수퍼스칼라 프로세서 설계 고려사항

1. Superscalar basics

Scalar pipelined processor는 IF-ID-RD-EX(MEM)-WB의 순서로 명령어가 실행된다. 이때, superscalar processor는 RD와 EX 사이에 dispatch buffer를 두고, EX와 WB 사이에 Reorder buffer를 두어 EX stage서 out of order(OoO또는 O3로 씀)수행을 할 수 있게 하며, EX서 여러 개의 실행 유닛을 병렬적으로 두어 여러 명령을 병렬적으로 처리할 수 있게 한다. 즉, 여러 명령어가 비순차적으로 병렬 실행될 수 있도록 하는 방법을 찾는 것이 superscalar 설계의 핵심이다.

Superscalar processor 구조는 IF-ID-RD-EX(MEM)-WB 순서보다 Fetch-Decode-Dispatch-Execute-Complete-Retire 순으로 표현하는 것이 더 편리할 수 있다.

Fetching 시 여러 instruction을 인출해야 한다(개인 의견으로, 3-wide이니 1cycle에 2개의 instruction을 인출하면 될 것 같음). 이때, 서로 다른 cache line 두 개 사이에 1cycle에 인출해야 하는 명령어가 겹치면 throughput이 줄어들기 때문에, alignment HW의 도입을 고려할 수 있다. 그 예시로 s개의 instruction fetch하는 구성에서, instruction memory를 s개로 쪼개고 인출하는 것을 들 수 있다(반드시 이러한 방법만 있는 것은 아님).

RISC 프로세서의 decoding은 주로 명령어 간 dependency를 찾고, control flow 변화를 확인하는 기능을 말한다. 이때, superscalar processor는 한 cycle에 여러 instruction을 decoding해야하므로 multiport register가 필요할 수 있다(설계 시 필요하면 multiport화 하면 됨).

Dispatching에서는 decoding stage에서 판별된 instruction의 종류에 맞추어 올바른 functional unit(execute stage)로 보내는 역할을 한다. 이때, decode는 됐지만 operand가 준비되지 않은 명령어를 위한 buffer가 필요하며, 이를 RS(reservation station)라고 한다. RS에는 centralized, distributed의 두 종류가 존재한다. 준비된 명령어는 FU로 issue된다.

Execution에서는 여러 FU서 병렬적으로 연산이 진행된다.

실행이 비순차적으로 일어나므로, register 또는 memory에 쓰기 전에 이를 다시 순차적으로 짜맞춰야 한다. 연산을 마친 명령어는 finish 상태라고 하며, 이 상태의 명령어는 completion buffer에 들어간다. Completion buffer는 reorder buffer(ROB)라고도 한다. 여기서 명령어들은 in-order로 빠져나간다. ROB에서 빠져나온 명령어들은 결과값을 레지스터에 쓰며, 이를 completion이라 한다. 이때, memory 쓰기 명령어, 즉 store 명령어들은 store buffer에 들어가며, store buffer에 있는 명령어들은 store가 끝나면 buffer 밖으로 방출되어 완료된다. 완전히 실행이 끝난 명령어는 retire되었다고 하며, store가 아닌 명령어들은 completion과 retire가 동시에 일어난다고 볼 수 있다.

Interrupt는 새로운 fetch 중단 후 pipeline 내 명령어를 모두 처리하고 상태를 저장한 후 OS에 권한을 넘겨 수행된다고 볼 수 있다.

Exception은 exception이 발생한 명령어를 표기하고 ROB에서 이전 명령어는 모두 completion 시킨다. Pipeline 내의 이후 명령어는 모두 flush하고 state 저장 후 예외처리 루틴으로 넘어간다.

1. Branch prediction

\*따로 조사하시는 분이 있으니 technique에 대한 자세한 내용보다 superscalar에 적용시 고려할 부분 위주 설명

Wide superscalar processor에서 fetch된 instruction 중 여러 개가 conditional branch일 수 있다. 이 경우 multiported BTB를 이용해 모든 branch를 한 번에 계산할 수도 있다(개인 의견: 이 부분에 대해 우리가 이렇게까지 할 이유는 없어 보임).

1. Register renaming: Handling false data dependencies

Scalar processer 설계를 배우면서, dependency는 RAW, WAR, WAW 3종류가 있음을 배운 바 있다. 뒤의 두 종류는 false dependency로서 상관이 없었지만, superscalar processor는 OoO수행을 하므로 이를 고려해야 한다(같은 위치에 A를 쓰고 B를 써야 하는데 B를 쓰고 나서 A를 쓰면 결과값이 달라진다). 이를 해결하기 위한 방법이 register renaming이다.

