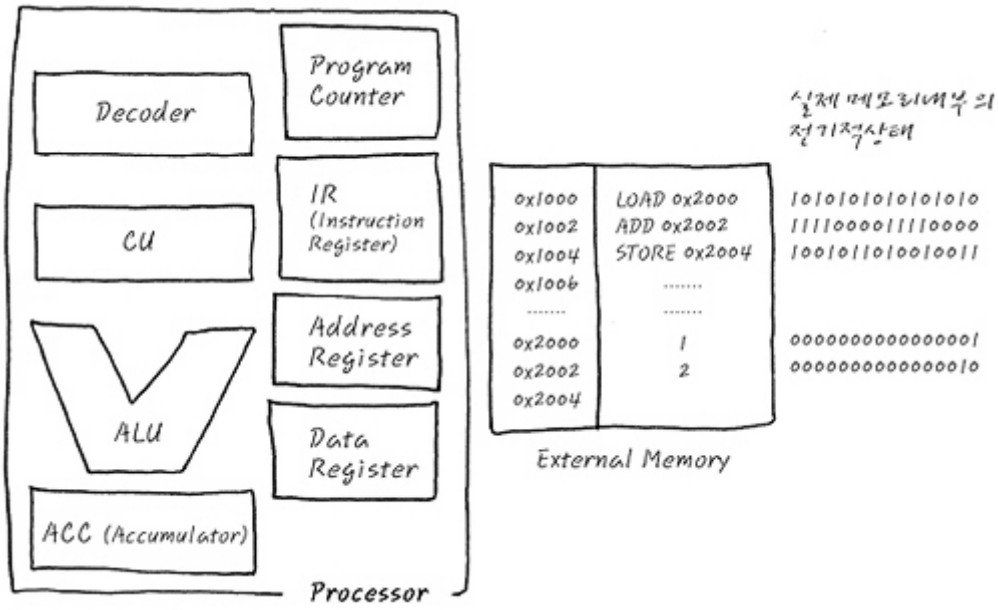
일반적인 CPU의 동작 예 (Core)와 Pipe Line

일반적인 CPU의 동작 예를 들기 위하여, 조금 더 Simplified된 CPU모델을 소개하고자 합니다. CPU는 최소한의 동작을 위하여 그림과 같이 , CU, ALU 이외에도 Program Counter, IR, Address Register, Data Register, ACC 등을 포함하고 있는 게 일반 적인 구조지요. 뭐 하나 하는데도 이렇게 많은 것들이 필요한가 싶은데, 뭐 일단은 두뇌나 다름없는 부분이니까, 이정도 복잡하다고 해서 그렇게 놀라서는 안됩니다. 태연자약하게 봐주세요. 뭐, 어떻게 보면 Decoder나 CU, ALU등은 앞의 논리회로로의 확장에서 보신, Transistor로 가득 찬 Digital 회로인데, 그것까지 다 보려면 머리가 아파질 것 같아서, 이 정도로만 하는 것입니다. - 사실 저도 머리가 아파서 이 정도로 그만 한다고 생각하니 어지러운 머리 속이 조금은 평화로워진 것 같기도 합니다. - 그림의 왼쪽부분은 그야말로 Processor (CPU) 부분이고, 오른쪽은 외부에 달려 있으며, 프로그램과 데이터를 담고 있는 External memory (RAM 또는 NOR flash따위의 ROM)를 의미합니다.



