Computer Architecture

SingleCycle

날 짜 : 2019. 04. 13

교수님 : 이성원 교수님

학 과 : 컴퓨터정보공학부

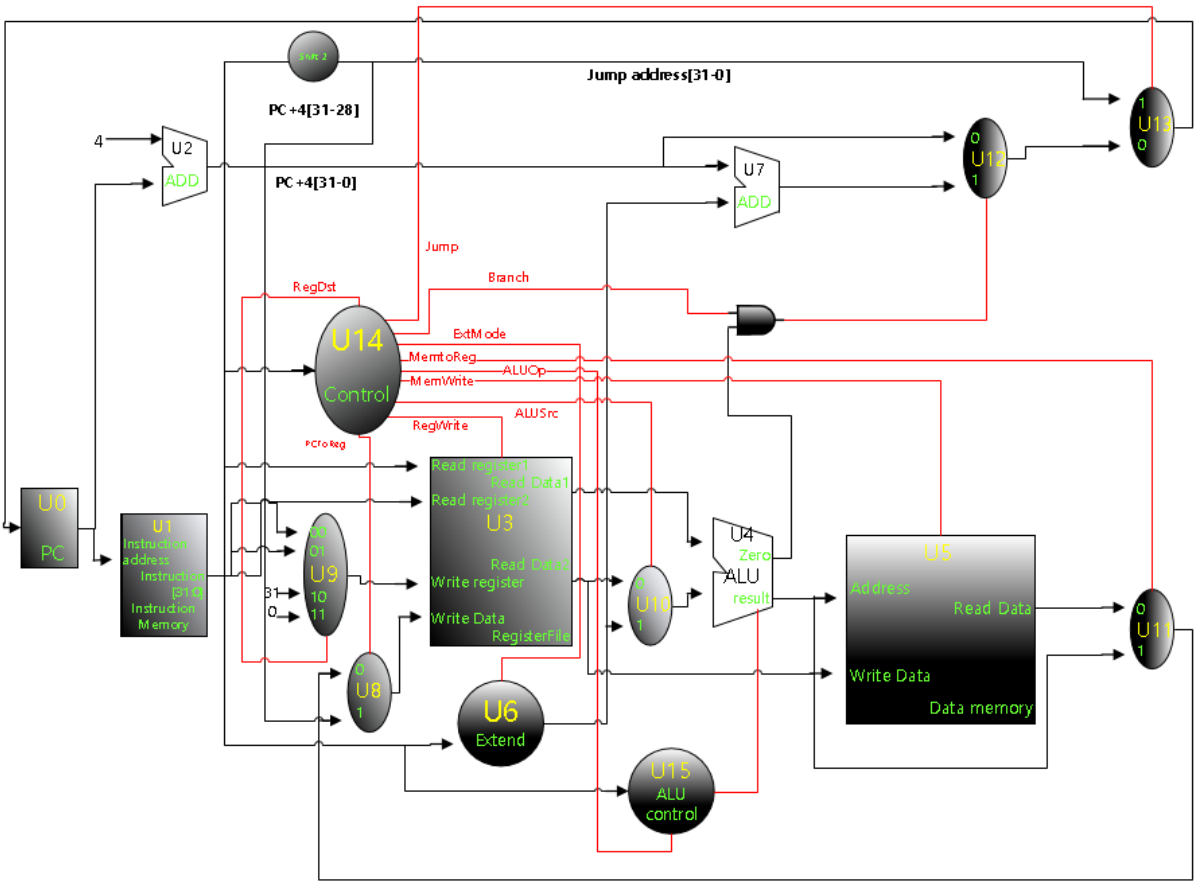
학 번 : 2015722025

이 름 : 정 용 훈

1. 문제의 해석 및 해결 방향

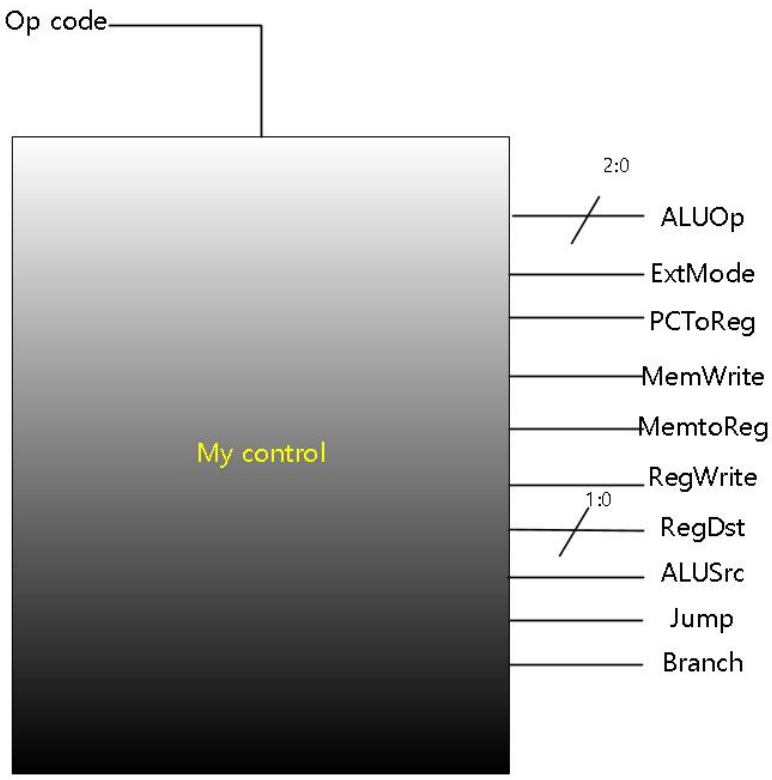
A. 실험 내용에 대한 설명

해당 프로젝트는 Single Cycle에서의 명령이 어떻게 동작하며, 이해하고 구현하는데 목적을 가지고 있다. 우선 주어진 코드에 Controltop은 Maincontrol과 Mycontrol로 구성되어있으며, Maincontrol에서 구현되지 않은 명령어는 권한을 Mycontrol로 넘겨주며, Mycontrol을통해 명령이 수행될 수 있도록 설계가 되어있다. Maincontrol에는 load, store, jump instruction들이 구현되어 있으며, 설계하게 될 Mycontrol에는 ADDI, XORI, BEQ, SLTI, JAL instruction들이 동작할 수 있는 알맞은 signal값들을 정의 해주어야 한다.



