

Project #2

MIPS Multi - Cycle CPU

Implementation

|  |  |  |
| --- | --- | --- |
| 과목명 |  | 컴퓨터구조 |
| 담당교수 |  | 이성원 교수님 |
| 학과 |  | 컴퓨터정보공학부 |
| 학년 |  | 3학년 |
| 학번 |  | 2019202009 |
| 이름 |  | 서여지 |
| 제출일 |  | 21.05.02 (일) |



1. 문제의 해석 및 해결 방향
   1. 실험내용에 대한 설명

제시된 Multicycle CPU는 ROM파일에서 읽은 data를 통해 state를 변화시키며, CPU의 module은 각 state에 따른 control signal을 사용한다. 이번 프로젝트의 내용은 Multicycle CPU의 FSM을 설계하고, micro programming을 이용하여 ROM을 작성하는 것이다. state number는 0부터 127 사이의 값을 가질 수 있고, ROM은 FSM state마다 40 bit의 control signal 정보를 갖는다. 해당 내용을 state number 순서대로 MICRO\_ROM.txt에 저장하고, MyFSM.v 가 명령어를 decoding하여 얻은 opcode에 따라 next state를 지정하는 동작을 수행하도록 작성한다.

* 1. 문제점 해결 방향

과제를 해결하기 위하여 Multicycle CPU의 FSM을 설계한다. NOR, SLT, SLTI, ADDI, BEQ, BGTZ, MULT, MFLO, XORI, JAL, JR 명령어를 3가지 종류로 나누어 설계하였다.

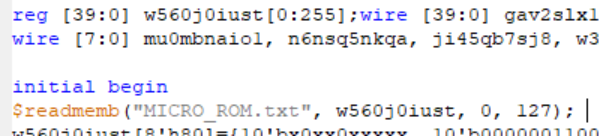
|  |  |  |
| --- | --- | --- |
| J | JAL, JL | 조건을 비교하지 않는 jump 명령어 |
| B | BEQ, BGTZ | 조건을 비교하여 PC 이동하는 Branch 명령어 |
| R | NOR, SLT, SLTI, SRAV,  ADDI, MULT, XORI, MFLO | ALU 결과를 RF에 저장하는 명령어  + MULT, MFLO 명령어 |

MULT와 MFLO의 경우 형태가 유사한 R 그룹으로 분류하였다.

J그룹의 명령어는 1 clock cycle동안 RF를 읽고 쓰는 동시에 PC를 변경하고, B그룹의 명령어는 ALU연산의 결과를 바로 이용하여 PC값을 변경할 수 있다. 따라서 두 경우 decoding이후 하나의 state만을 이용하여 명령어를 실행할 수 있다. 반면 R그룹 명령어는 ALU연산과 그 결과를 ALUOut reg에서 읽어 RF에 저장하는 것에 총 2 cycle이 필요하다. 예외적으로 MULT는 RF가 아닌 LO에 값을 저장하고, MFLO는 ALU연산을 하지 않는다.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| State | ALU | Reg | Mem | PC | Next |
| fetch (0) | pc+4 |  | inst. | ALU out | 1 |
| decode (1) | jump addr | read |  |  | J |B|R1 |
| J |  | read | write |  | ALUOut reg | 0 |
| B | 조건 비교 |  |  | ALUOut reg | 0 |
| R1 | alu op |  |  |  | R2 |
| R2 |  | write |  |  | 0 |

ROM file의 정보는 FSM의 state에 따른 control signal 정보이다. MultiCycle.v의 내용에서 ROM file이 각 state마다 40bit의 정보를 저장한다는 것을 확인할 수 있다.



이미 정의된 0번 state는 명령어를 메모리에서 읽어오는 동작을 한다. ROM file의 첫 40 bit의 내용을 signal별로 나누어 확인한 것은 다음과 같다.

