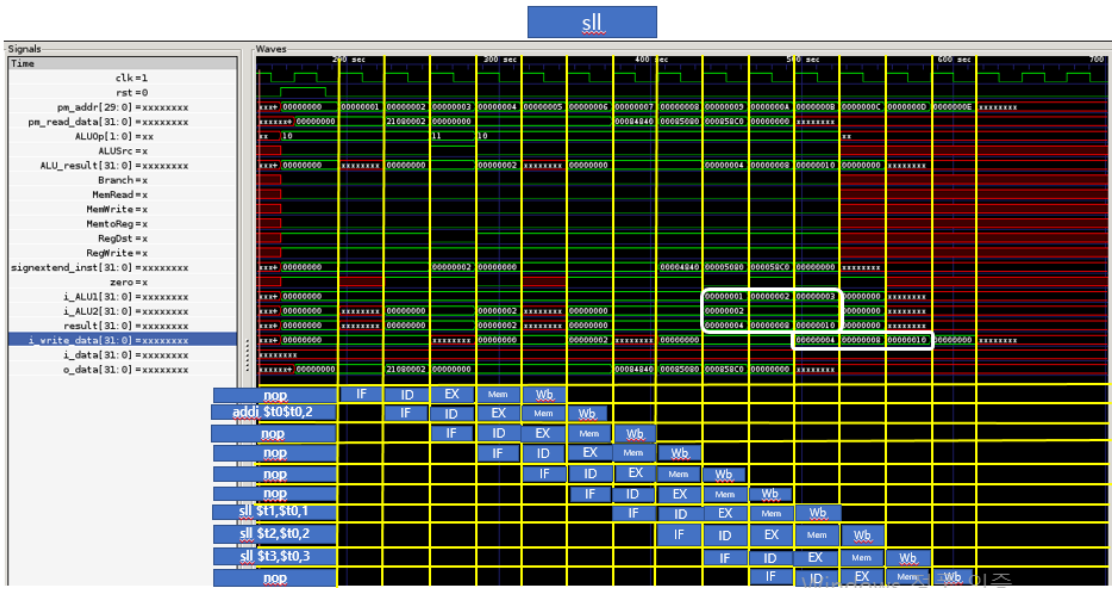
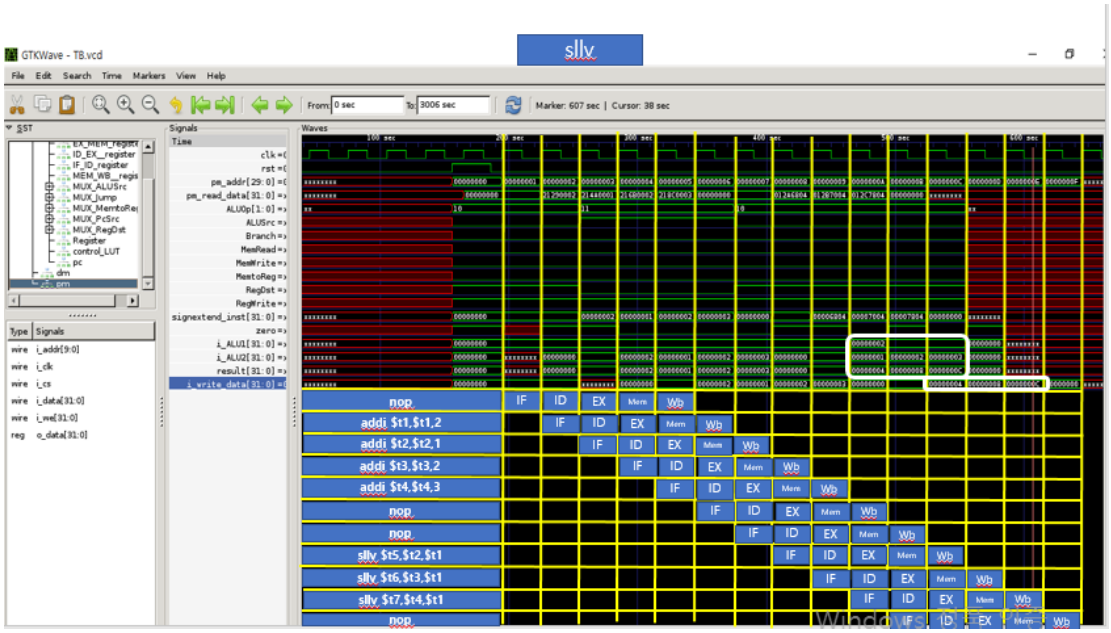
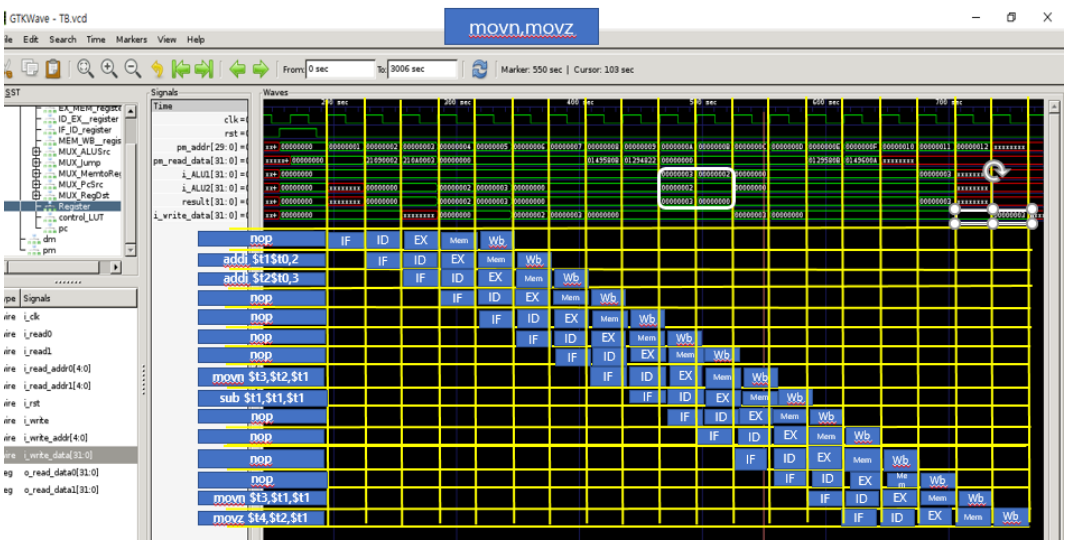
설계자	장윤봉 (2012122249), 오혜빈 (2015124129) 이다현 (2015124144) ; 5조 (작성요령, 작성예) 에 따라 작성 요령 및 작성 예에 대한 설명을 삭제하고 간단히 작성함.
Test Vector 1	어셈블리소스코드: hazard를 고려하지 않은 SLL(Shift Word Left Logical) 동작을 확인하기 위한 code nop addi $t0,$t0,2 // sll 검증을 위하여 shift left될 값을 $t0에 넣어준다. nop // 4개의 nop은 bypassing을 고려하지 않기 위함 nop nop nop sll $t1,$t0,1 // $t0(=2)의 값을 1bit 만큼 shhif left 시킨 후 $t1에 저장한다. sll $t2,$t0,2 // $t0(=2)의 값을 2bit 만큼 shhif left 시킨 후 $t2에 저장한다. sll $t3,$t0,3 // $t0(=2)의 값을 3bit 만큼 shhif left 시킨 후 $t3에 저장한다. nop 예상 결과 및 확인 방법: $t0의 값을 shift left 시키기 때문에 x2(1bit shift left),x4(2bit shift left),x8(3bit shift left)되는 결과를 얻을 수 있다. 따라서 $t1에는 2x2=4, $t2에는 2x4=8, $t3에는 2x8=16이 저장 될 것이다. 