위 회로는 Project를 위한 Single cycle block이다. 구현한 Mycontrol을 통하여 명령어들이 잘 동작할 수 있어야 한다. 아래는 구현한 Mycontrol에 대한 간단한 그림이다.

My control



Maincontrol에서 명령이 수행되지 않는다면, Mycontrol로 op이 넘어오며, 조건을 통하여 Single Cycle block으로 각각의 명령이 나가며, 명령에 대하여 필요한 block들이 실행이 된다. 구현 방법으로는 간단하게 조건 명령 If를 통하여 op코드를 판별하여, 각각의 signal의 상태를 판단하게 해주었다.

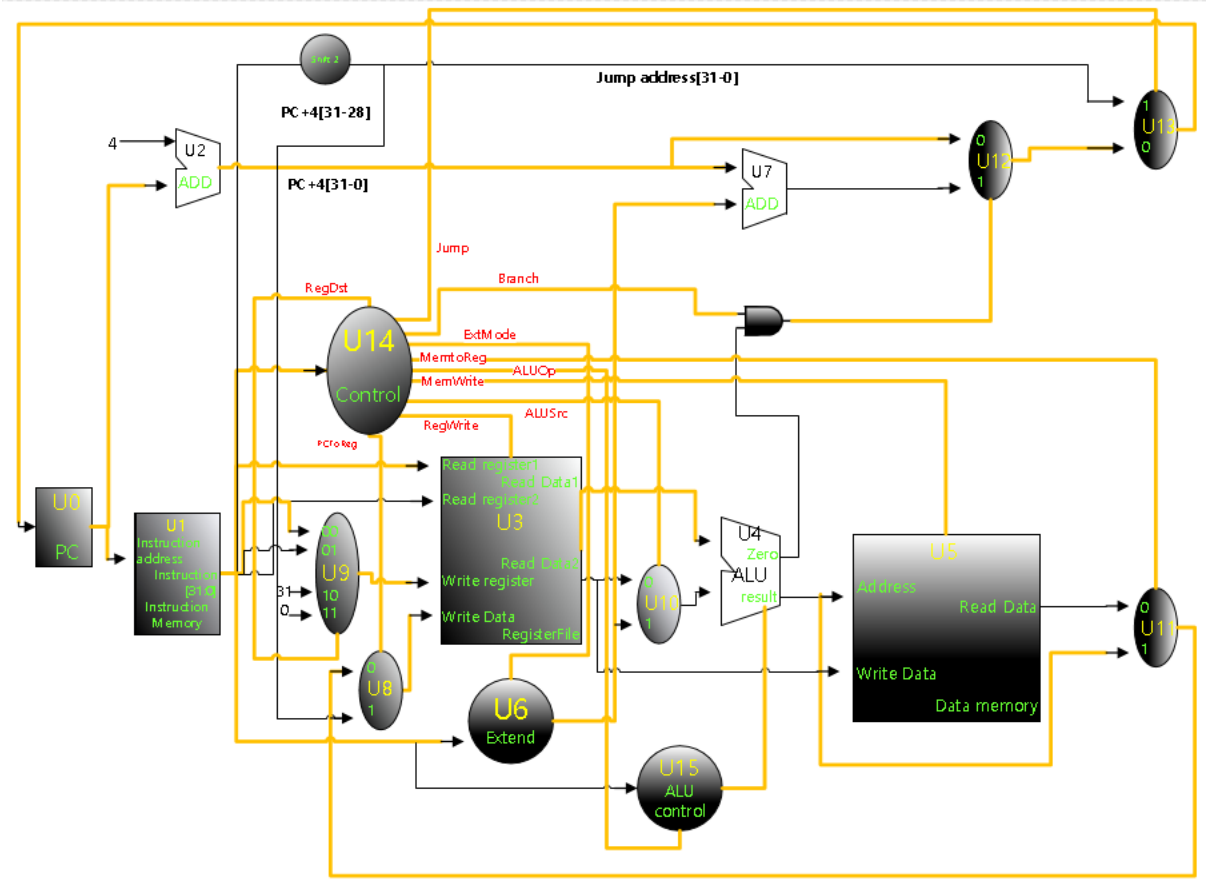
B. 문제해결 방안

설계 아이디어는 미리 구현되어있던 Maincontrol을 기반으로 설계 하게 되었으며, 사용되어야 할 signal의 대한 판단은 수업시간에 다룬 instruction과 관련된 내용을 기반으로 signal을 정하게 되었다. 각각의 data가 이동과 Signal의 동작은 각 명령어의 동작을 설명하는 block그림에서 살펴보도록 하겠다.

