Project_2021_1

Implementation and execution of the single cycle MIPS processor

컴퓨터구조

EEE3530.01

정의영 교수님

Team 25

2016132017 김보성

2016143515 양승현

2017142105 윤형탁

2021 년 06 월 08 일

1. ABSTRACT

Single cycle MIPS processor 를 Verilog 로 구현해 하노이의 탑에서 모든 원반을 다른 기둥으로 옮기는 데 필요한 이동의 최소 횟수를 구하는 작업을 수행했다. 주어진 소스에 jal, jr, addi instruction 을 수행할 수 있도록 데이터 패스를 설계하였으며, wire, control signal, parameter 등을 적절히 추가 및 수정하여 이를 구현하였다. 주어진 C code 를 assembly code, binary code 로 순차적으로 변환해 instruction memory 에 저장하고 시뮬레이션을 수행하였다. 명령이 수행됨에 따라 레지스터에 적절한 값들이 올라가는 것을 확인하고, 직접 구축한 모듈에서 중요한 신호들을 확인하여 정상적으로 동작하는 것을 확인했다.

2. OBJECT

주어진 데이터 패스와 구조를 분석하고 함수를 수행할 수 있도록 single cycle MIPS processor 를 완성한다. 주어진 C code 를 binary code 로 변환하고 설계한 프로세서에서 수행해봄으로써 프로세서가 정상적으로 작동하는지 확인한다.

3. PROCESSOR IMPLEMENTATION

3.1. ANALYZATION OF THE GIVEN CODES

코드는 tb_single_MIPS module 내에 mips_single module 이 들어가 있고, mips_single module 내에는 reg32 (PC), add32 (PCADD, BRADD), reg_file (RFILE), alu (ALU), rom32 (IMEM), mem32 (DMEM), mux2 (RFMUX, PCMUX, ALUMUX, WRMUX), control_single (CTL), alu_ctl (ALUCTL) module 들이 존재하는 계층구조로 이루어져 있다. Table 1 은 각 module 에 대한 설명이다.

Module name	Role
tb_single MIPS	250MHz 의 clock 을 생성
mips_single	Positive edge 를 따라 single cycle processor 가 잘 동작할 수 있도록 functional units (ALU, register, memory etc.)를 wire 로 연결
PC	PC register, PC 초기값은 0x4000_0000
PCADD	PC 에 4 를 더함
BRADD	(PC + 4)에 branch offset 을 더함
RFLIE	Register file, $a0 값은 5 로 지정되어 있음
ALU	ALU
IMEM	Instruction memory
DMEM	Data memory
RFMUX	$rt, $rd 중 하나를 선택하여 register 의 write register 로 입력
PCMUX	PC+4, branch target 중 하나를 선택하여 PC register 로 입력
ALUMUX	$rt 값, immediate 중 하나를 선택하여 ALU 로 입력
WRMUX	Data memory 의 read data 와 ALU result 중 하나를 선택하여 register file 의 write data 로 입력
CTL	Control 이 필요한 unit 들에 control signal 전달
ALUCTL	ALU 에서 수행할 명령을 control

Table 1 Original modules

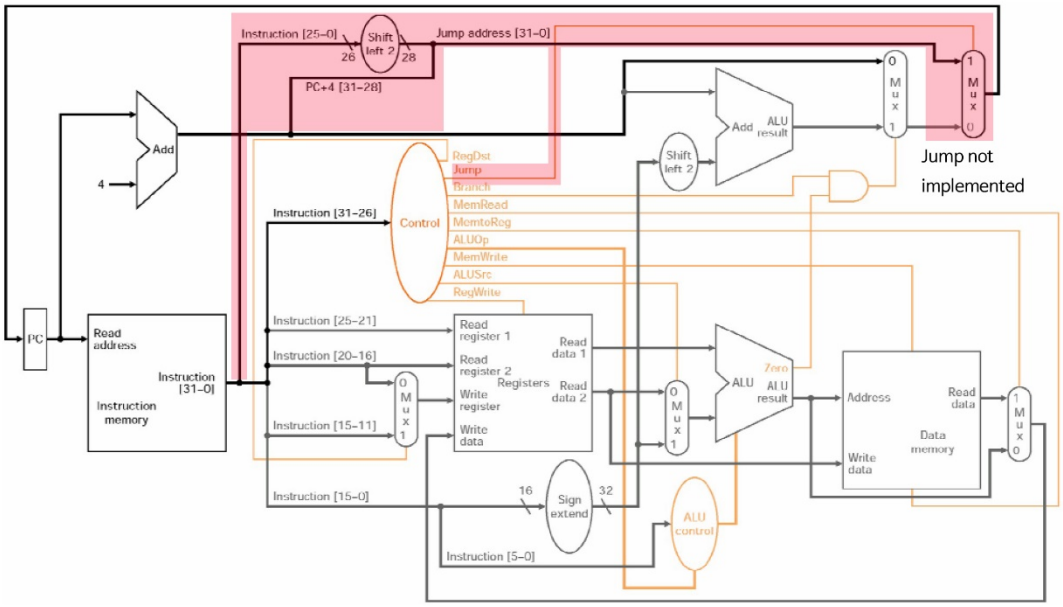

Figure 1 Original data path

이를 데이터 패스로 표현하면 Figure 1 과 같다. 주어진 프로세서에서 수행할 수 있는 instruction 은 add, sub, and, or, slt, lw, sw, beq 이다. Figure 1 에서 나타낸 데이터 패스 중 j instruction 에 관한 부분은 구현되어 있지 않았다.

