**Computer Architecture**

**Lab 10**

Build a 5 stage pipelined processor -KU-Dragon



**2013210128 변민재**

**1. Module : code 및 주석 (코드 파일 별도 첨부)**

|  |  |
| --- | --- |
| **registerfile** |  |
| **memory** |  |
| **components** |  |
| **pipeline\_**  **buffers** |  |
| **control\_**  **units** |  |
| **KUdragon** |  |

**2. 결과 waveform**

(별도 첨부)

* kudragon

**3. 결과 waveform 원인 분석**

1) Control Signal

main control unit에서는 Instruction Memory에서 읽어온 명령어의 Operation에 따라 Signal 값들을 결정해준다. 각 명령어에 따른 Signal 값은 다음과 같다.

표 1 명령어에 따른 Signal 값

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **명령어** | **Inst [31:26]** | **ALUSrc** | **ALUOp** | **Mem**  **Read** | **Mem**  **toReg** | **Mem**  **Write** | **Reg**  **Write** | **RegDst** | **Branch** | **Jump** |
| LW | 10 0011 | 0 | 00 | 1 | 1 | 0 | 1 | 0 | 0 | 0 |
| SW | 10 1011 | 1 | 00 | 0 | x | 1 | 0 | x | 0 | 0 |
| BEQ | 00 0100 | 0 | 01 | 0 | x | 0 | 0 | x | 1 | 0 |
| R-type | 00 0000 | 0 | 10 | 0 | 0 | 0 | 1 | 1 | 0 | 0 |
| JUMP | 00 0010 | x | x | x | x | 0 | 0 | x | 0 | 1 |

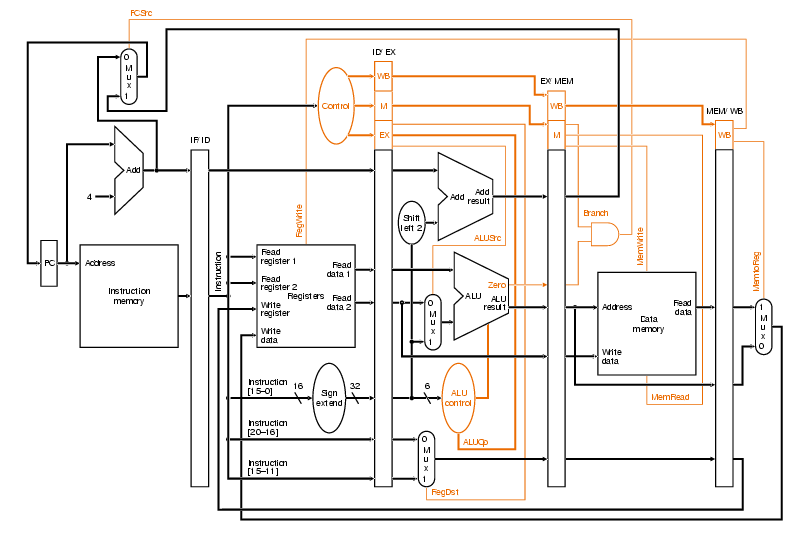
9개의 Signal값들은 각각 한 파이프라인 단계에서만 활성화된다.

표 2 단계별 활성화 Signal

|  |  |
| --- | --- |
| **단계** | **Signal** |
| IF | Instruction Memory에서 명령어를 읽어오기 위한 MemRead값은 항상 1이고, 쓰기 작업은 이루어지지 않기 때문에 MemWrite값은 항상 0이다. 이 값은 항상 인가되는 것으로 이 단계에서는 설정되는 Signal값이 없다. |
| ID | IF 단계와 마찬가지로 매 클럭 사이클마다 같은 일이 일어나기 때문에 설정되는 Signal값이 없다. |
| EX | 목적지 레지스터와 ALU연산을 선택하고 ReadData2와 SignExtendOut 중 하나를 ALU의 입력으로 선택하기 위해 RegDst, ALUOp, ALUSrc가 설정된다. |
| MEM | 분기일 때에는 Branch, 읽기일 때에는 MemRead, 쓰기일 때에는 MemWrite이 설정된다. |
| WB | ALU결과를 레지스터파일에 보낼지 메모리값에 보낼지를 결정하기 위해 MemtoReg가 설정되며, 선택된 값을 레지스터에 쓰기 위해 RegWrite이 설정된다. |

2) Buffer

buffer는 각 단계 사이에 존재하며 input, output은 다음과 같이 지정하였다.



