2016

# 모던프로세서 – 90 분가이드

원본: http://www.lighterra.com/papers/modernmicroprocessors/?repost

원작: Jason Robert Carey Patterson, last updated May 2015 (orig Feb 2001)

번역: sh

이 글을 읽는 당신은 전산학 전공자로서 학교 다닐때 하드웨어 과목 하나쯤은 들어 봤을겁니다. 물론 예전일이고, 졸업 후에는 프로세서 디자인에 대해선 심각하게 신경쓰지 않았겠죠. 특히나 요 근래 개발된 아래의 기술들에대해선 전혀 모르고 있을수도 있어요.

- pipelining (superscalar, OOO, VLIW, branch prediction, predication)
- multi-core and simultaneous multithreading (SMT, hyper-threading)
- SIMD vector instructions (MMX/SSE/AVX, AltiVec)
- caches and the memory hierarchy

뭐 그렇더라고 쫄필요는 없읍니다. 이 글이 순식간에 여러분을 캐쉬 구조라던가, 하이퍼쓰레딩, 인오더 vs 아웃오더 익스큐션 같은 디테일한것 들을 전문가들처럼 토론 할 수있게 해줄 꺼니까요.

그전에 미리 말해두고 싶은게, 이 글은 잡설이나 디테일한 설명없이 중요한 부분만 빠르고 간략하게 찝어줄겁니다. 쪼까 빠를수도 있어요. 그럼, 시작해 봅시다.

# 목차

**클럭 스피드가 전부는 아니다** ........ 2

**파이프라인과 인스트럭션레벨 병렬화** ........ 3

깊어지는 파이프라인 – 수퍼파이프라닝 ........ 5

다중 처리(Multiple Issue) – 수퍼스칼라 ........ 6

명시적 병렬화(Explicit Parallelism) – VLIW ........ 9

인스트럭션 의존성 과 지연시간(Instruction Dependency & latencies) ........ 10

분기와 분기예측(Branches & Branch Prediction) ........ 12

서술로 분기 없애기(Eliminating Branches with Predication) ........ 14

인스트럭션 스케줄링, 레지스터 리네이밍 & OOO (Instruction Scheduling, Register Renaming & OOO) ........ 15

브레니악 논쟁 (The Braniac Debate) ........ 17

전력 절벽과 인스트럭션 레벨 병렬화 절벽(The Power Wall & The ILP Wall) ........ 19

X86 에 대해서 (What About x86?) ........ 21

쓰레드-SMT, 하이퍼쓰레딩, 그리고 멀티코어(Threads-SMT, Hyper-Threading & Multi-Core) ........ 24

더 많은 코어 혹은 더 넓은 코어(More Cores or Wider Cores)? ....... 28
데이터 병렬화(Data Parallelism – SIMD Vector Instructions) ....... 32
메모리와 메모리 절벽(Memory & The Memory Wall) ....... 35
캐시와 메모리 계층(Caches & The Memory Hierarchy) ....... 36
캐시 충돌과 연계성(Cache Conflicts & Associativity) ....... 40
메모리 대역폭 vs 지연시간(Memory Bandwidth vs Latency) ....... 43
Acknowledgments ....... 44
More Information? ....... 45

# 클럭 스피드가 전부는 아니다

제일먼저 짚고 넘어가야할것이 클럭 스피드와 프로세서 성능 사이의 괴리 입니다. 클럭 스피드가 높다고 성능이 좋은게 아니란말이죠. 아래 90 년대 후반의 자료를 보겠읍니다.

| | | SPECint95 | SPECfp95 |
|---|---|---|---|
| 195 MHz | MIPS R10000 | 11.0 | 17.0 |
| 400 MHz | Alpha 21164 | 12.3 | 17.2 |
| 300 MHz | UltraSPARC | 12.1 | 15.5 |
| 300 MHz | Pentium II | 11.6 | 8.8 |
| 300 MHz | PowerPC G3 | 14.8 | 11.4 |
| 135 MHz | POWER2 | 6.2 | 17.6 |

**Table 1 – Processor performance circa 1997.**

200 MHz 짜리 MIPS R10000, 300 MHz 짜리 UltraSPARC 그리고 400 MHz 짜리 Alpha 21164 칩 모두, 대부분의 프로그램들을 돌리는데 거의 비슷한 성능을 보이죠. 클럭스피드차이가 많이 나는데도 말입니다.

300 MHz 짜리 Pentium II 도 정수 작업들에선 비슷합니다, 부동소수 연산에서는 거의 절반가까이 느린데도 불구하고 말이죠. 300 MHz 짜리 PowerPC G3 은 정수 연산작업은 다른것들보다 빠른데도 부동소수 연산은 상위 세개칩들보다 상당히 느립니다. 하물며, 135 MHz 짜리 IBM POWER2 칩은 부동소수 연산에선 400 MHz 짜리 Alpha 21164 칩이랑 비슷한 수준이죠, 정수연산은 절반정도 느린데도 말입니다.

어떻게 된 걸까요? 클럭스피드 말고도 다른 무언가가 이런 차이를 만드는거죠 당연히. 그 다른 무언가는 바로 한 클럭 사이클당 얼마나 많은 작업을 처리할수 있느냐에 달려 있읍니다.

# 파이프라인과 인스트럭션레벨 병렬화

인스트럭션들이 실행될때 순서대로 하나씩 실행되면 정말 좋겠지만, 실상은 그렇지 않습니다. 1980 년대 중반 이후론 그런적이없죠. 대신, 여러 인스트럭션들이 부분적으로 동시에 실행됩니다.

인스트럭션 하나가 어떻게 실행되는지 살펴보죠. 먼저 적재(fetch)되고 디코딩(decode)된후 적절한 펑셔널 유닛에 의해 실행됩니다 (Execute). 그리고 마지막으로, 실행된 결과가 저장되죠(Writeback). 이런식으로, 한 인스트럭션당 4 클럭 사이클이 소요됩니다 (CPI = 4).

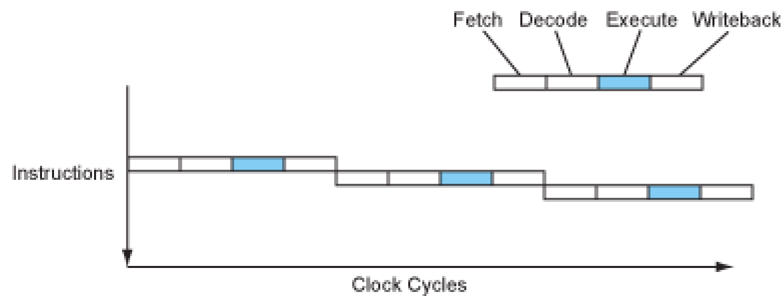


Figure 1 – The instruction flow of a sequential processor.

최근의 프로세서들은 이 4 단계(pipeline stage)의 과정을 파이프라인 안에서 동시에 처리합니다. 조립라인에 컨베이어 밸트처럼 말이죠. 한 인스트럭션이 실행되는 동안 다음 인스트럭션이 디코딩되고, 그 다음 인스트럭션이 적재 되는 식 입니다.

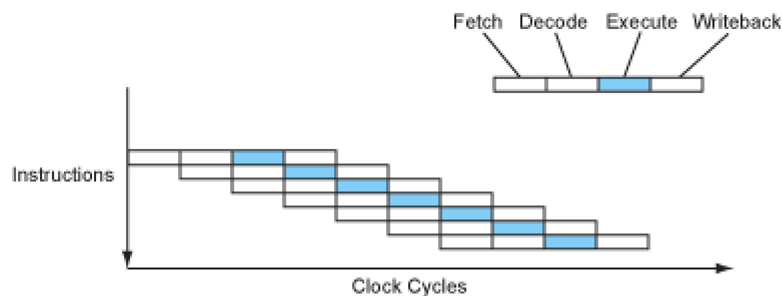


Figure 2 – The instruction flow of a pipelined processor.

이렇게하면 1 클럭 사이클로 1 인스트럭션을 처리할수있게되므로 (CPI = 1), 클럭스피드 변화없이도 4 배나 빨라지는거죠. 개이득!

하드웨어적으로 보면, 파이프라인 에서 수행되는 각 단계들은 몇개의 조합 로직 (combinatorial logic)과 레지스터 셋 또는 고속의 캐쉬메모리 엑세스로 구성됩니다. 각 단계(stage)들은 래치(latch)를 이용해 구분지어져 있습니다. 한개의 클럭 시그널이 래치들을 동기화 함으로써, 모든 래치들은 각 단계에서 처리된 결과를 동시에 받게 됩니다. 결과적으로, 클럭 시그널이 인스트럭션들을 파이프라인 안으로 펌프질하는 효과가 나는겁니다.

각 클럭 사이클의 초반에, 부분적으로 처리된 인스트력션을 위한 데이터와 컨트롤 정보가 래치안에 위치하게되고, 이 정보들이 다음 처리단계 내부의 로직 서킷들에 입력값이 됩니다.

클럭사이클 중반에는, 클럭 시그널이 그 단계의 조합로직 (combinatorial logic) 들로 퍼져가면서, 다음 단계의 래치로 넘어갈 결과물이 생성됩니다. 이 결과물들은 싸이클 후반부로 가면서 다음 래치의 입력이 되는 거죠.

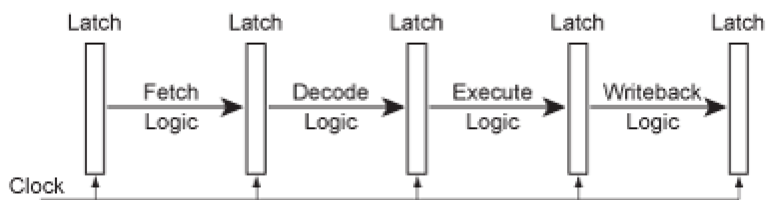


Figure 3 – A pipelined microarchitecture.

각 인스트럭션의 결과는 실행(Execute)단계가 끝난 후, 파이프라인 바이패스(Bypass)를 통해 다음 인스트럭션이 사용할수 있게 됩니다. 이전 인스트럭션의 결과 값이 레지스터에 저장(Writeback)되길 기다릴 필요가 없는것이죠.

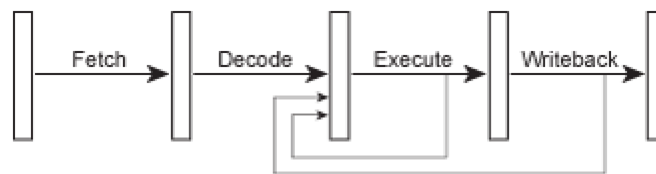


Figure 4 – A pipelined microarchitecture with bypasses.

그림에서는 파이프라인 단계들이 간단해보지만, 실행(Execute)단계는 여러가지 로직(게이트 셋) 그룹들을 사용해서, 프로세서가 처리해야할 다양한 오퍼레이션에 사용될 펑셔널 유닛 (functional unit)들로 구성되어 있읍니다.

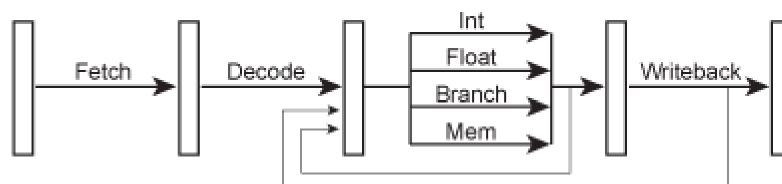

Figure 5 – A pipelined microarchitecture in more detail.

IBM 의 801 research prototype 이나 스탠포드 MIPS 를 기반으로한 MIPS R2000, 그리고 버클리 RISC 프로젝트를 통해 개발된 오리지널 SPARC 과 같은 초창기 RISC 프로세서들은 위의 그림들과 비슷한 5 단계의 파이프라인을 가지고 있었읍니다.

그 당시에, 80386, 68030 그리고 VAX 같은 CISC 프로세서들은아직 파이프라인을 사용하지 않고 있었지요. 그 이유는 RISC 칩들이 사용하는 인스트럭션 세트가 보다 간결해서 파이프라인을 적용하기가 쉬웠기 때문입니다.

그 결과, 20MHz 의 파이프라인이 적용된 SPARC 칩이 33MHz 의 직렬처리 방식의 386 보다 훨씬 빠른 성능을 보여주었죠. 그 이후의 칩들은 정도의 차이는 있지만 모두 파이프라인을 적용하게 되었읍니다.

오리지널 RISC 연구 프로젝트에 관한 잘 정리된 David Patterson 씨의 자료를 1985 CACM article 에서 찾아 볼수 있읍니다.

# 깊어지는 파이프라인 – 수퍼파이프라닝

클럭 스피드는 파이프라인에서 가장 길고 느린 단계의 길이에 제한받기 때문에, 각 단계를 구성하는 로직 게이트들은 더 작은 단위로 나눠질수 있었읍니다. 특히나 긴 것 들은말이죠. 이렇게 그 파이프라인은 많은 수의 짧은 하위 단계들로 구성된 보다 깊은 수퍼 파이프라인을 탄생 시키게 됩니다. 이런 방법으로 프로세서는 전반적으로 보다 높은 클럭 스피드를 낼 수있게 되었읍니다!

물론, 개별 인스트럭션을 완료하는데는 더 많은 싸이클을 필요로 하게 되었지만(latency), 프로세서는 여전히 1 인스트럭션을 1 싸이클에 처리할수 있게 되었읍니다(throughput). 그리고, 클럭 스피드가 올라가게 됨으로써, 초당 처리할수있는 인스트럭션의 갯수가 늘어나게 되었읍니다(실제 성능이 클럭수에 비례하게됨).

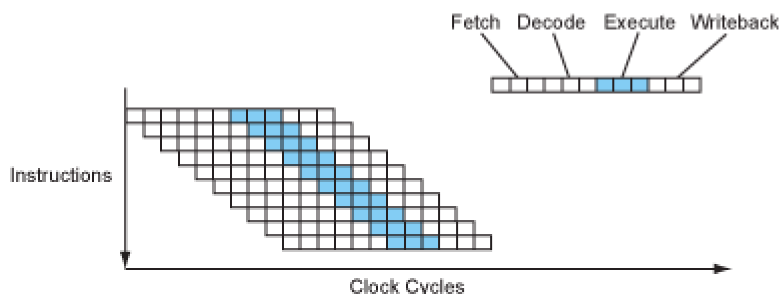


Figure 6 – The instruction flow of a superpipelined processor.

특히나 Alpha 프로세서 아키텍트들은 이 방식을 좋아했읍니다. Alpha 칩들이 당시에 다른 칩들에 비해 파이프라인이 깊고 고속의 클럭 스피드로 동작 하던 이유였지요.

요즘의 모던 프로세서들은 제법 깊은 파이프라인을 가지고있으며, 각 단계에서 발생하는 게이트 딜레이의 수를 적게 유지하려고 노력하고 있읍니다. 12~25 개 게이트 정도에 플러스 3~5 개 래치 정도로요.

| *Pipeline Depth* | *Processors* |
|---|---|
| 6 | UltraSPARC T1 |
| 7 | PowerPC G4e |
| 8 | UltraSPARC T2/T3, Cortex-A9 |
| 10 | Athlon, Scorpion |
| 11 | Krait |
| 12 | Pentium Pro/II/III, Athlon 64/Phenom, Apple A6 |
| 13 | Denver |
| 14 | UltraSPARC III/IV, Core 2, Apple A7/A8 |
| 14/19 | Core i*2/i*3 Sandy/Ivy Bridge, Core i*4/i*5 Haswell/Broadwell |
| 15 | Cortex-A15/A57 |
| 16 | PowerPC G5, Core i*1 Nehalem |
| 18 | Bulldozer/Piledriver, Steamroller |
| 20 | Pentium 4 |
| 31 | Pentium 4E Prescott |

**Table 2 – Pipeline depths of common processors.**

x86 프로세서들이 일반적으로 동급의 RISC 칩들보다 깊은 파이프라인을 사용하는데, 그 이유는 복잡한 x86 인스트럭션들을 디코딩하는 작업이 더해지기 때문입니다 (나중에 자세히).

UltraSPARC T1/T2/T3 Niagara 프로세서들이 최근들어 깊은 파이프라인을 사용하는 트렌드에서 벗어났는데요(6 – T1, 8–T2/3). 코어들을 최대한 작게 만들려고 하기 때문입니다(역시, 나중에 자세히).

# 다중 처리(Multiple Issue) – 수퍼스칼라

역주: Issue 의미가 여기서는 실행될 명령을 "시작" 시켜준다는 의미로 쓰였는데, 적당한 우리말 단어를 생각해 낼 수 가 없었읍니다.

파이프라인에서의 실행(Execute) 단계는 실제로 각자의 일을 독립적으로 실행하는 여러 펑셔널 유닛(Functional unit) 의 집합일 뿐이기 때문에, 많은 인스트럭션을 동시에 실행 시키기 딱 좋은 환경입니다.

그러기 위해선 먼저, 앞의 단계들(fetch/decode/dispatch)이 여러 인스트럭션들을 동시에 처리해서 실행 유닛쪽으로 넘겨주도록 발전해야합니다.

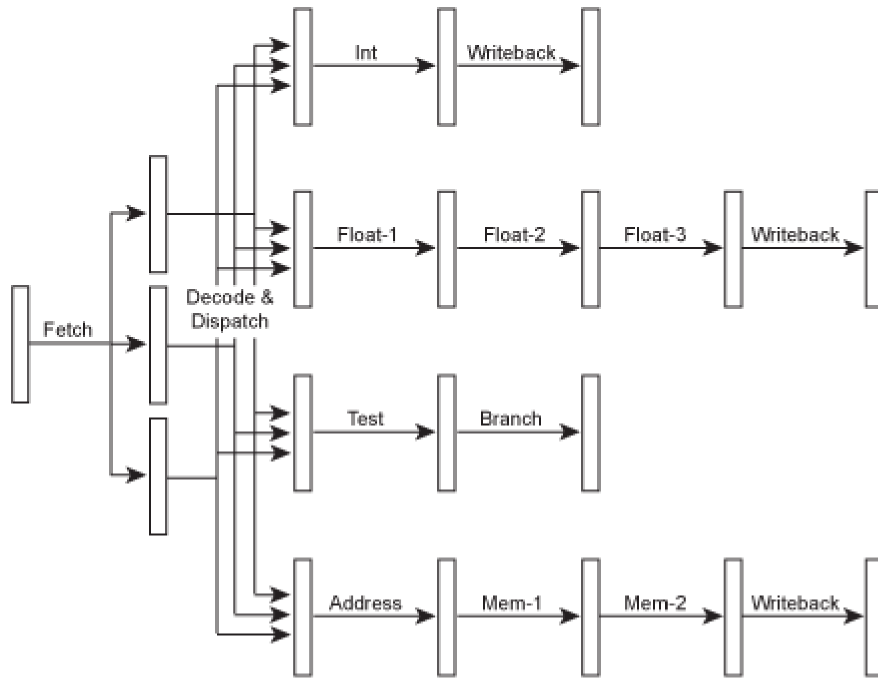


Figure 7 – A superscalar microarchitecture.