3.2. EXPLANATION OF OUR IMPLEMENTATION

Jump instruction 들을 수행하기 위해 데이터 패스를 수정하였다. 함수 종료를 선언하기 위해 j instruction 이 필요하고, 함수 호출을 위해 jal instruction 이 필요하다. 함수 종료 후 함수를 호출했던 기존 pc 값으로 돌아가기 위해 jr instruction 이 필요하고, 상수값을 더하는 연산인 addi instruction 을 추가로 사용하기로 한다. 수정된 데이터 패스는 Figure 2 와 같다.

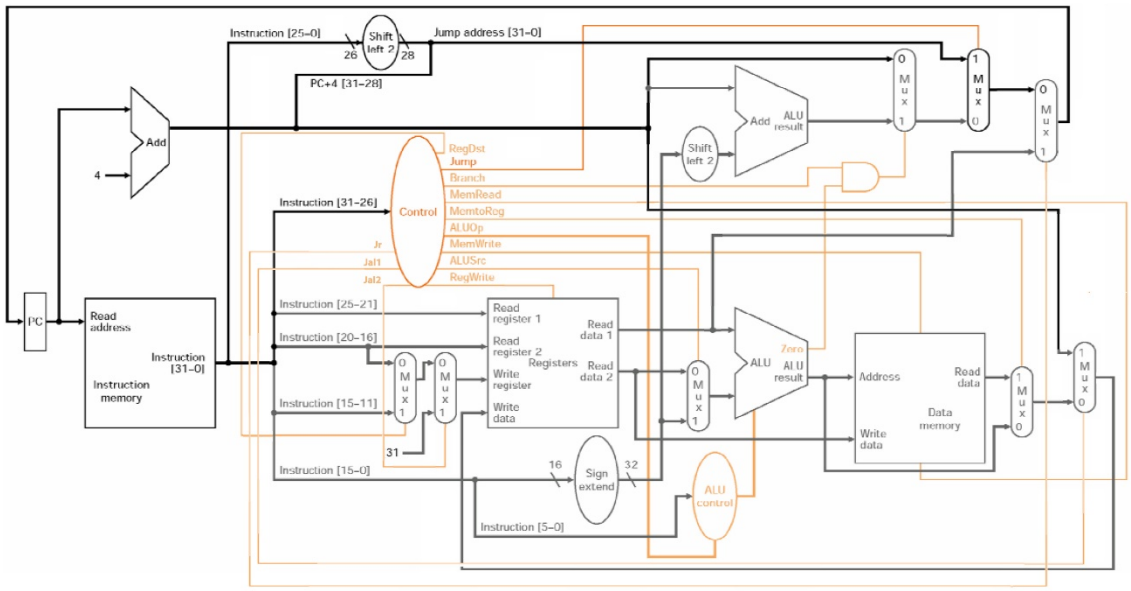

Figure 2 Modified data path

Module name	Role
JMUX	PC+4 와 PCMUX 의 출력값 중 하나를 선택하여 JRMUX 로 입력
JRMUX	JMUX 의 출력값과 Read data 1 중 하나를 선택하여 PC register 로 입력
JALMUX	WRMUX 의 출력값과 PC+4 중 하나를 선택하여 Register file 의 Write data 로 입력
JALMUX2	RFMUX 의 출력과 31 중 하나를 선택하여 Register file 의 Write register 로 입력

Table 2 Additional modules

j instruction 은 pseudo-direct addressing 방법으로 PC 값을 수정하며 jump offset * 4 의 값에 PC+4 값의 상위 4bit 를 추가하여 PC register 로 전달한다. 이 입력값이 기존 PCMUX 의 입력값과 같이 PC register 를 수정하기 때문에 JMUX 를 추가해 하나의 입력값을 선택하도록 했다. jal instruction 의 경우, pseudo-direct addressing 방법으로 PC 값을 수정하는 것은 같으나 동시에 PC+4 값을 $ra 레지스터에 저장해야 한다. 따라서 PC+4 값과 WRMUX 의 출력값 중 하나를 선택해 write data 에 입력할 수 있도록 JALMUX 를 추가하였다. jal instruction 을 수행할 때 write register 는 r31 = $ra 로 고정되므로 JALMUX2 를 추가해 RFMUX 와 31 중 하나를 선택하도록 하였다. jr instruction 은 R-format 으로 레지스터에 저장되어 있는 값을 출력해 다음 PC register 로 전달한다. JRMUX 를 추가해 레지스터에서 출력된 값과 JMUX 의 출력 중 하나를 선택하도록 하였다. 이처럼 설계한 데이터 패스를 프로세서에서 구현하기 위해서 Verilog code 를 추가, 수정하였으며 자세한 내역은 Appendix A 에 기재하였다.

함수 수행에 필요한 stack 은 0x7fff_ffff 부터 아래로 쌓일 수 있도록 하였다. mem32 (DMEM)의 base address 를 0xffff_ff 로 설정하였으며, 초기 $sp 값을 0x8000_0000 으로 설정하였다.

