# Advenced Computer Organization and Architecture

## Project 1 Report

**201220994 정기석**

### 1. Design and Implementation

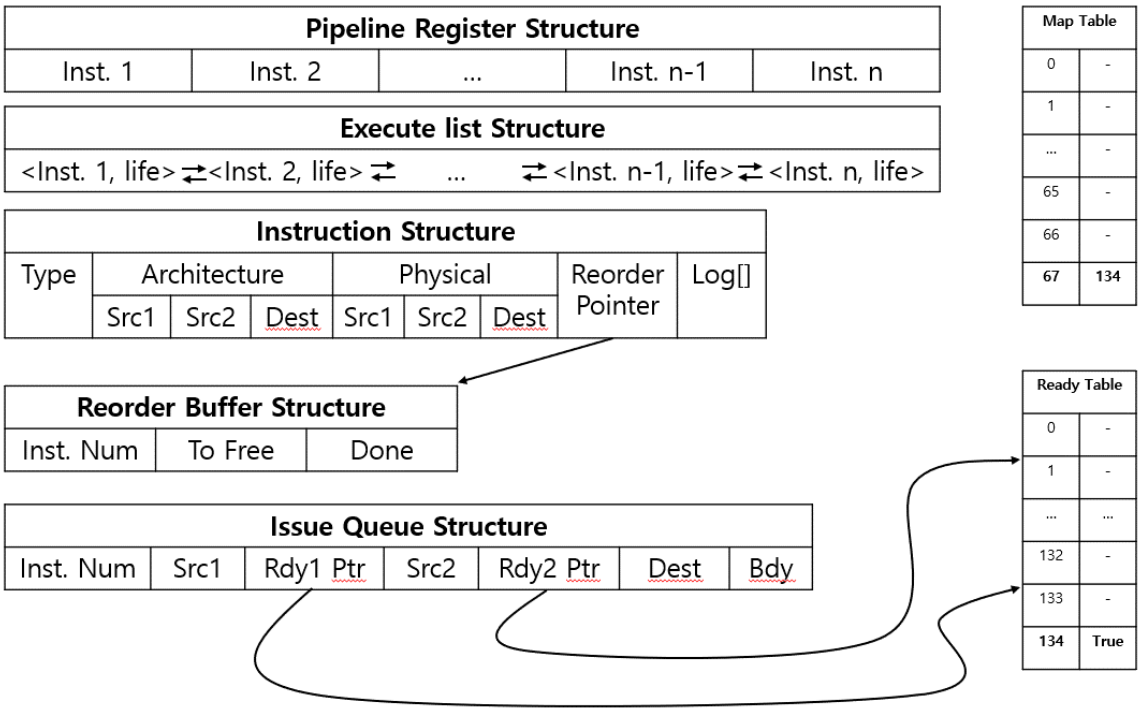



**Figure 1 Structures**

전반적인 데이터 구조는 **Figure 1**과 같다. Execute list를 제외한 Pipeline Register들은 Instruction Index를 저장한 Vector(배열) 구조이다. Register에 모든 instruction 정보를 저장하는 것이 아닌 index를 통해 instructions vector에서 명령어 정보를 가지고 온다.

Execute list structure도 다른 pipeline 구조와 비슷하다. 다만 실행 시간을 기억하기 위해 Instruction index와 life를 pair 형태로 저장하며 순서대로 삭제되는 것이 아닌 life가 0이 되는 명령어부터 삭제되므로 List(Linked list)로 기억한다.

Instruction Structure에서 Rename State 이후 확인할 수 있는 Old Physical Destination register를 알 수 있게 되므로 Reorder Pointer 저장하여 ROB 구조의 To Free의 변수를 변경할 수 있도록 한다. Log 배열은 State마다의 Cycle을 기록하여 저장할 수 있도록 한다.

Issue Queue structure에서 Ready table은 Dependency에 따라 바뀌므로 포인터로 참조하여 접근하도록 한다.

Map Table에서 Architecture Register 개수보다 1개 많은 67번 reg와 Ready Table에서 134번이 존재하는데 이는 reg가 존재하지 않는다는 뜻의 Reg -1번이 들어왔을 경우에 67번으로 재지정함으로써 예외처리를 하도록 한다. 이때 Ready Table 134번은 언제나 True를 나타낸다.