일반적으로 사용하는 Register를 architected register file(ARF)라 하고, renaming을 위해 사용하는 추가 register를 rename register file(RRF)라 한다. 기본적으로 dependency가 존재하는 instruction에 대해 RRF를 이용해 임시로 값을 저장하고 나중에 ARF로 update하는 식으로 register renaming이 이루어진다. 일반적으로, RRF의 entry 수는 ARF보다 적지만, RRF의 각 reg들은 어느 architected register에도 rename 가능하도록 flexibility를 주는 방식으로 구현된다. ARF는 기본적으로 data에 더해 각 entry별로 busy bit, tag field를 가진다. Busy bit은 ARF의 해당 entry가 renaming 되었는지를 표시하기 위해, tag field는 RRF entry를 가리키기 위해 이용된다. RRF는 data값과 valid, busy bit으로 구성되며, 위치에 따라 2가지 방식으로 구현 가능하다. 일반적인 경우를 가정하는 경우 ARF와 유사한 standalone 구조로 RRF를 생각할 수 있다. Worst case 가정 시에는 ROB의 각 entry마다 하나의 RR을 붙이는 방식으로 RRF의 구현을 생각해볼 수 있다.

Register renaming은 source read-destination allocate-register update의 3단계로 이루어진다.

먼저, source read는 일반적으로 decode stage서 행해진다. Register operand를 fetch하려고 하면 3가지의 가능성이 존재한다. 첫 가능성은 busy bit이 set되지 않은 경우이다. 이 경우 그냥 가져다 쓰면 된다. 두 번째 가능성은 busy bit이 set이고 해당하는 RRF entry가 valid 한 경우이다. 이 경우 register-updating instruction의 실행이 끝나고 아직 complete되지 않은 상태이므로 RRF에서 값을 가져다 쓰면 된다. 세 번째 가능성은 busy bit이 set되었으나 valid bit은 그렇지 않은 상태로, 아직 register-updating instruction이 실행 안 된 상태이므로 RS로 tag를 forwarding 하고, 해당 tag를 이용해 앞의 명령어가 완료되면 forwarding을 통해 data를 받는다.

Destination allocate는 decode stage서 행해지며, instruction의 destination register specifier를 통해 ARF에 접근하여 busy bit을 set 한다. 또한, unused(busy bit = 0인) RRF entry를 찾아 mapping을 수행한다.

마지막으로 register update는 complete 시 발생하며, complete 후 결과를 RRF서 ARF로 옮기는 것을 말한다.

발전된 기술로는 pooled register file이 존재한다. 이 기술은 ARF와 RRF를 물리적으로 구분하지 않고 하나의 register file을 만든 후, 그때그때 상황에 따라 ARF/RRF entry로 사용하는 것을 말한다. RRF/ARF간 data 전송이 불필요해지지만 HW 복잡도는 증가한다는 단점이 있다. (일단 이후에 추가해볼 수 있는 옵션으로 생각하고 구체적 설명은 제외하겠음)

1. Handling true data dependencies in registers

RAW dependence를 제어하기 위한 방법을 다룬다. 이를 해결하기 위한 전통적인 방식은 tomasulo algorihm이다. 이 알고리즘은 글만으로 설명하기 복잡한 부분이 있어 다음 링크로 대체한다. 참고로, FPU를 위한 알고리즘이었기 때문에 FPU를 예시로 드는 설명이 많다.

[Tomasulo's Algorithm - 임베디드컴퓨터구조 (tistory.com)](https://roomedia.tistory.com/entry/Tomasulos-Algorithm-%EC%9E%84%EB%B2%A0%EB%94%94%EB%93%9C%EC%BB%B4%ED%93%A8%ED%84%B0%EA%B5%AC%EC%A1%B0)

이를 정리해 일반화하면 RS서 tag를 통해 앞 instruction의 결과를 forwarding 받는 것이 가능하다라고 생각 가능하다. 또, instruction 간 dependency의 파악은 tag와 busy bit을 이용해 이루어진다. Tomasulo algorithm을 바로 가져다 쓰는 것이 아니라 예시를 통해 원리를 파악하고 접근하는 것이 편하다(예를 들어, 책의 예시로 쓰인 IBM 360은 memory 접근을 포함하므로 더 복잡).

Load/Store는 cache miss로 exception을 발생시킬 수 있으며, 이때 stall이 아닌 re-issue로 대응하는 것이 일반적이다. 즉, 다른 명령어와 달리 issue시에 RS entry를 비우는 것이 아니라 instruction의 실행이 종료된 경우에 RS entry를 비워야 한다.

RS는 operand와 busy bit, ready bit, 각 entry 별 valid bit으로 구성된다. Busy bit은 entry 할당 여부를, ready bit은 두 valid bit이 모두 1인지를 확인하게 된다. 또한 entry의 각 operand에 대한 tag를 저장하고 있어야 할 필요성이 있다. 동시에 준비된 entry간의 arbitration mechanism은 정책의 문제이다. 즉, 우리가 알아서 정하면 된다(FIFO?)

ROB는 모든 in-flight(=dispatch되었으나 아직 complete되지 않은)인 operation을 저장하며, 각 entry는 instruction의 status를 저장한다.

RS에 들어가기 이전에 operand를 가져오는 방식과 RS에서 나올 때 operand를 가져오는 두 가지 방법을 생각할 수 있다. 지금까지 논의하던 방식은 전자이다.