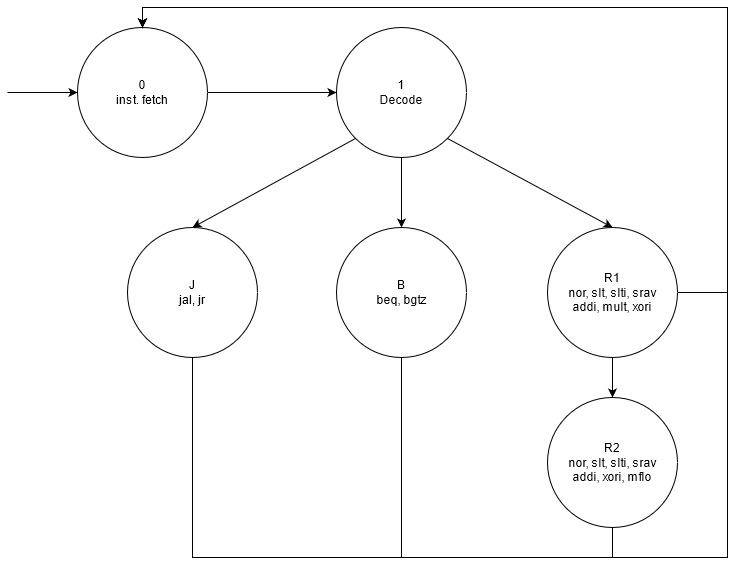
|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| State Num | 0 | 0 0 00 1 xx xxx 0 x 011 001 00100 0000 000 00 1 11 xxxxx | | | |
| lorD | 0 | inst mem access | ALUsrcA | 011 | ALU input A = PC |
| MemWrite | 0 | no mem write | ALUsrcB | 001 | ALU input B = 0x4 |
| DatWidth | 00 | 32 bit | ALUop | 00100 | a + b |
| IRwrite | 1 | inst reg write | ALUctrl | 0000 | (a, b), (shift amount) |
| RegDst | xx | don’t care | Branch | 000 | no Branch |
| RegDatSel | xxx | don’t care | PCsrc | 00 | from ALU out |
| RegWrite | 0 | no RF write | PCwrite | 1 | PC write enable |
| ExtMode | x | don’t care | StateSel | 11 | NextState += 1 |
|  |  |  | Reserved | xxxxx | don’t care |

각 signal이 instruction fetch를 위해 필요한 data path를 생성하도록 구성되어있다. 이와 같은 방법으로 주어진 12개의 명령어 수행에 필요한 signal을 작성할 수 있다. 이것은 프로젝트1의 내용과 유사하다.

