**Lab 6. Cache Lab**

**1. Introduction**

기존에 제작한 Pipelined CPU위에 cache를 구현하는 것을 목표로 한다. 기존에 제작했던 CPU에서 우리는 이상적인 Memory에 대한 접근이 하나의 cycle안에서 가능한 이상적인 상황을 가정했다. 그러나 실제로는 Memory stage에서 memory에 직접 접근하기 위해서는 더 많은 cycle이 필요하고, 매번 이만큼의 cycle동안 stall을 하기 보다는 cache 구조를 만들어서 최근에 사용한 데이터는 더 빠른 접근이 가능하도록 설계하는 구조를 Verilog로 구현한다. 이를 위해서 우리는 cache가 동작하는 원리를 비롯하여 여러가지 설계 가능한 데이터 정책들을 이해하고, 문제 조건에 맞도록 구현하여야 한다.

**2. Design**

|  |
| --- |
|  |
| Figure 1. Lab 6에서 구현하고자 하는 Cache의 기본 Design |

구현에 앞서 만들고자 하는 Cache의 구조와 Execute stage와 Write back stage 사이에 cache를 다음과 같이 새롭게 연결하고자 하였다. 주어진 방식대로 한번에 4word만큼의 데이터가 흐르고, 그 중에서 적절한 word를 offset에 따라 접근할 수 있도록 하였다. 전체 cache의 크기 (Data Tank)가 128Bytes의 크기를 가짐으로, 32개의 word-set들을 저장할 수 있는 공간을 만들었고, Offset과 index를 제외한 나머지 비트를 tag로 사용하여 같은 장소에 저장된 데이터들을 서로 다르게 표현할 수 있도록 하였다. 이렇게 구현한 cache를 EX/MEM register와 Date Memory File과 연결될 수 있도록 설정하였다.

**3. Implementation**

위에서 설계한 Cache을 주어진 여러 데이터 정책들에 맞추어 구현하였다. 연결과 구현의 편의를 위하여 별도의 module을 사용하여 따로 구현하였고 Single-Level direct-mapped, 4-words per cache-line, Write-Through, Write-allocate이 가능하도록 구현하였다. 또한 각 Case마다 주어진 Latency 수가 잘 충족하도록 구현하였다. Cache와 필요에 따른 memory 접근 시 latency가 발생하는데 해당 상황마다 적절한 수의 stall을 할 수 있도록 구현하였다. 구현한 Cache를 기존에 설계했던 pipeline과 연결될 수 있도록 RISC\_V\_TOP을 수정하였다.

**4. Evaluation**

완성된 코드를 컴파일한 이후 주어진 3개의 testbench 파일을 사용하여 시뮬레이션 결과를 확인하였다. Inst와 For-loop에 대해서는 정상적으로 동작하는 것을 확인하였고, Sort TB에 대해서는 31개의 Testcase를 통과하는 것을 확인하였다. Read Hit, Miss 그리고 Write Hit, Miss에 대해서 문제에서 주어진 Latency만큼을 가진다는 것을 확인하였다. 마지막으로 cache가 없는 상황의 cycle을 계산하여 현재 나오는 cycle을 비교하였을 때, cache로 인하여 cycle수가 현저히 줄었다는 것을 잘 확인할 수 있었다. 결과는 아래 figure와 같다.

|  |
| --- |
|  |
| Figure 2. 주어진 Testbench 파일로 실행한 시뮬레이션 결과 – inst |

Inst의 경우, Lab5에서 31 cycle이 나옴을 확인하였었다. 이 때 TB에 LW와 SW이 각각 한번씩 포함되어 있음으로, 새롭게 바뀐 D\_MEM에 Cache가 없는 경우를 가정하면, LW는 7번의 cycle이, SW의 경우 15번의 cycle이 증가하여야 하여, 만약 cache가 없었다면 총53 cycle을 가졌을 것이라고 계산할 수 있다. 구현한 코드의 경우 한번의 write miss, 한번의 read hit가 발생하여 31+17+0 = 48번의 cycle이 정확히 나옴을 확인할 수 있다.