1. Handling data dependencies in memory

Memory에 대해서는 register보다 간단한 설계가 적용 가능하다. Memory에 대한 접근은 address generation(즉, 다양한 addressing mode에 기반한 표기를 상수 주소로 변환)-address translation(virtual->physical)-memory accessing으로 구성된다. 이때, memory access를 OoO 수행 시 data dependence로 인해 문제가 발생할 수 있다. 이를 handle하는 가장 쉬운 방법은 모든 L/S가 program order로 실행되도록 하는 것이다. 이를 개선하면 true dependency가 존재하지 않는 L/S에 대해 OoO 실행을 하는 방안이 나온다. 이때, 설계를 쉽게 해주는 제약 조건이 존재한다. Exception에서의 복원을 위해 memory state가 sequential하게 변해야 한다는 점, multiprocessor system서 sequential consistency memory model을 가정한다는 점이다. 즉, store 명령은 in-order로 수행되어야 한다.

사용 가능한 OoO 방법은 크게 2가지이다. Load bypassing과 load forwarding이 그것이다. Load bypassing은 preceding store들과 load가 서로 충돌하지 않는 한 뒤쪽의 load를 앞으로 보내는 것이다. Load forwarding은 load가 data memory를 거치지 않고 store로부터 data를 바로 받아오는 것이다.

이를 위한 L/S unit의 구조는 다음과 같다(책 figure 5.33: 나중에 궁금하면 사진 요청해 주세요). Store unit과 load unit은 같은 RS를 공유하지만 다른 unit으로 분리된다. Store의 경우 address gen/translate가 끝나면 finish로 간주하고 finished store buffer에 저장한다. 이 명령어가 ROB에서 빠져나오면 completion으로 간주하고 completed store buffer에 저장한다. 일반적으로, 하나의 store buffer내에서 status bit/pointer를 통해 두 부분으로 구분한다. 구분을 하는 이유는 예외 발생 시 flush/완료를 구분하기 위해서이다.

L/S RS서 명령어가 in-order로 인출되므로, 특정 load의 앞에 있고 in-flight인 모든 store 명령어는 store buffer에 존재한다. 따라서, load bypassing/forwarding시 store buffer에 대해 associative search를 진행하는 것으로 preceding store와의 aliasing 확인이 가능하다.

추가적인 최적화로 load의 OoO issue, nonblocking cache, prefetching cache, load address prediction등이 존재하는데, 이것은 시간이 남으면 해볼 수 있는 것으로 생각하고 일단은 다루지 않겠다.

AHB

\*조사했는데 자료가 별로 없어서 아마 나중에 삽질해가면서 이것저것 가져다 붙여서 찾아야 할 듯 함.

AXI 보다 훨씬 간단한 프로토콜이어서 사용하지만, 10여개 이상의 신호가 필요한 등 간단하지는 않다. AHB 프로토콜 자체는 specsheet를 쉽게 찾아볼 수 있기 때문에 vivado에서 AHB를 사용하는 방법 위주로 조사하려고 했다.

먼저 (아마도 나중에 수강해야 할 듯 한) AMBA bus(AHB와 AXI를 모두 포함하는) 설계 강의이다. [AMBA (AXI/AHB/APB) 코딩 : 네이버 블로그 (naver.com)](https://m.blog.naver.com/PostView.naver?blogId=khshimto&logNo=222312260490&categoryNo=1&proxyReferer=)

실제로 AHB가 사용될 부분은 각 component간 연결이다. 즉, zynq processor가 slave, 우리가 설계한 processor가 master가 되는 구조이다. 문제는 zynq PS의 외부 버스는 AXI를 지원하도록 되어 있는 것으로 보인다는 것이다([PS–PL Miscellaneous Signals • Zynq 7000 SoC Technical Reference Manual (UG585) • Reader • AMD Technical Information Portal](https://docs.amd.com/r/en-US/ug585-zynq-7000-SoC-TRM/PS-PL-Miscellaneous-Signals)). 여기에서 참고해볼만 한 자료는 Xilinx 커뮤니티의 다음 질문글이다. 다만 내용 자체가 너무 오래되어 현재와는 조금 다른 듯 하다. [How to connect custom IP with AHB slave bus to zynq processor? (xilinx.com)](https://support.xilinx.com/s/question/0D52E00006hpTZmSAM/how-to-connect-custom-ip-with-ahb-slave-bus-to-zynq-processor?language=en_US)

위 링크에서는 AXI 연결 후 AXI-AHB bridge를 통해 AHB를 이용하는 IP와 zynq를 연결하고 있다. 여기에서 차이점은 IP가 slave라는 점이 차이점이다.

이 부분에 대해서 잘 동작할지는 실제로 만들어서 돌려봐야 확인이 가능할 것 같은데 그러기 위해서는 먼저 AHB master로 동작하는 IP가 필요하다. 따라서, L/S unit 설계를 마치고 나서 bus에 연결시켜 동작을 확인해봐야 할 것으로 생각된다.

AHB signal의 경우 전부 옮겨적기에는 복잡해서 적지 않는다.

추가적으로, AHB master interface를 어떻게 구현할지를 고민해봐야 하는데, 위에 링크로 적은 동영상 강의를 보는 것이 가장 좋을 것 같다고 생각된다.