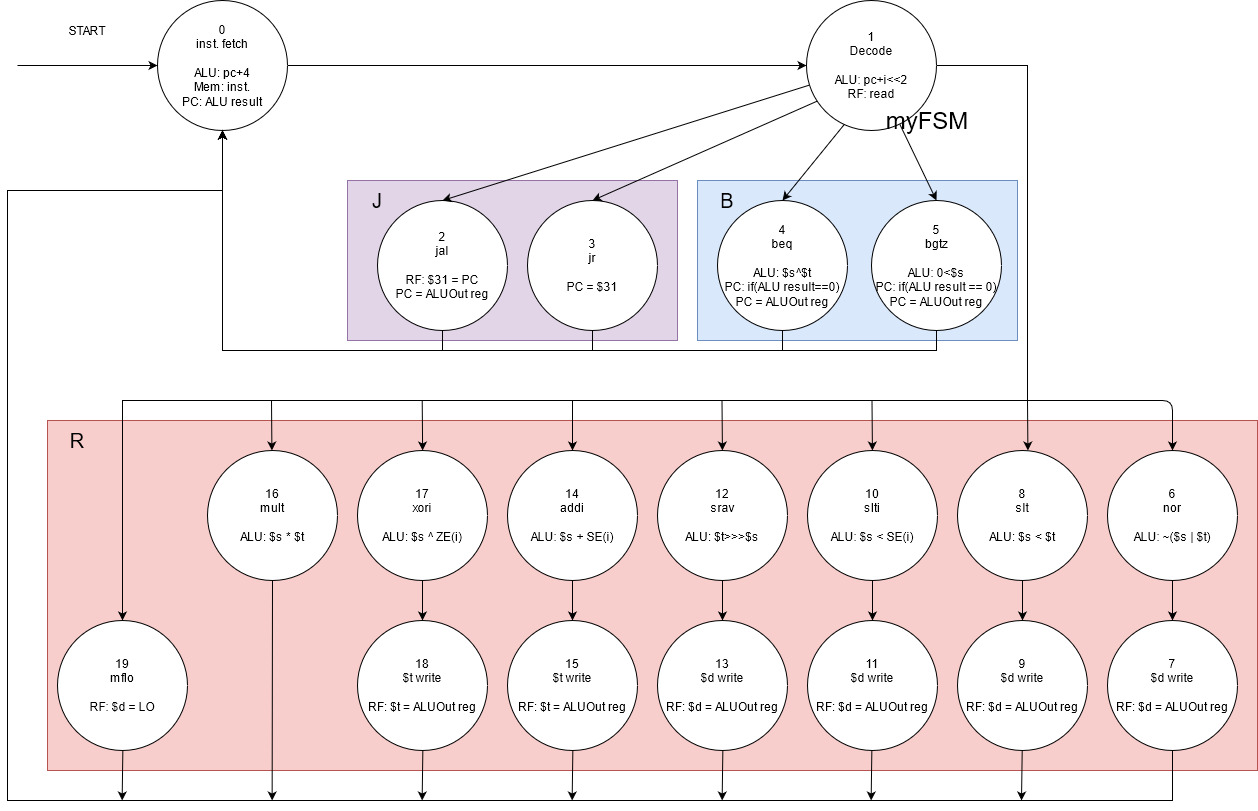
|  |  |  |
| --- | --- | --- |
| 0 | fetch | 0 0 00 1 xx xxx 0 x 011 001 00100 0000 000 00 1 11 xxxxx |
| 1 | decode | x 0 xx 0 xx xxx 0 1 011 100 00100 0000 xxx xx 0 01 00000 |
| 2 | jal | x 0 xx 0 10 100 1 x xxx xxx xxxxx xxxx 011 10 1 00 xxxxx |
| 3 | jr | x 0 xx 0 xx xxx 0 x 000 010 00001 0000 011 00 1 00 xxxxx |
| 4 | beq | x 0 xx 0 xx xxx 0 x 000 000 00011 0000 100 01 1 00 xxxxx |
| 5 | bgtz | x 0 xx 0 xx xxx 0 x 000 010 10000 0000 100 01 1 00 xxxxx |
| 6 | nor | x 0 xx 0 xx xxx 0 x 000 000 00010 0000 xxx xx 0 11 xxxxx |
| 7 | $d write | x 0 xx 0 01 000 1 x xxx xxx xxxxx xxxx xxx xx 0 00 xxxxx |
| 8 | slt | x 0 xx 0 xx xxx 0 x 000 000 10000 0000 xxx xx 0 11 xxxxx |
| 9 | $d write | x 0 xx 0 01 000 1 x xxx xxx xxxxx xxxx xxx xx 0 00 xxxxx |
| 10 | slti | x 0 xx 0 xx xxx 0 1 000 011 10000 0000 xxx xx 0 11 xxxxx |
| 11 | $d write | x 0 xx 0 01 000 1 x xxx xxx xxxxx xxxx xxx xx 0 00 xxxxx |
| 12 | srav | x 0 xx 0 xx xxx 0 1 000 000 01111 0001 xxx xx 0 11 xxxxx |
| 13 | $d write | x 0 xx 0 01 000 1 x xxx xxx xxxxx xxxx xxx xx 0 00 xxxxx |
| 14 | addi | x 0 xx 0 xx xxx 0 1 000 011 00100 0000 xxx xx 0 11 xxxxx |
| 15 | $t write | x 0 xx 0 00 000 1 x xxx xxx xxxxx xxxx xxx xx 0 00 xxxxx |
| 16 | mult | x 0 xx 0 xx xxx 0 x 000 000 01001 0000 xxx xx 0 00 xxxxx |
| 17 | xori | x 0 xx 0 xx xxx 0 0 000 011 00011 0000 xxx xx 0 11 xxxxx |
| 18 | $t write | x 0 xx 0 00 000 1 x xxx xxx xxxxx xxxx xxx xx 0 00 xxxxx |
| 19 | mflo | x 0 xx 0 01 010 1 x xxx xxx xxxxx xxxx xxx xx 0 00 xxxxx |

1. 설계 의도와 방법
   1. FSM Diagram

명령어를 그룹별로 나눈 FSM은 다음과 같다.



실제 구현한 fsm은 다음과 같이 나타난다.



* 1. Micro-Instruction의 Field 구분 및 Field 용도

ROM에 작성되는 Micro-Instruction의 구조는 다음과 같다.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| State Num | 0 | 0 0 00 1 xx xxx 0 x 011 001 00100 0000 000 00 1 11 xxxxx | | | |
| lorD | 0 | inst mem access | ALUsrcA | 011 | ALU input A = PC |
| MemWrite | 0 | no mem write | ALUsrcB | 001 | ALU input B = 0x4 |
| DatWidth | 00 | 32 bit | ALUop | 00100 | a + b |
| IRwrite | 1 | inst reg write | ALUctrl | 0000 | (a, b), (shift amount) |
| RegDst | xx | don’t care | Branch | 000 | no Branch |
| RegDatSel | xxx | don’t care | PCsrc | 00 | from ALU out |
| RegWrite | 0 | no RF write | PCwrite | 1 | PC write enable |
| ExtMode | x | don’t care | StateSel | 11 | NextState += 1 |
|  |  |  | Reserved | xxxxx | don’t care |