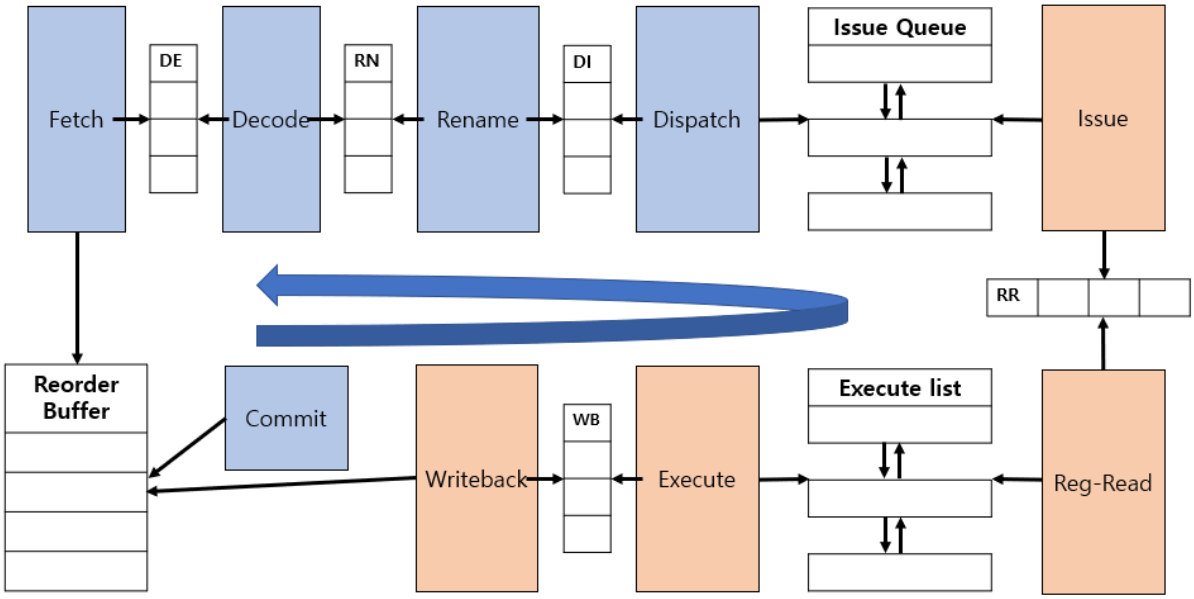



**Figure 2 Simplified program chart**

대략적인 프로그램의 흐름은 **Figure 2**와 같다. 화살표 방향은 각 State에서 데이터 접근을 나타낸다. 각 State는 Commit -> WriteBack -> ... -> Fetch와 같이 역순으로 진행한다.

### 1.1 Fetch

먼저 더 읽을 명령어가 있는지 확인한다. 읽을 수 있다면 Reorder buffer와 DE 레지스터의 공간이 비어 있는지 확인한다. 읽을 명령어가 더 이상 존재하지 않거나 ROB와 DE 모두 비어 있지 않으면 아무 일도 하지 않는다. 두 공간이 모두 비어 있다면 명령어를 읽어온 후 Reg -1번의 예외처리를 해준다. 이후 Reorder Buffer에 등록한 뒤 instruction에 존재하는 reorder pointer를 등록해준 뒤 명령어 배열에 등록한 후 DE Register에 등록한다. 이 과정을 WIDTH 횟수만큼 반복 후 다음 Cycle로 넘어간다.

### 1.2 Decode

이번 프로젝트에선 이번 State에서는 실질적인 동작이 없다. RN 레지스터에 공간이 있고 DE 레지스터에 값이 존재한다면 DE 레지스터의 값들을 등록한다.

위 과정이 더 이상 불가능 할 때까지 반복한다.

### 1.3 Rename

남은 Phy. Reg가 존재하고 DI가 비어 있다면 Architecture Reg를 Physical Reg로 바꾸어 준다.

Physical Dest. Reg가 사용된다면 false dependency를 제거하기 위해 Free Physical register queue에서 사용 중이지 않은 Reg를 가져와 Dest Reg로 대체한다. 이때 이전에 존재하던 Phy Dest Reg는 Reorder buffer에 To free에 등록됩니다. 또한 map table에서도 해당하는 reg가 변경된다. 이후 RN의 값을 DI에 등록한다.

위와 같은 과정을 Free Physical Register가 남아있고 DI Register가 비어 있고 RN 레지스터에 값이 존재하면 계속 반복한다.