4. CODE IMPLEMENTATION

4.1. IMPLEMENTATION OF CODES

```
int hanoi_tower(int n){
	if ( n == 1 )
		return(1);
	else
		return( hanoi_tower(n-1) * 2 + 1);
}
```

Given C code

```
0		slt $t0, $zero, $a0		# $t0 = 0 < $a0
1		beq $t0, $zero exit		# if $a0 < 1 goto exit
2		addi $t0, $zero, 6		# $t0 = 6
3		slt $t0, $a0, $t0		# $t1 = $a0 < 6
4		beq $t0, $zero, exit		# if $a0 >= 6 goto exit
```

```
5               jal hanoi_tower             # call hanoi_tower
6 exit:         j exit                      # exit (= j 0x4000_0004)
7 hanoi_tower: addi $t0, $zero, 1           # $t0 = 1
8               beq $t0, $a0, skip          # if $a0 = 1 goto skip
9               addi $a0, $a0, -1           # $a0 = $a0 – 1
10              addi $sp, $sp, -4           # stack down
11              sw $ra, 0($sp)              # save $ra
12              jal hanoi_tower             # call hanoi_tower
13              add $v0, $v0, $v0           # $v0 = 2 * $v0
14              addi $v0, $v0, 1            # $v0 = $v0 + 1
15              lw $ra, 0($sp)              # load $ra
16              addi $sp, $sp, 4            # pop stack
17              jr $ra                      # return
18 skip:        addi $v0, $zero, 1          # $v0 = 1
19              jr $ra                      # return
```

Converted assembly code

위의 코드는 C code 로 주어진 함수를 실행할 수 있도록 만든 assembly code 이다. 이 코드가 수행되는 과정은 다음과 같다.

4.1.1. (main) Validation of a given argument

```
0               slt $t0, $zero, $a0         # $t0 = 0 < $a0
1               beq $t0, $zero exit         # if $a0 < 1 goto exit
2               addi $t0, $zero, 6          # $t0 = 6
3               slt $t0, $a0, $t0           # $t1 = $a0 < 6
4               beq $t0, $zero, exit        # if $a0 >= 6 goto exit
```

위의 다섯 라인을 통해 input argument 인 $a0 의 값이 1 보다 작거나, 6 이상인 경우를 배제했다. 첫 번째 beq 를 통해 $t0 (= $a0 > 0)의 값이 false 인 경우와 두 번째 beq 를 통해 다음 $t0 (= $a0 < 6)의 값이 false 인 경우 exit 로 branch 하도록 설계했다.

4.1.2. (main) Function call and exit

```
5             jal hanoi_tower             # call hanoi_tower
6 exit:       j exit                      # exit (= j 0x4000_0004)
```

이후 jump and link instruction 이 실행되어 $ra (r31) 값이 PC+4 로 업데이트되고, hanoi_tower 의 instruction address 가 PC 에 업데이트 된다. exit 는 자기 자신으로 jump 하는 무한 루프로 구현하여 프로그램이 종료되도록 설계했다.

4.1.3. (hanoi_tower) Branch condition

```
7 hanoi_tower: addi $t0, $zero, 1          # $t0 = 1
8             beq $t0, $a0, skip          # if $a0 = 1 goto skip
```

해당 두 코드를 통해 C code 의 If (n == 1) 부분을 구현했다. Temporary register $t0 에 1 을 할당한 후 $a0 에 할당된 argument value (n)을 비교하여 n == 1 이면 skip 으로 branch 한다.

4.1.4. (hanoi_tower) Store register values and recursive call

```
9             addi $a0, $a0, -1           # $a0 = $a0 – 1
10            addi $sp, $sp, -4           # stack down
11            sw $ra, 0($sp)              # save $ra
12            jal hanoi_tower             # call hanoi_tower
```

Line 9 ~ 12 는 레지스터 값을 저장하고 재귀호출을 수행하는 부분이다. 이 때, 이 호출에서 $a0 값은 branch condition 을 체크한 이후로 더 이상 사용되지 않는다. 따라서 기존의 n 값이 들어가 있던 $a0 자리에 재귀 호출이 수행될 때의 argument value n - 1 값을 새로 배정한다. 따라서 다음 함수를 호출하기 전에 저장해야 하는 값은 $ra 레지스터 뿐이다. 값을 저장하기 위해 stack 을 4byte 확장한다. 새로 할당된 stack point 에서 offset 0 위치에 현재 $ra 의 값을 저장하고 재귀 호출을 수행한다.

4.1.5. (hanoi_tower) Return results and load data

```
13              add $v0, $v0, $v0           # $v0 = 2 * $v0
14              addi $v0, $v0, 1            # $v0 = $v0 + 1
15              lw $ra, 0($sp)              # load $ra
16              addi $sp, $sp, 4            # pop stack
17              jr $ra                      # return
```

Line 13 ~ 17 은 C code 의 return (hanoi_tower(n-1) * 2 + 1) 부분을 구현한 것이다. (new)$v0 = (old)$v0 * 2 + 1 을 수행하고 DMEM 에 저장했던 이전 $ra 값을 load 한 뒤 확보했던 stack 을 pop 시킨다. 모든 수행이 끝나면 함수를 종료한다.