위 그림을 보면, 각 펑셔널 유닛들을 위한 독립된 파이프라인들이 생겼읍니다, 그 파이프라인은 또 여러가지 단계로 나뉘어져 있지요. 이렇게 하면 간단한 인스트럭션들은 보다빠르게 완료 될 수있어, latency 를 줄여주게 됩니다. (나중에 자세히)

이런 방식을 사용하는 프로세서들은 다양한 깊이의 파이프라인를 가지고 있기때문에, 정수 계산용 인스트럭션 실행을 기준으로 파이프라인의 깊이를 이야기를 합니다. 이런 인스트럭션들의 파이프라인 길이가 가장 짧기때문이죠. 메모리 엑세스나 부동소수 연산등은 몇 단계가 더 필요해서 그만큼 파이프라인이 깊어집니다.

그래서 보통, 10 단계 파이프라인 을 가진 프로세서 라고하면, 정수 연산에 10 단계 정도, 메모리 관련작업에 12 ~13 정도, 그리고 14~15 단계정도를 부동소수연산에 사용한다고 하면 맞을겁니다. 다양한 파이프라인들 사이나/안쪽에 위치한 수많은 바이패스들은 다이어그램의 간결성을 위해 생략되었읍니다.

위 예의 프로세서는 1 싸이클당 3 가지 다른 종류의 인스트럭션(1 정수, 1 부동소수, 그리고 1 메모리 관련)을 실행할수 있읍니다. 심지어, 타겟 애플리케이션에 따라서, 펑셔널 유닛을 더 추가 해 한 싸이클당 처리 할 수있는 작업을 알맞게 조절할수 있읍니다.

슈퍼스칼라 프로세서상에서 인스트럭션 플로우는 다음과 같이 표현할수 있읍니다.

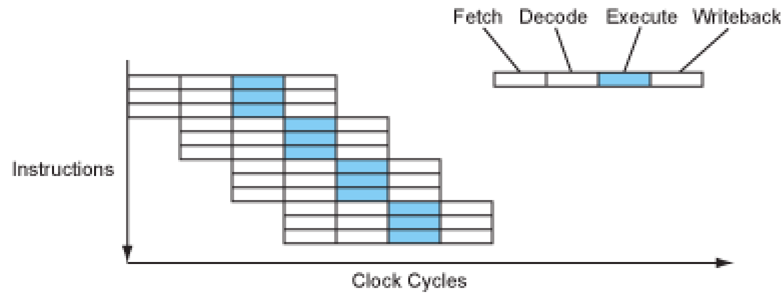


Figure 8 – The instruction flow of a superscalar processor.

이게좀 짱인게, 각 싸이클당 인스트럭션을 3 개씩이나 처리 할 수있게 되 버린것이죠 (CPI = 0.33, or IPC = 3, also written as ILP = 3 for *instruction-level parallelism*). 한 싸이클당 처리(issue), 실행 또는 완료되는 인스트럭션의 갯수를 프로세서의 넓이(processor's *width)* 라고 부릅니다.

처리넓이(issue width)가 보통 펑셔널 유닛 개수 보다 작다는거에 주목해야 합니다. 이유는 실행되는 일련의 코드들에 서로 다른 인스트럭션들이 배합되있기 때문이죠. 싸이클당 3 개의 인스트럭션을 수행하는게 목적인데, 그 인스트럭션들이 항상 똑같지는 않다는 겁니다. 예시에서 처럼 1 정수, 1 부동소수, 그리고 1 메모리인 경우도 있겠지만 항상 그렇지는 않기때문에, 3 개 이상의 펑션널 유닛이 필요한겁니다.

PowerPC 칩 이전의 IBM POWER1 프로세서가 최초의 메인스트림 수퍼스칼라 프로세서 였읍니다. 그후로 대부분의 RICS 칩들(SuperSPARC, Alpha 21064)이 수퍼스칼라 방식으로 바뀌었죠. 인텔도 오리지날 팬티엄 칩을 수퍼스칼라 방식으로 만들었지만, 복잡한 x86 인스트럭션 세트가 발목을 잡았읍니다 (나중에 자세히).

당연히, 깊은 파이프라인과 다중 인스트럭션 처리를 한 칩에서 동시에 할수 도 있게 되었죠.

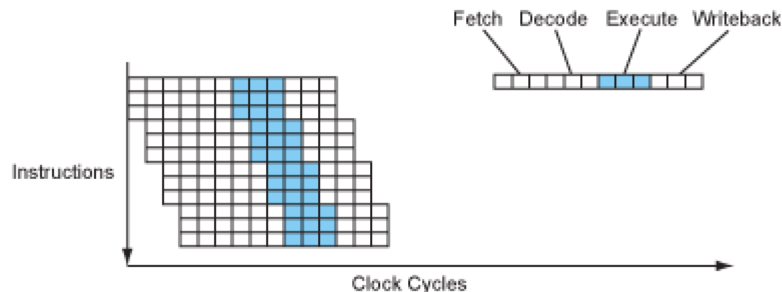


Figure 9 – The instruction flow of a superpipelined-superscalar processor.

요세 거의 모든 프로세서들이 슈퍼 파이프라인에 슈퍼스칼라 방식들입니다. 줄여서 그냥, 수퍼스칼라 프로세서 라고 통칭합니다. 엄밀하게 따지만 수퍼 파이프라이닝 이란게 그냥 파이프라인을 좀더 확장한거니까요.

| Issue Width | Processors |
|---|---|
| 1 | UltraSPARC T1 |
| 2 | UltraSPARC T2/T3, Scorpion, Cortex-A9 |
| 3 | Pentium Pro/II/III/M, Pentium 4, Krait, Apple A6, Cortex-A15/A57 |
| 4 | UltraSPARC III/IV, PowerPC G4e |
| 4/8 | Bulldozer/Piledriver |
| 5 | PowerPC G5 |
| 6 | Athlon, Athlon 64/Phenom, Core 2, Core i*1 Nehalem, Core i*2/i*3 Sandy/Ivy Bridge, Apple A7/A8 |
| 7 | Denver |
| 8 | Core i*4/i*5 Haswell/Broadwell, Steamroller |

**Table 3 – Issue widths of common processors.**

각 프로세서에 포함된 펑서널 유닛의 정확한 종류와 갯수는 타겟 마켓에 따라 다릅니다. IBM 의 POWER 계열은 부동소수 유닛을 강화했고, Pentium Pro/II/III/M 계열은 정수 유닛들에 좀더 중점을 두고 있으며, PowerPC G4e 계열은 대부분의 유닛을 SIMD 벡터 인스트럭션에 할애하고 있죠. 그 외의 프로세서들은 대부분 적절히 균형을 고려한 유닛 배합을 채택하고 있습니다.

# 명시적 병렬화(Explicit Parallelism) – VLIW

역주: 병렬화 (Parallelism) 가 여기선 여러 명령들이 동시에 실행되는 것을 이야기 합니다. 역시 적당한 단어가 떠오르지 않았습니다.

하위(backward)호환성이 중요치않을 때에는, 인스트럭션 세트를 병렬로 수행 될수있는 인스트럭션별로 명시적으로 그룹화해서 디자인하기도합니다.이런 방식은 디스패치 단계에서 복잡한 의존성 체크를 할 필요가 없게 만들죠. 따라서 프로세서를 디자인하기 쉬워지고, 크기를 줄여주며, 클럭 스피드를 쉽게 올릴수 있게 해주는 이점이 있읍니다, 이론적으로는 말이죠.

이런 종류의 프로세서들에서 사용되는 인스트럭션들은 보통 여러개의 작은 서브 인스트럭션들의 집합체 입니다. 따라서, 개별 인스트럭션의 길이가 길어지게 됩니다. 128 비트 이상되는것들도 있지요. 이러한 특징때문에 VLIW – *very long instruction word* 라고 불리게 되었읍니다. 각 인스트럭션은 여러개의 병렬 오퍼레이션을 처리하기위한 정보들을 포함하게 됩니다.

VLIW 프로세서의 인스트럭션 처리과정은 수퍼스칼라 방식과 비슷합니다만, 디코드/디스패치 단계가 훨씬 간단하며, 서브 인스트럭션의 그룹별로 적용됩니다.

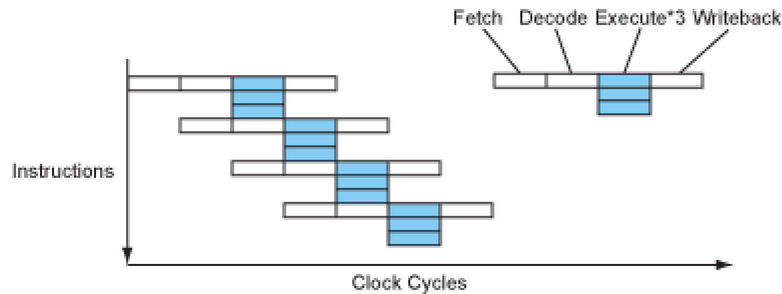


Figure 10 – The instruction flow of a VLIW processor.

디스패치 로직이 간단하다는점을 제외하면, VLIW 프로세서는 수퍼스칼라 프로세서들과 상당히 유사합니다. 특히나, 컴파일러의 관점에서는 더 그렇습니다(나중에 계속) . 이런 유사성에도 불구하고, 대부분의 VLIW 디자인이 인스트럭션간 상관관계를 고려하지 않는다는 점(not interlocked)에 점에 주목할 필요가 있습니다. 무슨 말 이냐면, VLIW 프로세서들은 수행되는 일련의 인스트럭션 사이의 의존성 검사를 하지않고, 캐시 미스발생시 관련된 인스트럭션 들만 정체 시킬수가 없어서 프로세서 전체를 정체해야 한다는 말입니다.

결과적으로, 컴파일러가 서로 의존적인 인스트럭션 사이에 적당한 싸이클을 채워줘야 할 필요가 생기는것이죠. 뭔가 다른 인스트럭션이 딱히 없다면, nops (no-operations, pronounced "no ops")으로라도 적당히 벌려줘야 합니다. 이런 특성이 VLIW 용 컴파일러들을 조금 더 복잡하게 만듭니다. 수퍼스칼라 프로세서들이 런타임에 수행하는 것들을 컴파일러가 처리 해줘야 하니까요. 그렇지만, 컴파일러에 가는 부담은, 프로서세 칩상의 리소스를 줄여주기 때문에 결과적으로는 이득입니다.

VLIW 디자인이 적용된 프로세서중 아직 상업적으로 성공한 CPU 는 없읍니다만, Itanium 프로세서로 시판되고 있는 인텔의 IA-64 아키택쳐가 이 디자인으로 x86 을 대체하려고 했었읍니다. 인텔은 IA-64 를 “EPIC(Explicitly Parallel Instruction Computing)” 디자인이라고 칭했었읍니다. 이 아키택처는 간단히 얘기해서 장기적으로 호환성을 유지할수 있도록 인스트럭션들을 그룹화 하고 분기예측(Prediction - 아래나옴) 기능을 추가한 VLIW 디자인이 었읍니다.

그외에도 VLIW 디자인은 GPU 에 내장된 프로그래머블 쉐이더나 DSP 칩들 에 채택했으며, Transmeta (곧 다룰 x86 섹션참조) 에도 적용되었읍니다.

# 인스트럭션 의존성 과 지연시간(Instruction Dependency & latencies)

역주: latency 라는 단어가 뭔가 지연되거나 정체되어 시간이 걸린다는 의미가 있읍니다만, 딱히 지연/정체등의 단어를 쓰기에는 어감이 이상 해 지는 부분이 있었읍니다. 따라서, 앞으로 이 두가지 단어를 섞어가며 쓰게 됩니다.

파이프라인과 다중처리는 과연 어디까지 갈수있을까요?

5 단계 파이프라인이 5 배 빠르다면, 20 단계 수퍼파이프라인을 만들면 어떨까요? 만약 다중으로 4 개씩 인스트럭션 처리가 가능한(4-issue) 수퍼스칼라가 좋은거라면, 8 처리짜리는 더 좋지 않을가요? 이런식으로, 한 싸이클에 20 개의 인스트럭션 다중처리를 가능하게 한 50 단계짜리 파이프라인을 사용한 프로세서를 만드는건 어떨까요?

아래의 두 인스트럭션을 살펴보도록 하죠…

```
a = b * c;
d = a + 1;
```

두번째 줄의 인스트럭션은 첫번째 것에 의존적입니다. 따라서, 프로세서는 처음 인스트럭션이 완료된후 그 결과 값을 받기전까지는 두번째 줄을 처리 할수가 없읍니다. 굉장히 심각한 문제죠, 왜냐면 서로 의존관계에 있는 인스트럭션들은 병렬로 실행하기가 불가능 하기 때문입니다. 따라서 다중처리(multiple issue) 가 불가능 합니다.

만약 첫번째 인스트럭션이 간단한 정수 더하기 였다면, 파이프라인을 사용한 개별 처리(single-issue) 프로세서상에서는 별 문제가 아니었을수도 있읍니다. 정수 덧셈은 빨라서, 두번째 인스트럭션이 바이패스를 통해 첫번재 인스트럭션의 결과값을 제시간에 받아 쓸수 있을테니까요.

그러나, 곱셈의 경우 완료하는데 여러 싸이클이 필요하기때문에 두번째 인스트럭션이 첫번째 인스트럭션보다 한 싸이클 늦게 실행단계(Execution stage)에 도착했을때, 결과값을 사용할 방법이 없읍니다. 그래서, 프로세서는파이프라인에 버블(bubble)을 주입해 두번째 인스트럭션의 실행을 정체 시킬수 밖에 없읍니다. 곱셈의 결과 값이 나올때 까지요.

한 인스트럭션이 실행단계(Execute stage)에 도착 했을때부터 그 실행 결과가 다른 인스트럭션들에 의해 사용될수있을 때까지 사용된 클럭 사이클의 수를 그 인스트럭션의 지연시간(latency) 라고 부릅니다.

파이프라인이 깊어질수록 거쳐야할 단계(Stage)가 늘어나며, 그 결과 지연시간(latency)도 길어 지게됩니다. 따라서, 매우 깊은 파이프라인이 짧은것보다 월등히 성능이 좋은것 만은 아닙니다. 인스트럭션간에 의존성때문에 발생되는 버블로 파이프라인이 꽉차버리기도 하니까요.

컴파일러 입장에서 보면, 모던 프로세서내에서 자주 발생하는 지연시간(latencies)들의 범위는 작게는 정수연산에 1 싸이클부터 부동소수 연산에 3~6 싸이클정도 그리고 곱셈연산에 비슷하거나 약간 더 걸리는 수준이며, 정수 나누기 연산에 12 싸이클 이상이 되기도 합니다.

메모리를 엑세스할때 발생하는 지연시간이 특히 문제가 됩니다. 일차적으로는 코드 도입부에서 발생 할 때인데, 이 경우에는 딜레이되는 부분을 쓸만한 다른 인스트럭션으로 채우기가 어렵습니다. 또한, 이런류의 작업이 얼마나 걸릴지 예상하기 약간 어려운것도 문제입니다. 캐시 힛/미쓰냐에 따라 지연시간이 많이 좌우 되기 때문이죠.(캐시 부분은 나중에 다룹니다)

*저자 - latency 라는 단어가 서로 연관성은 있지만 다른 의미로 사용되는게 혼란 스러울수도 있읍니다. 여기서는, 컴파일러의 입장에서의 지연시간(latency)을 뜻 합니다. 하드웨어 엔지니어들은 지연시간(latency)을 실행단계에 필요한 싸이클의 수로 생각해서 정수 파이프 라인은 1 throughput 에 5 latency 다 라고 말할수도 있읍니다. 하지만, 컴파일러 입장에서는 1 latency 입니다. 왜냐면 그 실행결과가 바로 뒷 따르는 싸이클의 인스트럭션에서 사용가능 하기 때문입니다. 컴파일러 에서 사용되는 지연시간(latency)의 의미가 보다 넓게 통용되고 있으며 하드웨어 메뉴얼에서도 그렇게 사용 합니다.*

# 분기와 분기예측(Branches & Branch Prediction)

파이프라이닝의 또한가지 문제점이 바로 분기문(branches) 입니다.  아래 일련의 코드를 예로 보겠읍니다.

```
if (a > 7) {
    b = c;
} else {
    b = d;
}
```

위의 코드는 아래의 코드로 컴파일됩니다.

```
    cmp a, 7    ; a > 7 ?
    ble L1
    mov c, b    ; b = c
    br L2
L1: mov d, b    ; b = d
L2: ...
```

이제, 프로세서가 파이프라인을 사용해서 이 코드들을 실행한다고 가정해보죠.

2 번째줄의 조건분기가 파이프라인의 실행단계(Execute state)에 도착 할 쯤에는, 다음 싸이클에 실행될 몇 개의 인스트럭션들이 적재(fetched)된후 디코드(decode) 되어 있어야 합니다. 하지만, 어떤 인스트럭션을 선택해야 할까요? if 문이 참일때 실행될 3, 4 번줄에 있는것들? 아니면 거짓일때 수행될 5 번 줄의 내용들? 조건분기문이 실행단계(Execute state)를 지날때까지는 알수있는 방법이 없죠. 깊은 파이프라인을 사용하는 프로세서 상에서는 아마 여러 사이클이 지난 다음에나 알수 있을겁니다. 그렇다고 마냥 기다릴수는 없는 노릇이죠. 일반적으로 이런 분기문들을 평균 6 개의 인스트럭션에 한번씩 만나게 되는데, 이때마다 몇 싸이클씩 기다리게 되면 파이프라인을 사용해 얻게되는 성능 향상을 모두 낭비 하는셈이 되는 것이죠.

따라서, 프로세서들은 다음단계를 추측해서 움직일수 밖에 없읍니다. 이 쪽이다 싶은 부분의 인스트럭션을 처리하기 시작하는거죠. 물론, 분기문의 결과가 나올때 까지, 추측을 통해처리한 결과들을 실제로 저장 (writeback) 하진 않읍니다. 운이 안좋을때는 추측이 틀려서 미리 처리했던 인스트럭션들을 모두 취소해야 하는 경우가 생기죠. 거기에 사용된 싸이클들은 낭비된것이 되지만, 운이 좋아서 맞으면 풀스피드로 계속 다음 인스트럭션들을 처리해 나갈수 있게 되는겁니다. 프로세서가 어떻게 바른방향으로 추측을 해야 하는가가 핵심 질문인데요. 두가지 방식이 떠오릅니다.