이때 몇 가지 signal은 서로 관련이 있다.

|  |  |
| --- | --- |
| lorD, MemWrite, DatWidth, IRwrite | 메모리 읽기/쓰기 |
| RegDst, RegDatSel, RegWrite | RF 쓰기 |
| ExtMode, ALUsrcA, ALUsrcB,  ALUop, ALUctrl | ALU 계산 |
| Branch, PCsrc, PCwrite | PC 쓰기 |
| StateSel, Reserved | Next state |

* 1. Micro-Instruction의 반복된 사용에 대한 분석

R그룹에 해당하는 명령어의 경우 MULT와 MFLO를 제외하면 모두 ALU계산을 수행하는 state와 RF쓰기를 수행하는 state로 나누어진다. 이때 RF에 ALUOut reg의 값을 쓰는 것은 두 가지 state로 정의한 뒤, 반복해서 사용할 수 있다. 그러나 이번 프로젝트에서 state를 이동하는 방법을 이해하지 못하여 적용하지 못했다. 따라서 각각 ALU계산 state이후 RF쓰기 state를 하나씩 생성하여 state # +1을 이용하여 다음 state로 넘어가는 방법을 이용하였다.

* 1. 시뮬레이션 결과와 예상 결과 비교 분석

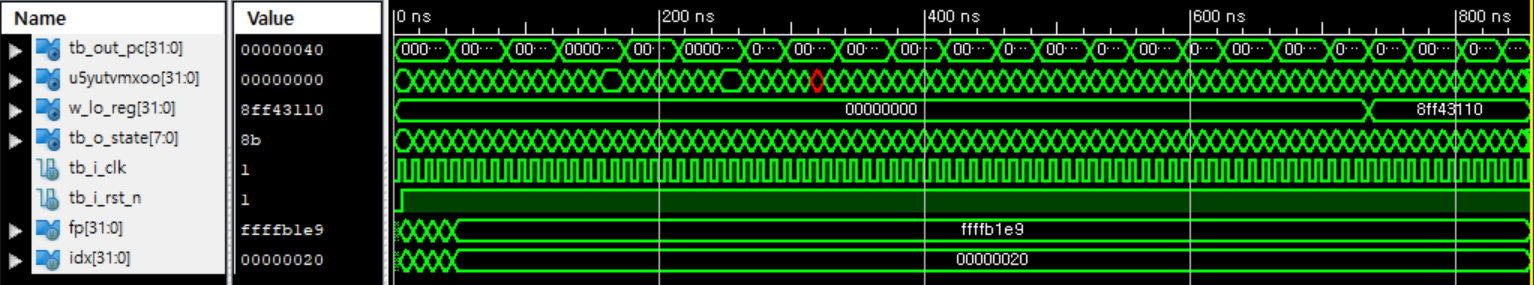
주어진 자료를 이용해 작성한 코드를 확인하였다. Sample Assembly Program은 다음과 같이 동작한다. 이 동작은 프로젝트1에서와 유사하다.

|  |
| --- |
| 0000: ori $r10, $r0, 0x110 |
| $r10 = 0x00000110 |
| 0004: add $r9, $r10, $r10 |
| $r9 = 0x00000220 |
| 0008: sub $r8, $r9, $r10 |
| $r8 = 0x00000110 |
| 000C: lw $r1, 0($r8) |
| 0x123435678가 저장되어있다.  $r1 = 0x12345678 |
| 0010: sw $r1, 4($r8) |
| 0014: lw $r2, 4($r8) |
| $r2 = 0x12345678 |
| 0018: j 0x020 |
| PC = 0x20 |
| 0020: lui $r3, 0x8765 |
| $r3 = 0x87650000 |
| 0024: ori $r3, 0x432C |
| 제안서와 달리 0x432C가 전달되었다.  $r3 = 0x8765432C |
| 0028: srav $r4, $r3, $r10 |
| $r4 = 0xFFFF8765 |
| 002C: nor $r5, $r4, $r0 |
| $r5 = 0x0000789A |
| 0030: beq $r1, $r2, 0x38 |
| PC = 0x38 // $r1 = $r2 = 0x12345678 |
| 0038: addi $r6, $r9, 1 |
| $r6 = 0x00000221 |
| 003C: jal 0x44 |
| PC = 0x44, $31 = 0x40 |
| 0044: slt $r7, $r9, $r6 |
| $r7 = 0x1 // $r9 = 0x220 < $r6 = 0x221 |
| 0048: bgtz $r7, 0x50 |
| PC = 0x50 // $r7 = 0x1 > 0 |
| 0050: lui $r2, 0x71e |
| $r2 = 0x071e0000 |
| 0054: xori $r2, $r2, 0x1301 |
| $r2 = 0x071e1301 |
| 0058: mult $r2, $r8 |
| hi = 0x7, lo = 0x8ff43110 |
| 005C: mflo $r7 |
| $r7 = 0x8ff43110 |
| 0060: slti $r10, $r7, 0x1 |
| $r10 = 1 // $r7 = 0x8ff43110 |
| 0064: jr $r31 |
| PC = 0x40 // $31 = 0x40 |
| 0x40: j 0x0 |
| PC = 0x0 |