일단은 CPU내부에는 무엇이 있을까 하나하나 나열하면서 다시 한번 살펴볼 필요가 있을 것 같습니다. 예시의 Processor 내부의 기억장소인 특정한 목적의 Register들만 먼저 살펴 본다면, 다음과 같습니다. PC : Program Counter : CPU가 현재 실행하고 있는 instruction의 주소를 가리킴.IR : PC가 가리키는 Instruction의 주소에서 읽어온 insturction을 담아두는 기억장소Address Register : 현재 사용하는 Data를 access하기 위한 data의 주소를 가리키는 값을 담아두는 기억장소. Data Register : Address Register가 가리키는 주소의 실제 값ACC : 특수한 register로서, 연산에 사용되는 값들을 저장하며, 연산의 결과값을 잠시 저장하는 일이 많으며, 외부 사용자가 직접 access할 수 있는 register가 아니고, CPU 혼자 독식하는 register입니다.Processor내부에 존재하는 위의 Register들은 이름에서도 킁킁 냄새 맡을 수 있듯이, 그 고유의 용도를 가지고 있습니다. Register들 이외의 기본적인 CPU의 Component들은Decoder : IR에 가져온 instruction을 해석하여 CU에 넘김.CU : Central Unit, Decoder에서 받아온 것을 각종 제어 신호로 변환하여 제어신호를 발생 시킴.ALU : 산술 연산을 담당하는 unit등이 있습니다. 이런 녀석들이 모여서 CPU (Processor)를 구성하고 있는거죠. 이런 CPU가 어떻게 동작하는지에 관련하여, 다음의 예제를 이용해서 더 가까이 성큼 다가서 보시지요. word a =1;word b =2;word c ;word **add** (void){ int temp; temp = a; c = a + b; return;}이 예는 CPU가 어떻게 동작하는지에 대해서 쉽게 다가서기 위해서 만든 예제 입니다. 일단 a,b,c는 전역변수 이므로, 절대 주소를 갖습니다. (전역변수가 절대 주소를 가질 수 있다는 성질은 언제 어디서나 access 가능하며, 값을 굳이 다른 값으로 바꾸지 않는 한 값을 유지 가능하게 할 수 있는 근거 이지요) 어쨌거나, 위의 예의 CPU는 16bit processor 즉, 1word = 2byte라고 가정하고 그 절대 주소를 제 마음대로 할당해 본다면, a는 0x2000, b는 0x2002, c는 0x2004 로 할당해 보시죠. 이때, 이 c code를 compile하면 다음과 같은 Assembly가 생성된다고 가정해 보면 어떨까요? 무리 되지 않는 범위 내에서 본다면, (순전히 예를 든 거니까, 이렇게 똑같게 compile되지 않는다고 하시면 곤