첫째로, 컴파일러가 프로세서에게 힌트를 주는겁니다. 이 방법을 정적 분기예측(*static branch prediction*) 이라고 부릅니다. 인스트럭션 포멧안에 분기예측을 위한 비트가 마련되어 있다면 좋겠지만, 예전 아키텍쳐들은 이런 옵션이 없었읍니다. 따라서, 컨벤션이 사용 되는데요. 예를들어, 백워드 분기문은 예측을 하고, 포워드 분기들은 않하는 방식이죠. 하지만 보다 중요한건, 이 방식은 매우 똑똑한 컴파일러가 필요 하다는 점이죠. 정확한 추측을 해야가기 때문 이지요. 루프안쪽의 분기문들은 쉽게 맞출수있지만, 다른 부분들은 어려울 수 있겠읍니다.

다른 방법으로는, 프로세서가 런타임에 추측을 하게 하는겁니다. 보통, 칩안에 내장된 분기예측 테이블(*branch prediction table*) 을 사용 합니다. 이 테이블 안에는 최근에 처리한 분기의 주소와 분기를 했는지 안했는지를 알려주는 비트값이 저장됩니다.

대부분의 프로세서들이 실제로는 2 비트를 사용해서, 한번 예측값을 어긋난 분기결과가 자주 맞는 얘측값을 뒤집지 못하게 합니다 (루프돌때 중요함). 물론, 이러한 동적 분기 예측 테이블은 프로세서 칩에서 귀중한 자원인 공간을 차지하지만, 분기 예측은 그 정도는 감수 할만 큼 중요합니다.

불행하게도, 최고의 분기예측 기법들도 틀릴때가 있고, 깊은 파이프라인을 사용한다면 상당한 양의 인스트럭션들을 취소 해야 할수도 있읍니다. 이렇게 낭비된 싸이클을 *mispredict penalty* 라고 부릅니다. Pentium Pro/II/III 가 좋은 예인데요. 12 단계 파이프라인을 채용했었고, 10-15 싸이클의 mispredict penalty 가 존재 했읍니다. 90% 정도의 적중률을 보이는 동적 분기 예측기로도, 이 정도의 패널티라면 약 30% 정도의

프로세서 성능을 허비하게 되는 것 입니다. 다시말해서, Pentium Pro/II/III 는 3 분의 1 의 프로세싱 시간을 쓸때없는 짓에 낭비 한 꼴이 되는것입니다.

모던 프로세서들은 보다 많은 하드웨어들을 분기예측에 할애하고 있읍니다. 예상 적중율을 더욱 끌어올리고 패널티 비용을 줄이기 위해서지요. 어떤 예측기들은 분기방향을 독립된 상태 보다는 그쪽으로 향하는 여러 분기문의 컨텍스트를 고려해서 저장합니다. 이들을 *two-level adaptive* predictor 라고 부릅니다. 또 다른 것들은 각 분기문에 대한 개별적인 히스토리 보다는 좀더 광범위한 분기 히스토리를 보관합니다. 코드 내에서 상당히 떨어져 있더라도, 분기문들 간에 연관성을 발견해 내기 위한 방편으로 말이지요. 이런 예측기를 *gshare* 또는 *gselect* predictor 라고 부릅니다.

대부분의 최첨단 모던 프로세서들은 여러가지 분기 예측기를 구비해놓고, 각 분기에 최적의 성능을 보이는 것을 선택해서 사용합니다. 어찌됐든, 정말 최고로 킹왕짱 좋은 분기 예측기를 사용하고있는 특 S 급 모던 프로세서들도 95% 정도의 예측 정확도를 보이고 있으며, 아직도 상당한 양의 성능 저하를 피할수 없읍니다.

한줄요약 – 아주 깊은 파이프라인은 자연적으로 수확체감(*diminishing returns)* 을 겪고 있으며, 그 이유는 파이프라인이 깊어질수록 예측해야 할 경우의 수가 늘어나고, 따라서 틀릴 확률도 높아지며, 그에 따른 mispredict penalty 가 증가하기 때문이다.

## 서술로 분기 없애기(Eliminating Branches with Predication)

역주: Predication 은 여기서 문제 해결의 실마리나, 힌트를 제공해 준다는 의미로 사용되었읍니다만, 역시 적당한 단어가 떠오르지 않았읍니다.

조건분기들은 골칫거리 들이라, 싹 없애버리면 정말 좋겠읍니다. 근데, if 문을 프로그래밍 언어들에서 없애기는 불가능 하니까, 분기들을 없애 버릴 다른 방법이 있을까요? 분기문이 사용되는몇몇 방식들에 해답이 있읍니다.

위에 예시로 들었던걸 다시봅시다. 5 인스트럭션중 2 개가 분기였고, 그중 한개는 조건없는 분기(Unconditional branch)였어요. 어떤 식으로든 mov 인스트럭션에게 어떨때만 실행되라고 힌트를 줄수있으면, 코드가 더 간단해 질수 있죠.

```
cmp a, 7        ; a > 7 ?
mov c, b        ; b = c
cmovle d, b     ; if le, then b = d
```

짜잔, 새로운 인스트럭션 *cmovle* (conditional move if less than or equal) 를 소개합니다. 이건 분기가 아닌듯이 평범하게 실행됩니다, 하지만 주어진 조건이 맞을때만 결과를 커밋하게 됩니다. 서술(*predicated*) 인스트럭션이라고 하는대요. True/false 테스트 결과에 따라 실행 여부가 좌우 됩니다.

이 새로운 인스트럭션을 이용해, 두개의 비싼 분기를 코드상에서 없애버렸읍니다. 게다가, 언제나 mov 를 실행하고 필요에따라 결과를 덮어쓰게 됨으로써, 코드의 병렬화가 더 수월 해 졌지요. 1 번과 2 번 줄이 병렬로 실행될수 있게 됐으니까요. 이로써, 50%의 속도 향상(3 싸이클 -> 2 싸이클)을 기대 할 수 있읍니다. 다 좋은데, 진짜 중요한건, 비싼 mispredict penalty 를 피할수 있게되었다는 점입니다.

물론, if 블럭이나 else 블럭이 길어지면, 이런식으로 하는게 분기 보다 많은 인스트럭션을 실행하게 되는 결과가 되겠죠. 프로세서가 양쪽 블럭을 모두 실행해서, 결과에 따라 한쪽을 버려야 할태니까요. 분기를 피하기 위해서 좀더 많은 인스트럭션을 수행하는게 합당한건지 결정하는게 사실 좀 까다롭습니다. 아주 작거나 아주 큰 블럭들은 그나마 결정하기 쉬운데, 어중간 한것들이 문제죠. 옵티마이져가 고려해야 할것들이 많아지거든요,

Alpha 아키택쳐는 원래부터 이런 조건부이동(conditional move) 인스트럭션을 갖추고 있었읍니다. MIPS, SPARC 그리고 x86 등에는 나중에 추가되었죠. 인텔은 IA-64 의 거의 모든 인스트럭션을 서술형으로 설계했읍니다. Inner loop 문 안쪽에 사용된 분기를 획기적으로 줄일 수 있을거라는 희망으로요. 특히, 컴파일러나 OS 커널등에 사용된 것들을 주 타겟 으로 생각했죠.

흥미롭게도, 핸드폰이나 타블렛에 사용되는 ARM 이 인스트럭션 전체를 서술형으로 구성한 첫번째 아키택쳐 였읍니다. 초창기 ARM 프로세서들이 짧은 파이프라인들을 사용해서 상대적으로 mispredict penalty 가 작았다는 점을 생각해보면 더 인상적이죠.

## 인스트럭션 스케줄링, 레지스터 리네이밍 & OOO (Instruction Scheduling, Register Renaming & OOO)

파이프라인내에서 분기나 긴 정체(long-latency) 때문에 발생한 버블들로 채워진 빈 싸이클들은 다른 일을 하는데 사용 될수도 있을겁니다. 그러기 위해선, 프로그램내의 인스트럭션들의 순서를 바꿔서, 한 인스트럭션이 기다리는 동안 다른 인스트럭션이 실행될수 있게 해야 합니다. 예를들면, 위에서 다뤘던 곱셈 예제에서, 두 인스트럭션 사이에서 대기하는 동안 코드내에 어딘가에 위치한 다른 작업들을 수행하는거지요.

대략 두가지 방식으로 이걸 가능하게 할 수있는데요. 첫번재는 하드웨서가 런타임에 인스트럭션들의 수행순서를 바꾸는 겁니다. 프로세서가 동적으로 *인스트럭션 스케줄링*(reordering) 을 하려면 디스패치(dispatch) 로직이 인스트럭션들을 그룹별로

처리 할 수있게 향상되야 합니다. 순서에 상관없이 프로세서의 펑셔널 유닛들을 최대한으로 사용할수 있게 말이죠. 이 방식을 OOO(*out-of-order execution*) 또는 OOE/OoO 라고 부릅니다.프로세서가 인스트럭션들을 순서에 관계없이 실행(OOO) 하려면, 인스트럭션들 사이의 의존성을 염두해 두여야 하는데요.  아키텍처상에 지정된 레지스터들을 쌩으로 사용하는 대신, 일련의 레지스터들을 조합하고 이름을 바꿔서(*renamed*) 사용하면 쉽게 해결할수 있읍니다.

예를들어, 한 레지스터값을 메모리로 저장하고 다른 메모리에서 값을 그 레지스터로 읽어오려면,  레지스터는 하나인데 담고있어야할 값은 두개가 되는거죠. 이 두개의 다른 값을 꼭 하나의 물리적 레지스터에 담을 필요가 없죠.  좀 더 나아가서, 서로다른 인스트럭션들이 각기 다른 물리적 레지스터를 사용하도록  매핑되면, 이것들은 병렬로 처리가 가능해지는 거죠. 이게 바로 OOO 의 핵심 포인트 입니다.

따라서, 프로세서는 항상 인스트럭션들에 사용된 물리적 레지스터들의 맵핑을 인지하고 있어야 합니다. 이런 과정을 레지스터 리네이밍(*register renaming)* 이라고 부릅니다. 코드 병렬화를 더 끌어올리고자하면, 쌩 레지스터들의 조합을 보다 많이 사용하면 되는거죠. 이러한 인스트럭션간의 의존성 분석, 레지스터 리네이밍, 그리고 OOO 모두 복잡한 로직들을 필요로 하기 때문에 프로세서를 더욱 디자인하기 힘들게하고, 칩의 크기를 키우며, 전력소모를 심하게 만듭니다. 이 로직들은 특히나 전력소모가 심한데요. 사용된 트렌지스터들이 항상 동작해야 하기때문이죠. 놀고 있을때는 전력을 최소로 사용하는 펑셔널 유닛들과는 다르게 말입니다(아예 꺼버릴수도 있음). 반면, OOO 는 최신 프로세서 디자인의 이점을 소프트웨를 재컴파일 하지 않고도 약간 이나마 얻을 수 있게 해주는 이점이 있읍니다.

이 문제를 접근하는 또다른 방법은, 컴파일러에게 인스트럭션들의 순서를 바꿔서 코드를 최적화 시키게 하는 겁니다.  정적 (*static)* 또는 컴파일 타임(compile-time) 인스트럭션 스케줄링 이라고 부르는 기법이죠. 이렇게 컴파일러에 의해 재배치된 인스트럭션들은 그 상태로 보다 간단한 순차적(*in-order*) 다중처리(multiple-issue) 로직에 의해 처리될수 있지요. 최적의 인스트럭션 실행순서를 컴파일러에게 전적으로 맡기는 겁니다. 복잡한 OOO 하드웨어 로직이 필요 없어지므로, 프로세서를 디자인하기 쉬워지고, 전력소비도 줄고, 빈 공간을 또 다른 코어든 엑스트라 캐시든 다른 유용한걸로 채울 수도있지요, 칩 사이즈는 그대로 유지하면서 말이죠. (나중에 다시 다룸).

이 방식의 또 다른 이점이있는데요. 하드웨어 보다 프로그램의 코드를 더 넓게 보고 판단할수 있게 됩니다. 예측하기 어려은 분기문의 다양한 패스들을 보다 다양하게 점검할수 있읍니다. 그렇지만, 컴파일러가 100% 용한 점쟁이가 될수는 없죠. 항상 완벽하게 모든 경우를 파악 할 수는 없는 겁니다.  OOO 하드웨어 없이는, 컴파일러가 캐시 미스같은걸 미처 예상하지 못했을때, 파이프라인이 정체 되는걸 피할수 없읍니다.

대부분의 초창기 수퍼스칼라 프로세서들(SuperSPARC, hyperSPARC, UltraSPARC, Alpha 21064 & 21164, the original Pentium)은 순차적 (in-order) 디자인을 채택

했었읍니다. 초기 OOO 디자인의 예로는 MIPS R10000, Alpha 21264 등이 있으며, 전체 POWER/PowerPC 계열들 도 넓은 범위에서 포함됩니다 (with their reservation stations). 요즘에는, 거의 모든 고성능 프로세서들이 비순차적 (out-of-order) 다자인을 채용하고 있읍니다. UltraSPARC III/IV, POWER6 그리고 Denver 는 예외구요. Cortex-A7/A53 그리고 Atom 같은 대부분의 저전력, 저성능 의 프로세서들은 아직도 순차적(in-order) 디자인을 채용하고 있는데, OOO 로직이 성능대비 전력소모가 심하기 때문입니다.

## 브레니악 논쟁 (The Braniac Debate)

비싼 out-of-order 로직이 정말로 필요한것인지 아니면 컴파일러들이 인스트럭션 스케줄링을 하드웨어 없이 충분히 잘 처리할수 있는지는 반드시 짚고 넘어가야 할 부분입니다.

역사적으로, 이 부분에 대한 논쟁은 계속되어 왔으며 브레니악 대 스피드 데몬(*brainiac* vs *speed-demon)* 이라는 이름으로 불려지고 있읍니다.  이런 식으로 디자인 스타일을 구분짓는 간단한(그리고 잼있는) 방식은 Linley Gwennap 의 1993 Microprocessor Report editorial 에 처음 등장하였으며,  Dileep Bhandarkar 의 저서 Alpha Implementations & Architecture 에 소개되며 널리 알려지기 시작 하였읍니다.

*브레니악(Brainiac)* 디자인들은 OOO 하드웨어를 사용해서 인스트럭션 레벨에서의 병렬화를 극강으로 끌어 올리는 부류입니다. 백만개 이상의 로직 트렌지스터 가 필요하고, 설계 하는데도 몇년씩 걸리는 데도 말이죠.

반면, *스피드 데몬(speed-demon)* 디자인들은 작고 간단한 하드웨어로 똑똑한 컴파일러에게 의지하는 방식입니다.  하드웨어의 간결함이 주는 이점들을 위해 인스트럭션 레벨 병렬화를 약간 포기하는 거죠.

예전에는, 스피드 데몬 디자인들이 좀더 빠른 클럭 스피드를 낼수있었는데요. 매우 간단한 하드웨어 구조였기 때문이지요. 빠른 클럭 스피드덕에 “스피드 데몬" 이라고 불린거죠, 하지만 오늘날은 꼭 그렇지많은 않읍니다. 클럭 스피드가 전력소모와 발열 문제에만 발이 묶인게 아니니까요.

OOO 하드웨어가 인스트럭션 레벨 병렬화를 좀더 끌어올리는 건 분명합니다. 캐시 미스같은 런타임에 발생할 수있는 비싼 작업들을 미리 예측 하기가 어렵기 때문이죠. 하지만, 간단한 순차적(in-order) 디자인은 더 작고 전력효율이 좋기 때문에 같은 크기의 칩안에 비순처적 (out-of-order) 코어들보다 더 많은 코어들을 심을수 있게 해줍니다. 여러분이라면 어떤걸 선택 할껀가요? 4 개의 파워풀한 브레니악 코어요? 아니면 8 개의 간단한 순차적 코어요??

정확히 어떤게 더욱 중요한 요소있지는 현재로써 매우 뜨거운 논쟁거리 입니다.

일반적으로, OOO 하드웨어의 이점과 비용모두 약간 부풀려져 있었던 적이 있읍니다. 비용측면에서 보면, 디스패지과정에서 적절한 파이프라인의 활용과 레지스터 리네이밍 로직을 통해 OOO 프로세서들은 1990 년대 들어 스피드 데몬들에 견줄만한 클럭 스피드를 확보하게 됐었으며, 요 근래들어서는 보다 발전된 엔지니어링을 통해서 전력수효를 주목 할 만큼 줄이게 됐지요. 남은건 칩의 크기정도 밖에 없읍니다. 프로세서 아키텍트들이 엔지니어링 문제를 약간이나마 해결해 준거지요.

불행하게도, OOO 하드웨어들의 인스트럭션 레벨 병렬화가 기대만큼 효율적이지 않다는것이 문제가 되고 있읍니다. 순차적 디자인에 비해 20~40% 정도로 상대적으로 기대에 미치지 못하는 성능을 내는거죠. OOO execution 의 선구자로 Pentium Pro/II/III 의 선임 아키택트중 한명인 Andy Glew 씨의 말을 빌리면: “OOO 의 감추고 싶은 비밀은 자주 OOO 를 쓸일이 없다는 겁니다.” 또한, Out-of-order execution 은 원래 기대했던 “재컴파일이 필요없는 성능향상(recompile independence)” 을 제대로 지원하지 못했읍니다. 어그레시브한 OOO 프로세서 상에서도, 재컴파일이 상당한 속도향상을 가져왔거든요.

브레니악 논쟁에서 보면, 많은 프로세서 벤더들이 두 디자인사이를 왔다리 갔다리 했었읍니다.

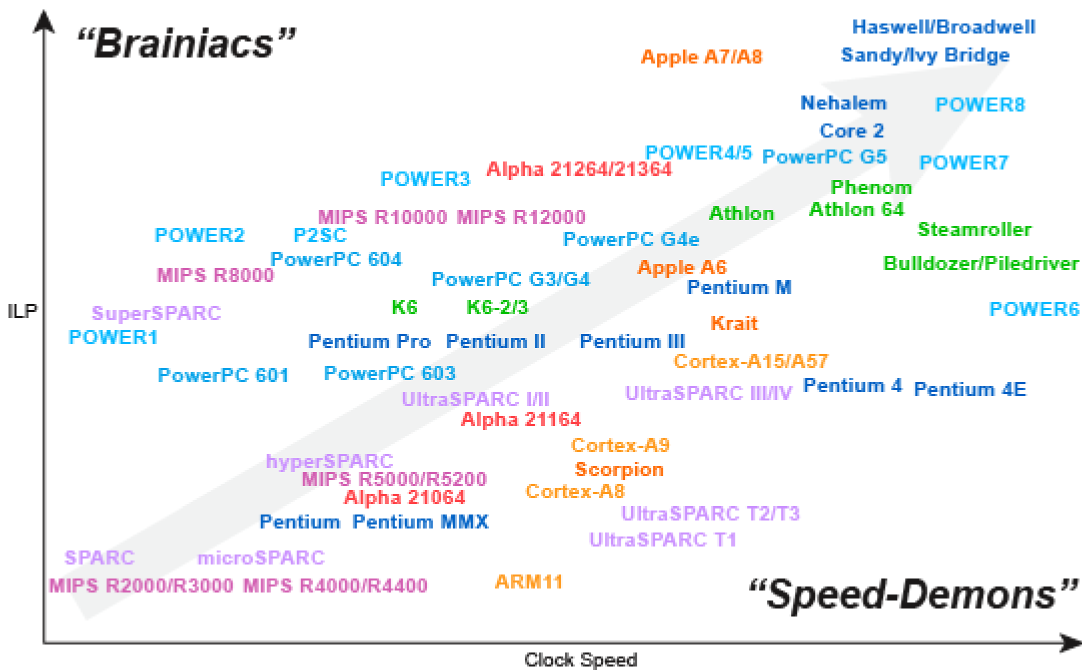


Figure 11 – Brainiacs vs speed-demons.