4.1.6. (hanoi_tower) Return 1 if n == 1

```
18 skip:        addi $v0, $zero, 1          # $v0 = 1
19              jr $ra                      # return
```

마지막 부분은 line 8 에서 n == 1 인 경우에 branch 해오는 부분이다. 앞에서 branch condition 을 확인했으므로 여기서는 return(1) 만 수행하면 된다. 수행이 끝나면 함수를 종료한다.

4.2. VERIFICATION OF EXECUTABILITY

작성한 assembly code 에서 사용한 instruction 은 slt, beq, addi, jal, j, sw, add, lw 로 총 8 개이다. 이들 중 이미 구현된 5 개의 instructions (slt, beq, sw, add, lw)를 제외한 addi, j, jal, jr instruction 은 3. 에서 구현했다. 따라서 앞서 설계한 single cycle MIPS processor 에서 모든 동작이 수행 가능함을 알 수 있다.

4.3. BINARY CODE GENERATION

```
5'd0 : data_out = { 6'd0, 5'd0, 5'd4, 5'd8, 5'd0, 6'd42 };       // slt $t0, $zero, $a0
5'd1 : data_out = { 6'd4, 5'd8, 5'd0, 16'd4 };                   // beq $t0, $zero, exit (= +4)
5'd2 : data_out = { 6'd8, 5'd0, 5'd8, 16'd6 };                   // addi $t0, $zero, 6
5'd3 : data_out = { 6'd0, 5'd4, 5'd8, 5'd8, 5'd0, 6'd42 };       // slt $t0, $a0, $t0
5'd4 : data_out = { 6'd4, 5'd8, 5'd0, 16'd1 };                   // beq $t0, $zero, exit (= +1)
5'd5 : data_out = { 6'd3, 26'd7 };                         // jal Hanoi_tower (= 0x4000_001c)
5'd6 : data_out = { 6'd2, 26'd6 };                         // j exit (= 0x4000_0018)
5'd7 : data_out = { 6'd8, 5'd0, 5'd8, 16'd1 };                   // addi $t0, $zero, 1
5'd8 : data_out = { 6'd4, 5'd4, 5'd8, 16'd14 };                  // beq $a0, $t0, skip (= +9)
5'd9 : data_out = { 6'd8, 5'd4, 5'd4, -16'd1 };                  // addi $a0, $a0, -1
5'd10 : data_out = { 6'd8, 5'd29, 5'd29, -16'd4 };               // addi $sp, $sp, -4
5'd11 : data_out = { 6'd43, 5'd29, 5'd31, 16'd0 };               // sw $ra, 0($sp)
5'd12 : data_out = { 6'd3, 26'd7 };                        // jal Hanoi_tower (= 0x4000_001c)
5'd13 : data_out = { 6'd0, 5'd2, 5'd2, 5'd2, 5'd0, 6'd32 };      // add $v0, $v0, $v0
5'd14 : data_out = { 6'd8, 5'd2, 5'd2, 16'd1 };                  // addi $v0, $v0, 1
5'd15 : data_out = { 6'd35, 5'd29, 5'd31, 16'd0 };               // lw $ra, 0($sp)
5'd16 : data_out = { 6'd8, 5'd29, 5'd29, 16'd4 };                // addi $sp, $sp, 4
5'd17 : data_out = { 6'd0, 5'd31, 5'd0, 5'd0, 5'd0, 6'd8 };      // jr $ra
5'd18 : data_out = { 6'd8, 5'd0, 5'd2, 16'd1 };                  // addi $v0, $zero, 1
5'd19 : data_out = { 6'd0, 5'd31, 5'd0, 5'd0, 5'd0, 6'd8 };      // jr $ra
```

Binary code

위 code 는 assembly code 를 binary code 로 변환한 것으로 IMEM : rom32 (rom32.v) 에 입력하였다.

5. Execution Result

5.1. Instruction – Register match

Instruction 이 line 0 = 0x4000_0000 에서 line 19 = 0x4000_004c 까지의 address 를 가지므로 pc 값의 LSB 8 bits 를 변수 pc_in 과 pc_out 으로 지정하였고, 레지스터에서 실제로 사용된 $v0, $a0, $t0, $sp, $ra 값을 변수로 지정하여 시뮬레이션을 실행하였다.