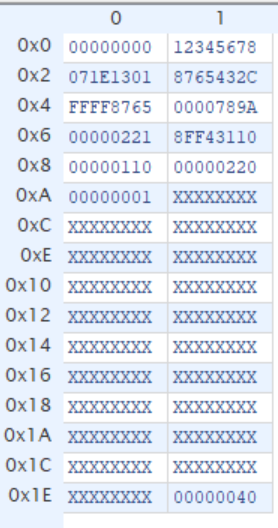
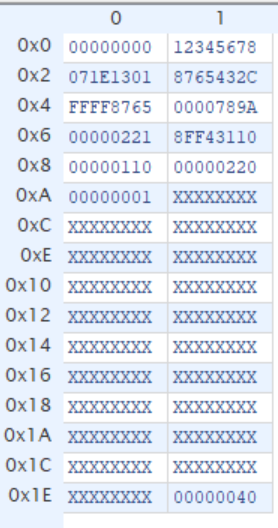
따라서 예상 결과는 다음과 같다.

|  |  |  |  |
| --- | --- | --- | --- |
| $r0 | 0x00000000 | $r1 | 0x12345678 |
| $r2 | 0x071e1301 | $r3 | 0x8765432C |
| $r4 | 0xFFFF8765 | $r5 | 0x0000789A |
| $r6 | 0x00000221 | $r7 | 0x8ff43110 |
| $r8 | 0x00000110 | $r9 | 0x00000220 |
| $r10 | 0x00000110 | $r31 | 0x00000040 |
| hi | 0x00000007 | lo | 0x8ff43110 |

시뮬레이션 결과는 다음과 같이 나타났다. 주어진 testbench에 ALU와 MULT module의 계산결과를 출력에 추가하여 값을 확인하였다.



프로그램은 모든 명령어를 시행한 다음 다시 처음으로 되돌아가기 때문에 최초 한 번 실행 이후인 850ns일 때 register에 저장된 값을 예상한 결과와 비교하면 다음과 같다.



|  |  |  |  |
| --- | --- | --- | --- |
| $r0 | 0x 0000 0000 | $r1 | 0x 1234 5678 |
| $r2 | 0x 071e 1301 | $r3 | 0x 8765 432C |
| $r4 | 0x FFFF 8765 | $r5 | 0x 0000 789A |
| $r6 | 0x 0000 0221 | $r7 | 0x 8ff4 3110 |
| $r8 | 0x 0000 0110 | $r9 | 0x 0000 0220 |
| $r10 | 0x 0000 0110 | $r31 | 0x 0000 0040 |
| hi | 0x 0000 0007 | lo | 0x 8ff4 3110 |

<예상 값> <실제 값>

두 값이 일치하는 것을 확인할 수 있다. 따라서 해당 프로젝트의 내용을 정상적으로 작성한 것을 확인할 수 있다.

1. 고찰

전체적인 과제는 1차 프로젝트와 유사하지만 FSM을 작성하는 과정이 추가되었고, Multicycle CPU의 Microprogramming을 해볼 수 있는 과제였다. 저번 프로젝트에서의 경험을 통해 signal을 구성하는 것에 큰 어려움은 없었으나 ROM의 내용을 이해하는 것에서 어려움을 겪었다. 이 문제는 MultiCycle.v의 내용을 확인하여 해결하였지만, Micro instruction의 마지막 5bit가 의미하는 것은 이해하지 못했다. 다음 cycle에 사용할 state를 지정하는 것으로 추측하였으나 실제 사용법을 알아내지 못해서 R그룹의 명령어에 대해 동일한 동작을 하는 state를 여러 개 만들어서 사용해야했다. 따라서 FSM의 구조가 복잡해지고, ROM의 길이가 늘어나는 문제가 발생하였다.

1. 참고자료

이성원교수님, 컴퓨터구조 강의자료, 광운대학교 컴퓨터정보공학과 2021