일례로, DEC 은 Alpha 칩을 초반 두세대는 스피드데몬 쪽으로 설계했다가, 삼세대 칩은 브리니악 디자인으로 바꿨었죠. MIPS 도 비슷했읍니다. 반면에, Sun 은 첫번째 superscalar SPARC 칩을 브레니악 방식으로 디자인했다가, 나중 모델들은 스피드 데몬 쪽으로 설계를 변경 했구요. POWER/PowerPC 진영 역시 최근까지 서서히

브레니악 디자인에서 멀어지는 양상을 보여왔읍니다. POWER/PowerPC 에서 사용하는 reservation stations 들이 서로 다른 펑셔널 유닛들 간 OOO execution 을 어느 정도 지원 하고 있기는 하지만 말이죠(각 펑셔널 유닛 내에서는 엄격하게 순차적으로 실행됨).

반면, ARM 프로세서들은 저전력 저성능의 임베디드 분야에서 시작해서, 브레니악 디자인 쪽으로 꾸준하게 이동 해 왔지만, 아직 모바일 분야를 주무대로 하고 있기 때문에 클럭 스피드를 크게 향상시키는 어려운 면이 있읍니다.

모든 칩메이커들 중에 인텔이 가장 흥미로운데요. 모던 x86 프로세서들은 아키택서상의 제약(나중에 자세히)으로 인해 적게라도 브레니악 디자인을 택할수 밖에 없었읍니다. Pentium Pro 는 그 제약을 전폭적으로 감내했지요. 그때마침, 1 GHz 를 향한 AMD 와의 경주가 본격화되고, AMD 가 2000 년 3 월에 간발의차로 이기게 됩니다.

그걸 계기로, 인텔은 클럭 스피드에 목숨을 걸게되고, 깊은 20 단계짜리 파이프라인을 적용하고, 어느정도의 ILP(Instruction Level Parallelism)를 희생한 decoupled x86 아키택처 디자인으로 Pentium 4 를 스피드 데몬들 만큼 빠르게 만들게됩니다. 2GHz 를 넘어서 3GHz 가지 도달하게 되었고, 급기야는 31 단계짜리 파이프라인으로 3.8GHz 의 속도를 내게 됩니다.

그와 동시에, 인텔은 IA-64 Itanium (위 그림에는 안나왔지만)으로 한번 더 스마트 컴파일러 방식에 확실한 베팅을 하게 되는데요. 간단한 구조의 하드웨어로 정적(static) 컴파일 타임 스케줄링에 의지하는 디자인으로 말이죠. 결국 IA-64 는 망했죠. Pentium 4 에서 발생한 전력효율및 발열문제가 상당했었고, AMD 의 2GHz 대 느린 클럭의 Athlon 프로세서가 실제 현장에서 Pentium 4 보다 월등한 성능을 보였거든요. 이를 계기로, 인텔은 디자인 방향을 다시한번 뒤집읍니다. 예전의 Pentium Pro/II/III 에서사용된 브래니악 디자인을 바탕으로 Pentium M 과 그 후속작인 Core 제품들을 내놓게되고, 이 제품들은 큰성공을 거두게되죠.

# 전력 절벽과 인스트럭션 레벨 병렬화 절벽(The Power Wall & The ILP Wall)

역주: Wall 이라는 단어는 여기서 넘사벽쯤의 의미로 쓰였읍니다. 그냥 벽 보다는 절벽이 더 절망적인 느낌이라 사용했읍니다.

Pentium 4 에서 발생한 심각한 전력 및 발열 문제들은 클러스피드에 한계가 있다는걸 여실히 보여 주었읍니다. 전력소모량이 클럭 스피드향상 보다 훨씬 빠르게 증가 한다는 점이죠, 어떤 칩 기술을 사용하던 말이에요. 예를들면, 20%의 클럭 스피드 향상은 50% 정도의 전력소모를 증가시킨다는 겁니다.

서킷전체가 향상된 속도로 움직인다고 가정했을때, 짧아진 타이밍 요구조건을 만족시키기 위해서는 트랜지스터들이 20% 이상 더 자주 스위칭 되야하고, 더 높은 전압이 필요하기 때문이죠. 필요전력은 클럭진동수에 선형으로 증가 하지만, 필요전압은 제곱(square)단위로 늘어나서, 결과적으로 매우 높은 클럭스피드에서는 "삼중고(triple whammy)" 를 발생시킵니다($f*V*V$).

이게 끝이아니죠, 트랜지스터를 스위칭 하는데 드는 전력외에도, 트렌지스터가 꺼져있을때도 새어나가는 전력(*leakage* power)도 있읍니다. 트랜지스터가 꺼져있다고 하더라도, 아주 적은양의 전류는 흐르게되 있는데요. 이 전류들도 전압이 올라가면 늘어나게 되는거지요.

여기까지도 봐줄만 하다고 생각한다면, 한가지가 더 있읍니다. 일반적으로 새나가는 전력은 온도에 따라 높아지게 되는거거든요. 이는 실리콘 안쪽의 더 뜨겁고 활발한 전자들의 움직임이 늘어나면 트랜지스터의 효율은 떨어지게 되기때문입니다.

결과적으로, 오늘날 모던 프로세서들의 클럭 스피드를 30% 정도 올리면 전력은 현재의 거의 두배 가까이 들고 발열도 두배쯤 늘어나게 됩니다.

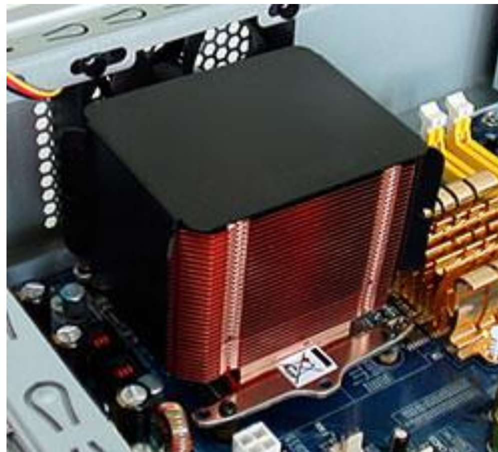

Figure 12 – The heatsink of a modern desktop processor, with front fan removed.

어느정도 까지는 전력소모가 늘어나도 상관없읍니다만, 150 ~ 200 watts 쯤 부터는 전력및 발열 문제가 처치 곤란해집니다. 실리콘 칩에 그정도 전력과 쿨링을 제공하기가 현실적 으로 불가능하기 때문입니다. 설령, 서킷이 보다 높은 클럭 스피드로 동작할수 있다고 해도 말이죠. 이 문제를 전력절벽(*power wall)* 이라고 부릅니다.

Pentium 4, IBM's POWER6 그리고 최근의 AMD  Bulldozer/Piledriver 프로세서들처럼 클럭 스피드에 초점을 맞춤 제품들은 전력절벽(power wall) 에 금새 부딪히고 말았고, 계획했던 클럭 스피드를 낼수 없다는걸 알게 됐읍니다. 결과적으로, 인스트럭션 레벨 병렬화를 보다 강화한 저속의 똑똑한(*smarter)* 프로세서들 에게 뒤쳐지고 말았죠.

따라서, 클럭 스피드만 보고 내달리는 전략은 그리 좋은게 못 됩니다. 당연히, 랩탑이나, 태블릿 또는 스맛폰 같은 포터블, 모바일 장비들은 전력절벽에 더 빠르게 다다릅니다. 랩탑은 50W 정도, 태블릿은 10W, 맛뽄들은 5W 보다 적은 값에도 말이죠. 이유는 배터리 용량과 제한된 (혹은 팬없는) 쿨링등에서 오는 제한들 때문입니다.

그럼, 클럭 스피드에만 몰두하는게 문제라면, 브래니악쪽에 몰빵하는게 맞을까요? 안타깝게도 그렇지 않읍니다. ILP 를 끌어올리는데도 한계가 있읍니다. 왜냐면, 일반적인 프로그램들은 로드지연(load latency)이나, 캐시 미스, 분기, 그리고 인스트럭션간 의존성 등의 이유로 아주 잘 정제된 병렬화 작업이 많지 않기 때문 입니다. 이런 식으로 제약이 걸리는걸, 인스트럭션 레벨의 병렬화 절벽(*ILP wall*) 이라고 합니다.

초창기의 POWER, SuperSPARC, 그리고 MIPS R10000 와 같이 ILP 에만 몰빵했던 프로세서들은, 얼마 못가서 ILP 를 뽑아내는데 한계를 느끼게 됩니다. ILP 를 위해 복잡도가 증가해서, 클럭 스피트를 올리기가 어려워 졌거든요. 결과적으로, 이 칩들은 ILP 에 상관치 않는 멍청하지만 클럭스피드가 높은 칩들에 뒤쳐지게 되어 버렸지요.

다중 4 처리(4-issue) 수퍼스칼라 프로세서의 경우 각 싸이클 마다 의존성이나 지연시간(lantencies) 이 일정 기준에 맞는 4 개의 독립적인 인스트럭션이 준비 되어 있어야 합니다. 하지만, 실제로 이런 조건을 만족시키기는 거의 불가능하죠. 특히, 로드지연(load latencies) 이 3 ~ 4 싸이클이 걸리는 점을 감안 하면 말입니다.

현재, 주로 사용되는 싱글 쓰레드 애플리케이션들에 대한 실제 ILP 효율은 싸이클당 2~3 개의 인스트럭션 정도로 제한되어 있읍니다. 사실, SPECint 밴치마크를 돌려보면, 모던 프로세서들은 한 싸이클당 평균적으로 2 개 이하의 ILP 효율을 보입니다. 이 SPEC 밴치마크들은 실생활에 사용되는 대형 메인스트림 애플리캐이션들 보다 ILP 를 뽑아내기 쉬운 편인데도 말이죠.

과학 계산(scientific)용 코드 같은건 병렬화율이 조금 높긴하지만, 메인스트림 애플리케이션들과는 약간 거리가 있지요. 또한, 어떤 프로그램들은 포인터 참조가 빈번한(pointer chasing) 코드가 많아서, 1 싸이클을 유지하기도 벅찰때가 있읍니다. 이런류의 프로그램들은, 메모리 절벽( *memory wall*) 이라는 또 다른 문제에 봉착 하게 되지요. (나중에 다룸)

# X86 에 대해서 (What About x86?)

그럼, x86 은 이런 것들중에 어디쯤에 속할까요? 그리고 인텔과 AMD 는 35 년도 더된 아키택처로 어떻게 경쟁력을 유지하고 있을까요?

수퍼스칼라 x86 이었던 오리지날 Pentium 은 놀랄만한 엔지니어링 작품 이었지만, 복잡하고 지저분한 인스트럭션 세트는 커다란 문제 였읍니다. 복잡한 메모리 주소

지정(addressing) 방식들과 적은 수의 레지스터들은 잠재적인 의존성으로 인해 병렬처리가 거의 불가능 하다는걸 의미 했지죠. RISC 아키택쳐와 경쟁하기 위해선 x86 인스트럭션 세트의 단점을 어떤식으로든 뛰어 넘어야 했읍니다.

이 문제의 해결 방법이 NexGen 이라는 회사와 Intel 의 엔지니어들의해 거의 비슷한 시점에 개별적으로 개발 되게 됩니다. X86 의 복잡한 인스트럭션들을 보다 간단한 RISC 칩들이 쓰는 것 과 비슷한 모양의 마이크로 인스트럭션들로 동적으로 디코딩하는 방법 이었지요. 그 마이크로 인스트럭션들을 RISC 스타일의 레지스터 리네이밍과 OOO 수퍼스칼라를 사용하는 빠른 코어들을 사용해 실행 할수 있게 되었읍니다. 이 마이크로 인스트럭션들은 마이크로 옵스(*µops*) 라고 불렸고, 대부분의 x86 인스트럭션들이 1 ~ 3 개의 마이크로 옵스들로 디코드 되게 되었읍니다. 물론 아주 복잡한 것들은 더 많이 쪼개 지겠죠.

이렇게 decoupled 된 수퍼스칼라 x86 프로세서들에게, 레지스터 리네이밍은 절대적으로 필요한 기술 이었읍니다. 32 bit 모드에서 x86 아키택처는 꼴랑 8 개의 레지스터 밖에 없었으니까요 (64bit 모드에서 8 개 더 추가됨). 이 기술의 효과는 리네이밍을 통해 얻는 이점이 미미했던 RISC 아키택쳐들에서 와는 극명한 차이를 보였는데요. 기발한 레지스터 리네이밍을 통해 RISC 칩들이 사용하던 거의 모든 트릭들을 x86 세계에서도 사용 할 수있게 되었으니까요. 물론, 마이크로 옵스들이 x86 레이어에 가려져서 컴파일러가 볼수있는 부분이 적었던 관계로, 보다 진보된 정적 인스트럭션 스케쥴링과 메모리 엑세스를 피하기위해 보다 큰 레지스터 세트를 사용하는 트릭들은 제외 하고 말이죠. 기본적인 동작구조는 아래 그림에서와 같읍니다.

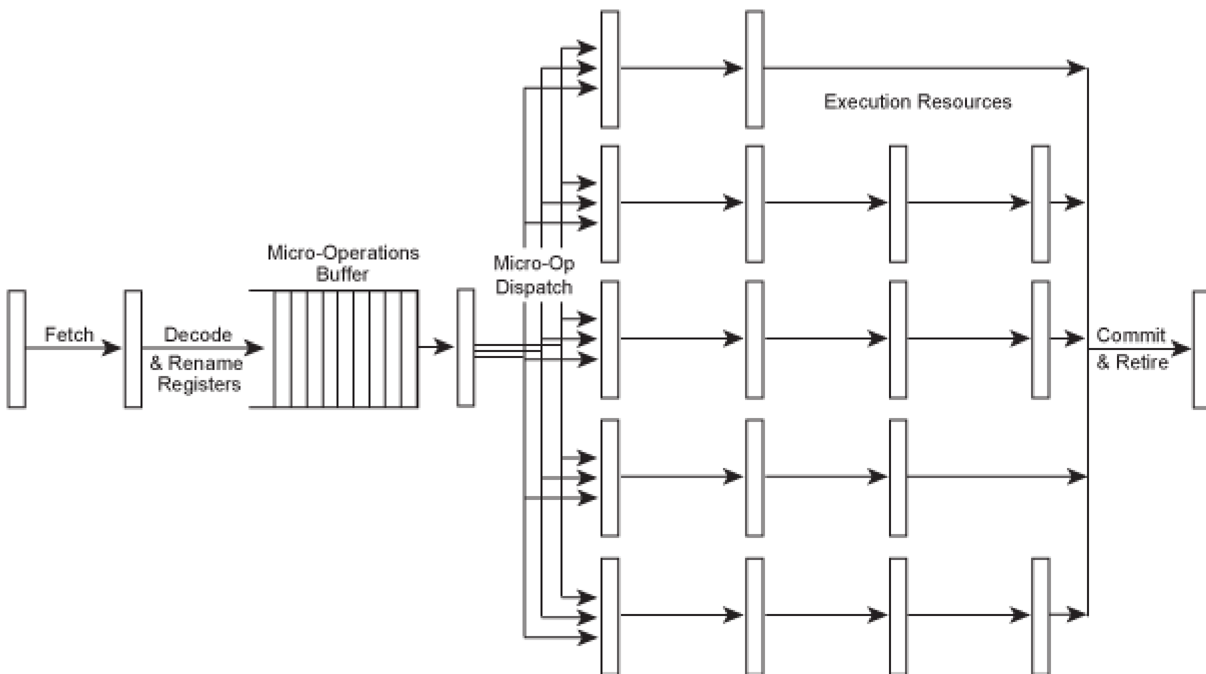


Figure 13 – A "RISCy x86" decoupled microarchitecture.

NexGen 의 Nx586 과 Intel 의 Pentium Pro (P6 로도 알려짐)가 이런식의 decoupled x86 마이크로 아키텍처를 채용한 최초의 프로세서들 이었으며, 요즘에는 모든 x86 프로세서들이 채용하고 있지요. 물론, 코어 파이프라인 이나 펑셔널 유닛 디자인같은 세세한것들은 프로세서들마다 다 다릅니다. 개별 RISC 프로세서들이 그랬던것 처럼 말이죠. 하지만, x86 인스트럭션을 RISC 인스트력션 형태의 마이크로 옵(μop)으로 변환하는 큰틀은 모두 동일 하죠.

몇몇 최신 x86 프로세서들은 변환된 마이크로 옵(μop) 들을 작은 버퍼나 전용 “L0” 인스트럭션 캐시등에 저장 해놓고 재 사용하기도 합니다. 루프를 돌때 똑같은 변환 작업을 반복하지 않도록 말이죠. 이렇게 해서 수행시간과 소비전력을 줄일수도 있었읍니다. 저 위쪽에 수퍼스칼라 섹션에서 예로들었던 파이프라인의 깊이 차트상에, Core i*2/i*3 Sandy/Ivy Bridge 그리고 Core i*4/i*5 Haswell/Broadwell 의 값이 14/19 단계로 표시된 이유도 여기있는데요. L0 μop 캐시를 사용하면 파이프라인이 14 단계가 되지만, L1 인스트럭션 캐시를 사용해야하면, 마이크로 옵들을 재변환 해야 하기때문에 19 단계로 늘어나게 되는거죠.

X86 인스트럭션 적재(fetch) 와 디코드(decode)를 좀 더 안쪽에서 수행되는 RISC 와 비슷한 마이크로 옵 인스트럭션 디스패치(dispatch) 와 실행(execution)단계로부터 분리 해 내는(decoupled) 이 방식은 모던 x86 프로세서들의 처리넓이(*width*)를 정의하기 어렵게 만듭니다. 더우기, 이런 프로세서들은 내부적으로 몇몇 마이크로 옵들(load-and-add or compare-and-branch)을 추적하기 쉽도록 그룹화 하기 때문에 처리넓이(*width*)를 정의 하기가 훨씬 모호해 지는 경향이 있읍니다.

