, XOR, SLL, SRL, OR 연산이 8번씩 사용된다. 총 클락 수는 $4 * 5 * 8 = 160$ 클락이다.

변경된 구조를 사용했을 때는 ADD 연산 8번, DXOR 연산 4번 ROT 연산 8번이 사용된다. 총 클락 수는 $4 * 8 * 2 + 5 * 8 = 84$ 클락이다.

2개의 Quarter-Rounds에서 47.5%의 클락 수 감소 효과를 확인할 수 있다.

ChaCha20 알고리즘은 총 80개의 Quarter-Rounds로 구성되며 80개의 Quarter-Rounds에서도 47.5%의 클락 수 감소 효과를 확인할 수 있을 것이다.

(Simulation까지만 회로 구현을 진행하여 Gate Delay를 확인할 수 없어 클락 수로만 성능 비교를 진행하였다.)

기존 MIPS 구조에서 동작 결과

Name		Value	1,935.000 ns
clk	1	002c00	1,940.
> dbg_ctrl[21:0]	8	002c00	
> dbg_state[3:0]	8	002c00	
> dbg_z[31:0]	21e39053	21e39053	

변경된 CPU 구조에서 동작 결과

Name		Value	1,335.000 ns
clk	1	002c00	1,340.
> dbg_ctrl[21:0]	8	002c00	
> dbg_state[3:0]	8	002c00	
> dbg_z[31:0]	21e39053	21e39053	

예상 연산 결과인 21e39053이 나오며 기존 MIPS 구조에서의 동작 결과와 변경된 CPU 구조에서의 동작 결과가 동일한 것을 확인하였으며, 실행 시간은 1935ns에서 1335ns로 기존 구조에서보다 600ns 빠르게 실행 결과를 출력하였으며 31%의 속도 개선이 이루어졌다.

VPN 소프트웨어, TLS, HTTPS 통신을 지원하는 서비스, 파일 암호화 프로그램에서 ChaCha20 알고리즘이 높은 비중으로 사용되며, 새로운 구조의 CPU로 해당 프로그램에서 성능 향상을 이룰 수 있다.

ChaCha20
전체 어셈블리
코드

변경된 구조로 작성된 코드	기존 구조로 작성된 코드
<pre> addi \$t8 \$t0 22136 addi \$t9 \$t0 17185 addi \$t12 \$t0 57072 addi \$t13 \$t0 48282 addi \$t16 \$t0 52105 addi \$t17 \$t0 57072 addi \$t20 \$t0 39871 addi \$t21 \$t0 44288 add \$t8 \$t8 \$t12 add \$t9 \$t9 \$t13 dxor \$t8 \$t20 \$t9 \$t21 addi \$t7 \$t0 16 rot \$t20 \$t20 \$t7 rot \$t21 \$t21 \$t7 add \$t16 \$t16 \$t20 add \$t17 \$t17 \$t21 dxor \$t16 \$t12 \$t17 \$t13 addi \$t7 \$t0 12 rot \$t12 \$t12 \$t7 rot \$t13 \$t13 \$t7 add \$t8 \$t8 \$t12 add \$t9 \$t9 \$t13 dxor \$t8 \$t20 \$t9 \$t21 addi \$t7 \$t0 8 rot \$t20 \$t20 \$t7 rot \$t21 \$t21 \$t7 add \$t16 \$t16 \$t20 add \$t17 \$t17 \$t21 dxor \$t16 \$t12 \$t17 \$t13 addi \$t7 \$t0 7 rot \$t12 \$t12 \$t7 rot \$t13 \$t13 \$t7 </pre>	<pre> addi \$t8 \$t0 22136 addi \$t9 \$t0 17185 addi \$t12 \$t0 57072 addi \$t13 \$t0 48282 addi \$t16 \$t0 52105 addi \$t17 \$t0 57072 addi \$t20 \$t0 39871 addi \$t21 \$t0 44288 add \$t8 \$t8 \$t12 xor \$t20 \$t20 \$t8 sll \$t22 \$t20 16 srl \$t23 \$t20 16 or \$t20 \$t22 \$t23 add \$t16 \$t16 \$t20 xor \$t12 \$t12 \$t16 sll \$t22 \$t12 12 srl \$t23 \$t12 20 or \$t12 \$t22 \$t23 add \$t8 \$t8 \$t12 xor \$t20 \$t20 \$t8 sll \$t22 \$t20 8 srl \$t23 \$t20 24 or \$t20 \$t22 \$t23 add \$t16 \$t16 \$t20 xor \$t12 \$t12 \$t16 sll \$t22 \$t12 7 srl \$t23 \$t12 25 or \$t12 \$t22 \$t23 add \$t9 \$t9 \$t13 xor \$t21 \$t21 \$t9 sll \$t24 \$t21 16 srl \$t25 \$t21 16 or \$t21 \$t24 \$t25 add \$t17 \$t17 \$t21 xor \$t13 \$t13 \$t17 sll \$t24 \$t13 12 srl \$t25 \$t13 20 or \$t13 \$t24 \$t25 add \$t9 \$t9 \$t13 xor \$t21 \$t21 \$t9 sll \$t24 \$t21 8 srl \$t25 \$t21 24 or \$t21 \$t24 \$t25 add \$t17 \$t17 \$t21 xor \$t13 \$t13 \$t17 sll \$t24 \$t13 7 srl \$t25 \$t13 25 or \$t13 \$t24 \$t25 </pre>