이는 $t1, $t2, $t3의 WB stage에서 update되는 값(register write 값)을 확인하여 검증한다. 결과:  $t1에는 2x2=4, $t2에는 2x4=8, $t3에는 2x8=16이 각각 EX 단계의 ALU에서 계산 된 걸 확인하였고 (0x00000004=32'd4, 0x00000008=32'd8, 0x00000010=32'd16), 각각 WB 단계에서 $t1, $t2, $t3에 shift left 결과 값이 update된 걸 확인 하였다. (하얀색 BOX)
Test Vector 2	어셈블리소스코드: hazard를 고려하지 않은 SLLV(Shift Word Left Logical Variable) 동작을 확인하기 위한 code nop addi $t1,$t1,2 // sllv 검증을 위하여 shift left 될 값 2를 $t1에 넣어준다. addi $t2,$t2,1 // 1bit를 shift left 할 수 있게 $t2에 1을 저장한다. addi $t3,$t3,2 // 2bit를 shift left 할 수 있게 $t2에 1을 저장한다. addi $t4,$t4,3 // 3bit를 shift left 할 수 있게 $t2에 1을 저장한다. nop nop sllv $t5,$t2,$t1 // $t2(=1)를 $t1(=2) bit 만큼 shift left 시킨 후 $t5에 저장한다. sllv $t6,$t3,$t1 // $t3(=2)를 $t1(=2) bit 만큼 shift left 시킨 후 $t6에 저장한다. sllv $t7,$t4,$t1 // $t4(=3)를 $t1(=2) bit 만큼 shift left 시킨 후 $t7에 저장한다.