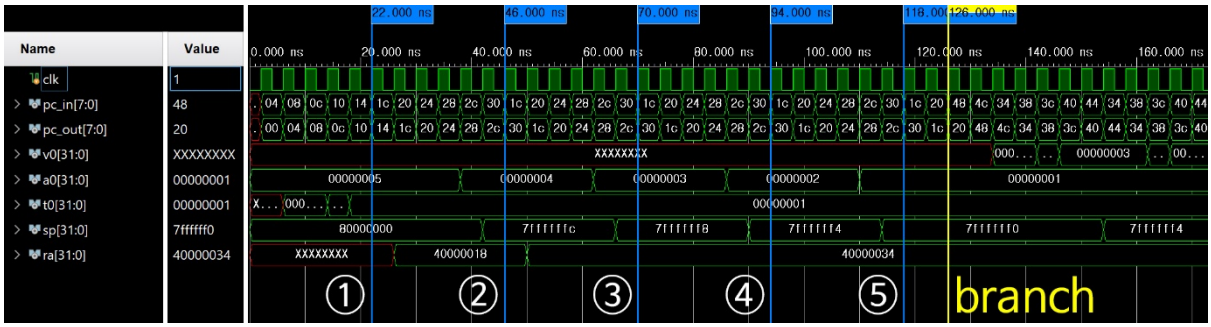

Figure 3 PC – Register value waveform 1

그림에서 ① ~ ⑤ 로 표시한 부분은 jal instruction 을 수행하는 부분이다. Insturction 수행 중 pc_in, 수행 후 업데이트된 pc_out 값이 line 7 = (0x4000_00)1c 가 나오는 것을 확인할 수 있다. 첫번째 jump 를 수행할 때는 hanoi_tower 함수를 부른 지점의 다음 주소인 line 6 = 0x4000_0018 을 $ra 에 저장했고, 두번째 이후로는 재귀 호출을 수행하는 지점의 다음 주소인 line 13 = 0x4000_0034 를 $ra 에 저장하는 것을 확인할 수 있다.

addi instruction 은 $t0 에 1, 6 을 저장하거나, $a0 에 -1 을 더하고, $sp 를 -4 만큼 이동시키는 역할을 수행한다. 처음 $a0 값이 유효한지 확인하는 부분에서 $t0 값이 바뀌는 것이나 $a0 가 1 씩 줄어드는 것, $sp 가 4 씩 내려가는 것을 확인할 수 있다.

노란색으로 강조표시한 부분은 line 8 에서 branch instruction 이 수행되는 부분이다. $a0 값이 1 인 것을 확인하고 line 18 = (0x4000_00)48 로 이동하여 $v0 에 1 값을 반환하는 것을 확인할 수 있다.

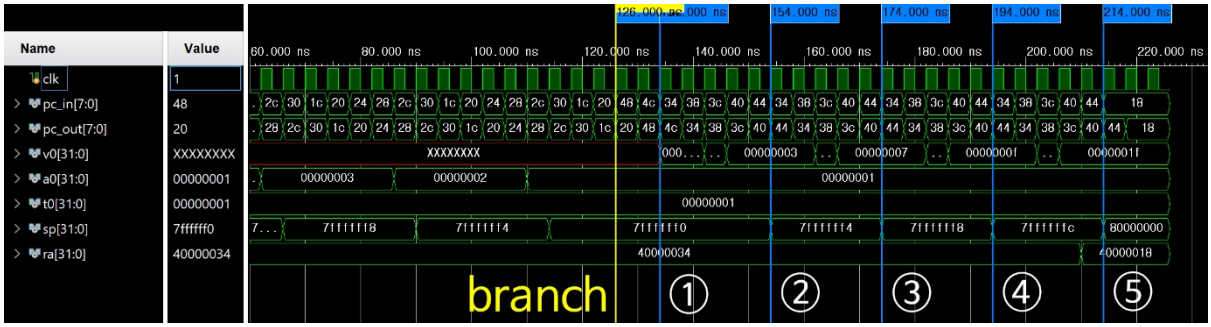

Figure 4 PC – Register value waveform 2

branch instruction 을 수행한 이후 ① ~ ⑤ 부분에서는 jr instruction 을 수행한다. 첫번째 ~ 네번째 insturction 수행 중 pc_in, 수행 후 업데이트된 pc_out 값이 line 13 = (0x4000_00)34 가 나오는 것을 확인할 수 있다. 마지막 수행에서는 처음 hanoi_tower 함수를 호출한 지점의 다음 주소인 line 6 = (0x4000_00)18 으로 복귀한다.

함수 종료 후 결과값 0x1f = d31 이 정상적으로 반환되는 것을 확인할 수 있다.

5.2. OPERATION OF IMPLEMENTED INSTRUCTIONS

직접 추가한 instruction 이 수행되는 지점에서 input, output, control signal 등이 제대로 작동하는지 확인하였다.