**1.4 Dispatch**

DI에 값이 있고 Issue Queue가 비어 있다면 아래 과정을 반복한다.

DI의 Inst. Num을 이용해 Issue Structure를 생성하여 Queue에 저장한다. 이때 ready의 포인터와 Physical src1, src2, dest 그리고 Bdy가 같이 저장된다.

**1.5 Issue**

RR 레지스터가 비어 있으면 Issue Queue에 존재하는 모든 명령어에 대하여 Src1 Ready와 Src2 Ready가 모두 True일 때 RR 레지스터에 값을 등록한다.

**1.6 Reg-Read**

RR 레지스터에 값이 존재하고 execute list가 비어 있다면 execute list에 값을 저장한다. 이때 Latency에 해당하는 life값을 같이 저장해준다.

Reg-Read에서의 작업은 Decode State에서 동작하는 것이 실제 구조에선 맞지만 별도로 Latency를 저장해야 하는 Structure의 추가로 인한 복잡성 증가로 Reg-Read에서 Latency를 저장하도록 한다.

**1.7 Execute**

Execute list에 명령어가 존재하고 WB 레지스터가 비어 있다면 Execute list에 있는 모든 명령어의 life를 1씩 감소시키면서 life가 0이 된 명령어들의 Physical Dest Reg의 Ready table을 true로 변경한 뒤 WB 레지스터에 명령어를 등록하도록 한다.

WB 레지스터는 non-blocking을 위해 항상 비어 있지만 차후 변경사항에 대비하여 따로 변경하지 않는다.

**1.8 WriteBack**

Execute 단계의 non-blocking을 위해 WIDTH와 관계없이 WB레지스터에 있는 모든 명령어에 대하여 Reorder buffer의 done 값을 ture로 변경한다.

### 1.9 Commit

In-order를 위해 Reorder Buffer의 명령어가 done인지 확인하여 최대 WIDTH개 만큼의 done 명령어를 Reorder Buffer에서 제거하고 Log를 출력하도록 한다. 이때 done이 아닌 명령어를 만나면 작업을 하지 않고 Commit State를 종료한다.

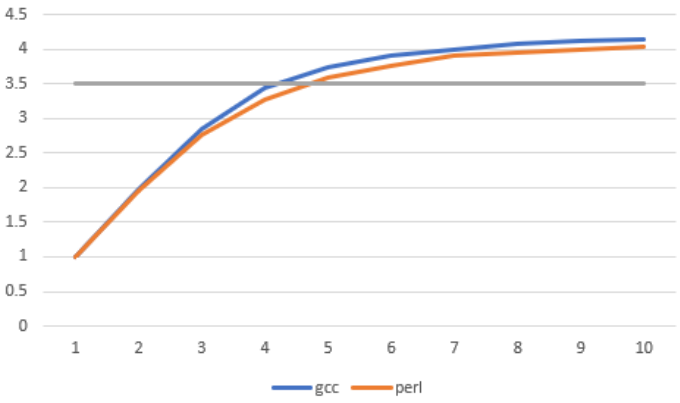
### 1.10 Advance Cycle

위 모든 State가 한 번씩 실행되면 Cycle를 1 더해주는 과정을 거친다. 이때 Reorder Buffer가 비어 있다면 모든 명령어가 끝났다고 판단할 수 있기 때문에 Simulation을 종료하도록 한다.

## 2. Analysis

Width, ROB, IQ parameter들에 대한 gcc와 perl instruction들의 IPC 성능 관계를 비교하고 원인을 파악하고자 한다.

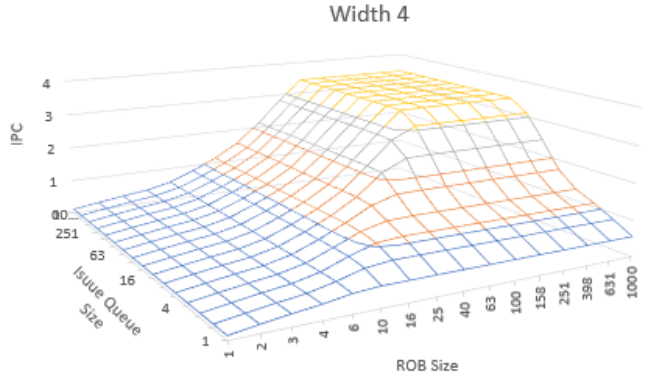
### 2.1 Width 크기에 따른 성능 향상




ROB와 IQ가 최적인 상황에서 Width에 따른 성능 향상은 왼쪽 그림과 같이 약 4정도에서 급격히 효율이 떨어진다.