1

2

1

2

3

4

5(5bit)

6(5bit)

1

2

3

4(5bit)

5(1bit)

1

2

buffer에서는 9개의 Signal값 중 다음단계에 필요한 Signal값을 전달 받는다. 또한 buffer에는 데이터가 항상 쓰여야 하므로 모든 buffer의 WE값은 1이다.

표 3 IDEX buffer의 Signal

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **IDEX\_**  **Signal[8]** | **IDEX\_**  **Signal[7]** | **IDEX\_**  **Signal[6]** | **IDEX\_**  **Signal[5]** | **IDEX\_**  **Signal[4]** | **IDEX\_**  **Signal[3]** | **IDEX\_**  **Signal[2]** | **IDEX\_**  **Signal[1]** | **IDEX\_**  **Signal[0]** |
| Branch | RegDst | RegWrite | ALUSrc | ALUOp[1] | ALUOp[0] | Mem  toReg | Mem  Write | Mem  Read |

표 4 EXMEM buffer의 Signal

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **EXMEM\_**  **Signal[4]** | **EXMEM\_**  **Signal[3]** | **EXMEM\_**  **Signal[2]** | **EXMEM\_**  **Signal[1]** | **EXMEM\_**  **Signal[0]** |
| Branch | RegWrite | MemtoReg | MemWrite | MemRead |

표 5 MEMWB buffer의 Signal

|  |  |
| --- | --- |
| **MEMWB\_**  **Signal[1]** | **MEMWB\_**  **Signal[0]** |
| RegWrite | MemtoReg |

3) KUdragon

KUdragon에서 사용되는 module에 대한 자세한 설명이다.

표 6 pipeline 단계별 module

|  |  |  |
| --- | --- | --- |
| **IF** | mux2to1 MUX0 | 새 PC값을 정해준다. |
| single register PC | WE가 인가되었을 때, PC값을 instruction memory로 전달해준다. |
| ALU ADD0 | 명령어를 인출한 후 PC값을 증가시킨다. |
| **ID** | registerfile RF | R 형식 ALU 연산 구현을 위한 레지스터파일로, 항상 read register입력이 지정하는 레지스터 내용을 출력하므로 제어 입력이 필요하지 않다. 그러나 쓰기 제어는 해주어야 한다. |
| signextd SE | 적재와 저장 구현에 필요하다. |
| forwardingunit FU | 필요할 때에만 전방전달하기 위해 RegWrite 활성화를 확인해야 한다. 명령어의 목적지가 $0이면 결과값을 전방전달할 필요가 없다. 파이프라인 레지스터와 ALU 입력 사이의 종속성을 해결하기 위해 ALU 입력에 mux를 추가해 해결한다. |
| hazarddetectionunit HDU | PC와 IF/ID 레지스터에 쓰는 것을 제어할 뿐 아니라 mux가 실제 signal값과 0중 하나를 선택하게 한다. 적재-사용 hazard 검사가 참이면 pipeline을 지연시키고 signal field를 0으로 만든다. |
| **EX** | ALU ADD1 | output값이 branch target으로 쓰인다. |
| mux2to1 MUX1 | ALU의 두번째 입력을 레지스터에서 (산술/논리 또는 분기 명령어의 경우) 받을 것인지 아니면 명령어의 변위필드 (적재 또는 저장 명령어의 경우)에서 받을 것인지 선택한다. |
| ALU ALU0 | Zero값이 branch control logic에 사용된다. |
| **MEM** | mem D\_Mem | 적재와 저장 구현에 필요하다. 읽기와 쓰기 중 하나만 인가가 가능하다. |
| **WB** | mux2to1 MUX3 | ALU출력 (산술/논리 명령어의 경우)와 데이터 메모리 출력(적재 명령어의 경우) 중 어떤 것을 register file에 쓸 것인지 결정한다. |
| mux2to1 MUX7 | 점프 목적지 (jump target)와 분기 목적지나 다음 주소 명령어 중 하나를 선택하는데 쓰인다. |
| shiftleft2 SL1 | 점프 명령어의 하위 26bit를 왼쪽으로 2bit만큼 자리이동하고 하위 bit에 00을 추가한다. |
| concatenate4to28 Con | PC+4의 상위 4bit에 SL1의 output 하위 28bit를 덧붙여 32bit 주소를 만든다. |