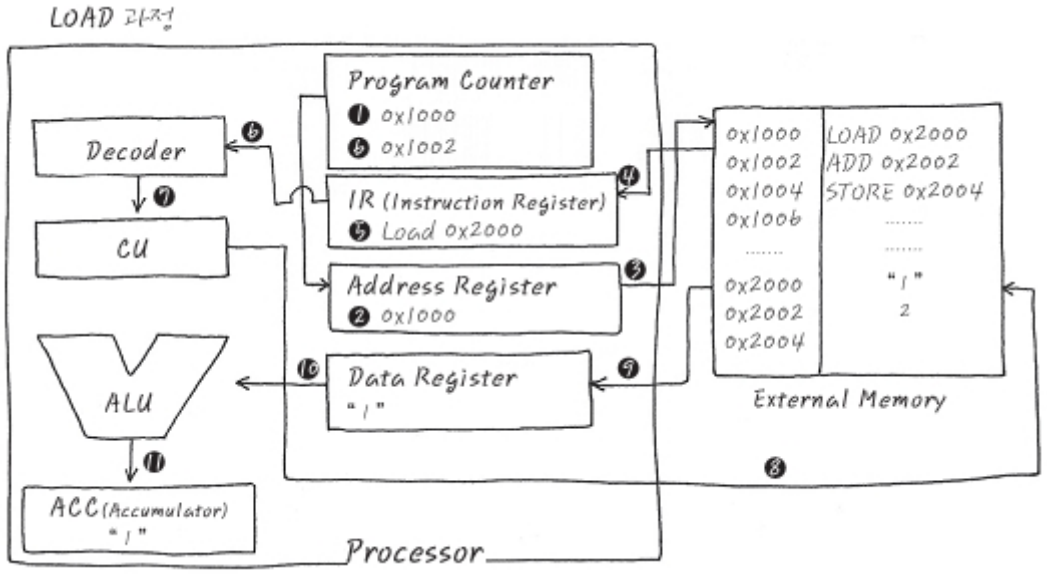
란해요)주소 Assembly
0x1000 LOAD 0x2000

0x1002 ADD 0x2002

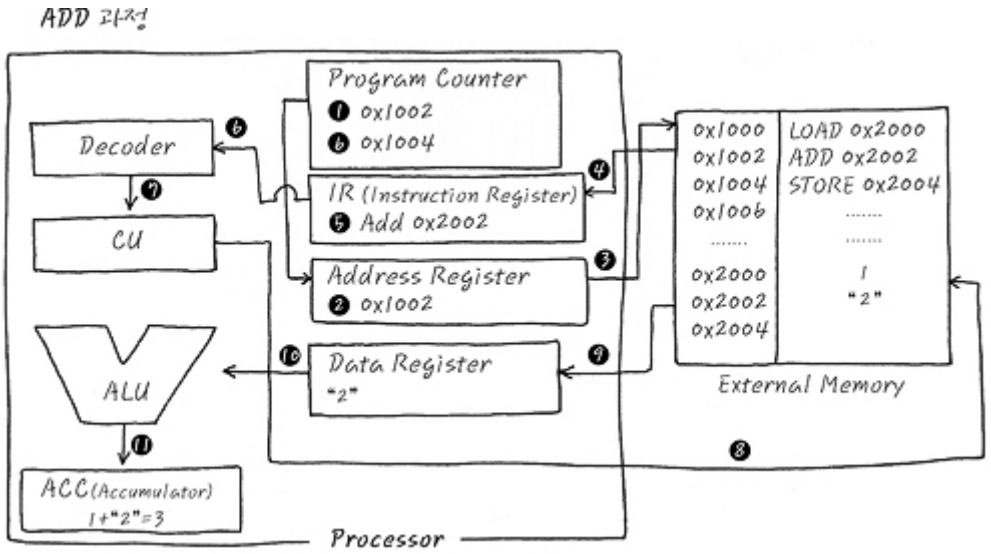
0x1004 STORE 0x2004 설명은.. LOAD 0x2000 → a값을 Data Regster에 load해서 (약속된 전기적 신호는 1010101010101010)ADD 0x2002 → 이전에 Load되어 있던 a값과 새로 load하는 b값을 더한 후, (약속된 전기적 신호는 1111000011110000)STORE 0x2004 → 그 더한 결과값을 c의 주소에 저장 (약속된 전기적 신호는 1001011010010011) 오호, a의 값을 load하여, b의 값과 더해서 c의 주소에 저장하는 Assembly라고 보면 되겠습니다요. 위의 그림에서 오른쪽 external memory에 있는 내용과 같습니다. 그림을 미리 그려놓으니 편리하네요.