2. 설계 의도와 방법

A. 구현한 Single Cycle CPU 블록도

(1) ADDI

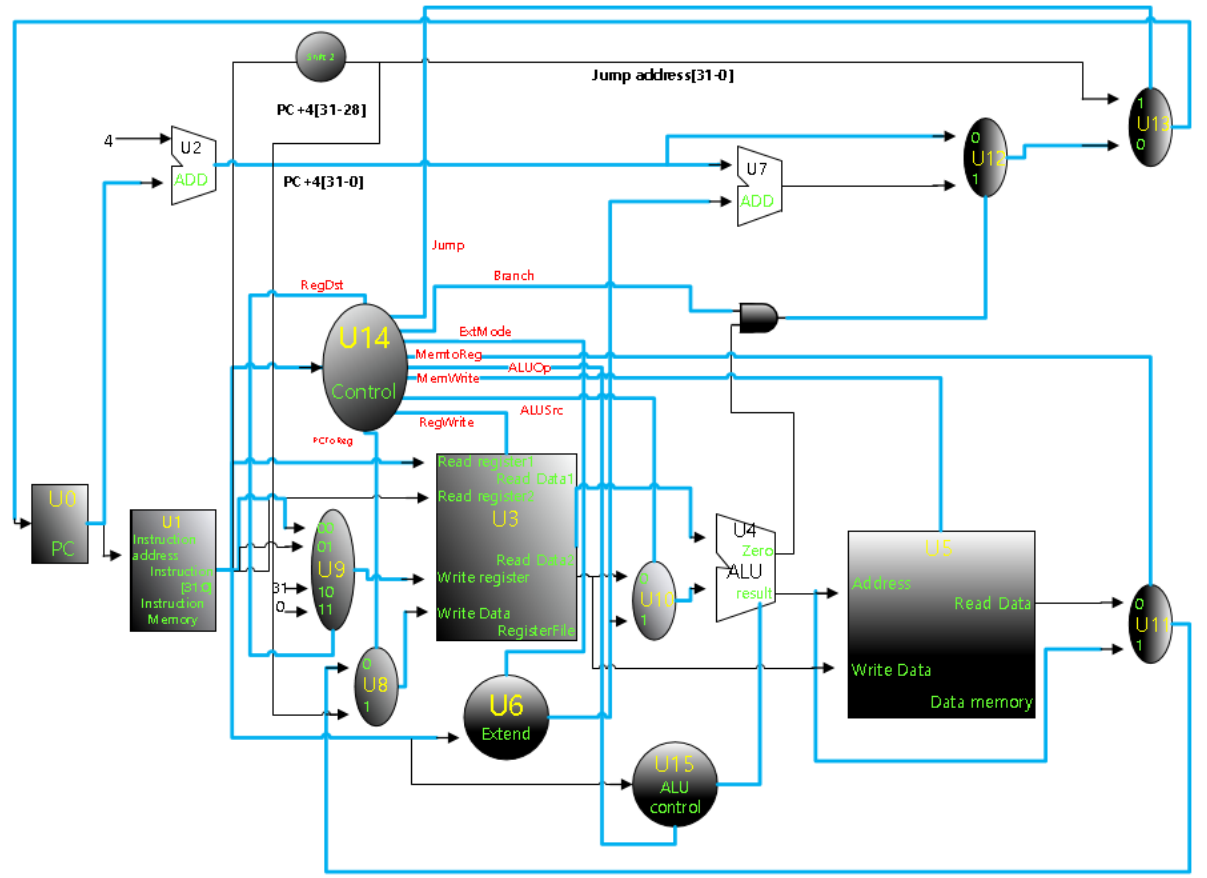


ALUOp	ALUSrc	MemWrite	ReqWrite	ReqDst	MemtoReq	Jump	Branch	PCToreg	Extmode
000	1	0	1	00	0	0	0	0	1

Addi명령의 경우 위와 같은 path로 명령이 진행 된다. PC에서 address가 나오면, Instruction memory에서 instruction을 뽑아 위와 같은 path로 동작하게 된다.