이는 명령어 간의 의존성에 따른 결과로 보인다.

### 2.2 ROB와 IQ 크기의 상관관계

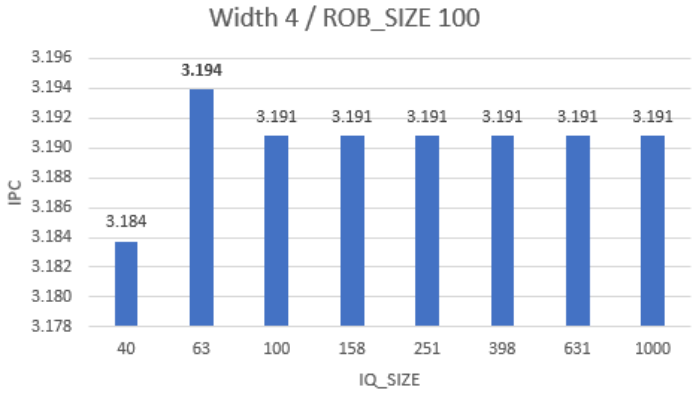



ROB와 IQ 크기의 상관 관계는 왼쪽 그림에서 모서리에 해당하는 부분을 비율로 나타내면 ROB : IQ ≒ 1:0.6와 같아진다. 즉, ROB의 0.6배 이상의 IQ 사이즈는 성능에 영향을 끼치지 못한다.

이러한 관계의 원인으로 판단되는 것으로는 ROB는 9 Cycle(1 Latency Inst ~ 13Cycle(5Latency Inst)가 소요될 때까지 하나의 명령어가 유지되고 IQ는 4 ~ 8 Cycle동안 하나의 명령어가 유지된다. 즉, IQ에서 하나의 명령어가 최대로 저장되어 있는 시간은 ROB에 비해 0.615 (8/13)배 이므로 ROB의 크기의 비해

0.615배 이상으로는 성능 향상이 없어져 **0.615 ROB = IQ**라는 결론에 도달한다.

### 2.3 Physical Register의 개수와 IQ의 상관관계




Perl Instruction들에서 왼쪽 차트와 같이 IQ Size가 커지면 오히려 성능이 약간 낮아지는 모습을 보인다.

이러한 원인으로 판단되는 것으로는 Physical Register가 고갈되어 Rename State에서 더 이상 새로운 Register를 지정해주지 못해 그 다음 Register인 DI에 명령어를 전달하지 못해 하나의 State가 완전히 비어 있게 되는 것이다. 이러한 이유로 IQ Size가 크게 늘어나도 성능 향상이 없는 것이고 또한 이러한 고갈이 생길 때마다 DI Register를 채우지 못해 Dispatch State가 동작하지 않아 1 Cycle이 낭비되게 된다.

지금 프로젝트에서 Architecture Register 67개에 Physical Register를 각각 할당하여 남는 Free Physical Register들은 67개이다. 이때 DI Register가 비어 있지 않도록 Width개를 제외한 나머지 모든 명령어에 Physical register를 할당한다고 하면 63개의 명령어에 할당할 수 있으므로 IQ Size는 63으로 설정할 수 있을 것이다.

즉, 최적의 **IQ Size = # Physical register - # Architecture Register – Width**라고 볼 수 있다.

### 2.4 결론

위의 내용들로 분석해본 결과 두 종류의 Instruction들(gcc / perl) 모두 width는 4-5정도에서 성능과 비용에서 타협을 볼 수 있을 것이다. 그리고 ROB와 IQ크기의 관계는 0.615ROB = IQ이다. 다만 ROB와 IQ 부분에서 최대 실행 Cycle과 관련이 있으므로 Cycle의 변화에 따라 숫자(0.615)가 바뀌게 될 것이다. 마지막으로 Physical Register의 개수에 따라 최적의 IQ Size에 영향을 끼친다는 사실을 확인할 수 있었다.

이러한 분석을 통해 이번 프로젝트의 조건에서 Width는 4, IQ size = 63, ROB = 102로 최적의 Parameter들을 확인할 수 있다.