어쨌든, 이 Assembly가 0x1000, 0x1002, 0x1004에 저장되어 있다고 하고,

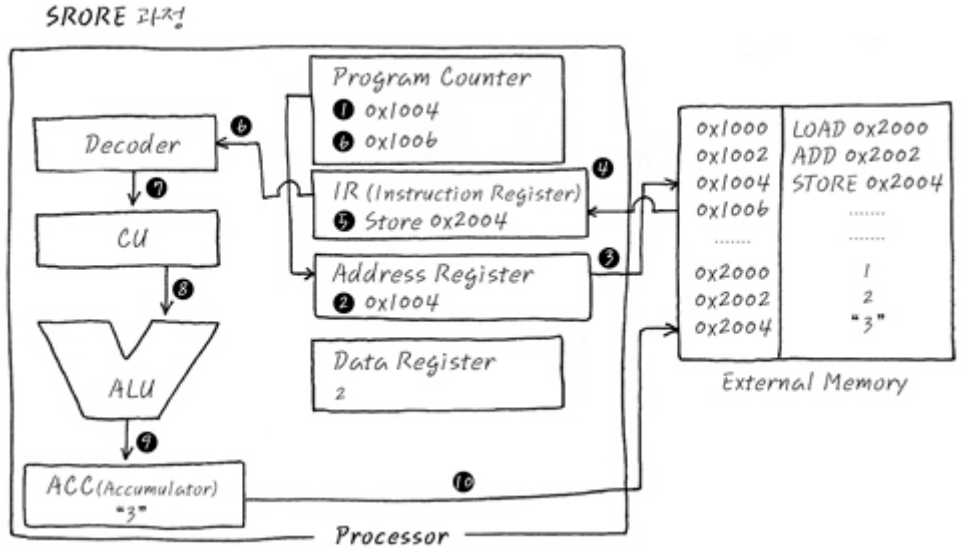
이 Assembly를 CPU가 처리하는 과정을 LOAD, ADD, STORE과정으로 나누어서 보시지요- 지금부터는 그림과 설명을 잘 match해서 보셔야 하니까 두 눈을 크게 뜨시시고고고 레츠고 -첫 번째 LOAD과정



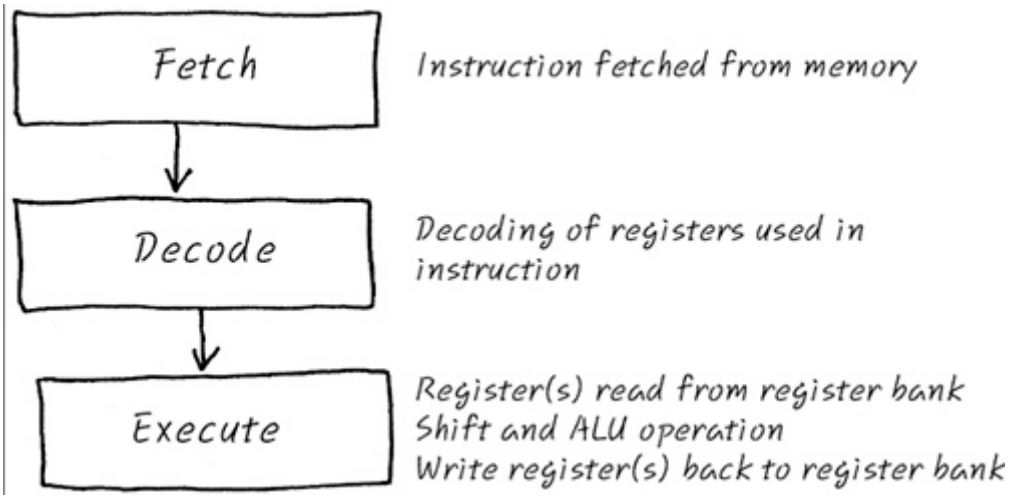
1. 현재 CPU가 실행하려는 주소는 PC에 들어 있는 0x10002. Address Register에 0x1000을 넣고서, 3. Address Register에 0x1000이 들어가는 순간 자동으로 Memory의 0x1000을 Access하여4. 그 곳에 있는 Instruction이 Memory로 부터 읽어 짐. (LOAD 0x2000) 5. Memory로 부터 읽혀진 Instruction은 6. Decoder로 흘러 들어가 무슨 내용인지 해석 되는 동시에 PC는 다음을 실행하기 위하여 증가됨. 7. 오호라, 0x2000번지의 값을 가져오라는 내용임을 파악하여, 8. Memory로 부터 0x2000의 값을 읽어 오라고 CU가 제어 신호를 발생 시킴, (ACC에게는 임시 저장토록 제어 신호발생). 9. CU가 발생시킨 제어 신호에 의하여 1이라는 값이 Data Register에 들어가고, 10. 이 값은 ALU를 통하여 연산을 할지도 모르니까 ACC에 임시 저장됨. 자자, 첫 번째 LOAD과정이 끝났습니다. 천천히 안단테(andante)로 그림과 step에 대한 설명을 mapping시켜 보세요. 간단하지요?두 번째 ADD 과정으로 진출합시다. 레츠고.