Xorshift는 정수 상태에 대해 논리 시프트와 XOR 연산을 순차적으로 적용하여 난수 값을 생성하는 단순 구조의 PRNG이다. 구현이 간단하고 계산 속도가 빠르다. Xorshift는 “A 변수를 비트 이동 후 A 변수와 XOR해서 A에 저장” 연산이 반복된다.

Xorshift 알고리즘은 새로운 명령어 SLXO, SRXO를 이용하여 최적화하였다.

변경된 구조로 작성된 코드	기존 구조로 작성된 코드
addi \$t1, \$t0, 10	addi \$t1, \$t0, 10
slxo \$t1, \$t1, \$t1, 13	sll \$t2, \$t1, 13
srxo \$t1, \$t1, \$t1, 17	xor \$t1, \$t1, \$t2
slxo \$t1, \$t1, \$t1, 5	srl \$t2, \$t1, 17
	xor \$t1, \$t1, \$t2
	sll \$t2, \$t1, 5
	xor \$t1, \$t1, \$t2

기존 CPU 구조에서는 ADDI 1회, SLL 3회, XOR 3회 연산되며 변경된 CPU 구조에서는 ADDI 1회, SLXO 2회, SRXO 1회 연산된다.

총 클락수는 기존 CPU 구조에서는 총 $4 * 7 = 28$ 클락이며, 변경된 구조에서는 $4 + 5 * 3 = 19$ 클락이다.

Xorshift 연산에서 32%의 클락 수 감소 효과를 확인할 수 있다.

Xorshift

기존 MIPS 구조에서 동작 결과



변경된 CPU 구조에서 동작 결과



예상 연산 결과인 0029414a가 나오며 기존 MIPS 구조에서의 동작 결과와 변경된 CPU 구조에서의 동작 결과가 동일한 것을 확인하였으며, 실행 시간은 295ns에서 205ns로 기존 구조에서보다 90ns 빠르게 실행 결과를 출력하였으며 30%의 속도 개선이 이루어졌다.

