프로세서 설계 초안입니다.

기본적으로 3-way superscalar machine, 매 cycle마다 2개의 instruction fetch를 가정합니다.

Fetch-Decode-Dispatch-Execute-Complete-Retire 순의 파이프라인을 가정합니다.

Execute 단계는 3개의 functional unit으로 구성됩니다.

명령어를 3종류로 나누겠습니다: ALU instruction, Control instruction, Memory instruction

1. ALU instruction

Memory에 접근하지 않으며, control flow(즉, 다음 PC의 값)을 변경하지 않는 모든 instruction을 말합니다. 이 instruction들은 fetch후 decode stage서 source register의 read를 수행한다. 이때, Register file의 busy/valid bit을 읽고, 값을 바로 가져오거나 아니면 값이 준비되지 않았으므로 해당 operand에 대한 tag만 주어서 RS로 보내거나 하는 두 가지 행동을 취할 수 있다(data dependency handle). RS에 들어가면 dispatch stage이며, 이때 적절한(나중에 정해야 함) 알고리즘에 의해 operand가 준비된 instruction을 issue한다. (Centralized RS의 경우)이 시점에서 어떤 FU로 issue할지를 결정하게 된다. (\*issue를 어떤 FU로 할지, 동시에 준비된 경우 어떤 알고리즘에 따라 issue할지도 정해야 합니다.) Execute stage에서 곱셈/나눗셈/FP연산 등은 multiple cycle이 걸릴 수도 있습니다. 이를 어떤 방식으로 구현할지(pipeline화 할지 아니면 단순히 multi cycle이 걸리는 연산으로 할지 등… 곱셈 나눗셈의 경우 다른 정수 명령어들과 같은 파이프라인을 공유한다는 것 역시 생각해 보아야겠죠)고민해볼 필요가 있습니다. Completion stage에서는 ROB에서 in-order 출력이 이루어지며, Retire는 store 명령어를 포함하지 않으므로 동시에 이루어집니다.

1. Control instruction

Jump, conditional branch instruction이 해당됩니다. 추가로 system call을 호출하는 ecall등도 해당됩니다. 이 경우 어느 시점에 PC가 바뀌어야 하는지는 명령어에 따라 달라집니다. 먼저, jump, ecall등의 경우 jump 조건에 대한 판단(연산)없이 해당 위치로 jump하면 되므로, decode stage서 jump하는 것이 가장 좋습니다. Conditional branch의 경우 조건 판단이 필요하며, 학교 수업을 기준으로 branch prediction 여부에 따라 execution stage 혹은 그 이후에 pc값의 update가 수행됨을 배웠을 것입니다. 우리 프로세서의 경우 decode 단계에서 분기 예측이 수행될 경우 dispatch 단계에서 여기에서 따로 고려해야 할 사항은 분기예측을 수행할 경우 이에 대해서 반드시 검증이 계속되어야 하며, 잘못된 분기 시에는 flush가 필요하다는 것입니다. 이때, branch instruction 이후의 모든 명령어에 대해 tag를 달아서 해당 예측이 맞았다는 점이 확인될 때까지 ROB에서 빠져나가지 못하게 하는 것이 필요합니다.

1. Memory instruction

Load/Store 명령이 해당됩니다. ALU와 비슷한 진행을 가집니다. Execute stage서 address generation이 발생하고, 그 이후에 memory 접근이 이루어집니다. 따라서, FU 자체가 여러 개의 pipe로 이루어져 있을 수 있습니다. 주된 차이점은 memory instruction들의 경우 cache hit fail시 re-issue된다는 점, 그리고 store instruction의 경우 store buffer를 가지고 있으며, 자기들끼리의 dependency handling mechanism을 추가로 가져야 한다는 점을 들 수 있습니다.

위처럼 명령어의 종류별 제어를 알아보았기 때문에, 각 stage 별로 어떤 기능이 들어가야 할지 대략은 명확해졌다고 생각합니다.

실제로 위와 같은 기능을 구현하기 위한 많은 디테일들이 누락되어 있습니다. 예를 들어 register renaming을 위해 필요한 RRF의 숫자나 bit에 대한 정확한 명시는 하지 않았습니다. 다만, 기본적인 개념들은 이전 superscalar+AHB 파일에서 좀 더 자세히 다룬 바가 있으니, 그것을 기반으로 알고리즘을 익히고 설계하면서 수정해 나가면 될 것 같습니다.

위 설명은 uncore 부분은 다루지 않았습니다. 그러나 cache의 경우 크게 복잡하지 않으며, memory와의 통신 및 I/O의 경우 bus문제가 해결되어야 해서 일단은 제외했습니다.

간략하게만 설명하자면, cache의 경우 instruction cache와 data cache를 쪼개서 구현합니다. 문제는, OS 등 상위의 software 구조가 잡히기 전에는 data와 instruction의 구분이 어려울 수 있다는 점입니다. Interrupt, 즉 비동기 예외 처리는 실질적으로 fetch를 중단하고 코어에서 현재 실행 중인 명령어만 처리하면 되기 때문에 크게 복잡한 것이 없을 것으로 생각됩니다. I/O의 경우가 많이 복잡한 편인데, 먼저 우리는 Memory-mapped I/O를 채택하고 있고, 이 의미는 load/store 명령어를 통해 I/O 읽기/쓰기를 수행한다는 것이며 따라서 load/store시 특정 주소에 대해서는 cache에 대한 접근이 아니라 I/O 버스에의 접근을 수행해야 합니다. 즉, memory쪽을 개발하는 사람이 이에 대해서 신경써줄 필요가 있습니다. 또, cache의 update도 bus에 대한 접근을 필요로 하므로 이 부분도 bus에 대한 해결이 필요합니다. DMA의 경우 간단하게 설계하면 쉽게 짤 수 있을 것으로 보입니다(시작 주소, 길이 주고 I/O 대행시키면 되니까… 특정 peripheral처럼 생각하고 제어 레지스터에 값 쓴다는 생각으로 설계하면 될 듯함). Cache와 함께 가장 쉬우면서도 중요한 부분은 (TLB)를 포함한 address translation으로, cache와 함께 이 부분이 가장 먼저 준비되어야 할 것으로 생각됩니다.