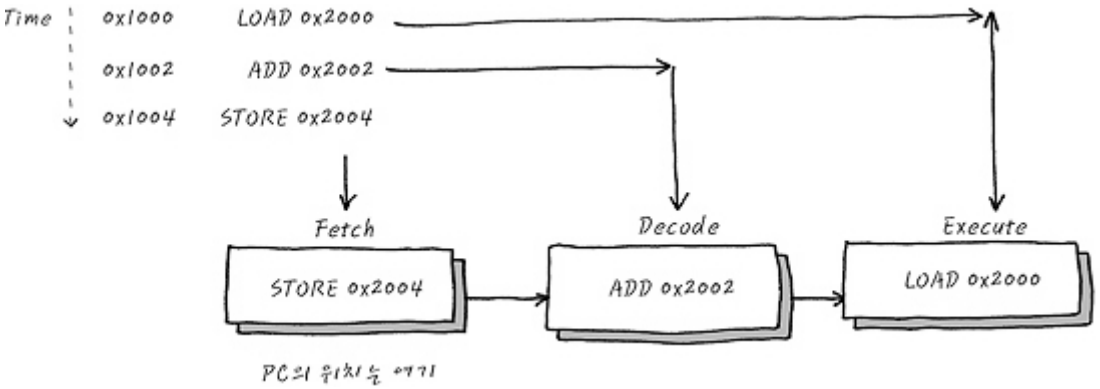
1. LOAD와 마찬가지로, 현재 CPU가 실행할 주소는 앞에서 이미 증가한 0x1002이므로2. 이 값을 Address Register에 넣음. 3. 자동으로 0x1002에 위치하고 있는 ADD 0x2002가 Load되며, 4. 이 값은 IR에 전달 된다. 5. 전달된 값은 IR에 저장되며, 6. IR의 값이 Decoder에 전달되는 동안 PC도 자동으로 다음 instruction을 가리킬 수 있도록 증가하며, 7. Decoder는 0x2002번지의 값을 더하라는 해석을 완료하여 CU에 전달한다. 8. CU는 Decoder의 해석에 의거하여 0x2002에 있는 값을 읽어 오도록 제어 신호를 발생시키며, ALU에게는 더하라는 제어 신호도 발생시킨다. 9. 0x2002에 있는 Data 2를 Load해서 Data Register에 저장한다. 10. CU가 발생시킨 제어 신호에 의하여, ALU는 Data Register에서 읽어온 data 2를 이미 있던 ACC의 값과 더하여 ACC에 결과 값을 저장한다. 두 번째 ADD과정을 살펴 보았습니다. 어떤가요. 간단하지요? 이번엔 알레그로 (Allegro)로 마지막 Store 과정을 같은 원리로 잘 따져 보시죠.