예를들어서, Core i*4/i*5 Haswell/Broadwell 같은 칩들은 싸이클당 5 개의 x86 인스트럭션들을 디코딩 할수 있는데요. 최대 4 개의 마이크로 옵 그룹(fused μops)을 생성하게 됩니다. 이것들은 또 L0 μop 캐시에 저장되고, 여기서는 한 싸이클당 4 개 까지의 μop 그룹이 적재되고, 레지스터 리네이밍을 거쳐, 리오더 버퍼에 저장됩니다. 그럼 거기서 는 싸이클당 8 개까지의 분리된 개별 μop 들이 펑셔널 유닛들로 보내 집니다(issued) , 그기서 완료될때까지 다양한 파이프라인을 거치게되고, 또 거기에선 한 싸이클당 4 개까지의 μop 그룹들이 저장 (committed) 되고, retired 될 수 있지요.

그래서, Haswell/Broadwell 칩의 처리넓이는 어떻게 될까요?

이 프로세서는 실질적으로 8 다중처리(8-issue) 넓이를 가진다 라고 말할 수 있읍니다. 한 싸이클당 8 개까지의 분리된(un-fused) 마이크로옵(μop)을 적재(fetched), 처리(issue), 그리고 완료 할 수있기 때문이죠. 그것들이 최적의 방식으로 배합(paired/fused) 되어 있다는 가정 하에서요. 그리고, 분리된(un-fused) 마이크로옵(μop)은 간단한 RISC 인스트럭션에 거의 직접 대응 되기 때문이기도 합니다.

하지만, 전문가들 사이에서 조차도 이런 디자인의 처리넓이를 정확이 어떻게 정의 해야 하는지에 대해 의견이 분분합니다. 그룹화된 마이크로옵(fused μops) 단위로

따지면 4-issue 도 맞읍니다. 그 프로세서가 트래킹용 단위로 얘네를 쓰거든요. 오리지널 x86 인스트럭션 기준으로 생각하면 5-issue 도 맞는얘기가 되죠. 물론 이 처리넓이 정의에 관한 이 난제는 다분히 학술적인 겁니다. 그 어떤 프로세서도 실제 사용되는 코드를 돌릴때 여기서 예시된거 처럼 고도의 ILP 효율을 내진 못 하니까요.

RISC 스타일의 x86 그룹에서 가장 흥미로운 멤버중 하나는 바로 Transmeta 의 Crusoe 프로세서 였는데요. 이 칩은 x86 인스트럭션들을 내부적으로 수퍼스칼라 용 인스트럭션이 아닌 VLIW 형식으로 변환했읍니다. 이 변환작업을 소프트웨어를 이용해 런타임에 처리했죠, JVM 처럼말이죠.

이 방식은 프로세서 구조를 간단한 VLIW 가 될수있도록 만들었읍니다. 복잡한 x86 디코딩및 레지스터 리네이밍 하드웨어가 필요치 않았고, 수퍼스칼라 디스패치혹은 OOO 로직도 필요없었죠. 소프트웨어 기반의 x86 변환기법은 하드웨어 기반보다 성능을 떨어트리긴 했지만, 결과적으로는 매우 적은 전력을 사용하면서 가볍고 빠르며 발열이 적게 동작하는 프로세서를 탄생 시켰읍니다.

600 MHz 짜리 Crusoe 프로세서가 저 전력 모드(300 MHz)로 동작하는 500 MHz Pentium III 랑 비슷한 성능을 냈었읍니다. 비교할수 없을정도로 적은 전력소비와 발열양을 내면서 말이죠. 이러한 특성은 이 칩을 랩탑이나, 핸드핸드 컴퓨터등 배터리 라이프가 중요한 기기들에 이상적으로 만들어 주었읍니다.

물론 요즘엔, Pentium M 이나 Core 후속 같은 x86 프로세서들이 저전력 소비를 중점으로 고려해 설계 되었기 때문에, Transmeta 스타일의 소프트웨서 기반 접근 방식을 무용지물로 만들었죠. 그래도, 이와 매우 비슷한 방식이 지금도 NVIDIA 의 Denver ARM 프로세서들에 사용되고 있읍니다. 저전력 소비로도 높은 성능을 내겠다는 목표를 가지고 말이지요.

## 쓰레드-SMT, 하이퍼쓰레딩, 그리고 멀티코어(Threads-SMT, Hyper-Threading & Multi-Core)

앞서도 얘기했듯이, 수퍼스칼라를 통한 ILP 효율성은 대부분의 일반 프로그램들이 잘 정제된 병렬화 가능성을 거의 가지고 있지않다는 사실에 의해 급격히 감쇠되죠. 이 때문에, 스마트하고 어그레시브한 컴파일러의 지원을 받는 매우 극단적인 브레니악 디자인의 OOO 수퍼스칼라 프로세서 조차도, 일상적으로 사용되는 메인스트림 소프트웨어를 돌렸을때, 평균적으로 싸이클당 2~3 개의 인스트럭션 밖에 처리하지 못 합니다. 적제단계에서의 정체, 캐시미스, 분기, 그리고 인스트럭션간 의존성등의 복합적인 문제들 때문에요.

동일 싸이클 내에서 여러개의 인스트럭션이 처리되는건 잘해봐야 몇 사이클정도의 짧은 찰라 일뿐이고, 이나마도 저효율 ILP 코드를 실행하는 여러 싸이클로 나눠지게 됩니다. 따라서, 이론적인 최고 성능에는 발끝 만큼도 못미치는 겁니다.

만약 현재 실행되고 있는 프로그램안에 부수적으로 실행 될수있는 개별 인스트럭션들이 없다면, 그 프로그램내의 또 다른 쓰레드나, 주변에서 실행중인 다른 프로그램이 이러한 인스트럭션들을 제공해줄 잠재적인 공급원이 될수도 있읍니다. 이런 형태의 쓰레드레벨 병렬화를 활용한 프로세서 디자인 기법을 *Simultaneous multithreading* (SMT) 이라고 합니다.

재차 얘기하지만, 파이프라인속에 비어있는 버블들을 뭔가 유용한 인스트럭션으로 채워넣는 것이 목표입니다. 하지만, 이번에는 같은 프로그램안 어딘가에 존재하는 인스트럭션들을 사용하지 않고, 해당 코어에서 돌고있는 여러 쓰레드들로 부터 인스트럭션들을 끌어 오는거죠.

따라서, 하나의 SMT 프로세서는 시스템의 다른 부분들에게 마치 여러개의 독립된 프로세서들 처럼 보여지게됩니다. 마치, 진짜 멀티 프로세서 시스템 처럼요. 물론 진짜 멀티프로세서 시스템 역시 여러 쓰레드를 동시에 실행시키지만, 한 프로세서당 하나의 쓰레드만 돌리죠. 이는 한개의 칩에 여러개의 프로세서 코어를 올린 멀티코어 프로세서들 에게도 해당됩니다.

반면, SMT 프로세서는오직 하나의 물리적 프로세서 코어만 사용해 여러개의 논리적 프로세서처럼 보여지게 하는거지요. 이는 SMT 칩을 멀티코어 프로세서보다 공간 활용, 제작단가, 전력소모 그리고 발열감쇠 등에서 월등하게 효율적 이도록 만들어 줍니다. 그리고 당연히, 개별코어에 SMT 디자인을적용해 멀티코어 시스템을 구성해도 아무런 문제가 없지요.

하드웨어적 측면에서 보면, SMT 를 구현하기 위해선 프로그램 카운터, 아키텍쳐상에 보여지는 레지스터들(리네임 레지스터말고), TLB 에 저장되는 메모리 맵핑등과 같이 쓰레드의 실행상태(execution state)들을 저장하는 실제 프로세서의 각 부품들을 모두 중복시키는 작업이 필요합니다.

운좋게도, 이런 부품들은 프로세서의 전체 하드웨어중 아주 작은 부분을 구성합니다. 정말 크고 복잡한 디코더들이나, 디스패치 로직, 펑셔널 유닛들, 그리고 캐시들과 같은 부품들은 모두 쓰레드들끼리 공유할수 있는 부분입니다. 물론, 프로세서 역시 어느 인스트럭션들과 리네임 레지스터들이 어떤 쓰레드들에 소속 되어있는지 항상 파악하고 있어야 하는데, 그걸 가능하게 하기위해 추가되는 코어 로직의 복잡도는 그렇게 크지 않읍니다.

따라서, 코어에 10% 정도 더 추가되는 로직때문에 발생하는 상대적으로 저렴한 디자인 비용과, 무시할 수있을 정도로 적게 증가된 전체 트렌지스터 수및 최종 생산 비용만으로도, 이 프로세서는 여러 쓰레드들을 동시에 실행할수 있게되었고, 이는

펑셔널 유닛의 활용성 및 클럭당 처리할 수 있는 인스트럭션의 수를 눈에띄게 올려줄것이라 여겨졌읍니다 (당연히 전체적인 성능도요). SMT 프로세서의 인스트럭션 플로우는 아래와 같읍니다.

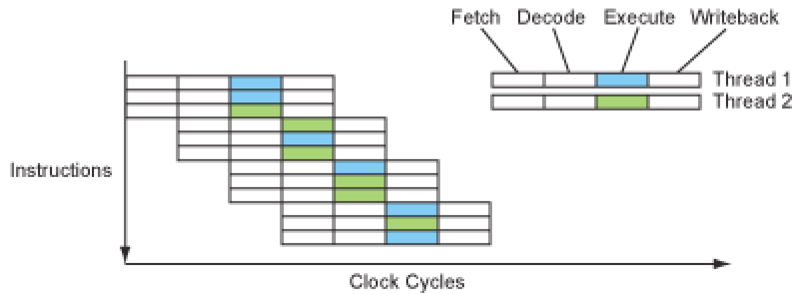


Figure 14 – The instruction flow of an SMT processor.

정말 멋지죠! 이제 여러개의 쓰레드를 돌림으로써, 파이프라인속 버블들을 채워넣을수 있겠네요. 싱글쓰레드 프로세서에 있는것보다 많은 펑셔널 유닛을 추가 할수도 있겠구요. 다중 인스트럭션을 실행과 관련된 문제도 해결할 수 있을것 같읍니다. 경우에 따라서는, 싱글 쓰레드의 성능향상도 기대해 볼수있을것 처럼보이네요 (특히 ILP-프랜들리한 코드에서요). 20 다중처리(20-issue) 정도는 문제도 아니겠네요, 그렇죠? 불행스럽게도, 대답은 ‘아니요’ 입니다.

SMT 의 성능을 제대로 내기는 좀 까다로운 면이 있읍니다. 먼저, SMT 는 많은 프로그램들이 동시에 실행되거나(놀고 있는게 아니라), 만약 하나의 프로그램만 돈다치면, 그안에 여러개의 쓰레드가 동시에 실행되고 있다는 상황을 가정하고 만들어 진것입니다.

현존하는 멀티 프로세서 시스템들을 에게서 목격된 바 로는, 그 가정이 항상 사실은 아니 라는걸 보여주죠. 실제로, 적어도 데스크탑들이나 랩탑들 그리고 소규모 서버들에 한해서는 이러한 경우가 극히 드물죠. 그래서, 보통은 그 장비가 주로 사용되는 한 가지의 작업내에서 해결 봐야하는 경우가 많죠.

데이터베이스 시스템이나, 이미지/비디오/오디오 프로세싱, 3D 렌더링 그리고 과학계산용 애플리션같은 것들은 쉽게 병렬화할수있는 가능성이 높은 코드들이 여기저기 널려 있지만, 아쉽게도 멀티프로세서를 활용하기위한 멀티 쓰레드사용을 염두하지 않고 작성된 경우가 많습니다. 더군다나, 이런류의 애플리케이션들은 프로세서 때문이아니라 메모리 대역폭에 의해 병렬화 제한을 받는경우가 허다하죠.

따라서, 쓰레드나 프로세서를 늘린다고 도움이 되지 않읍니다. 메모리 대역폭이 극적으로 증가되지 않는다면 말이죠(나중에 다룸). 뿐만 아니라, 웹브라우져나 멀티미디어 디자인툴, 인터프리터, 하드웨어 시뮬레이션툴 등등 의 많은 애플리케이션들이 현재로썬 아예 병렬화를 염두하고 작성되지 않았거나, 설사 그렇더라고 멀티 프로세서를 효과적으로 활용하기엔 부족함이 있읍니다.

거기다가, SMT 디자인에서는 모든 쓰레드들이 오직 한개의 프로세서 코어와 한 세트의 캐시를 공유하기때문에 진정한 멀티 프로세서(혹은 코어)에 비해 성능상의 단점이 상당합니다. SMT 프로세서의 파이프라인 안에서, 한 쓰레드가 다른 쓰레드들이 써야되는 펑셔널 유닛을 독차지 해버리면, 그 쓰레드들 전체가 기다려야 하는 상황이 발생합니다. 해당 펑셔널 유닛을 아주 잠깐만 쓰면 되는것들도 말이죠. 따라서, 쓰레드들의 작업이 균형있게 진척되도록 하는게 매우 중요합니다. SMT 를 가장 효율적으로 사용하려면 매우 변동성이 큰 코드 배합들로 구성된 애플리케이션을 돌리는게 좋읍니다. 쓰레드들이 동일한 하드웨어 리소스를 놓고 경쟁하는 상황을 최소화 하는게 관건인거죠.

또한, 캐시 영역을 두고 여러 쓰레드들이 경쟁하는 상황은 한 쓰레드가 모든 캐시영역을 독점 할때 보다도 않좋은 결과를 낼수도 있읍니다. 특히나, 크리티컬 워킹셋이 캐시용량에 민감한 하드웨어 시뮬레이터/에뮬레이터, 버츄얼머신들 그리고 고화질 비디오 인코딩 애플리케이션 같은 것들은 말이죠. 결론은, 사용하는 애플리케이션에 따라 SMT 의 성능이 싱글 쓰레드의 성능이나 전통적인 쓰레드간 컨택스트 스위칭 성능보다 더 나쁠수도 있다는 얘깁니다.

다른 한편으론, 데이터 베이스 시스템이나, 3D 그래픽스 렌더링, 그리고 많은 수의 제네럴 퍼포스의 코드들처럼 메모리 대역폭 말고 메모리 정체(latency)때문에 제약을 받는 애플리케이션들은 SMT 를 사용해 굉장한 이득을 볼수있지요. 캐시미스나 로드정체(load latency) 같은 상황에서 놀고있을 싸이클을 잘 활용할수 있도록 해 주니까요.

따라서, SMT 의 성능은 정의하기 매우 복잡하고 애플리케이션에 따라 극명한 차이를 보이게 됩니다. 이런 특성은 제품 마케팅을 힘들게도 하죠. 어떤때는 물리적인 두개의 프로세서 만큼 빠르다가도, 또 어떤때는 진짜 후진 두개의 프로세서 같기도하고, 심지어는 한개의 프로세서 보다 못할때도 있으니까 말이죠.

Pentium 4 가 SMT 기술을 처음 적용한 프로세서 였읍니다. 인텔은 이걸 “하이퍼 쓰레딩” 이라고 불렀죠. 2 개의 쓰레드가 동시에 동작할수 있도록 해주는 디자인 이었죠. 이 칩의 초기 리비전들은 버그때문에 SMT 기능을 꺼두기도 했었지만 말이죠. SMT 를 적용했던 Pentium 4 는 애플리캐이션에 따라 -10% 에서 +30% 정도의 성능차를 보였읍니다.

그후 인텔이 브레니악 디자인으로 회기하던 때에 나온 Pentium M 와 Core 2 는 SMT 를 버리고, 멀티 코어 디자인으로 진화했지요. 그 당시에 많은 SMT 디자인들이 버림을 받았는데. Alpha 21464 나 UltraSPARC V 같은 것들이 그 예입니다. 그후로 오랫동안, SMT 는 완전히 사장되는가 싶었지만, POWER5 이 멀티코어와 함께 2 쓰레드 SMT 디자인을 채용하면서 재기 하였읍니다(2 쓰레드/core * 2 코어/칩 = 4 쓰레드/칩).

인텔의 Core i 시리즈 역시 2-thread SMT 기술을 적용해서, 쿼드코어 Core i 프로세서는 8-쓰레드 칩이 되었읍니다. Sun 이 가장 공격적으로 쓰레드레벨 병렬화를

사용했었는데, UltraSPARC T1 Niagara 는 각기 4 쓰레드 SMT 를 적용한 8 개의 간단한 순차적(in-order) 코어들을 장착해 한 칩에서 총 32 개의 쓰레드를 지원 했었읍니다. UltraSPARC T2 에서는  코어당 8 쓰레드를 지원 하게 되었고, UltraSPARC T3 에 이르러선 칩하나당 16 개의 코어를 장착 함으로써 128 쓰레드를 지원하는 경지에 다다르게 됩니다.

## 더 많은 코어 혹은 더 넓은 코어(More Cores or Wider Cores)?

비슷한 넓이라면 (equally wide) 쓰레드 레벨 병렬화를 인스트럭션 레벨로 변환시킬수 있는 능력과 ILP 친화적인 코드에서 보다 나은 싱글 쓰레드 성능을 낼수있는 장점이 결합된 SMT 가 존재하는 상황에서, 누가 구지 멀티 코어 프로세서를 만들까 의아해 하는 독자들도 있을텐데요. 이게 그렇게 간단한게 아닙니다.

앞서 보았듯이, 매우 넓은 수퍼스칼라 디자인은 칩에 필요한 공간과 클럭 스피드 측면에서 봤을때 확장성(scale)이 매우 떨어집니다. 가장 문제가 되는것중 하나가, 복잡한 다중처리(multiple-issue) 디스패치 로직을 효과적으로 확장하기가 어렵다는 점입니다. 로직의 크기가 처리넓이(issue width) 의 제곱단위로 늘어나게 되는데, 다중처리 후보인 $n$ 개의 인스트럭션을 그 후보군내의 모든 다른 인스트력션과 비교해 봐야 하기 때문이지요.

인스트럭션들이 배열되는 순서에 제한을 둔다던가, 어떤 “처리 규칙(issue rules)” 들을 사용 한다던가, 혹은 진짜 기발한 엔지니어링 기술로 약간 줄일수는 있겠지만, 결국 $n^2$ 를 벗어날 수가 없읍니다. 이 말인즉슨, 4 다중처리 디자인과 비교했을때, 5 다중처리(5-issue) 프로세서는 50% 이상, 6 다중처리는 두배이상, 7 다중처리는 세배이상 크기가 증가하게 되는 것이죠. 8 다중처리를 구현하려면 로직의 크기가 4 배이상 커지게 되겠죠, 처리 넓이는 단지 2 배 밖에 증가하지 않는데 말이죠.

이에 더해서, 이 디자인은 고도의 멀티 포티드(multi-ported) 레지스터 파일들과 캐쉬들이 필요합니다. 동시에 요청된 엑세스들을 처리하기 위해서말이죠. 이 두가지가 결합되서, 크기를 키울 뿐만아니라 서킷 레벨에서 와이어링의 거리를 굉장히 길어지게 만듭니다. 결과적으로 클럭 스피드에 심각한 제약을 걸게 되는 거죠.