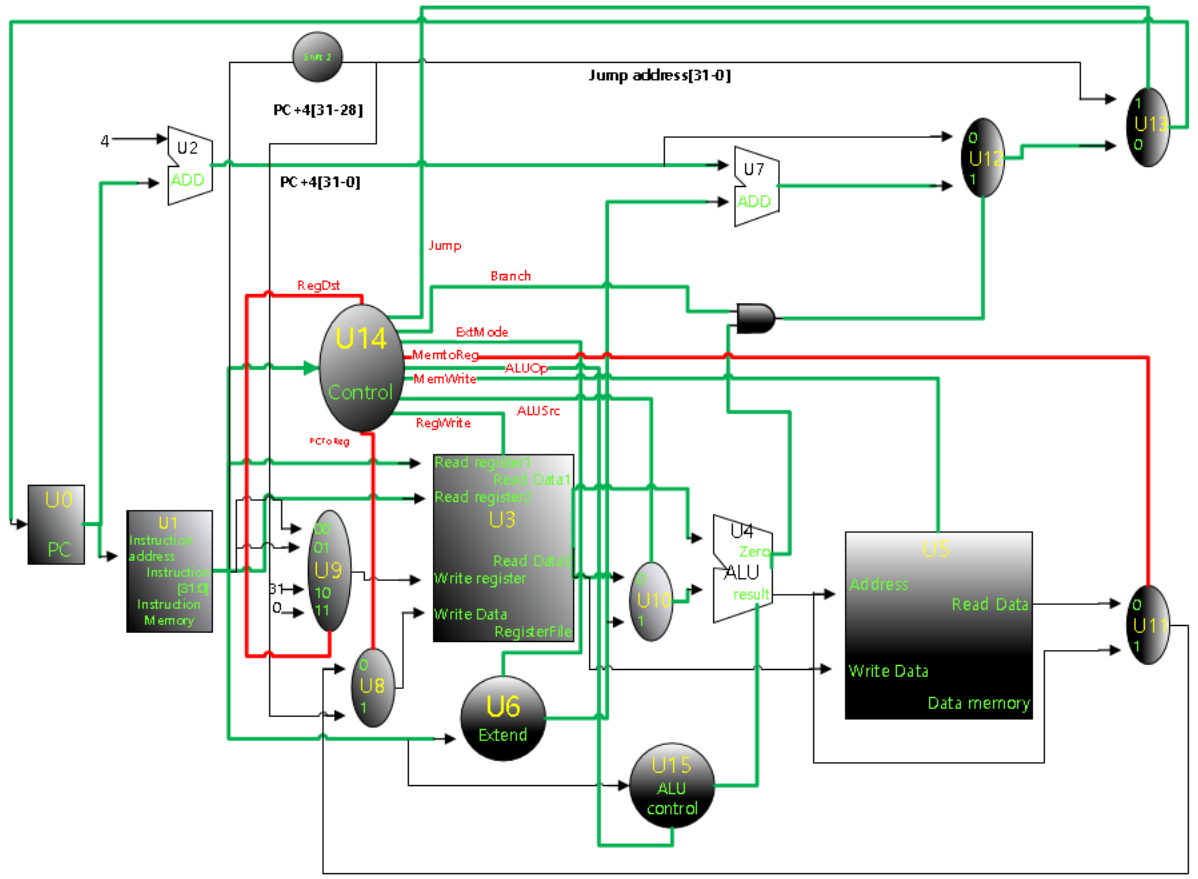
아래 나오는 각각의 명령들도 path에 따라 동작하게 된다.

(2) XORI



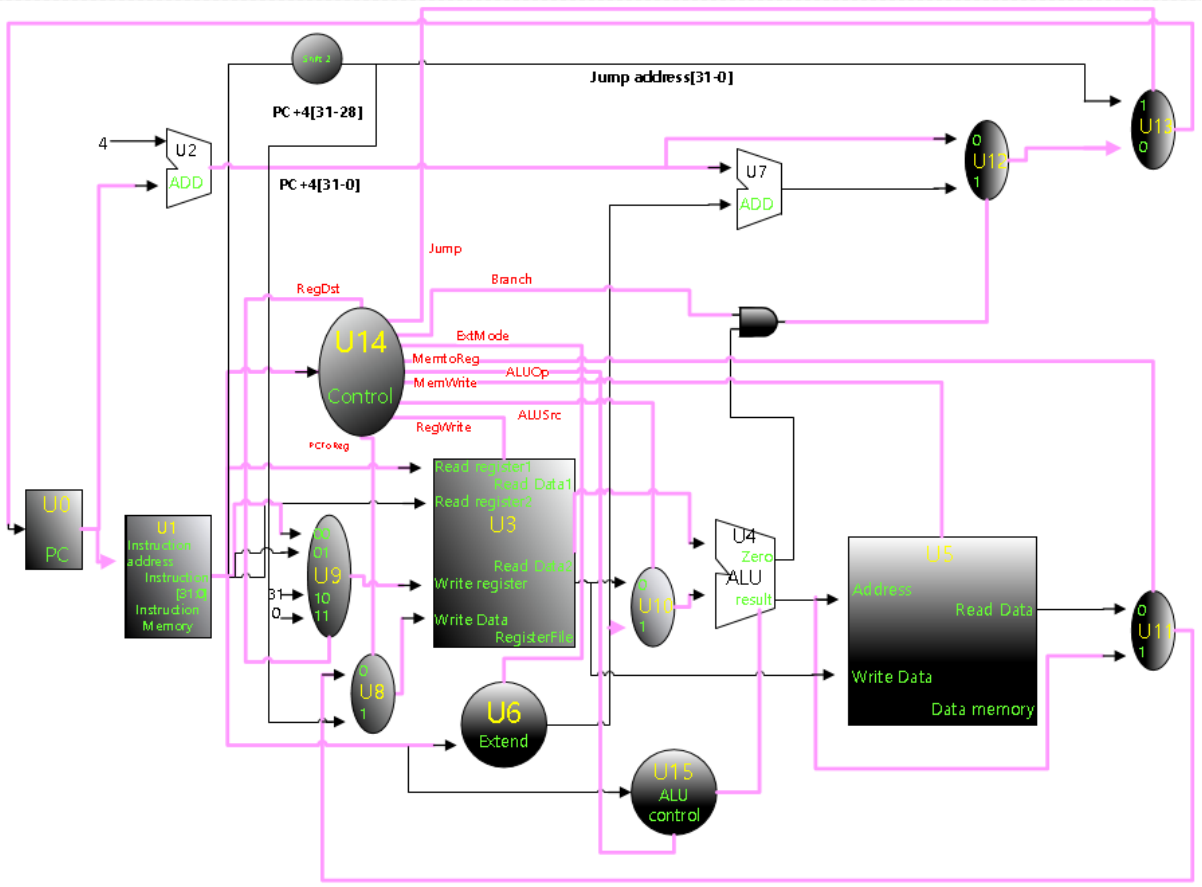
ALUOp	ALUSrc	MemWrite	ReqWrite	ReqDst	MemtoReq	Jump	Branch	PCToreg	Extmode
011	1	0	1	00	0	0	0	0	0