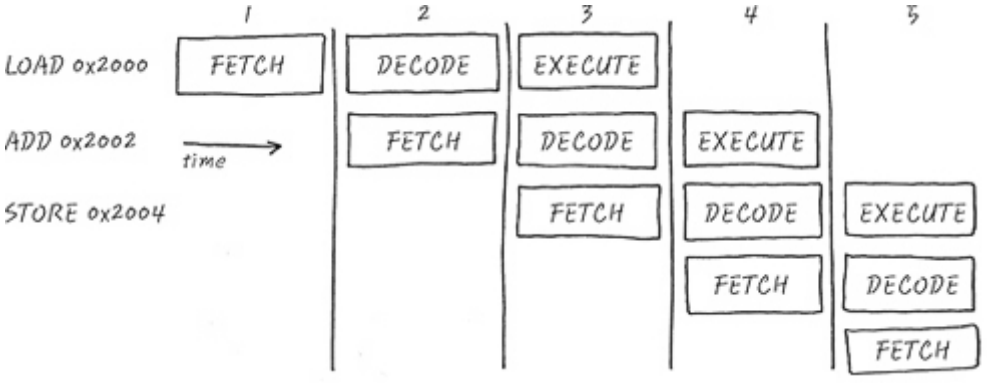
1. 현재 실행해야 할 Instruction은 PC의 값인 0x1004이며,2. 이 값은 Address Register로 전달된다. 3. 결국 instruction을 load하기 위하여 0x1004를 access하여, 4. 0x1004에 값은 CPU로 loading된다. 5. Loading된 값은 IR에 저장되며 이 값은 Decoder로 전달되는 동시에 PC는 또 한번 증가한다. 6. Decoder는 해석 하여 , 7. CU로 그 내용을 전달하여8. CU는 ALU에게 ACC에 있는 값을 0x2004에 저장할 수 있도록 제어 신호를 발생시켜 9. ACC에 있는 값은 CU가 발생시킨 제어 신호에 의하여 0x2004번지에 결과 값 3을 저장함. 이제까지 읽어보니, CPU라는 건 항상 정해진 규칙에 의해서 일을 하죠.항상 똑같은 일을 하고 있습니다. 그 step으로는1. instruction을 메모리로부터 가져오고 (Fetch)2. 가져온 Instruction을 해석해서 어떤 일을 하는 녀석이냐를 알아보고 Register 값들도 확인, (Decode)3. Decoding된 Instruction을 실행한다. (Execution)위의 CPU구조를 Simple하게 나누어 본다면 Fetch하는 Unit, Decode하는 Unit, Execute하는 unit으로 나눌 수 있겠지요.위의 CPU구조에서 일을 하는 순서는..



일단 무엇을 하든 이런 식으로 Fetch/ Decode/ Execution을 순서대로 한다라는 의미에서라면, 다시 말해, 이렇게 Sequential하게 3가지 일을 한다면, 한가지 일을 열라 할 때 나머지 2개가 놀고 있겠네요. 조금 더 CPU가 효율적으로 움직이게 할 수 있지 않을까 하는 idea에서부터 pipe line이라는 게 생긴 거죠. (알뜰한 인간들, 나머지 2개가 놀고 있는 게 눈꼴 신 게죠) 이 idea는 Decode하는 동안에는 그 다음 번 instruction을 Fetch해 오면 되지 않을까? 하는 idea로 부터 출발하여, 한번 동작할 때 서로 CPU의 다른 부분을 사용하는 Fetch/ Decode/ Execution을 한꺼번에 실행해 보자 까지 생각의 흐름이 흘러 버린 거죠. 한 순간의 시간을 멈추어 그림으로 그려보면!



뭐 이런 식인 거에요. 세 개의 Instruction의 각 stage (fetch, decode, execution)를 한방에 처리할 수 있는 게지요. 현재 이 순간 STORE 0x2004를 메모리에서 가져오고 있고요, ADD 0x2002가 무신 명령어인지 Decode하고 있고요, 실제로 실행은 LOAD 0x2000을 하고 있는 거죠. 으랏차.이제 시간이 막 흘러가는 상황을 따져보져. 유명한 그림 하나를 따서 보여 드릴께요. ARM7 기준으로 Fetch/ Decode/ Execute 3stage pipe line이에요.왼쪽으로 갈수록 시간이 지나는 거고, 한 칸당 1 cycle, 즉 1 clock이라고 보시면 됩니다.



자, 처음 cycle에서부터 시간이 흘러 3 cycle까지 갔을 때는 첫 번째 opcode의 execute가 실행되고 있고, 두 번째 opcode의 decode, 그리고 세 번째 opcode의 fetch를 한꺼번에 시행하고 있습니다. 오오, 그냥 눈으로만 봐도 performance가 좋아지겠죠. 예를 들어, Fetch/ Decode/ Execute를 pipe line의 개념을 사용하지 않고 실행한다면 2개 opcode실행하는데 6개 cycle이 걸릴 테지만, Pipe line을 이용하면 2개 opcode 실행하는데 4개