Xoshiro는 Xorshift의 개선 계열로, 로테이트와 XOR·곱셈을 포함한 상태 업데이트를 사용하여 더 나은 통계적 품질을 제공하는 PRNG이다. 길고 균일한 주기를 가진다. Xoshiro 알고리즘 의사코드를 C언어로 표현한 구문이다.

```
uint32_t xoshiro128pp(void) {
    uint32_t result = rotl(s[0] + s[3], 7) + s[0];

    uint32_t t = s[1] << 9;

    s[2] ^= s[0];
    s[3] ^= s[1];
    s[1] ^= s[2];
    s[0] ^= s[3];

    s[2] ^= t;

    s[3] = rotl(s[3], 11);

    return result;
}
```

새로운 명령어 ROT와, DXOR 명령어를 이용하여 코드를 최적화할 수 있다.

기존

```
xor $t3, $t1, $t3
xor $t4, $t2, $t4
xor $t2, $t3, $t2
xor $t1, $t4, $t1
```

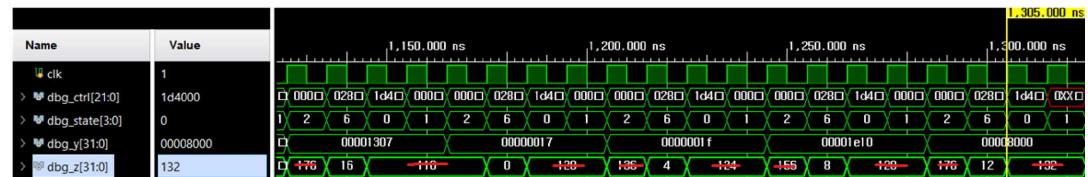
변경

```
DXOR $t1, $t3, $t2, $t4
DXOR $t3, $t2, $t4, $t1
```

Xoshiro

Load, Store 연산을 제외하고 계산하는 부분만 분석했을 때, 기존 CPU 구조를 사용하면 총 클락수는 56클락이며, 변경된 CPU 구조를 사용하면 총 클락수는 42클락이다. 변경된 CPU 구조 사용시 연산 과정에서 21%의 클락 수 감소 효과를 확인할 수 있다.

기존 MIPS 구조에서 동작 결과



변경된 CPU 구조에서 동작 결과



기존 MIPS 구조에서의 동작 결과와 변경된 CPU 구조에서 동일한 주소(Z Reg의 값)로 동일한 값(Y Reg의 값)을 저장하는 것을 확인하였으며, 실행 시간은 1305ns에서 1165ns로 기존 구조에서보다 140ns 빠르게 실행 결과를 출력하였으며 11%의 속도 개선이 이루어졌다.