(3) BEQ



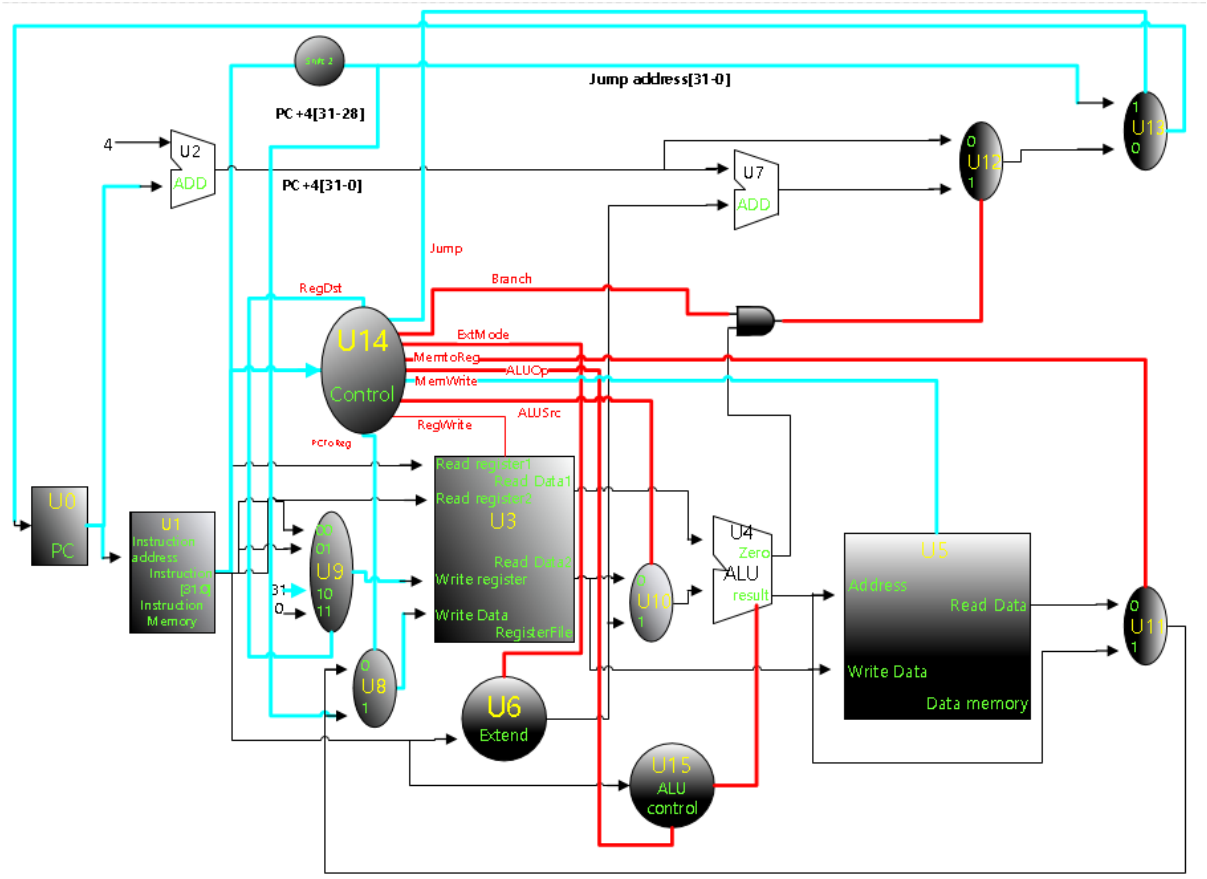
ALUOp	ALUSrc	MemWrite	ReqWrite	ReqDst	MemtoReq	Jump	Branch	PCToreg	Extmode
001	0	0	0	xx	x	0	1	x	1