cycle만 있으면 됩니다. 더 많은 연속적인 opcode를 실행한다면 더욱 많은 양을 처리할 수 있겠지요. 그림만 봐서도 5개 cycle에 3개의 opcode를 처리할 수 있는 것입니다. 그러면, Pipe line 단계가 많아지면 좋아질 것인가? 하는 의문이 들지 않는 겝니까?. 네, 실은 Pipe line이 많아지면 많을 수록 좋아질 것입니다만 너무 많은 stage는 효율성 면에서 성능이 더 좋아지지 않고, 나빠진다고 봐야 합니다. 이런 것은 시스템이 어떻게 생겼느냐와 어떤 전략을 사용했느냐에 따라 달라지고, 마구잡이로 늘린다고 좋아지는 건 아니오니, 그 점 양해해 주시기 바랍니다. ㅋ그 예로 ARM에서의 pipe line은 ARM7의 경우 3단계 pipe line, Fetch → Decode → Execution으로 3단계를 거치며, ARM9의 경우에는 5단계 pipe line, Fetch → Decode → Execute → Memory → Write 의 pipe line을 가집니다. 이 부분은 ARM의 pipe line에 대해 다시 자세히 언급하려고 계획 중이니 여유롭게 가십시다요. 상당히 중요한 사실 한가지가 숨어 있는데, 눈치를 챘나요? 이런 pipe line을 채용하게 되면 현재 PC값 (Program Counter)는 어디를 가리키고 있는 것 입니까? 하고 질문할 수 있겠습니다. PC는 어디에 있을까요? PC는 항상 Fetch하고 있는 곳을 가리키고 있습니다. 고로, 현재 Execution하고 있는 곳보다 앞서가고 있겠습니다. ARM의 경우라면, 32bit가 1 word니까 PC는 항상 +8만큼 앞서가고 있겠죠. 이 내용은 Exception Debugging을 하려면 아주 요긴하게 쓰이는 Concept 이니까 꼭 기억해 두세요. 자자, 어떻습니까? CPU라는 것도 하나의 rule을 가지고 모든 것을 처리하고 있습니다. 실은 이런 과정은 아주 간단하게 설명한 것이고 이 것을 기본으로 하여 훨씬 더 복잡한 과정을 거칩니다만, 이 정도만으로도 충분히 CPU의 동작을 이해 할 수 있으리라 생각하고 있습니다만. 이정도 읽는 것으로도 눈이 아파오고 뻑뻑할 것 같습니다. - 실은 제가 그러네요 -

CPU 동작예의 내용은 제가 감명 깊게 읽은 책에서 아이디어를 따와서 재 구성하였습니다. Reference는 "초보 프로그래머가 꼭 알아야 할 컴퓨터 동작원리", 김종훈, 한빛미디어 p77 입니다. 감사합니다.

Pipeline을 사용하게 되면, Control Hazard와 Data Hazard라는 용어에 부딪치게 되는데, 이건 무엇이냐! 하면, 이런 Pipe line이 가장 좋을 때는 연속적인 메모리 영역에 대해서 쫘아악~ 아무 생각 없이 긁어와서 실행하는 환경이 최상으로 빠르고 좋겠습니다. 만 실상은 그렇지 만은 않죠. 그러다 보니, Pipe line하면서 막상 branch - 다른 곳으로 jump - 하게 되면, execute하는 단계에서 이미 fetch해 놓은 녀석을 버려야 하는 일이 벌어지는데 이런걸 Control Hazard라고 하고요, Execute하는 단계에서 어떠한 Register값을 조작했는데 동시에 다음 opcode의 Decode에서 조작하고 있는 Register 값을 가져오려고 하면 뭔가가 틀어지죠. 이런걸 Data Hazard라고 부릅니다. Hazard라고 하니까 무시무시하죠. 그래서 가끔 이런 문제 때문에 pipe line이 정상적으로 쭈으으으욱 긁어서 하지 못하는 경우도 생기지만, 그래도 안 쓰는 것 보다는 나으니까 열심히 pipe line을 사용한답니다.