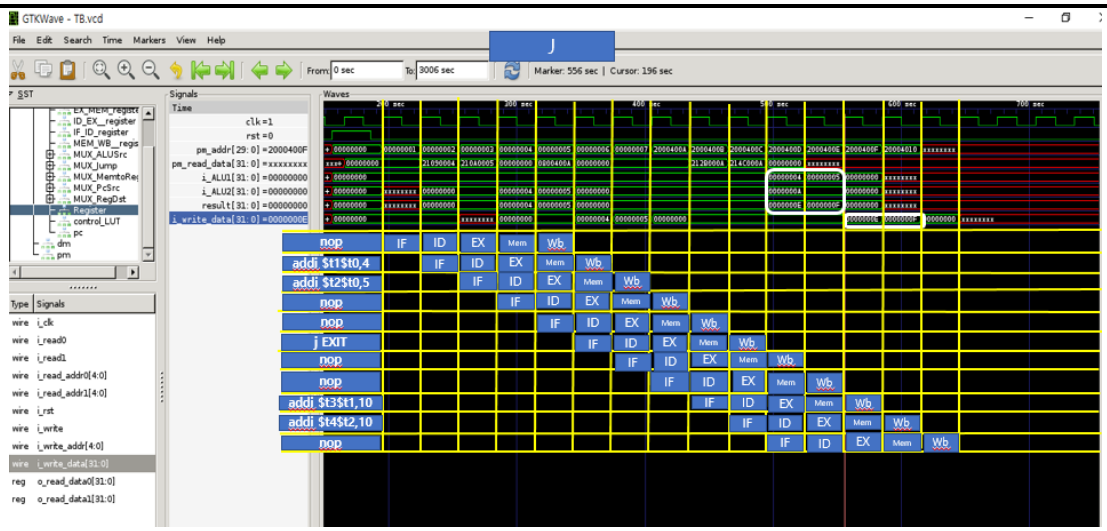
	nop 예상 결과 및 확인 방법: $t2(=1), $t3(=2), $t4(=3)를 $t1(=2) bit 만큼 shift left 시키기 때문에 각 resgiter 값에 x4를 한 결과가 나올 것이다. 따라서 $t5에는 1x4=4, $t6에는 2x4=8, $t7에는 3x4=12이 저장 될 것이다. 이는 $t5, $t6, $t7의 WB stage에서 update되는 값(register write 값)을 확인하여 검증한다. 결과:  $t5에는 1x4=4, $t6에는 2x4=8, $t7에는 3x4=12이 각각 EX 단계의 ALU에서 계산 된 걸 확인하였고 (0x00000004=32'd4, 0x00000008=32'd8, 0x0000000C=32'd12), 각각 WB 단계에서 $t5, $t6, $t7에 shift left 결과 값이 update된 걸 확인 하였다. (하얀색 BOX)
Test Vector 3	어셈블리소스코드: hazard를 고려하지 않은 MOVN(Move Conditional on Not Zero), MOVZ(Move Conditional on Zero) 동작을 확인하기 위한 code nop addi $t1,$t0,2 // move 여부 확인하기 위해 parameter를 설정, $t1=2. addi $t2,$t0,3 // move 여부 확인하기 위해 parameter를 설정, $t1=3. nop // bypassing을 고려하지 않았기 때문에 넣어준 4개의 nop nop nop nop movn $t3,$t2,$t1 // $t1=2이므로(not zero) $t3이 $t2(=3)로 update 된다. sub $t1,$t1,$t1 // $t1=0으로 update nop // bypassing을 고려하지 않았기 때문에 넣어준 4개의 nop nop nop nop movn $t3,$t1,$t1 // $t1=0이므로(zero) $t3이 update 되지 않는다. movz $t4,$t2,$t1 // $t1=0이므로(zero) $t4가 $t2(=3)으로 update된다. 예상 결과 및 확인 방법: $t1의 값이 변함에 따라 move 가 결정되는 code 이다. 처음 movn은 $t1값이 2이기 때문에 (not zero) $t3의 값이 $t2(=3)으로 update될 것이며. Sub를 거쳐 $t1=0이 되고 movn 에서 $t1은 zero이기 때문에 movn에 의해 move 하지 않을 것이다. 마지막으로 movz에 의해 $t4에 $t2(=3)이 update 될 것이다. 결과:

<table>
<tr><td></td><td> 

처음 movn에서$t1이므로(not zero) $t3의 값이 $t2(=3)로 저장된다. .sub를 통해 $t1이 0으로 update되고 마지막 movn에서 $t1(=0)이므로 movn은 무시하고 movz에서는 $t1(=0)이므로 $t4에 $t2(=3)이 update되었다. 각각의 EX단계와 wb 단계에서 WB 됨을 확인하였다.(하얀 BOX)</td></tr>
<tr><td>Test Vector 4</td><td>어셈블리소스코드: Control Hazard를 고려하지 않은 j 명령어 동작을 확인하기 위한 code
nop
addi $t1,$t0,4 // j 검증을 위한 register 값 $t1=0+4 설정
addi $t2,$t0,5 // j 검증을 위한 register 값 $t1=0+5 설정
nop // bypassing을 고려하지 않았기 때문에 넣어준 nop
nop
j EXIT // jump 하여 EXIT 레이블로 제어를 옮김
nop // Control hazard를 고려하지 않았기 때문에 넣어준 nop
nop
addi $t3,$t1,5
addi $t4,$t2,5
EXIT:
addi $t3,$t1,10
addi $t4,$t2,10
nop

예상 결과 및 확인 방법: $t1과 $t2에 각각 4와 5값을 넣어준 후 jump를 하여 EXIT 레이블로 branch 하기 때문에 addi $t3,$t1,5와 addi $t4,$t2,5가 수행되지 않고 아래의 addi $t3,$t1,10과 addi $t4,$t2,10 만 수행되어 $t3와 $t4에는 5가 더해지는 것이 아니라 각각 10이 더해 질 것이며 $t3에는 14, $t4에는 15가 저장 될 것이다. 검증은 ALU_result값이 5를 더한것인지 10을 더한것인지 확인하였고 각각의 WB stage에서 update 되는 값을 확인하여 j 명령어 작동을 검증하였다.