(4) SLTI



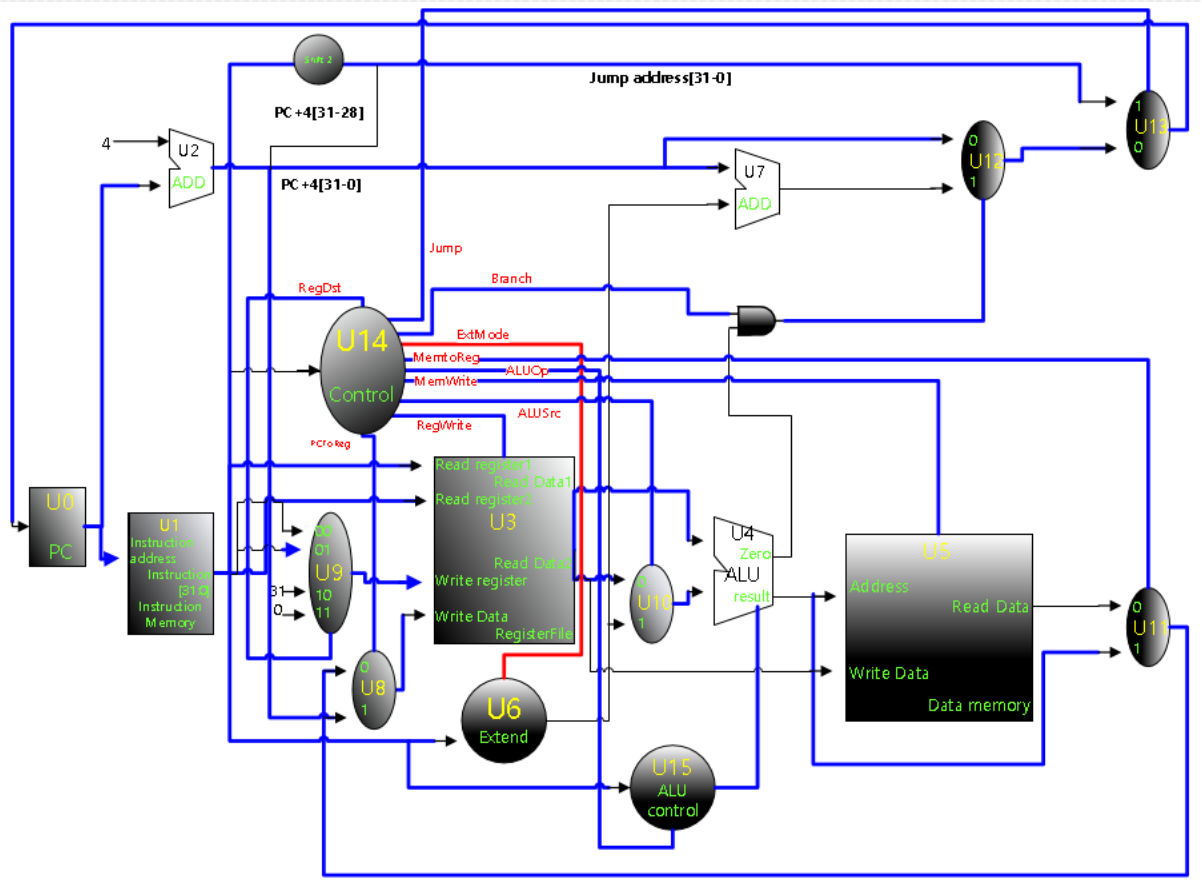
ALUOp	ALUSrc	MemWrite	ReqWrite	ReqDst	MemtoReq	Jump	Branch	PCToreg	Extmode
100	1	0	1	00	0	0	0	0	1

(5) JAL



ALUOp	ALUSrc	MemWrite	ReqWrite	ReqDst	MemtoReq	Jump	Branch	PCToreg	Extmode
xxx	x	0	1	10	x	1	x	1	x

(6) R-type



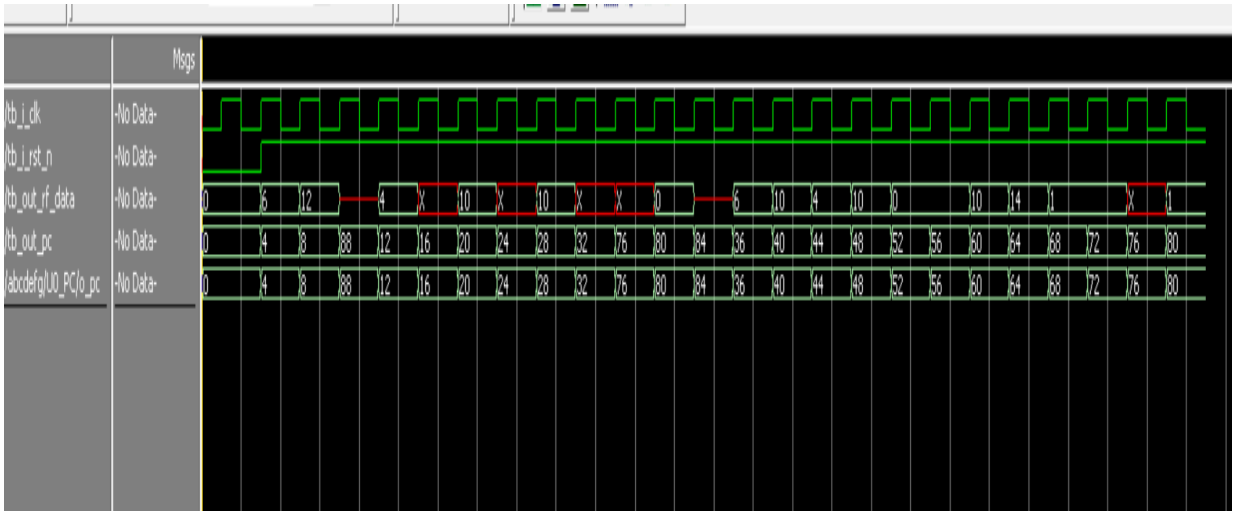
ALUOp	ALUSrc	MemWrite	ReqWrite	ReqDst	MemtoReq	Jump	Branch	PCToreg	Extmode
010	0	0	1	01	0	0	0	0	x