IF(Instruction Fetch) 단계에서는 PC에 있는 주소를 사용하여 명령어 메모리로부터 명령어를 읽고 IFID buffer에 저장한다. PC 주소는 Adder에 의해 4씩 증가되어 다시 PC에 저장됨으로써 다음 클럭 사이클에 저장된다. 증가한 주소는 BEQ와 같은 명령어에서 필요할 수 있으므로 IFID buffer에 쓰인다. ID(Instruction Decode) 단계에서는 명령어 해독 및 register file 읽기가 실행된다. IFID buffer에서 32bit로 signextend된 명령어 값과 레지스터 번호 두 개를 받아 증가된 PC값과 함께 IDEX buffer에 저장한다. EX(Execute) 단계에서는 실행 또는 주소 계산이 이루어진다. IDEX buffer로부터 register1의 내용과 signextend된 수치를 읽고 ALU를 사용하여 이들을 더한 후 EXMEM buffer에 저장한다. MEM(Memory Access) 단계에서는 메모리 접근이 이루어지는데 적재 명령어의 경우 EXMEM buffer에서 주소를 받아서 Data Memory를 읽고 읽은 데이터를 MEMWB buffer에 저장한다. 저장 명령어의 경우에는 IDEX buffer에 있는 저장되어야 하는 값을 받아 EXMEM buffer에 저장한다. 마지막 WB(Write-Back) 단계에서는 적재 명령어의 경우 MEMWB buffer에서 데이터를 읽어서 registerfile에 쓴다. 저장 명령어의 경우에는 아무 일도 일어나지 않는다.

표 7 pipeline 단계별 수행

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **IF** | **ID** | **EX** | **MEM** | **WB** |
| **8d42005a 인출**  LW R2, 90(R10)  PC값이 4로 증가 |  |  |  |  |
| **8d43005e 인출**  LW R3, 94(R10)  PC값이 8로 증가 | **(8d42005a)** 10과 2를 읽어옴 |  |  |  |
| **00432022 인출**  SUB R4, R2, R3  PC값이 12로 증가 | **(8d43005e)** 10과 3을 읽어옴 | **(8d42005a)** 10과 90을 ALU를 사용하여 더한 후 (주소값 100) EXMEM buffer에 저장 |  |  |
| **ad44005a 인출**  SW R4,90(R10)  PC값이 16으로 증가 | **(00432022)** 2와 3을 읽어옴 | **(8d43005e)** 10과 94를 ALU를 사용하여 더한 후 (주소값 104) EXMEM buffer에 저장 | **(8d42005a)** EXMEM buffer에서 받은 주소값 100에 해당하는 값을 Data Memory에서 읽고 (11223344) MEMWB buffer에 저장 |  |
| nop | nop | nop | nop | nop |
| **014c2820 인출**  ADD R5, R10, R12  PC값이 20으로 증가 | **(ad44005a)** 10과 4를 읽어옴 | **(00432022)** SUB연산 수행 : R2에 저장된 값(11223344) – R3에 저장된 값(55667788)=  bbbbbbbc | **(8d43005e)** EXMEM buffer에서 받은 주소값 104에 해당하는 값을 Data Memory에서 읽고 (55667788) MEMWB buffer 에 저장 | **(8d42005a)** 읽어온 데이터를 R2로 보냄 |
| **012d3020 인출**  ADD R6, R9, R13 | **(014c2820)** 10과 12를 읽어옴 | **(ad44005a)** 10과 90을 ALU를 사용하여 더한 후 (주소값 100) EXMEM buffer에 저장 | **(00432022)** 결과값 bbbbbbbc를 WB단계로 전달 | **(8d43005e)** 읽어온 데이터를 R3로 보냄 |
| **10a60005 인출**  BEQ R5, R6, R5 | **(012d3020)** 9와 13을 읽어옴 | **(014c2820)** ADD 연산 수행 : R10에 저장된 값+ R12에 저장된 값 | **(ad44005a)** buffer에서 받은 주소값 100에 해당하는 메모리에 R4의 값을 저장 | **(00432022)** bbbbbbbc를 R4로 보냄 |
| JUMP 연산에서는 항상 주소값 0을 가리키므로 @000에서부터 다시 명령어를 읽어와서 연산을 무한반복한다. | | | | |