	<p>시뮬레이션 프로그램에서는 몬테카를로 시뮬레이션 방법으로 시뮬레이션을 진행하는데, 몬테카를로 시뮬레이션을 진행하는 과정에서 수많은 난수 생성을 진행한다. Xorshift와 Xoshiro 알고리즘을 이용하여 난수 생성을 할 때 새로운 CPU 구조를 이용하면 시뮬레이션 프로그램에서 성능을 개선할 수 있다.</p>																												
Xoshiro 전체 어셈블리 코드	<table border="1"> <thead> <tr> <th>변경된 구조로 작성된 코드</th><th>기존 구조로 작성된 코드</th></tr> </thead> <tbody> <tr> <td>addi \$t15, \$t0, 7 addi \$t16, \$t0, 15 addi \$t17, \$t0, 23 addi \$t18, \$t0, 31</td><td>addi \$t15, \$t0, 7 addi \$t16, \$t0, 15 addi \$t17, \$t0, 23 addi \$t18, \$t0, 31</td></tr> <tr> <td>sw \$t15, 0(\$t0) sw \$t16, 4(\$t0) sw \$t17, 8(\$t0) sw \$t18, 12(\$t0)</td><td>sw \$t15, 0(\$t0) sw \$t16, 4(\$t0) sw \$t17, 8(\$t0) sw \$t18, 12(\$t0)</td></tr> <tr> <td>lw \$t1, 0(\$t0) lw \$t2, 4(\$t0) lw \$t3, 8(\$t0) lw \$t4, 12(\$t0)</td><td>lw \$t1, 0(\$t0) lw \$t2, 4(\$t0) lw \$t3, 8(\$t0) lw \$t4, 12(\$t0)</td></tr> <tr> <td>add \$t5, \$t1, \$t4</td><td>add \$t5, \$t1, \$t4</td></tr> <tr> <td>addi \$t6, \$t0, 7</td><td>sll \$t6, \$t5, 7</td></tr> <tr> <td>rot \$t7, \$t5, \$t6</td><td>srl \$t7, \$t5, 25 or \$t7, \$t6, \$t7</td></tr> <tr> <td>add \$t8, \$t7, \$t1</td><td>add \$t8, \$t7, \$t1</td></tr> <tr> <td>sll \$t9, \$t2, 9</td><td>sll \$t9, \$t2, 9</td></tr> <tr> <td>dxor \$t1, \$t3, \$t2, \$t4 dxor \$t3, \$t2, \$t4, \$t1</td><td>xor \$t3, \$t1, \$t3 xor \$t4, \$t2, \$t4 xor \$t2, \$t3, \$t2 xor \$t1, \$t4, \$t1 xor \$t3, \$t3, \$t9</td></tr> <tr> <td>XOR \$t3, \$t3, \$t9</td><td></td></tr> <tr> <td>addi \$t10, \$t0, 11</td><td></td></tr> <tr> <td>rot \$t4, \$t4, \$t10</td><td>sll \$t11, \$t4, 11 srl \$t12, \$t4, 21 or \$t4, \$t11, \$t12</td></tr> <tr> <td>sw \$t8, 16(\$t0) sw \$t1, 0(\$t0) sw \$t2, 4(\$t0) sw \$t3, 8(\$t0) sw \$t4, 12(\$t0)</td><td>sw \$t8, 16(\$t0) sw \$t1, 0(\$t0) sw \$t2, 4(\$t0) sw \$t3, 8(\$t0) sw \$t4, 12(\$t0)</td></tr> </tbody> </table>	변경된 구조로 작성된 코드	기존 구조로 작성된 코드	addi \$t15, \$t0, 7 addi \$t16, \$t0, 15 addi \$t17, \$t0, 23 addi \$t18, \$t0, 31	addi \$t15, \$t0, 7 addi \$t16, \$t0, 15 addi \$t17, \$t0, 23 addi \$t18, \$t0, 31	sw \$t15, 0(\$t0) sw \$t16, 4(\$t0) sw \$t17, 8(\$t0) sw \$t18, 12(\$t0)	sw \$t15, 0(\$t0) sw \$t16, 4(\$t0) sw \$t17, 8(\$t0) sw \$t18, 12(\$t0)	lw \$t1, 0(\$t0) lw \$t2, 4(\$t0) lw \$t3, 8(\$t0) lw \$t4, 12(\$t0)	lw \$t1, 0(\$t0) lw \$t2, 4(\$t0) lw \$t3, 8(\$t0) lw \$t4, 12(\$t0)	add \$t5, \$t1, \$t4	add \$t5, \$t1, \$t4	addi \$t6, \$t0, 7	sll \$t6, \$t5, 7	rot \$t7, \$t5, \$t6	srl \$t7, \$t5, 25 or \$t7, \$t6, \$t7	add \$t8, \$t7, \$t1	add \$t8, \$t7, \$t1	sll \$t9, \$t2, 9	sll \$t9, \$t2, 9	dxor \$t1, \$t3, \$t2, \$t4 dxor \$t3, \$t2, \$t4, \$t1	xor \$t3, \$t1, \$t3 xor \$t4, \$t2, \$t4 xor \$t2, \$t3, \$t2 xor \$t1, \$t4, \$t1 xor \$t3, \$t3, \$t9	XOR \$t3, \$t3, \$t9		addi \$t10, \$t0, 11		rot \$t4, \$t4, \$t10	sll \$t11, \$t4, 11 srl \$t12, \$t4, 21 or \$t4, \$t11, \$t12	sw \$t8, 16(\$t0) sw \$t1, 0(\$t0) sw \$t2, 4(\$t0) sw \$t3, 8(\$t0) sw \$t4, 12(\$t0)	sw \$t8, 16(\$t0) sw \$t1, 0(\$t0) sw \$t2, 4(\$t0) sw \$t3, 8(\$t0) sw \$t4, 12(\$t0)
변경된 구조로 작성된 코드	기존 구조로 작성된 코드																												
addi \$t15, \$t0, 7 addi \$t16, \$t0, 15 addi \$t17, \$t0, 23 addi \$t18, \$t0, 31	addi \$t15, \$t0, 7 addi \$t16, \$t0, 15 addi \$t17, \$t0, 23 addi \$t18, \$t0, 31																												
sw \$t15, 0(\$t0) sw \$t16, 4(\$t0) sw \$t17, 8(\$t0) sw \$t18, 12(\$t0)	sw \$t15, 0(\$t0) sw \$t16, 4(\$t0) sw \$t17, 8(\$t0) sw \$t18, 12(\$t0)																												
lw \$t1, 0(\$t0) lw \$t2, 4(\$t0) lw \$t3, 8(\$t0) lw \$t4, 12(\$t0)	lw \$t1, 0(\$t0) lw \$t2, 4(\$t0) lw \$t3, 8(\$t0) lw \$t4, 12(\$t0)																												
add \$t5, \$t1, \$t4	add \$t5, \$t1, \$t4																												
addi \$t6, \$t0, 7	sll \$t6, \$t5, 7																												
rot \$t7, \$t5, \$t6	srl \$t7, \$t5, 25 or \$t7, \$t6, \$t7																												
add \$t8, \$t7, \$t1	add \$t8, \$t7, \$t1																												
sll \$t9, \$t2, 9	sll \$t9, \$t2, 9																												
dxor \$t1, \$t3, \$t2, \$t4 dxor \$t3, \$t2, \$t4, \$t1	xor \$t3, \$t1, \$t3 xor \$t4, \$t2, \$t4 xor \$t2, \$t3, \$t2 xor \$t1, \$t4, \$t1 xor \$t3, \$t3, \$t9																												
XOR \$t3, \$t3, \$t9																													
addi \$t10, \$t0, 11																													
rot \$t4, \$t4, \$t10	sll \$t11, \$t4, 11 srl \$t12, \$t4, 21 or \$t4, \$t11, \$t12																												
sw \$t8, 16(\$t0) sw \$t1, 0(\$t0) sw \$t2, 4(\$t0) sw \$t3, 8(\$t0) sw \$t4, 12(\$t0)	sw \$t8, 16(\$t0) sw \$t1, 0(\$t0) sw \$t2, 4(\$t0) sw \$t3, 8(\$t0) sw \$t4, 12(\$t0)																												