B. 시뮬레이션 결과와 예상결과 비교분석

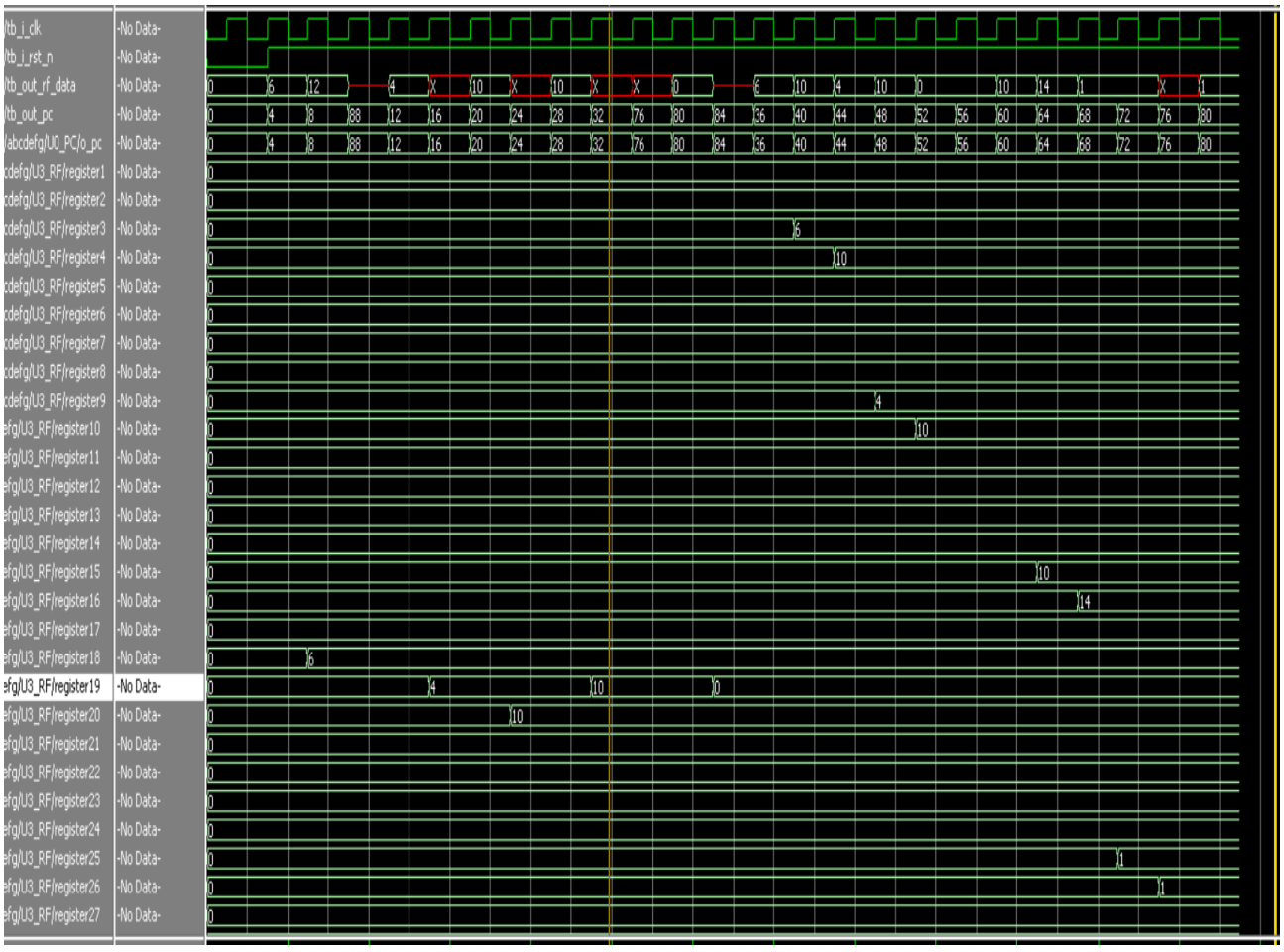
시뮬레이션은 다음과 같은 어셈블리 코드를 기반으로 동작하게 된다.

Assembly code

```
addi $s1,$s1,6
jal 22
addi $s2,$s2,4
beq $s1,$s2,25
addi $s3,$s3,10
sw $s3, 4($zero)
lw $s2, 4($zero)
beq $s2,$s3,10
add $v0,$s1,$s2
add $v1,$s2,$s3
sub $t0,$v1,$v0
sub $t1,$s3,$s2
and $t4,$t1,$t0
and $t5,$s1,$s2
or $t6,$t4,$t1
or $t7,$v0,$s3
slt $t8,$t5,$t0
slt $t9,$t6,$t7
sw $t9,40($zero)
lw $s2,40($zero)
j 9
j 3
```



다음 사진은 위 어셈블리 코드를 동작시킬 때 나오게 되는 wave form이다. 처음 검증하게 될 값은 PC값으로 주어진 어셈블리 코드를 순차적으로 잘 수행하는지 확인하는 것이다. 우선 PC값이 0 -> 8 이후 jal 22 명령이 있으므로 22를 왼쪽으로 2번 shift하여 88의 값을 갖게 되고, 22번째 명령인 j 3이 실행되어 다시 addi를 실행하게 된다, 그 후 각 명령을 실행하게 되는데 아래는 레지스터의 값과 비교한 사진이다.

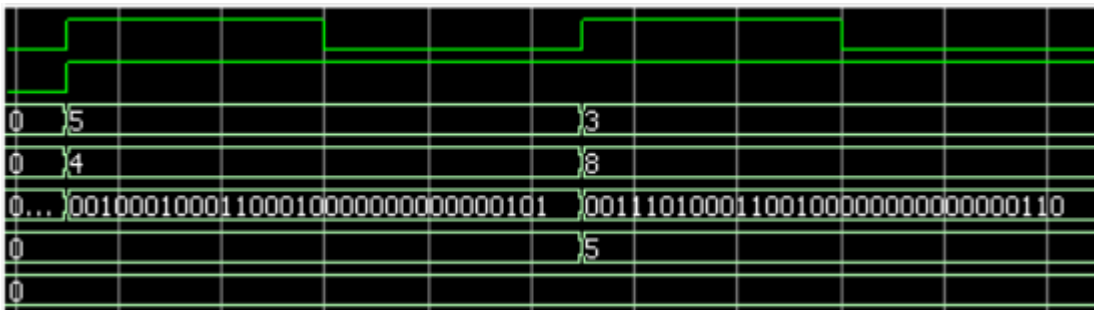


Branch, jump, jal, lw, sw를 동작하는 경우는 out data가 없는 것을 확인할 수 있다.