결과:</td></tr>
</table>



Gtkwave에 addi $t3,$t1,5와 addi $t4,$t2,5가 수행되지 않고 바로 addi $t3,$t1,10과 addi $t4,$t2,10 만 수행되어 ALU_Result 값이 14,15 (0x0000000E=32'd14, 0x0000000F=32'd15) 출력되고 각각의 WB stage에서 $t3=14, $t4=15 값이 update 되는 것을 확인하였다.

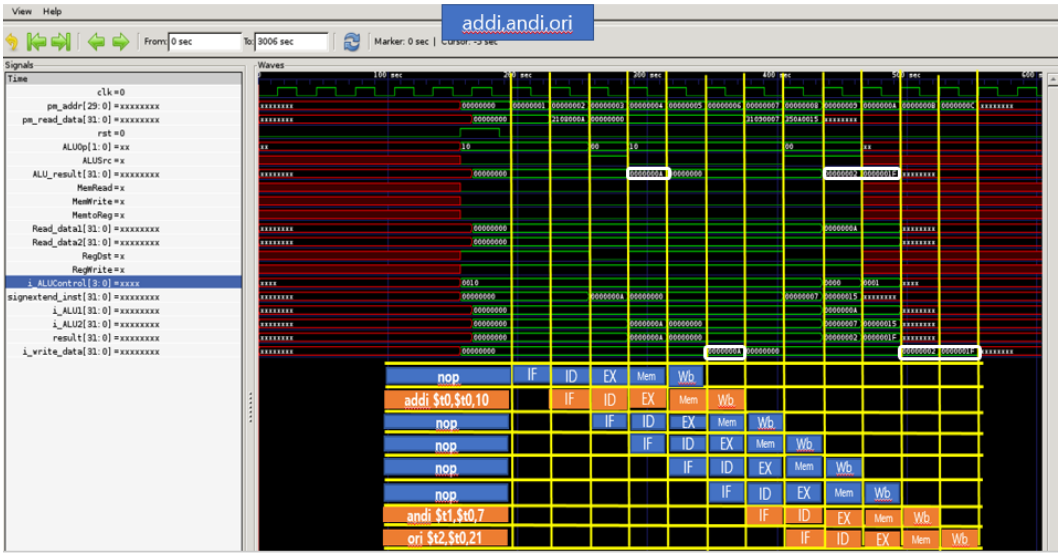
Test Vector 5

어셈블리소스코드: hazard를 고려하지 않은 addi, andi, ori 동작을 확인하기 위한 code .

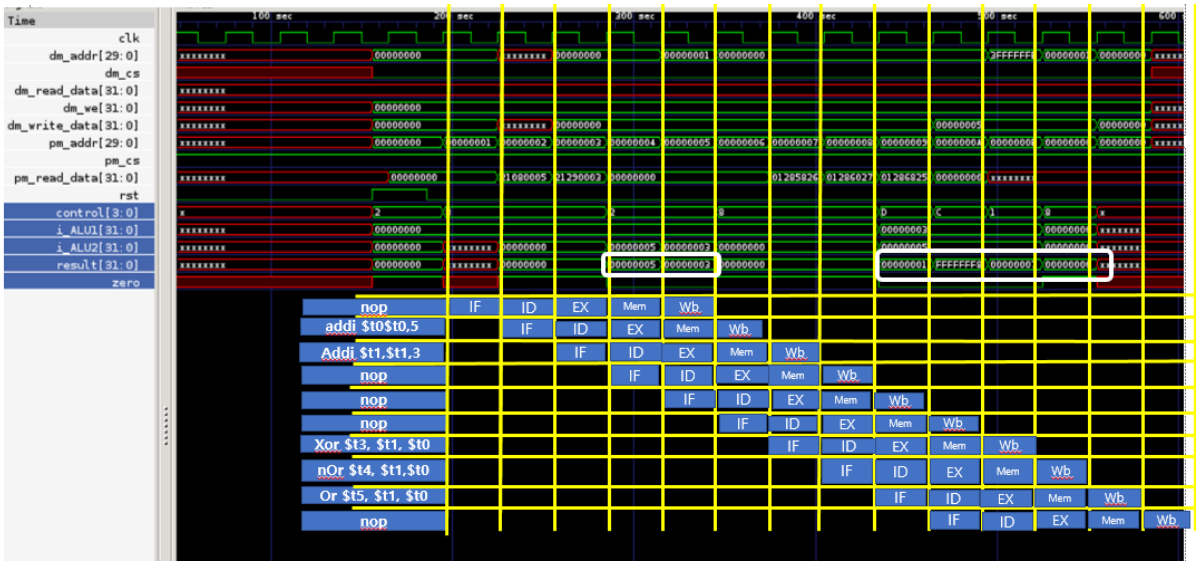
```
nop
addi $t0,$t0,10 // $t0=10
nop             // bypassing을 고려하지 않았기에 넣어준 nop
nop
nop
nop
andi $t1,$t0,7 // decimal 10과 7을 32 bit binary로 bitwise and연산하여 $t1에 넣어준다.
ori $t2,$t0,21 // decimal 10과 21을 32 bit binary로 bitwise or연산하여 $t2에 넣어준다.
```