따라서, 하나의 10 다중처리(10-issue) 코어가 실제로는 두개의 5 다중처리 코어들 보다 더 크고 느려지는 불쌍사가 벌어지게 됩니다.  고로, 우리 꿈속의 20 다중처리(20-issue) SMT 디자인은 서킷 디자인 제약으로 인해 실현 시킬수 없는것 입니다.

결국에는, 타겟 애플리케이션의 특성에 따라 SMT 와 멀티 코어 디자인의 이점이 결정되기 때문에 다양한 수준으로 이 둘을 적절히 채용하는 넓은 범주의 디자인이 사용될 수 밖에 없는 것이죠. 오늘날, 전형적인 SMT 디자인은 넓은 실행 코어와 OOO

실행 로직을 둘다 채용 하고 있습니다. 여러개의 디코더들과, 크고 복잡한 수퍼 스칼라 디스패치 로직등도 포함해서 말이죠. 그 결과, SMT 코어는 칩에 필요한 공간이 제법 크지요.그정도의 공간이면, 여러개의 보다 간단한 순차적(in-order) 단일처리(single issue) 방식의 코어들을 들여놓을 수도 있을겁니다. 아마 6 개까지도 가능할듯 싶네요!

이제, 인스트럭션 레벨 병렬화(ILP)나 쓰레드 레벨 병렬화(TLP)모두 기대한만 큼의 성능은 못 뽑아준다는 걸 알았고요(diminishing returns in different ways), SMT 는 기본적으로 TLP 를 ILP 로 변환해 주는 거라는 사실도 알게됐고, 넓은 수퍼스칼라 디자인은 칩의 크기 측면에서 선형으로 확장될수 없다는것도 배웠읍니다. 물론 디자인 복잡도와 전력 소모 문제도 있었구요.

그럼 적정선(sweet spot) 은 어디쯤이 될까요? 코어들의 넓이가 어느정도 여야 ILP 와 TLP 사이의 균형을 맞출수 있을까요? 현재도 여러 가지 다른 방식들이 실험되고 있읍니다…

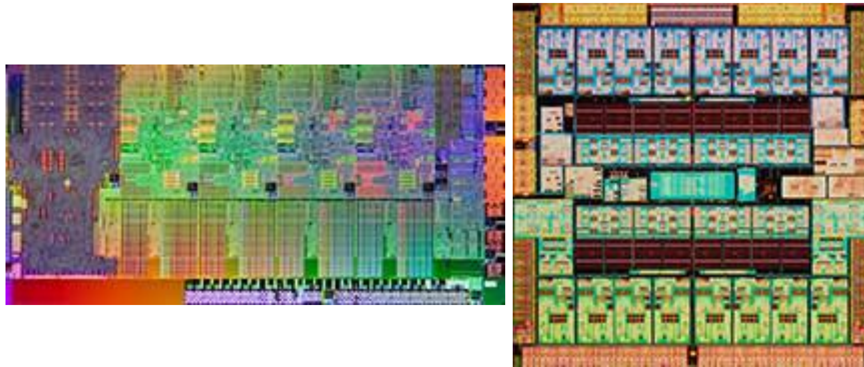

Figure 15 – Design extremes: Core i*2 "Sandy Bridge" vs UltraSPARC T3 "Niagara 3".

넓은 코어 진영의 끝판왕인 인텔의 Core i*2 Sandy Bridge  프로세서는 4 개의 크고 넓은 비순차적(out-of-order) 6 다중처리(6-issue)를 지원하는 극강(aggressive)의 브래니악 방식 코어들을 적용하고 있읍니다. 왼쪽 그림의 위쪽에 보이는게 코어들이고, 아래쪽 L3 캐쉬를 공유하고 있지요.  각 코어가 2 개의 쓰레드를 돌릴수 있어서 총 8 개의 "빠른" 쓰레드를 지원합니다.

그 반대 진영의 끝판왕은 Sun/Oracle 의 UltraSPARC T3 Niagara 3 프로세서로, 16 개의 작고 간단한 순차적(in-order) 2 다중처리(2-issue) 코어들을 사용하고 있읍니다. 오른쪽 그림에서 위에 8 개, 아래 8 개의 코어가 중간의 L2 캐쉬를 공유하고 있는게 보이는군요. 각 코어가 8 쓰레드를 돌릴수 있어서, 총 128 의 어마어마한 수의 쓰레드를 지원합니다. Sandy Bridge 의 쓰레드들 보다는 눈에띄게 느릴테지만 말이죠.

두 칩모두 2011 초에 나온 것들 이구요.  둘다 약 10 억개의 트랜지스터를 포함하고 있습니다. 위 의 두 그림은 트랜지스터 밀도는 비슷하다고 가정하고, 진짜랑 비슷한 스케일로 줄인 겁니다. 간단한 순차형 코더들이 얼마나 작은지 확인해 보세요!

어느 쪽이 더 좋은 방식이냐구요? 글세요, 간단히 답 할수가 없네요. 다시한번 말하지만, 애플리케이션에 따라 많이 달라집니다. 액티브 상태이지만 메모리 정체(latency) 에 제약을 받는 쓰레드가 많은 데이터 베이스나 3D 렌더링용 애플리케이션들 에겐, 간단한 코어가 많은게 좋겠죠. 크고 넓은 코어들은 어차피 메모리를 기다리는데 거의 대부분의 시간을 보낼꺼니까요. 하지만, 동작중인 쓰레드가 많지않은 대부분의 애플리케이션들에겐, 개별 쓰레드의 성능이 더 중요합니다. 따라서 적은 수의 넓고 보다 브래니악한 코어들이 더 어울리게 되는겁니다 (적어도 요새 애플리케이션들은 그렇습니다).

물론, 이 두 극단적인 디자인들 사이에서 다른 많은 옵션들 역시 시도되고 있는데요. 예를들어, IBM 의 POWER7 은 위 둘과 같은 세대의 칩이었고, 역시 약 10 억개에 가까운 트랜지스터를 가지고 있었지만, OOO execution 하드웨어를 보다 덜 사용한 4 쓰레드 SMT 디자인의 8 개의 코어를 채택 하였읍니다.  AMD 의 Bulldozer 디자인은 보다 혁신적인 방식을 사용했는데, 한개의 SMT 스타일 프론트 엔드를 2 개의 백엔드 코어들이 공유 하도록 설계 하였읍니다. 이 백엔드 코어들은 각자 멀티코어 스타일의 정수 실행(execution) 유닛들을 사용했지만, 부동소수 유닛은 SMT 스타일을 공유해서 SMT 와 멀티코어의 경계를 넘나들게 되었읍니다.

오늘날(2015 년 초반)에는, Moore's Law 덕분에 몇십억개의 트랜지스터를 집적할수 있게 되어서, 매우 공격적인 브래니악 디자인들도 제법 많은 수의 코어들을 채용 할 수 있게 되었읍니다. 인텔의 Xeon Haswell EP (Core i*4 Haswell 의 서버용 버젼) 프로세서는 5 십억 7 백만개의 트랜지스터를 사용해 18 개의 매우 공격적인 브래니악 방식의 2 쓰레드 SMT 를 지원하는 8 다중처리(8-issue) 코어를 탑재 하였읍니다 (Xeon Sandy Bridge EP 는 6 다중처리 코어 8 개였음).

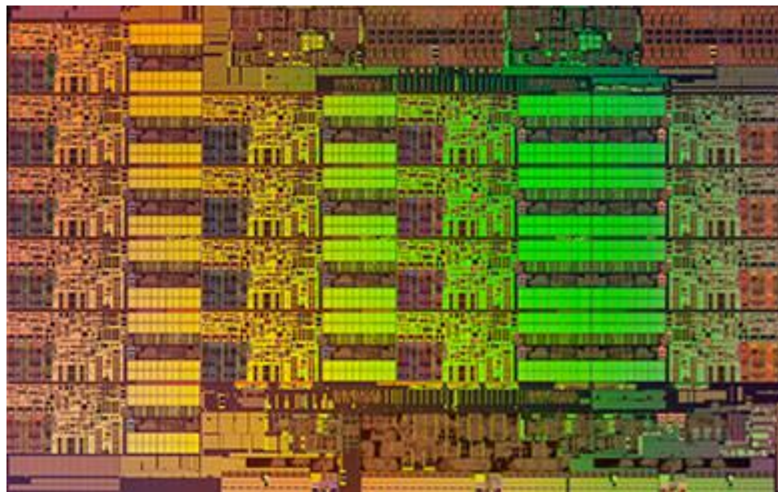

Figure 16 – Xeon Haswell EP with 18 brainiac cores, the best of both worlds?

반면에, IBM 의 POWER8 프로세서는 4 십억 4 백만개의 트랜지스터를 사용해서, POWER7 과 비교 했을때 매우 브레니악적인 방식의 디자인을 채택 하고도, 보다 많은 코어를 가지고 있었읍니다. 각기 8 개의 쓰레드를 지원하는 SMT 방식의 12 개의 코어를 탑재 했었으니 까요(POWER7 은 4 쓰레드 지원 SMT 를 이용한 8 개의 코어). 물론, 이런 브래니악 방식의 커다란 코어디자인이 그 많은 트랜지스터들을 얼마나 효과적으로 사용 했었는지는 또 다른 이야기 입니다.

어쩌면 미래에는, 면적당 멀티코어 성능이 좋은 작은 코어들과, 개별 쓰레드의 성능을 최대로 끌어올린 큰 코어들을 이용한 비대칭형 디자인을 기대해 볼수도 있을겁니다. 한 두개쯤의 크고 넓은 브래니악 코어들과 여러개의 작고 간단한 코어들이 섞여있는 그런 디자인 말이죠. 여러 측면에서, 이런식의 디자인이  제일 타당성이 있어보이죠. 고도로 병렬화된 프로그램들은 여러개의 작은 코어들덕을 보게될것이고, 순차적이고 단일 쓰레드방식의 프로그램들은 크고 넓은 브래니악 코어들 덕을 볼수 있을테니까 말이죠. 설령, 싱글 쓰레드 성능을 두배 정도 늘리기위해 4 배나 큰 공간이 필요하더라도 말입니다.

Sony PlayStation 3 에 사용된 IBM 의 Cell 프로세서가 논란의 여지와 함께, 이런 방식의 디자인을 채택한 첫번째 시도였지만, 불행하게도 프로그램 하기가 아주 심각하게 어려웠던 문제가 있었읍니다. Cell 에 탑재된 간단한 소형 코어들이 커다란 메인 코어와 인스트럭션 세트 호환이 안되었던 것이죠. 또한, 이 소형 코어들은 메인 메모리를 엑세스하는데 제한적이고 이상한 방식을 사용 할수 밖에 없었어서, 범용 CPU 보다는 특별한 용도의 코프로세서 역할 밖에 할수가 없었읍니다.

몇몇 모던 ARM 디자인들역시 비대칭형 방식으로 설계 됐었읍니다. 여러개의 대형 코어들을 소형코어 한 두개 정도와 결합시킨 형태였죠. 이 방식은 멀티코어 성능의 최대화가 목적이 아니라, 모바일 기기들 상에서 필요에따라 전력소모가 큰 대형코어들을 끌수 있게 함으로써 배터리 수명을 늘리는데 목적을 둔 디자인 이었읍니다. ARM 은 이 방식을  "big.LITTLE"이라고 불렀죠.

물론, 트랜지스터들의 집적도가 높아짐에 따라,  메인 CPU 칩안에 입출력이나 네트워킹 또는 비디오 인코딩/디코딩 전용 하드웨어, 아니면 아예 저성능의 GPU 전체와 같은 부가기능을 통합하는 것도 좋은 방법이죠.  이런식의 통합 방식은 칩의 갯수나 물리적 공간 혹은 비용등이, 메인 CPU 에 많은 코어들 심어서 얻을수 있는 성능상의 이득보다 훨씬 중요한 전화기, 타블렛, 그리고 소형의 저 성능 랩탑들과 같은 제품군에 매우 적합 합니다. 이런 혼합형 디자인을 *system-on-chip* 또는 줄여서 SoC 라고 부르죠.

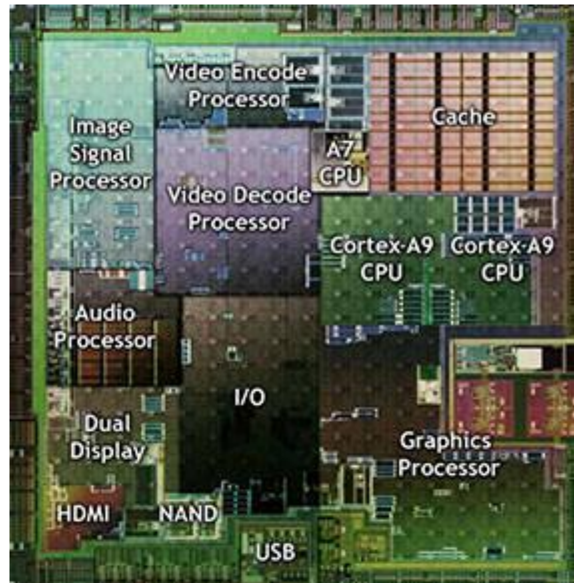


Figure 17 – A typical modern SoC: NVIDIA Tegra 2.

# 데이터 병렬화(Data Parallelism – SIMD Vector Instructions)

앞서 소개했던 인스트럭션/쓰레드 레벨 병렬화 와 더불어, 또 다른 병렬화 가능성이 여러 프로그램 내에 존재 하는데요. 바로 데이터 병렬화(data parallelism) 입니다. 이는 인스트럭션들을 그룹별로 병렬실행 할수 있는 방법들을 찾는대신, 한개의 인스트럭션에 그룹화된 데이터 값들을 병렬로 적용시킬 수 있는 방법을 찾는것이 핵심 입니다.

이 방식은 때때로 SIMD(Single Instruction, Multiple Data) 병렬화 라고 불리지만, 그보다 벡터 프로세싱(*vector processing*) 으로 더 많이 예기되죠. 주로 수퍼컴퓨터들이 이주 긴 벡터를 이용한 벡터 프로세싱을 많이 사용해 왔는데요. 수퍼컴퓨터들에서 돌아가는 과학계산 프로그램들의 형태가 벡터 프로세싱에 상당히 적합했기 때문이지요.

그러나, 오늘날엔, 벡터 수퍼컴퓨터들은 일반용 CPU 를 사용하는 멀티프로세서 디자인들에게 많이 밀려나 있는 상태입니다. 그럼 왜 벡터프로세싱을 다시얘기 하느냐? 많은 상황들에서, 특히 이미징, 비디오, 그리고 멀티 미디어 애플리케이션 같은것들에서, 프로그램은 똑같은 인스트럭션을 서로 연관성있는 값(짧은 벡터: 간단한 스트럭쳐 또는 배열)들로 이루어진 작은 그룹들에 실행해야 할때가 있읍니다.

예를들면, 어떤 이미지 프로세싱 애플리케이션에서 한 픽셀의 빨강, 초록, 파랑, 또는 알파(투명도) 값을 나타내는 8 비트 숫자들로 이루어진 그룹들을 더하거나 할 경우 처럼 말이죠.

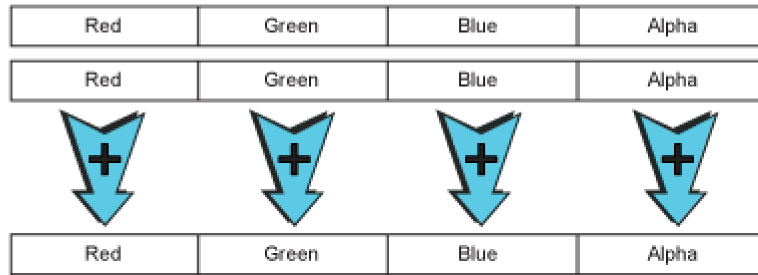


Figure 18 – A SIMD vector addition operation.

이때 수행되는 작업들은 한개의 32 비트 덧셈을 수행할때 와 똑같읍니다. 매 8 번마다 발생하는 캐리(carry)가 퍼져나가지(propagated) 않는다는 점만 제외하면 말이죠. 또한, 8 비트가 모두 찼을때 값들이 0 으로 넘어가지(wrap)않고, 최대값인 255 를 유지하고 있는게 더 좋을때도 있어요 (포화연산- saturation arithmetic). 바꿔 말하면, 매 8 번째 캐리(carry)가 발생하는 대신 모든 비트가 1 인 결과를 유발하게 되는겁니다. 따라서, 위에 나온 벡터 덧셈은 사실 변형된 32 비트 더하기에 불과 한거죠.

하드웨어 측면에서 보면, 이런류의 벡터 인스트럭션들 추가하는게 그렇게 어려운일은 아닙니다. 기존의 레지스터들이 재사용될 수 있고, 많은 경우들에서 펑셔널 유닛들도 기존의 정수 또는 부동소수형 유닛들과 공유될수 있으니까요. 바이트 셔플링(suffling) 같은 작업들에 유용한 패킹/언패킹 인스트럭션들도 추가될수 있을 꺼고요, 비트 마스킹(bit-masking) 작업등을 위해 서술형 인스트럭션들도 몇개 정도 추가 할수 있겠죠. 약간만 생각해보면, 소수의 벡터 인스트럭션 세트로도 인상적인 속도 향상을 낼수 있읍니다.

당연히, 32 비트에서 멈출 이유는 없읍니다. 만약 64 비트 레지스터들이 존재한다면, 64 비트 벡터를 제공하기위해 사용될수 있겠죠 (대부분의 아키텍쳐들이 적어도 부동소수 연산을 위해서 64 비트 레지스터들을 구비하고 있음). 이렇게 하면, 결과적으로, 병렬화를 두배로 올리게 되는겁니다. SPARC VIS 와 x86 MMX 가 이미 이렇게 하고있죠. 만약 완전히 새로운 레지스터들을 만드는게 가능하다면, 이왕이면 더 넓은게 좋겠죠. SSE 는 128 비트 짜리 레지스터 8 개를 추가했었구요. 나중에 64 비트 모드에서 16 개로 늘렸고, AVX 를 들여놓으면서 256 비트로 넓혔읍니다. 반면, PowerPC AltiVec 는 처음부터 32 개의 128 비트 레지스터들을 제공했었죠(Power PC 의 좀 나눠놓는 디자인 스타일을 유지함, 예네는 분기 인스트럭션용 레지스터도 따로있음 심지어).