SHA-256은 임의 길이 입력을 받아 256비트 길이의 해시값을 생성하는 암호학적 해시 함수이며, 충돌·역상·2차 역상이 사실상 불가능하도록 설계된 SHA-2 계열 알고리즘이며, 비밀번호 저장, 디지털 서명, 블록체인 등에서 널리 사용된다.

SHA-256 과정에서 내부 함수를 이용하여 상태값 8개를 업데이트 하는 과정이 있고, 내부 함수들에는 Σ_0 , Σ_1 , Ch, Maj, add, round constants $K[i]$ 가 있다. 이 중 Σ_0 , Σ_1 함수는 Rotate와 XOR 연산의 반복으로 이루어져 있다.

이 Σ_0 , Σ_1 함수 연산 과정에서 ROT 명령어와 DXOR 명령어를 이용하여 성능을 개선할 수 있다.

SHA-256

변경된 구조로 작성된 코드	기존 구조로 작성된 코드
addi \$t16, \$t0, 5	addi \$t16, \$t0, 5
addi \$t17, \$t0, 10	addi \$t17, \$t0, 10
addi \$t10, \$t0, 30	sll \$t20, \$t16, 30
rot \$t4, \$t16, \$t10	srl \$t21, \$t16, 2
addi \$t10, \$t0, 19	or \$t4, \$t20, \$t21
rot \$t5, \$t16, \$t10	sll \$t20, \$t16, 19
addi \$t10, \$t0, 10	srl \$t21, \$t16, 13
rot \$t6, \$t16, \$t10	or \$t5, \$t20, \$t21
addi \$t10, \$t0, 26	sll \$t20, \$t16, 10
rot \$t7, \$t17, \$t10	srl \$t21, \$t16, 22
addi \$t10, \$t0, 21	or \$t6, \$t20, \$t21
rot \$t8, \$t17, \$t10	sll \$t20, \$t17, 26
addi \$t10, \$t0, 7	srl \$t21, \$t17, 6
rot \$t9, \$t17, \$t10	or \$t7, \$t20, \$t21
dxor \$t4, \$t5, \$t7, \$t8	sll \$t20, \$t17, 21
dxor \$t6, \$t5, \$t9, \$t8	srl \$t21, \$t17, 11
	or \$t8, \$t20, \$t21
	sll \$t20, \$t17, 7
	srl \$t21, \$t17, 25
	or \$t9, \$t20, \$t21
	xor \$t18, \$t4, \$t5
	xor \$t19, \$t7, \$t8
	xor \$t18, \$t18, \$t6
	xor \$t19, \$t19, \$t9

(다른 알고리즘에서는 코드로 작성한 부분이 알고리즘 전체를 나타내어서 암달의 법칙을 사용하지 않았지만, SHA-256 연산에서는 코드로 작성한 부분이 알고리즘의 일부분이므로 암달의 법칙을 통해 성능 개선치를 계산하였다.)

SHA-256 내부 연산을 명령어 단위로 분석한 결과, 64회 반복되는 라운드 연산은 대시그마 22 instr, Ch 4 instr, Maj 5 instr, ADD 7 instr, 상태 갱신 8 instr로 총 46 instr이 필요하며, 이를 모두 합하면 2,944 instr이 수행된다. 메시지 스케줄은 48회 반복되며, σ 함수(ROTR·SHR·XOR) 18 instr와 ADD 3 instr로 반복당 21 instr이 필요하여 총 1,008 instr이 수행된다. 전체 SHA-256 처리 과정은 총 3,952 instr로 구성되며, 이 중 대시그마 1,408 instr가 전체의 35.63%를 차지한다. 대시그마 구간의 실행 시간이 약 1.8배 개선된다고 가정하면, 암달의 법칙에 따라 전체 성능 향상은 $1 / (0.6437 + 0.3563 / 1.44) =$ 약 1.122배로 계산되어, 변경된 CPU 구조 사용 시 전체

성능이 약 12.2% 향상된다.

기존 MIPS 구조에서 동작 결과

Name	Value	975.000 ns
clk	1	980.00
> dbg_ctrl[21:0]	002c00	002c00
> dbg_state[3:0]	8	8
> dbg_z[31:0]	29400500	29400500

변경된 CPU 구조에서 동작 결과

Name	Value	675.000 ns
clk	1	680.00
> dbg_ctrl[21:0]	003c00	003c00
> dbg_state[3:0]	c	c
> dbg_z[31:0]	29400500	29400500

예상 연산 결과인 29400500이 나오며 기존 MIPS 구조에서의 동작 결과와 변경된 CPU 구조에서의 동작 결과가 동일한 것을 확인하였으며, 실행 시간은 975ns에서 675ns로 300ns 빠르게 실행 결과를 출력하였으며 31%의 속도 개선이 이루어졌다.