예상 결과 및 확인 방법: addi의 경우 EX stage에서 $t0=0+10=10으로 계산되어 WB stage에서 update 될 것이고, $t0(=10)과 7을 bitwise and연산하여 $t1에 넣어주기 때문에 $t1=2가 될 것이다. 또한 $t0(=10)과 21을 32 bit binary로 bitwise or연산하여 $t2에 넣어주기 때문에 $t2=31이 될 것이다. 이는 각각의 EX stage와 WB stage의 값을 보고 확인 할 수 있다.

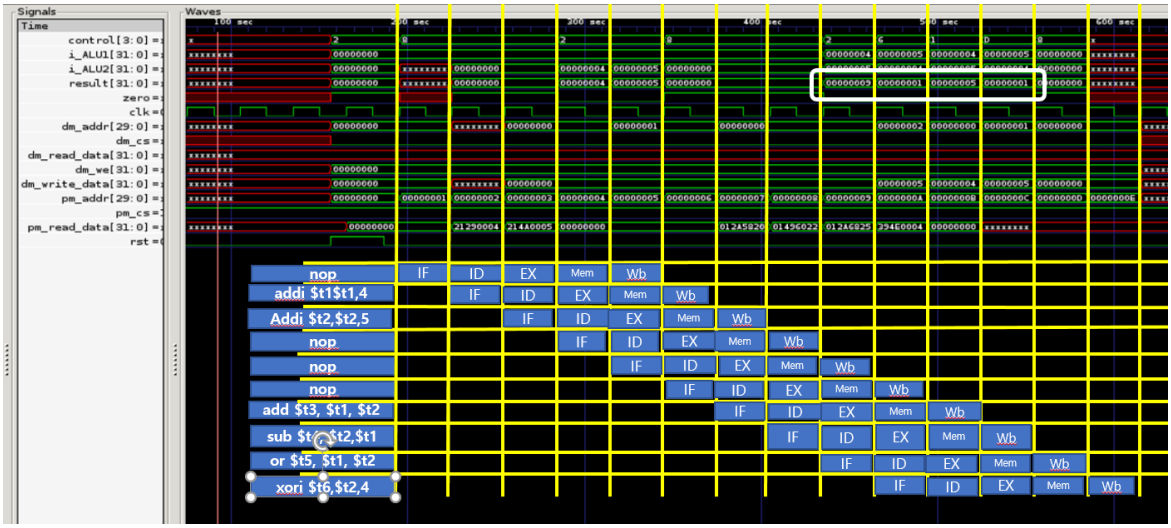
결과:

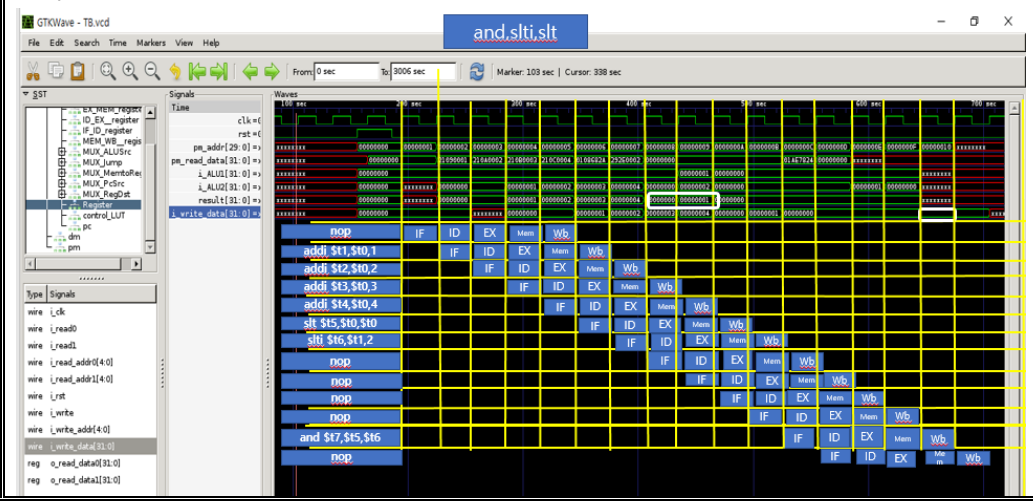


Addi로 $t0 값이 10(0x0000000A)계산되고(EX) $t0 값이 새롭게 update 됨(WB)을 확인하였다.
그리고 andi로 인해 $t1이 2(0x00000002)로 계산되고 $t1 값이 새롭게 update 됨(WB)을 확인하였다.
또한 ori로 인해 $t2가 31(0x0000001F)로 계산되고 $t2 값이 새롭게 update 됨(WB)을 확인하였다

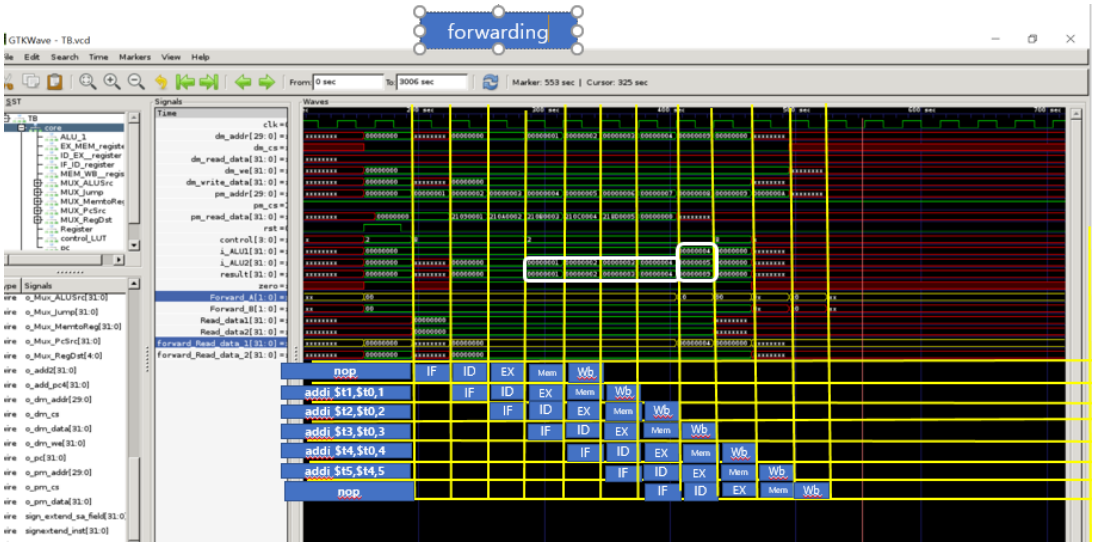
Test Vector 6	어셈블리소스코드: Hazard를 고려하지 않은 xor, or, nor 연산
	Nop
	Addi $t0,$t0,5
	Addi $t1,$t1,3
	Nop
	Nop
	Nop // bypassing을 고려하지 않았기에 넣어준 nop
	Xor $t3,$t1,$t0 // decimal 5와 3을 32bit binary로 bitwise xor연산하여 $t3에 넣어준다
	Nor $t4,$t1,$t0 // decimal 5와 3을 32bit binary로 bitwise nor연산하여 $4에 넣어준다
	Or $t5,$t1,$t0 // decimal 5와 3을 32bit binary로 bitwise or연산하여 $t5에 넣어준다
	Nop
	