다음은 각각의 명령에 대한 값이 잘 들어가는지 확인하는 과정이다.

(1) ADDI

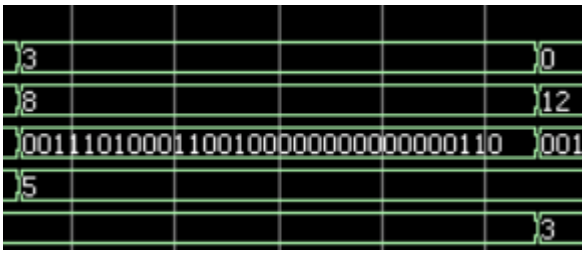
R18<-R18+imm



Imm의 값은 5로 설정하였으며, 레지스터는 18번으로 값이 잘 들어가는 것을 확인할 수 있다.

(2) XORI

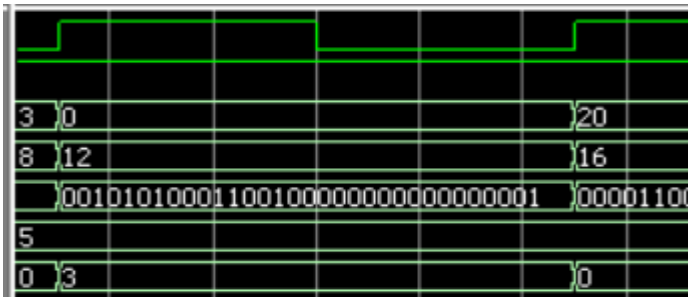
R19<-R18 xor imm



다음은 XORI명으로써 19번레지스터에 R18에 저장되어있는 5의 값과 imm로 들어가는 6의 값을 xor하는 명령이다. 결과 값으로 3이 잘 들어가는 것을 확인할 수 있다.

(3) SLTI

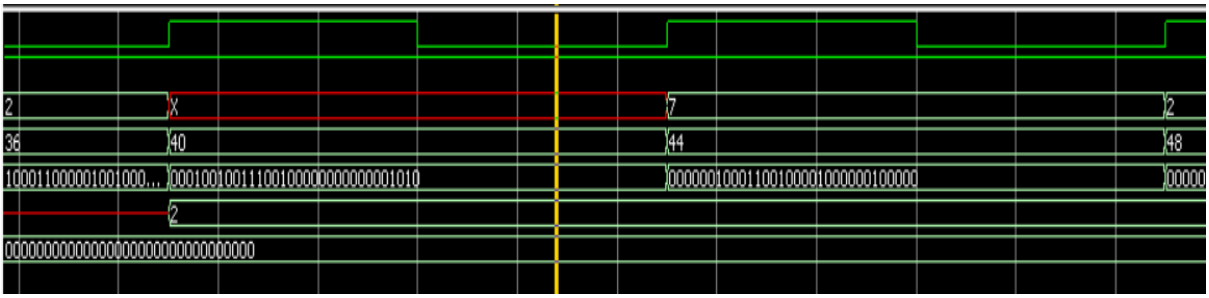
R19<-R18 less than 1



R18에 들어있는 5는 imm값으로 들어온 1보다 크기 때문에 R19가 0으로 set된 것을 확인 할 수 있다.

(4) BEQ

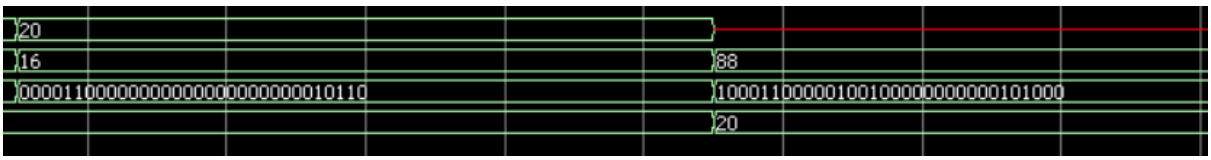
R19 R20 10



R20과 R19의 값이 같지 않기 때문에 PC의 값이 바뀌지 않고 순차적으로 명령이 계속 실행되는 것을 확인할 수 있다.

(5) JAL

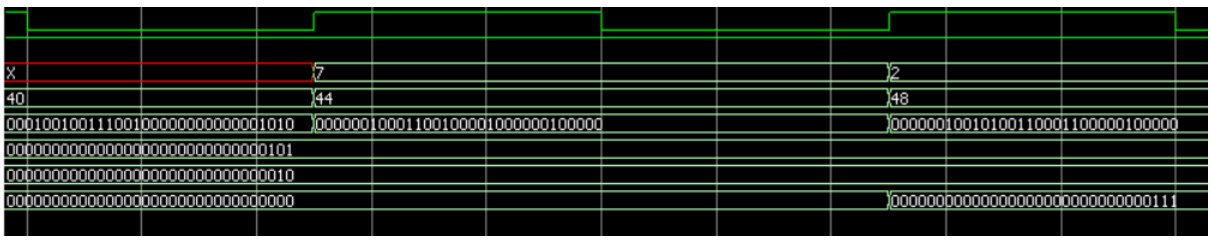
22



점프하는 값이 22이므로 왼쪽으로 두 번 shift하여 88의 값으로 PC값이 변경된 것을 확인할 수 있다. 또한 마지막 레지스터인 링크레지스터의 다음 PC값인 20이 저장되는 것을 확인할 수 있다.

(6) R-type

ADD R3 R18 R19



R-type의 명령으로 R18과 R19의 값을 더하여 R3에 넣는 것을 확인할 수 있다.