5.2.1. j instruction

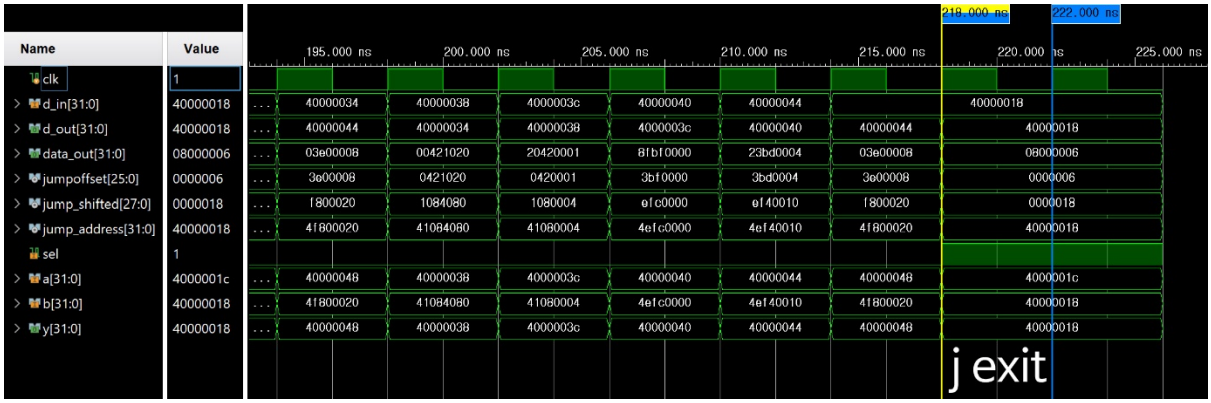

Figure 5 j instruction

Line 6 (j exit) instruction 을 수행하는 부분이다. d_in 은 PC 레지스터의 input, d_out 은 PC 레지스터 output 이다. data_out 은 IMEM 에서 출력되는 신호이다. jumpoffset, jump_shifted, jump_address 는 각각 26 bits, 28 bits, 32 bits data 로 instruction value, shifted

value, pseudo-direct addressing 으로 이어지는 변환과정이다. sel, a, b, y 는 JMUX 에서 select 신호와 input, output 을 모니터한다.

data_out 에서 0x0800_0006 신호가 출력되는데, 이 중 앞의 6 bits 는 opcode, 뒤의 26 bits 는 word aligned 를 고려한 jump address 를 나타낸다. 여기서 jumpoffset 값을 2 bits shift 하면 0x6 * 4 = 0x18 이 jump_shifted 값이 된다. Jump_shifted 값과 PC+4 의 MSB 4 bits 를 합해 목표 주소인 0x4000_0018 이 jump_address 가 된다. 이 값이 JMUX input a 로 입력되고 select 신호가 1 이 입력되면서 output 으로 나가 d_in 으로 입력되는 것을 확인할 수 있다.

5.2.2. jal instruction

Figure 6 jal instruction

Line 5 (jal hanoi_tower) instruction 을 수행하는 부분이다. PC 값을 업데이트하는 과정은 앞의 j instruction 과 같으므로 여기서는 생략했다. 첫번째 select, a, b, y 신호는 JALMUX 의 input, output 이며, 두번째 select, a, b, y 신호는 JALMUX2 의 input, output 이다. RegWrite 신호는 레지스터에 데이터를 업데이트하도록 하는 신호이고, WN 은 데이터를 쓸 레지 스터 번호, WD 는 레지스터에 쓸 데이터이다.

JALMUX 에서는 d_out + 4 = 0x4000_0018 값을 선택하여 출력하며, JALMUX2 에서는 0x1f = d31 값을 선택하여 출력하는 것을 확인할 수 있다. 레지스터에서는 RegWrite = 1, WN = 0x1f, WD = 0x4000_0018 으로 앞에서 선택한 값들이 $ra 레지스터에 쓰이는 것을 확인할 수 있다. PC 값 또한 d_in = 0x4000_001c 으로 정상적으로 업데이트 된다.

5.2.3. jr instruction

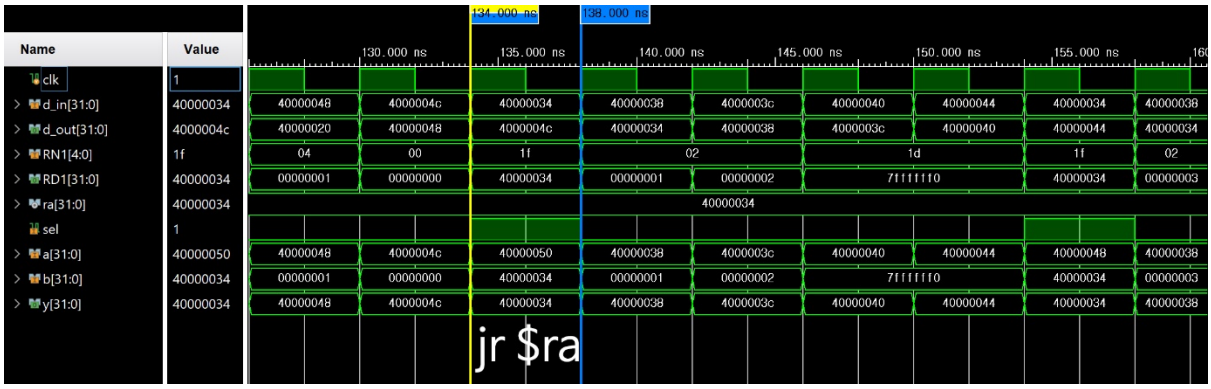

Figure 7 jr instruction

Line 17 (jr $ra) instruction 을 수행하는 부분이다. RN1 과 RD1 은 $ra 레지스터 번호를 지정해 데이터를 불러오는 과정이다. 뒤의 sel, a, b, y 신호는 JRMUX 의 input, output 이다.

RN1 에서 0x1f = d31 = $ra 레지스터를 지정하여 R[$ra] = RD1 = d_in = 0x4000_0034 로 $ra 값이 RD1 으로 출력된다. JRMUX 에서는 이 값을 선택해 d_in 으로 입력하는 것을 확인할 수 있다.