	예상 결과 및 확인 방법: $t0와 $t1 register에 있는 data 값인 5와 3의 연산으로 xor,or,nor instruction의 ex 단계에서 연산 값들을 볼 수 있다. 또한 i_ALU1과 i_ALU2 port에서 instruction에서 필요한 data 값들이 연결되어 있는 것을 볼 수 있다.
	
	결과:
	 
	$t3에 xor 연산되어 0x00000001 $t4에 nor 연산되어 0xFFFFFFF8 $t5에 nor 연산되어 0x00000007이 연산되어 들어간 것을 확인할 수 있었다.

Test Vector 7	어셈블리소스코드: hazard를 고려하지 않은 add sub or xori 연산
	nop
	addi $t1,$t1,4
	addi $t2,$t2,5
	nop
	nop
	nop
	add $t3 ,$t1,$t2
	sub $t4,$t2,$t1
	or $t5,$t1,$t2
	xori $t6,$t2,4 // 연산을 수행한다
	nop
	
	예상 결과 및 확인 방법: 4와 5를 2진수로 나타내면 100과 101 이므로 $t3에 4+5=9의 값이 $t4에는 5-4=1의 값인 0x00000001이 $t5에는 5와 4를 or 한 5인 0x00000005의 값과 $t6에는 5와 4를 xori한 1인

	0x00000001의 값이 들어가 있게 된다 결과:  $t3에 4+5=9의 값이 $t4에는 5-4=1의 값이 $t5에는 5와 4를 or 한 5의 값과 $t6에는 5와 4를 xori한 1의 값이 들어가 있게 된다

Test vector 8	어셈블리소스코드: and와 slt(Set on Less Than),.slti의 동작을 검증하기 위한 code
	nop
	addi $t1,$t0,1
	addi $t2,$t0,2
	addi $t3,$t0,3
	addi $t4,$t0,4
	slt $t5,$t0,$t0 // slt의 동작검증, $t5<-0
	slti $t6,$t1,2 // slti의 동작검증, $t6<-1
	nop
	nop
	nop
	nop
	and $t7,$t5,$t6 // 1과 0을 and 하여 나온 결과값을 $t7에 저장
	nop
	
	예상 결과 및 확인 방법 : slt $t5,$t0,$t0에서 $t0는 0이므로 $t5=0이 될것이다. slti $t6,$t1,2에서 $t1(=1)이므로 $t6=1이 될 것이다. EX Stage 와 WB Stage에 update되는 값을 확인하여 검증을 확인한다.
	결과:
	 

	결과적으로 and에 의해 1과 0이 and된 0이 $t7에 update 됨을 확인 할 수 있었다.

Test Vector 9	어셈블리소스코드: forwarding을 검증하기위한 code
	nop
	addi $t1,$t0,1
	addi $t2,$t0,2
	addi $t3,$t0,3
	addi $t4,$t0,4
	addi $t5,$t4,5 // bypassing에 의해 $t5=9로 알맞게 저장 될 것이다.
	nop
	
	예상 결과 및 확인 방법 : bypassing에 의해 $t5=9로 알맞게 저장 될 것이다.
	결과:
	 
	
	하얀색 BOX 오른쪽에 보면 addi $t5,$t4,5 의 EX 단계에서 계산이 예상한 것과 마찬가지로 알맞게 되는 것을 알 수 있다.