**5. 구현시 어려웠던 점**

1) 모듈 이름을 코드 모듈의 이름과 같게 하거나 이름을 써주지 않았더니 시뮬레이션 자체가 되지 않고 Error Loading Design이 떴다. 이름을 바꾸어주었더니 시뮬레이션이 됐다. 컴파일러가 이런 사소한 실수를 잡아주지 않아서 고생했다.

2) 대부분 책을 참고하면서 코드를 짰는데 책에서 참고한 그림이 잘못된 부분이 있어서 고쳐야 하는 불편함이 있었다. 또한 책에서 각 단계별로 설명된 그림들이 다른 단계에 대한 부분은 생략되어있어 그림마다 회로구성이 달라서 맞추기가 힘들었다.

3) kustar에서는 control unit에서 current state와 next state가 있어서 각 단계에서 넘어가는 흐름을 잘 이해할 수 있었는데 kudragon에서는 그런 state가 없어서 흐름을 이해하기가 어려웠다.

4) buffer에서 각 선마다 크기가 달랐는데 그림에 따라 개수도 달라서 처음에는 맞추기가 어려웠다. buffer에 선을 잘못 연결해서 다시 코드를 짰는데 책의 303쪽 그림부터 짠 후에 책의 그림을 따라 모듈을 추가해주었더니 조금 더 수월하게 완성할 수 있었다.

5) 책에서는 OP가 JUMP일 때signal값이 나와있지 않았다. JUMP일 때 모듈 구성은 책의 272쪽 단일 싸이클 회로 그림을 참고했고, signal값은 MemRead, RegWrite, Branch, Jump만 설정해주었다.

6) ControlSignal에 제어값들을 묶어서 넣어준 뒤 한 비트씩 꺼내서 사용했는데 값을 넣어줄 때 거꾸로 넣어주어서 0비트 자리에 저장되어야 할 값이 8비트 자리에 저장되어서 결과가 이상하게 나왔다. 그래서 assign할 때 넣어주는 값을 반대로 넣어주었다.

7) 가장 문제가 되었던 점은 PC값이 4씩 증가해야 하는데 1050ns 지점에서 이상한 값이 나와서 그 후 주소 연산이 이상하게 되어서 명령어를 제대로 읽어오지 못하는 것이었다. 이상한 값이 나오는 부분을 보니 MUX0에서 input값은 정확한데 output값을 잘못 가져오는 것이었다. MUX0은 PCSrc를 select값으로 갖는데 PCSrc는 Branch&Zero이다. 그래서 Branch와 Zero값을 확인해보았더니 ALU에서 input값을 잘못 주어서 Zero값도 잘못 나오는 것이었다. ALU 모듈을 수정해주니 PC값이 잘 증가하였다.