5.2.4. addi instruction

Figure 8 addi instruction

Line 2 (addi $t0, $zero, 6) instruction 을 수행하는 부분이다. 앞의 RegDst ~ ALUOP 신호는 Control unit 에서 generate 된 신호이고, 뒤의 ctl, a, b, result 는 ALU 의 control signal 과 input, output value 이다. 실제로 업데이트 되는 $t0 값도 함께 모니터링 하였다.

addi instruction 이 수행될 때 RegDst 는 rt 를 선택해야 하므로 0, ALUSrc 는 immediate 값을 선택해야 하므로 1, MemtoReg 는 ALU output 을 선택해야 하므로 0, 출력된 값을 레지스터에 저장해야 하므로 RegWrite 는 1 이다. I-format 에서 ALUOp 는 funct 값을 받지 않고 직접 add operation 을 지정하게 되므로 00 값을 전달하게 된다. ALU 쪽에서는 ctl = 2 (= add), input a = 0, input b = 6, result = 6 으로 addi instruction 이 정상적으로 수행되어 $t0 값을 업데이트 하는 것을 확인할 수 있다.

6. CONCLUSION

주어진 소스 파일에 j, jal, jr, addi 의 4 가지 추가 instruction 을 구현한 single cycle MIPS processor 가 잘 작동하는 것을 확인하였다. 시뮬레이션 결과에서 PC, 레지스터의 값이 정상적으로 업데이트되는 것을 확인하였으며, instruction 이 수행되는 동안 control signal, input value, output value 들이 예상한대로 잘 작동함을 확인하였다. 구현한 프로세서를 통해 실행한 전체 함수 동작시간은 222ns 였다.

Figure 9 Comparison to the real device

Figure 9 은 주어진 코드를 Python code 로 변환해 실제 컴퓨터에서 동작시간을 측정한 것이다. 실제 컴퓨터에서 동작시간은 514 ± 11.7 ns 로 여기서 구현한 프로세서의 222 ns 와 비교해 훨씬 느린 것을 알 수 있다. 이는 여기서 구현한 single cycle processor 가 매우 적은 instruction set 만을 가지는 대신, 동작속도는 매우 빠르다는 것을 의미한다. 그러나 이번 프로젝트에서 수행한 함수는 매우 단순한 함수로, 더 복잡한 프로그램을 실행한다면 상황이 상당히 달라질 수 있다.

이번 프로젝트를 통해 설계한 데이터 패스를 실제 프로세서 구조에 맞게 구현할 수 있게 되었으며, 코드가 프로세서에서 어떻게 수행되는지 확인하였다. 또한 단순한 함수를 수행할 때는 프로세서의 ISA 가 단순한 편이 cost 와 동작시간에서의 이득을 줄 수 있음도 함께 확인하였다.

7. REFERENCE

- E. Y. Chung, Course EEE3530 Lecture Notes and Given Sources, 2021.
- D. A. Patterson & J. L. Hennessy, *Computer Organization and Design MIPS Edition: The Hardware/Software Interface*. Morgan Kaufmann, 2013.

A. APPENDIX: MODIFIED VERILOG CODES

tb_single_MIPS module

```
initial begin
    forever #2 clk <= ~clk;
end

initial begin
    clk <= 1'b0; reset <= 1'b0;
    #2 reset <= 1'b1;
    #2 reset <= 1'b0;
    #222 $stop;
end
```

전체 instruction 이 수행되는 데에 필요한 시간은 약 222ns 이다.

mips_single module

```
// break out important fields from instruction
wire [5:0] opcode, funct;
wire [4:0] rs, rt, rd, shamt;
wire [15:0] immed;
wire [31:0] extend_immed, b_offset, jump_address;
wire [25:0] jumpoffset;
wire [27:0] jump_shifted;
// datapath signals
wire [4:0] rfile_wn, wn_result;
wire [3:0] pseudo;
wire [31:0] rfile_rd1, rfile_rd2, rfile_wd, alu_b, alu_out, b_tgt, pc_next, j_result, b_result,
            pc, pc_incr, dmem_result, dmem_rdata;
```

```
// control signals

wire RegWrite, Branch, PCSrc, RegDst, MemtoReg, MemRead, MemWrite, ALUSrc, Zero, Jump, Jr, Jal1, Jal2;
wire [1:0] ALUOp;
wire [2:0] Operation;

// module instantiations

mux2 #(32)  JMUX(Jump, b_result, jump_address, j_result); //jr instruction mux 추가

mux2 #(32)  JRMUX(Jr, j_result, rfile_rd1, pc_next); //jr instruction mux 추가

mux2 #(32)  JALMUX(Jal1, dmem_result, pc_incr, rfile_wd); //jal instruction mux 추가

mux2 #(5)   JALMUX2(Jal2, wn_result, 5'd31, rfile_wn); //jal instruction mux 추가

control_single CTL(.opcode(opcode), .funct(funct), .RegDst(RegDst), .ALUSrc(ALUSrc), .MemtoReg(MemtoReg),
                   .RegWrite(RegWrite), .MemRead(MemRead), .MemWrite(MemWrite), .Branch(Branch),
                   .ALUOp(ALUOp), .Jump(Jump), .Jr(Jr), .Jal1(Jal1), .Jal2(Jal2));

  // jump address
assign pseudo = pc_incr[31:28];
assign jump_shifted = jumpoffset << 2;
assign jump_address = { pseudo , jump_shifted};
```