레지스터를 넓히는것 외에도 짝을지어서(pairing) 사용하는 방법도 있읍니다. 각 레지스터 쌍을 SIMD 벡터 인스트력션들을 위한 개별 피연산자(operand)로 사용하는

방식이죠. ARM NEON 이 이런식입니다. 레지스터들을 32 개의 64 비트 레지스터나 16 개의 128 비트 레지스터로 사용 할 수 있죠. 자연스럽게, 이런 레지스터내의 데이터는 꼭 8 비트단위 바이트가 아닌 다른 방식으로 나뉘어질 수도 있습니다. 예를 들어서, 고화질 이미지 프로세싱을 위한 16 비트 정수라던가, 과학계산용 부동소수 등으로 말이죠. 실제로, AltiVec 은 4 중 병렬 부동소수 곱셈-덧셈(4-way parallel floating-point multiply-add)연산을 완전 파이프라인 처리가 가능한 독립된 인스트럭션으로 실행 할 수 있었습니다.

이런류의 데이터 병렬화가 내재되어 있고, 쉽게 추출될수 있는 애플리케이션들에선, SIMD 벡터 인스럭션들이 놀라운 속도향상을 낼수 있습니다. 원래 이미지나 비디오 프로세싱 용 애플리케이션들이 주 대상이었지만, 오디오 프로세싱, 음성인식, 3D 렌더링 의 특정 부분, 그리고 여러 종류의 과학계산용 코드들이 대상에 포함될수 있습니다. 컴파일러나 데이터베이스 시스템 같은 부류의 애플리케이션들에선 큰 속도향상을 기대하긴 어렵습니다, 아마 거의 없을거라고 봅니다.

불행히도, 정말 쉬운 케이스를 제외하면 컴파일러가 벡터 인스트럭션들을 자동으로 사용하기에는 많은 어려움이 있습니다. 가장 큰 문제가 프로그래머들이 코드를 작성할때 모든걸 순차적(serialize)으로 만들어 놓는거죠. 컴파일러가 어떤 인스트럭션들이 독립적 이어서 병렬로 수행할수 있는것들인지 판단하기 어렵게 말입니다. 이 부분(컴파일러의 능력)에 있어서 아주 천천히 발전이 이루어 지고는 있지만, 기본적으로 현시점에서, 벡터 인스트럭션의 이점을 누리기 위해선 프로그램들이 일일히 재작성 되어야 합니다. 과학계산용 코드안에 배열 기반의 간단한 루프들은 제외하고 말이죠.

하지만, 다행히도, 여러분이 좋아하는 운영체제의 그래픽스나 비디오/오디오 라이브러리들의 주요 부분들을 부분적으로 재작성 하는것 만으로도 많은 애플리케이션 들에 커다란 영향을 미칠수 있습니다. 오늘날, 이미 대부분의 OS 들은 이런식으로 주요 라이브러리들을 향상시켜 놔서, 거의 모든 멀티미디어, 3D 그래픽스 애플리케이션들이 매우 효율적인 벡터 인스트럭션들을 사용하고 있습니다. 추상화(abstraction)의 또 다른 승리죠!

거의 모든 아키텍쳐들이 SIMD 벡터 익스텐션들을 이미 추가 했습니다. SPARC (VIS), x86 (MMX/SSE/AVX), PowerPC (AltiVec) 그리고 ARM (NEON) 이 대표적이죠. 하지만, 각 아키텍쳐 내에서도 오직 비교적 최신의 프로세서들만이 이 새로운 인스트럭션들을 부분적으로 실행 할수 있습니다. 이는 하위 호환성(backward-compatibility) 문제들을 야기하기도 했습니다. 특히나 SIMD 벡터 인스트럭션들이 약간 난잡하게 발전된 x86 아키텍쳐에서 말이죠 (MMX, 3DNow!, SSE, SSE2, SSE3, SSE4, AVX, AVX2).

# 메모리와 메모리 절벽(Memory & The Memory Wall)

앞서도 얘기했듯이, 정체현상/지연시간(latency)은 파이프라인을 사용하는 프로세서들에게 큰 문제입니다. 특히나 메모리에서 값을 읽어오는 과정(load)이 특히 문제죠. 전체 인스트럭션의 25% 정도가 관련 되있거든요.

로드 작업들은 코드 시퀀스(기본 블럭들)의 시작부분에 일어나는 경향이 강합니다. 로딩 되고 있는 데이터에 의존적인 다른 인스트럭션들과 함께 말이죠. 이는 그 인스트럭션들 모두를 지연시키는 결과를 초래하고, 효율적인 ILP 를 달성하기 어렵게 만들죠.

별거 아닌듯이 보이지만, 대부분의 수퍼스칼라 프로세서들이 싸이클당 한두개의 로드 인스트럭션들 만을 처리 할 수 있기 때문에, 실제론 훨씬 심각하답니다.

메모리 엑세스와 관련된 가장 핵심 문제는 빠른 메모리 시스템을 만들기가 어렵다는 사실이죠. 부분적으로는 빛의 속도처럼 고정된 제한요소(신호가 램 안팎으로 이동시에 지연됨)들 때문이기도 하지만, 보다 근본적으로는 메모리 셀들을 구성하는 소형 콘덴서(capacitors)들을 채우고 비우는 (charging and draining) 과정이 상대적으로 느리기 때문입니다. 이런 자연의 법칙들은 우리가 어떻게 바꿀수 있는게 아니죠. 비켜가는(work around)법을 배워야 합니다.

예를들어, [CAS(Column Access Strobe) latency] 가 11 인 SDRAM 을 사용하는 메인 메모리를 엑세스할때 발생하는 지연시간(latency)은 24 개의 메모리 시스템 버스 사이클이 걸리죠. 에드레스를 DIMM(메모리 모듈)로 보낼때 1 개, 해당 row 를 엑세스하는데 RAS-to-CAS delay of 11 개, column 엑세스에 CAS latency of 11 개, 그리고 마지막으로 첫 블럭의 데이터를 프로세서(또는 E-cache)에 전달하는데 1 개씩 말이죠. 남은 데이터 블럭들은 이후 몇 개의 버스 싸이클을 통해 읽어들여 지게 됩니다.

멀티 프로세서 시스템의 경우, 프로세서간 캐시 일관성(coherency)을 유지하기 위해 더 많은 버스 싸이클이 필요할 수도 있읍니다. 거기다가, 메모리 컨트롤러에 어드레스를 보내기전에 프로세서 내의 여러 온칩(on-chip) 캐시들을 체크하는데 필요한 프로세서 싸이클과 램에서 메모리 컨트롤러로 데이터가 도착해서 관련된 프로세서 코어로 보내질때 추가되는 부분도 생각해야죠. 다행히, 이 부분은 메모리 버스 싸이클이 아니라 보다 빠른 내부 CPU 싸이클이긴 합니다. 그렇다쳐도, 대부분의 모던 프로세서들에서 20 CPU 싸이클 정도는 걸리는 실정이죠.

가령 800 MHz SDRAM 메모리 시스템(DDR3-1600) 과 2.4 GHz 의 프로세서가 있다고 치면, 메인 메모리를 엑세스하는데 필요한 비용은 자그만치 92 CPU 클럭 싸이클, (1+11+11+1) * 2400/800 + 20, 이나 됩니다! 정말 비싸죠! 더 높은 클럭의 프로세서들에선 이 비용이 더욱 커지게 됩니다. 2.8 GHz 프로세서는 104 싸이클, 3.2 GHz 는 116 싸이클, 3.6 GHz 프로세서는 128 cycles, 그리고 4.0 GHz 짜리는 140

싸이클이라는 어마어마한 시간을 메인 메모리 엑세스를 기다리는데 허비하게 되는 겁니다!

여기서 짚고 넘어가야될 것이, DDR SDRAM 메모리는 클럭 시그널의 상향엣지와 하향엣지 모두에서 데이터를 전송(ie: at *double data rate*)할 수 있지만, 메모리 시스템 버스의 실제 클럭 스피드는 절반 밖에 되지 않는다는 점입니다. 그리고, 버스 클럭 스피드가 컨트롤 신호에 적용되는 실제 속도랍니다. 따라서, DDR 메모리 시스템의 지연시간(latency)은 non-DDR 시스템과 차이가 없읍니다. 대역폭은 두배지만 말이죠 (대역폭과 정체도의 차이에 대해 나중에 설명함).

이전세대 프로세서들은 상황이 더 나빴었읍니다. 메모리 컨트롤러가 온칩(on-chip) 형태가 아니라, 메인보드 칩셋의 일부였기 때문이죠. 프로세서와 메인보드 칩셋간에 주소나 데이터 전송을 위해 버스 싸이클 2 개가 추가로 필요 했었던 데다가, 메모리 버스들이 200 MHz 보다 느리게 동작했었죠. 결과적으로, 추가된 2 버스 싸이클은 총 20+ CPU 싸이클을 더 필요로 하게 됐었던거죠. 몇몇 프로세서들은 칩셋과 프로세서 사이의 프론트 사이드 버스(FSB) 스피드를 올리는 방식으로 이 문제를 최소화 하려 했었읍니다(800 MHz QDR in Pentium 4, 1.25 GHz DDR in PowerPC G5).

모든 모던 프로세서들에 사용되는 훨씬 나은 방식은, 메모리 컨트롤러를 프로세서 칩안에 통합해서 저 2 버스 싸이클을 훨씬 빠른 CPU 싸이클로 변환 되도록 하였읍니다. UltraSPARC IIi 와 Opteron/Athlon 64 가 이 방식을 처음 도입한 메인스트림 프로세서들 이었죠. 인텔은 한발 늦게 Core i 씨리즈에 와서야 이렇게 바꿨구요.

불행하게도, DDR SDRAM 과 온칩 메모리 컨트롤러 모두 메모리 latency 를 향상 시키는데는 한계가 있었기에, 이 문제는 여전히 중요한 숙제로 남아있읍니다. 프로세서와 메인 메모리간 속도차가 크게 벌어지는 이 문제는 메모리 절벽(*memory wall*) 이라고 불리웁니다. 전력절벽(*power wall*) 때문에 프로세서의 클럭스피드가 예전처럼 빠르게 증가하지 못하기 때문에, 상황이 약간 개선 되고는 있지만, 메모리 절벽을 여전히 커다란 문제입니다.

# 캐시와 메모리 계층(Caches & The Memory Hierarchy)

역주: locality 라는 단어가 이제 나오기 시작하는데, 재 사용성등의 의미로 사용됩니다. 지역성 같은 말로 표현하기는 좀 어색한듯 해서, 그냥 내버려뒀음.

모던 프로세서들은 메모리 절벽 문제를 캐시들을 사용해 해결 중입니다. 캐시(cache)란 프로세서 칩안이나 혹은 매우 가까운 거리에 위치한 작지만 빠른 형태의 메모리를 말합니다. 메인 메모리의 작은 부분들의 복사본을 저장하는 역할을 하죠. 프로세서가 메인메모리의 특정 영역을 요청하면, 해당 데이터가 캐시내에 존재할 경우, 메인 메모리보다 빠르게 요청을 처리 할수 있게되는 겁니다.

일반적으로, 8-64k 정도의 작지만 빠른 Level-1(L1) 캐시가 프로세서 내의 각 코어안에 위치하고 있읍니다. 몇백 KB 에서 몇 MB 정도 되는 조금 더 큰 level-2 (L2) 캐시가 약간 떨어져서 칩안에 위치 하고있고요, 그리고 더 크고 느린 L3 캐시를 채용하기도 하죠.

온칩 캐시들과, 칩 바깥의 캐시(E-cache), 그리고 메인 메모리(RAM) 모두가 메모리 계층을 형성 하고 있지요. 각 단계가 그 전 단계보다 크고 느린 형태입니다. 이 메모리 계층의 최하위에는 물론 가상메모리 시스템 (paging/swapping)이 위치하고 있지요. RAM 의 페이지들을 엄청나게 느린 파일 시스템에 연계하여 무제한의 메인 메모리가 존재하는 것같은 착각을 일으키게 하는데 쓰입니다. 마치 도서관 책상에서 책안의 뭔가를 찾아 볼때와 비슷한데요. 두 세권의 책이 펼쳐져있고, 그안에서 뭔가를 찾는건 빠르죠. 하지만, 필요한 책 모두를 그렇게 한 책상에 펼쳐 놓을 수는 없는 노릇이죠. 설령 가능하다고 해도, 수백권 사이를 옮겨 다니는게 더 많은 시간이 걸릴 겁니다.

대신, 책상한켠에 여러 책들을 쌓아놓고서 필요할때만 찾아 보는거죠. 물론 한권씩 꺼내고 열어 보는데 시간이 걸릴 테지만 말입니다. 또한, 새로운 책을 펼쳐 볼때마다, 먼저 펴놨던 책들을 정리 해서 공간을 만들어야 할 수도 있겠구요. 결국에는, 현재 책상에 없는 책이 필요할때도 생길 텐데요. 이렇게 되면, 책상에서 일어나서 도서관 책장사이를 돌아다니면서 찾아야 될테니 정말 정말 느리겠죠.  하지만, 도서관이 크다는건 선택할수 있는 책이 엄청 많다는걸 의미하죠, 책상 하나에 다 못 올릴 만큼요.

메모리 계층은 일반적으로 다음과 같은 형태입니다.

| *Level* | *Size* | *Latency* | *Physical Location* |
|---|---|---|---|
| L1 cache | 32 KB | 4 cycles | inside each core |
| L2 cache | 256 KB | 11 cycles | beside each core |
| L3 cache | 6 MB | ~21 cycles | shared between all cores |
| L4 E-cache | 128 MB | ~58 cycles | separate eDRAM chip |
| RAM | 4+ GB | ~117 cycles | SDRAM DIMMs on motherboard |
| Swap | 100+ GB | 10,000+ cycles | hard disk or SSD |

**Table 4 – The memory hierarchy of a modern desktop/laptop: Core i*4 Haswell.**

요세는 전화기들도 이런식이죠.

| *Level* | *Size* | *Latency* | *Physical Location* |
| --- | --- | --- | --- |
| L1 cache | 64 KB | 4 cycles | inside each core |
| L2 cache | 1 MB | ~20 cycles | beside the 2 cores |
| L3 cache | 4 MB | ~107 cycles | beside the memory controller |
| RAM | 1 GB | ~261 cycles | separate SDRAM chip |
| Swap | N/A | N/A | paging/swapping not used on iOS |

**Table 5 – The memory hierarchy of a modern phone: Apple A8 in the iPhone 6.**

캐시들에 관한 놀라운점은 이들이 정말 정말 잘 동작 한다는 겁니다. 메인 메모리를 L1 캐시만큼 빠르게 보이도록 해주죠, 크기는 그대로인데 말이에요.

모던 L1 캐시는 2~4 개 정도의 프로세서 싸이클 지연시간(latency)을 보입니다. 메인 메모리를 엑세스 하는 것보다 월등히 빠르죠. 게다가, 대부분의 애플리케이션들에서 약 90% 정도의 히트율(hit rate)을 달성 하고 있읍니다. 따라서, 메모리 엑세스할때 열에 아홉은 단지 몇 싸이클밖에 안걸리게 되죠!

캐시들이 이렇게 놀라운 히트율을 낼수 있는건 프로그램들이 동작하는 방식 때문 입니다. 대부분의 프로그램들이 시간과 공간의 *locality* 를 보이는데요. 메모리의 일부를 엑세스 할때, 그 부분을 금방 다시 접근할 가능성(temporal locality)이 높고, 그 근처의 메모리를 이용해야 가능성(spatial locality)또한 상당하게 됩니다.

Temporal locality 의 이점을 살리려면 최근에 사용된 데이터를 캐시안에 보관하고 있기만 하면되죠. Spatial locality 의 효과를 보려면, 메인 메모리에서 캐시로 데이터를 전송할때 몇십 바이트의 블럭단위로 처리하면 됩니다. 이 단위를 캐시라인 (*cache line*) 이라고 부르죠.

하드웨어 측면에서 보면, 캐시는 두개의 컬럼으로 구성된 테이블처럼 동작 합니다. 한 컬럼은 메모리 주소를 다른 하나는 데이데 값들의 블럭을 저장하는 거죠 (각 캐시 라인은 개별 값이 아니라, 여러 데이터로 이루어진 블럭임을 기억하세요). 물론, 실제로는, 캐시에 메모리 주소중 필요한 상위부분만 저장하면 충분합니다. 하위 부분은 캐시 테이블을 인덱스 하는데 사용되니까요. 태그(tag)라고 불리우는 그 상위부분이 테이블에 저장된 값과 동일하면, 히트(*hit*) 로 인정되어 적절한 데이터 조합이 프로세서 코어로 보내지게 됩니다.

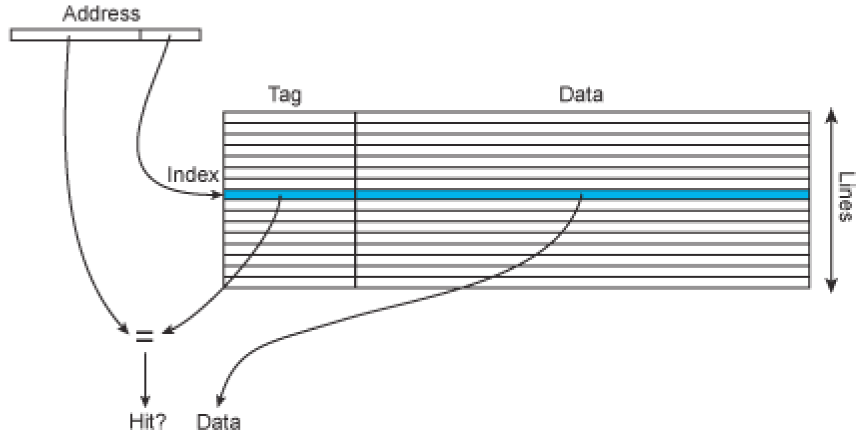


Figure 19 – A cache lookup.

물리주소나 가상주소 모두 캐시룩업(cache lookup)에 사용될 수 있읍니다. 각기 장단점이 있지요(컴퓨팅의 모든 타 분야들처럼요). 가상주소를 사용하면 문제가 될 수도 있읍니다. 왜냐면, 서로 다른 프로그램이 동일한 가상 주소를 사용해 다른 물리 주소들을 지정 할 수도 있으니까요. 그렇게 되면, 컨택스트 스위칭이 발생할때 마다 캐시를 비워야 될 수도 있읍니다. 반면, 물리주소를 사용하면, 가상에서 물리 주소로의 맵핑이 캐시룩업의 일부분으로 반드시 실행되어야 합니다. 이는 모든 캐시룩업을 느리게 만들죠.

따라서, 현재 널리 사용되는 트릭은 가상주소를 인덱싱에 사용하고 물리주소를 태그로 저장하는 방식입니다. 이렇게 하면, 가상주소를 물리주소로 매핑( TLB lookup) 하는 작업이 캐시 인덱싱과 병렬로 수행될수 있기 때문에, 태그를 비교해야 할때쯤에는 사용 가능한 상태가 됩니다. 이런 방식을 *virtually-indexed physically-tagged* 캐시라고 부릅니다.