|  |
| --- |
|  |
| Figure 3. 주어진 Testbench 파일로 실행한 시뮬레이션 결과– forloop |

|  |
| --- |
|  |
| Figure 4. Forloop에 포함된 LW, SW 개수 확인 |

For Loop의 경우, Lab5에서 97번의 cycle이 나옴을 확인하였다. 이번 랩에서 cache가 없었다면 몇 개의 cycle이 나왔을 지 파악하기 위해서 해당 testbench에 몇 개의 LW와 SW가 들어가는지 프린트해주는 코드를 작성하여 돌렸고 그 결과 (figure 4) LW는 22개, SW는 12개가 나옴을 확인할 수 있었다. 따라서 cache가 없었더라면 97번의 cycle + LW당 7cycle (7 \* 22) + SW당 15cycle (15 \* 12) = 431 cycle이 나옴을 계산할 수 있다. 그러나 cache를 구현한 코드에서는 (figure 3) 220번의 cycle이 실행되었음을 알 수 있고, 현저히 cycle수가 줄었다는 것을 확인할 수 있다. 해당 TB에서는 9번의 write hit, 3번의 write miss, 22번의 read hit가 발생하였으며, 각각의 경우에 필요한 cycle수를 더해보면 (write hit당 8cycle 증가, write miss당 17cycle 증가, read hit는 증가 없음) 97 + 72 + 51 = 220 cycle이 정확하게 나온다는 것을 확인할 수 있었다. 마지막으로 sort의 경우 30개의 test를 통과하고, 31번째의 testbench부터 답이 다르게 나오는 것을 확인하였다. 결과는 아래 figure 5와 같다.

|  |
| --- |
|  |
| Figure 5. 주어진 Testbench 파일로 실행한 시뮬레이션 결과- sort |

|  |
| --- |
|  |
| Figure 6. Read – Hit Waveform (1 cycle) |

|  |
| --- |
|  |
| Figure 7. Read – Miss Waveform (10 cycles) |

|  |
| --- |
|  |
| Figure 8. Write – Hit Waveform (9 cycles) |

|  |
| --- |
|  |
| Figure 9. Write – Miss Waveform (18 cycles) |

Figure 6부터 9까지는 가능한 Cache의 동작 경우들에 대하여 waveform을 분석한 내용들이다. 각각의 경우 주어진 cycle 수에 맞추어 잘 동작하는 것을 알 수 있다. \*Figure 7의 Read-Miss의 경우, 그림에서 볼 수 있듯, 해당 LW inst로 인한 Data Hazard가 발생하여 Cache로 인한 stall 10 cycle 직후에 1 cycle stall이 추가된 것을 알 수 있다. 따라서 해당 경우에 Read-Miss인 경우에 필요한 cycle 수는 10 cycle이라고 볼 수 있다.

**5. Discussion**

구현해야 하는 Cache가 어디에 어떤 형태로 들어갈지, 어떻게 구현할 수 있을지에 대한 고민이 필요하였다. Pipeline CPU Lab과 마찬가지로 디버깅에 많은 어려움을 겪었다. Write-hit, miss, Read-hit, miss 각각의 상황에 대해 각각 다른 cycle로 정상 동작하도록 설계하고 구현하는데 어려움이 있었다. 구현 이후에는 Cache가 없는 상황과 비교하기 위한 또 다른 전략이 필요하여 이를 구현하는 것 또한 시간이 소요되었다. Sort를 디버깅하는데 굉장히 많은 시간이 소요되었는데 특히 classum에 올라온 CSN과 관련된 내용을 파악하는데 많은 시간을 소요하였다. 그 결과, 30개의 test가 통과하게 되었다. Cache에서 올바른 값이 나옴에도 불구하고 31번째 test의 경우, Output port로 signal을 보내는 타이밍이 정상적으로 작동하도록 lab5에서 조정하여 구현하였던 것이, D\_MEM과 Cache가 Negative Edge에서 작동하면서 해당 경우에는 output이 나가는 타이밍이 맞지 않는 것으로 보여진다.

**6. Conclusion**

복잡한 Pipeline 구현에 이어서 실제 메모리와 유사한 상황에서 더 좋은 성능을 내도록 하는 Cache까지 모두 구현을 해봄으로써 현재 사용되는 컴퓨터 아키텍처가 어떤 형태로 구현되는지 더욱 깊게 이해할 수 있었다. 특히 이번 랩에서는 cache가 어떤 원리로 어떻게 동작해야 하는지, 이 것이 어떤 효과가 있는지에 대해 직접 경험이 가능하였다. 지난 과제들을 통해 컴퓨터 구조에 대한 지식은 물론, Verilog에 대한 이해도와 숙련도가 모두 올라갔다.