각 instruction 이 올바르게 작동하도록 wire 들을 추가하였으며, JMUX, JRMUX, JALMUX, JALMUX2 module 을 추가하였다. Control_single CTL module 에 MUX select signal parameter 들을 추가하였고, j 및 jal instruction 이 진행될 수 있게 pseudo-direct address 값을 지정해주었다.

control_single module

```
module control_single(opcode, funct,  RegDst, ALUSrc, MemtoReg, RegWrite, MemRead, MemWrite, Branch, ALUOp, Jump, Jr, Jal1, Jal2);
    input [5:0] opcode, funct;
    output RegDst, ALUSrc, MemtoReg, RegWrite, MemRead, MemWrite, Branch, Jump, Jr, Jal1, Jal2;
    output [1:0] ALUOp;
    reg    RegDst, ALUSrc, MemtoReg, RegWrite, MemRead, MemWrite, Branch, Jump, Jr, Jal1, Jal2;
    reg    [1:0] ALUOp;

     parameter R_FORMAT = 6'd0;
     parameter ADDI = 6'd8;
     parameter LW = 6'd35;
     parameter SW = 6'd43;
     parameter BEQ = 6'd4;
     parameter JUMP = 6'd2;
     parameter JAL = 6'd3;
```

```
always @(opcode or funct)
begin
    case (opcode)
      R_FORMAT :
      begin
        if (funct == 6'd8)
        begin
          RegDst=1'bx; ALUSrc=1'bx; MemtoReg=1'bx; RegWrite=1'b0; MemRead=1'b0;
          MemWrite=1'b0; Branch=1'b0; ALUOp = 2'bxx; Jump=1'b0; Jr=1'b1; Jal1=1'b0; Jal2=1'b0;
        end
        else
        begin
          RegDst=1'b1; ALUSrc=1'b0; MemtoReg=1'b0; RegWrite=1'b1; MemRead=1'b0;
          MemWrite=1'b0; Branch=1'b0; ALUOp = 2'b10; Jump=1'b0; Jr=1'b0; Jal1=1'b0; Jal2=1'b0;
        end
      end
      JUMP :
      begin
          RegDst=1'bx; ALUSrc=1'bx; MemtoReg=1'bx; RegWrite=1'b0; MemRead=1'b0;
          MemWrite=1'b0; Branch=1'b0; ALUOp = 2'bxx; Jump=1'b1; Jr=1'b0; Jal1=1'b0; Jal2=1'b0;
      end
      JAL :
      begin
          RegDst=1'bx; ALUSrc=1'bx; MemtoReg=1'bx; RegWrite=1'b1; MemRead=1'b0;
          MemWrite=1'b0; Branch=1'b0; ALUOp = 2'bxx; Jump=1'b1; Jr=1'b0; Jal1=1'b1; Jal2=1'b1;
      end
      ADDI :
      begin
          RegDst=1'b0; ALUSrc=1'b1; MemtoReg=1'b0; RegWrite=1'b1; MemRead=1'b0;
          MemWrite=1'b0; Branch=1'b0; ALUOp = 2'b00; Jump=1'b0; Jr=1'b0; Jal1=1'b0; Jal2=1'b0;
      end
      default
      begin
          $display("control_single unimplemented opcode %d", opcode);
          RegDst=1'bx; ALUSrc=1'bx; MemtoReg=1'bx; RegWrite=1'bx; MemRead=1'bx;
          MemWrite=1'bx; Branch=1'bx; ALUOp = 2'bxx;
      end
```

Module parameter 에 MUX control signal 인 Jump, Jr, Jal1, Jal2 를 추가하였다. 또한 jr instruction 은 R-format 이지만 기존 R-format instruction 들과 다르게 작동하므로 funct value 를 추가로 확인해서 jr instruction 을 구별할 수 있도록 하였다. jr, jump, jal, addi instruction control signal 값들은 위와 같이 설정하였다.

reg_file module

```
initial
begin
  file_array[4] = 5;
  file_array[29] = 32'h80000000;
end
```

Stack 의 맨 첫번째 word 가 0x7fff_fffc ~ 0x7fff_ffff 에 들어가도록 $sp 의 초기값을 0x8000_0000 으로 설정하였다. 실제로 stack 을 사용할 때는 스택 공간을 먼저 확보하고 데이터를 저장하므로 0x8000_0000 의 바로 아래인 0x7fff_ffff 부터 아래 방향으로 데이터가 쌓이게 된다.

mem32 module

```
module mem32(clk, mem_read, mem_write, address, data_in, data_out);
  input  clk, mem_read, mem_write;
  input  [31:0] address, data_in;
  output [31:0] data_out;
  reg    [31:0] data_out;

  parameter BASE_ADDRESS = 25'hffffff; // address that applies to this memory - change if desired
```

base_address parameter 는 DMEM 에서 stack 으로 사용할 부분을 지정한다. 상위 25 bits 를 0xffff_ff 로 설정하면 b0111_1111_1111_1111_1111_1111_1000_0000 = 0x7fff_ff80 에서부터 b0111_1111_1111_1111_1111_1111_1111_1111 = 0x7fff_ffff 까지의 범위를 stack 으로 사용할 수 있다. Stack 은 0x7fff_ffff 부터 내려오면서 사용할 예정이므로 base address 값을 0xffff_ff 로 설정하였다.