모던 프로세서 내의 다양한 캐시들의 크기와 속도는 성능에 절대적으로 중요한데요. 이중에서도 으뜸으로 중요한것이 primary L1 data cache (D-cache) 와 L1 instruction cache (I-cache) 입니다. 어떤 프로세서들은 소형 L1 캐시들을 탑재 했었읍니다. Pentium 4E Prescott, Scorpion 그리고 Krait 은 16k L1 캐시들을(각개의 I- 와 D-cache) 장착했었고, 초기의 Pentium 4 와 UltraSPARC T1/T2/T3 들은 보다 작은 8k 짜리를 사용 했었읍니다. 대부분 32k 를 적정선(sweet spot)으로 삼게 됐지만, 몇몇은 보다 큰 64k (Athlon, Athlon 64/Phenom, UltraSPARC III/IV, Apple A7/A8) 혹은 때때로 128k (the I-cache of Denver, with a 64k D-cache)까지 다양하게 사용되고 있읍니다.

모던 L1 데이터 캐쉬들의 로드 지연시간(load latency)은 프로세서의 제네럴 클럭 스피드에 따라 대체로 3~4 싸이클 정도이지만, 가끔씩 짧아지기도 합니다(UltraSPARC III/IV 은 클럭리스 “웨이브” 파이프라인덕에 2 싸이클).

로드 지연시간이 한 싸이클정도 늘어나는게(3->4 또는 4->5), 별거 아닌거 같아 보이지만, 실제 성능에 심각한 영향을 끼칩니다. 이는 대다수 일반 사용자들이 이해하지 못하는 부분 인데요. 일상적으로 사용되는 포인터 체이싱(pointer chasing) 코드들에 있어서, 프로세서의 로드지연은 체감 성능을 좌우하는 중요한 요소입니다.

대부분의 모던 프로세서들은 두세 단계(level)의 커다란 온칩 캐시를 채용하고 있습니다. 이를 모든 코어들이 공유하는게 일반적 인데요. 이 캐시 역시 매우 중요 합니다만, 최적의 크기는 실행중인 애플리캐이션의 종류와 액티브 워킹 세트(active *working set)*의 크기에 좌우됩니다. 어떤 애플리케이션 들에게는 L3 캐시의 크기가 2MB 이던 8MB 이던 별 차이가 없을수도 있지만, 다른 것들에겐 매우 큰 차이일 수도 있죠. 여러 모던 프로세서 코어들에서 상대적으로 작은 L1 캐시들이 상당한 크기의 칩공간을 차지하는걸 감안하면, 커다란 L2/3 캐시들은 얼마나 많은 공간을 차지할지는 안봐도 비디오지요. 하지만, 이들은 메모리 절벽과 대적하기 위해선 절대적으로 필요한 것들입니다.

종종, 대형 L2/L3 캐시가 전체 칩공간의 절반 정도를 차지하기도 합니다. 너무 커서 칩사진에 확연히 보여질 정도죠. 코어들과 메모리 컨트롤러를 구성하는 좀 지저분해 보이는 로직 트랜지트터들에 비하면 상대적으로 깔끔하고 반복적인 구조가 눈에 띄입니다.

## 캐시 충돌과 연계성(Cache Conflicts & Associativity)

역주: Associativity/Associative 라는 단어가 선택할 수있는 옵션이 여러개 모여있다는 의미로 사용되었읍니다. 이걸 결합또는 연계라고 번역을 하긴했는데 어색함. 그렇다고 냅두기는 좀 그렇고해서 섞어서 사용하다가 포기함 T.T.

이상적으로는, 미래에 사용될 가능성이 가장 높은 데이터가 캐시에 보관 되는것이 좋겠죠. 하지만, 캐시들이 점쟁이는 아니기 때문에, 가장 최근에 사용된 데이터를 보관하는 것이 그나마 이상적일 가능성이 약간이나마 있겠죠.

불행하게도, 가장 최근에 사용된 데이터를  그 상태 그대로 저장하면, 메모리상의 한 지점에서 읽어온 데이터가 캐시라인상의 어떤 한 지점에 (무작위로) 위치하게 됩니다. 그럼, 캐시는 n KB 정도의 최근 데이터를 원래 모양그대로 유지하게 되서 locality 측면에서는 좋을지 모르지만 엑세스 속도는 느려지게 됩니다. 모든 캐시라인들을 일일히 검사해 봐야되기 때문이죠. 이런식이면, 수백개의 라인을 가지고 있는 모던 캐시들은 매우 느려지게 될것입니다.

따라서 이렇게 하는대신, 메모리의 특정 주소에서온 데이터가 하나 또는 오직 소수의 캐시라인 만을 사용하도록 제한합니다. 그러면, 검사해 봐야할 범위가 줄어들어서, 캐시 엑세스 속도 를 빠르게 유지 할 수 있읍니다 (애초에 캐시를 쓰는 이유이기도 하죠;). 하지만, 이 방식도 단점이 있읍니다. 캐시안에 최근 사용된 데이터의 최적 세트 (absolutely best set)가 저장되지 않는다는 점입니다. 왜냐면, 메모리내의 여러 다른

지점들이 모두 캐시상의 같은 지점에 매핑될 것 이기 때문입니다. 이렇게 캐시라인을 공유 하게될 두개의 메모리 위치가 동시에 요청되는 시나리오를 캐시충돌(*cache conflict*) 이라고 부릅니다.

캐시충돌은 "병적인(pathological)" worst-case 성능 문제들을 발생 시킬 수도 있읍니다. 하나의 프로그램에서 같은 캐시라인에 매핑될 두개의 메모리 위치들을 반복해서 엑세스 하게 되면, 캐시는 계속해서 메인 메모리에서 데이터를 읽어와서 저장 해야하고, 이때마다 매번 긴 메인 메모리 정체(main-memory latency)를 격을 수 밖에없죠(100 cycles or more, remember!). 이런 상황을 *thrashing* 이라 부르며, 캐시는 아무 도움이 되지 못하고 오히려 짐만 되는 것 입니다. 명백하게 Temporal locality 와 데이터 재사용이 가능함에도 불구하고, 메모리 위치와 캐시라인 사이의 간단한(simplistic) 맵핑에서 오는 제약 때문에 이런식의 엑세스 패턴에서 발생되는 locality 이점을 살릴수 없게 되는것입니다.

이 문제를 해결하기 위해, 보다 발전된 캐시들은 한개의 캐시라인에만 매핑하지 않고 소수의 다른 위치들에 데이터를 저장 할수 있게 되었읍니다. 한 조각의 데이터가 저장될 수 있는 캐시내의 위치들의 수를 그 캐시의 연계성(*associativity*) 이라고 부릅니다. "연계성(associativity)" 이라는 말은 캐시 룩업이 연계(association)에 의해 작동 하기 때문에 붙혀진 이름입니다. 메모리의 특정 주소가 캐시의 특정 위치에 연계되는 방식이죠 (혹은 세트대 세트로).

앞서 설명했듯이, 가장 간단하고 빠른 캐시는 메모리상의 각 주소가 캐시안의 오직 한 곳으로만 매핑되도록 합니다. 캐시 내에서 각각의 데이터가 메모리 주소의 하위 비트들만 사용해 간단하게 *address % size* 형식으로 매핑되는 것이죠(위에 그림에서 처럼). 이런 종류들을 *direct-mapped* 캐시라고 부릅니다. 주소값의 하위비트가 동일한 두 메모리 위치가, 동일한 캐시라인으로 매핑 되어 캐시충돌을 일으킬 수 있읍니다.

데이터의 주소값에 따라 사용가능한 2 개의 위치중 하나에 데이터를 저장하는 방식의 캐시를 2-way *set-associative* 캐시라고 부릅니다. 비슷하게, 4-way set-associative 캐시는 한 데이터에 4 개의 저장 가능한 위치를 제공하고, 8-way 캐시는 8 개의 가용 위치를 사용 합니다. Set-associative 캐시들은 direct-mapped 캐시들과 매우 비슷 하게 동작합니다. 단지, 여러개의 태이블이 존재하고, 모두 병렬로 인덱싱 될수있으며, 각 태이블의 태그가 원하는 값인지 대조되는게 다른 점이죠.

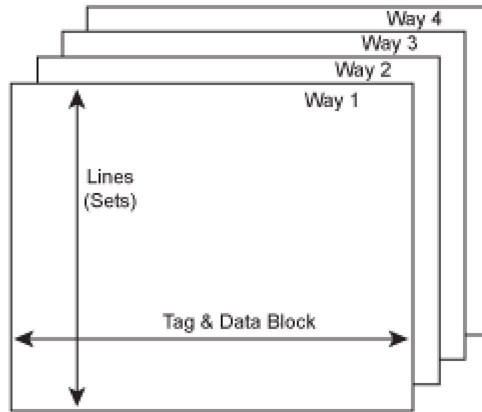


Figure 20 – A 4-way set-associative cache.

또한, 각 테이블(or *way)*은 마커(marker) 비트들을 이용해서 사용된지 가장 오래된 캐시라인을 표시해두고, 새 라인이 읽혔을때나 보다 빠른 동작을 위해 지워버립니다.

대체로, set-associative 캐시들은 캐시충돌 문제를 피할수 있읍니다. 테이블을 더 추가하는 만큼 더 많은 충돌을 피할 수 있게 되지요. 하지만, 캐시가 고도로 연계화 될수록, 엑세스 속도가 느려지게 됩니다. 태그 비교작업은 병렬로 이루어지지만, 적절한 히트(맞는게 있을경우)를 선택하는데 추가적인 로직이 필요하고, 각 테이블내의 마커 비트들을 업데이트 해줘야 하기 때문입니다. 또한, 필요한 칩 공간도 늘어나게 되는데, 데이터 블럭보다 태그정보들이 상대적으로 많이 차지하고, 각 테이블을 병렬로 엑세스 하기위해 데이터 패스(datapaths)들이 추가되야 하기 때문입니다. 이 요소들 모두 혹은 그 어떤것도 엑세스 타임에 부정적인 영향을 끼칠 수 있읍니다. 따라서, 2-way set-associative 캐시가 direct-mapped 캐시보다 똑똑하지만 느리고, 4-way 와 8-way 역시 비슷하게 더 똑똑 하지만 더 느려지게 됩니다.

대부분의 모던 프로세서들에서, 인스트럭션 캐시는 highly set-associative 될 수 있읍니다. 파이프라인내의 다른 더느린 작업들에 의해 캐시의 latency 가 약간 묻히기 때문입니다. 반면, 데이터 캐시는, 보통 어느정도 까진 set-associative 되어 있지만, 언제나 골치거리인 로드 지연시간(load latency) 을 최소화 하기위해 너무 고도화 시키지는 않읍니다.

대부분의 프로세서들이 4-way set-associative 를 최적점(sweet spot)으로 보고 채택하였지만, 몇몇 제품은 더 낮은 연계화를 사용하기도 하고(2-way in Athlon, Athlon 64/Phenom, PowerPC G5, 그리고 Cortex-A15/A57), 또 다른 제품들은 오히려 더 높은것 들을 사용하죠(8-way in PowerPC G4e, Pentium M 과 Core 후속들).

매우 멀리있는 메인메모리로 데이터를 가지러 떠나기전 마지막 방편인, 커다란 L2/L3 캐시(때에따라 LLC - "last-level cache"라고 불림) 역시 보통 고도로 연계화되어 있읍니다. 12- 또는 16-way 까지도요. 하지만, 외부의 E-cache 는 크기및 구현상의 유연성 때문에 가끔씩 direct-mapped 캐시를 사용합니다.

캐시 컨셉은, 또한, 소프트웨어 시스템까지 확장 되었읍니다, 예를들면, 운영체제는 메인메모리를 파일시스템의 내용을 캐시하는데 사용하여 파일 I/O 작업의 속도를 올립니다. 웹 브라우져들은 최근에 방문한 페이지들을 캐시해 놓고, 다시 방문 할때 속도 향상을 위해 사용하죠. 그리고, 웹 캐시 서버들은(proxy caches) 원격 서버상의 내용들을 로컬 서버에 캐시하죠 (대부분의 ISP 가 이거를 반드시 씁니다).

메인 메모리와 가상 메모리(paging/swapping)에 관련해서는, 가상 메모리를 똑똑하고 완전 연계화(fully associative cache)된 캐시로 생각하면 됩니다, 이 챕터의 도입부에 얘기했던 이상적인 캐시처럼 말이죠. 가상 메모리 시스템은 운영체제 커널처럼 인텔리젼트한 소프트웨어에 의해 관리 됩니다.

# 메모리 대역폭 vs 지연시간(Memory Bandwidth vs Latency)

역주: show stopper 는 말그대로 쇼를 보여줘야대는데 뭔가 중요한게 아직 안돼있다는 의미인데, 우리말로 뭐라할지 몰라서 내비둠. Effective latency 는 스펙상에 나와있는 거 말고 실제 체감되는거 라는 의미가 있음, 상황에 따라 변할수 있는 그런거.

메모리는 블럭단위로 전송되고, 캐시미스는 프로세서를 작업도중 멈추게 할 수도 있는 긴급한 “show stopper” 이벤트 이기때문에, 메모리에서 블럭단위로 데이터를 가져오는 속도는 매우 중요합니다. 메모리 시스템의 데이터 블럭 전송율을 대역폭(*bandwidth*) 이 라고 부릅니다. 그럼 지연시간/정체율(*latency)* 와는 어떻게 다를까요?

고속 도로에 비교해 보면 좋겠네요. 100 마일 떨어져 있는 도시로 운전을 한다고 가정해 보죠. 차선을 두배로 늘려서, 시간당 지나갈 수 있는 통행량(bandwidth)은 두배로 늘릴수 있겠지만, 여러분의 여행시간(the latency)을 줄일 수는 없죠. 시간당 지나 갈 수 있는 차들의 수를 늘리는게 목적이면, 더 많은 차선(wider bus)을 추가 하는게 답 이겠지만, A 지점에서 B 지점으로 한대의 차량이 이동 하는데 걸리는 시간을 줄이려면, 뭔가 다른걸 해야 되겠죠. 제한속도(bus 와 RAM 속도)를 높힌다던가, 거리를 줄인다던가, 혹은 가까운 곳에 쇼핑몰을 지어서 사람들이 먼곳의 도시로 자주 갈 필요 없게(cache) 만들던가 말이죠.

메모리 시스템에는, 종종 대역폭과 지연시간 사이에 미묘한 tradeoffs 가 존재 합니다. 낮은 지연시간 디자인은 컴파일러나 데이터 베이스같은 포인터 연산이 많은(pointer chasing) 코드들에 더욱 적합할 것이고, 대역폭 중심의 디자인은 이미지 프로세싱이나 과학계산처럼 간단하고 선형 엑세스 패턴을 가진 프로그램들에 우위를 가지게 됩니다.

물론, 대역폭을 늘리는게 상당히 쉽죠. 간단하게, 더 많은 메모리 뱅크를 추가하고 버스들을 넓히면 대역폭을 두배 ~ 네배까지 쉽게 늘릴 수 있읍니다. 실제로, 여러

하이엔드 시스템들이 성능향상을 위해 이렇게 하고 있습니다만, 단점이 없진 않습니다. 특히나, 넓어진 버스들은 보다 비싼 마더보드와 RAM 을 늘리는 방식에서의 제한 (짝으로 또는 네개의 그룹으로 설치), 그리고 보다 높은 최소 RAM 구성(용량)을 의미합니다.

불행하게도, 지연시간(*latency)은* 대역폭 보다 향상 시키기나 훨씬 어렵습니다. "신을 매수 할수는 업다(*you can't bribe god)*"는 말 처럼요. 그래도, 요 근래 들어, 실제 지연 시간(effective memory latency) 부분에서 제법 쓸만한 발전이 이루어져 왔는데요. 메모리 버스와 동일 클럭으로 작동하는 synchronously clocked DRAM (SDRAM) 을 사용해서 말이죠. SDRAM 의 특장점은 메모리 시스템의 파이프라인화(*pipelining of the memory system)*를 가능하게 했다는 점 이었죠.

SDRAM 칩 오퍼레이션의 내부 타이밍 특성(internal timing aspects)과 interleaved 구조가 시스템에 노출됨으로써, 그 것들을 잘 활용할 수 있게 되었습니다. 현재 진행중인 메모리 엑세스 작업이 끝나기 전에 새로운 작업을 시작할 수 있도록 해서 effective latency 를 줄여 주었습니다. 이런식으로 예전의 비동기 DRAM 시스템에서 나타나던 대기시간을 약간 이나마 없앨 수 있었습니다 (평균적으로, 비동기 메모리 시스템은 새로운 작업을 시작하기 전에 그 전 엑세스에서 캐시라인을 반정도 채울때까지 기다려야 했고, 이는 매우 느린 여러개의 버스 사이클을 필요로 했었음).

실제 지연시간(effective latency)을 줄인것에 더해서, 대역폭 또한 상당히 늘어 났습니다. SDRAM 메모리 시스템상에선 여러개의 메모리 요청이 매우 효율적으로, 완전히 파이프라인화된 형식으로 동시에 처리될 수 있었기 때문입니다. 메모리 시스템의 파이프라이닝은 대역폭에 드라마틱한 영향을 미쳤읍니다. 비동기 메모리 시스템과 동일한 메모리 셀 기술을 사용한  SDRAM 시스템의 지연시간은 아주 약간 줄었지만, 일반적으로 동시대의 비동기 메모리 시스템의 두배에서 세배정도의 대역폭을 제공 할 수 있었읍니다.

메모리 기술의 발전이 보다 고도화된 캐싱과 더불어, 점점 더 늘어나는 프로세서 코어들에 요구되는 더 높은 대역폭을 지원 하면서 동시에 메모리 절벽을 이겨 낼수 있을까요?아니면, 프로세서 아키텍쳐나 코어의 수가 더이상의 큰 차이를 보이지 않게되고, 메모리 시스템만이 뜨거운 감자가 되어, 우리는 곧 메모리(대역폭과 지연시간 모두) 때문에 끊임없이 제약받게 될까요?? 어떻게 될지 지켜보면 흥미로울 겁니다. 그리고, 미래를 예측하는건 절대 쉽지않지만, 긍정적이어도 좋을 이유들이 많읍니다.

## Acknowledgments

The overall style of this article, particularly with respect to the style of the processor "instruction flow" and microarchitecture diagrams, is derived from a combination of a well-known 1989 ASPLOS paper by Norman Jouppi and David Wall, the book POWER & PowerPC by Shlomo Weiss and James Smith, and the

two very famous Hennessy/Patterson textbooks Computer Architecture: A Quantitative Approach and Computer Organization and Design.

There have, of course, been many other presentations of this same material, and naturally they are all somewhat similar, however the above four are exceptionally good (in my opinion). To learn more about these topics, those books are an excellent place to start.

# More Information?

If you want more detail on the specifics of recent processor designs, and something more insightful than the raw technical manuals, here are a few good articles...

- Intel's Haswell CPU Microarchitecture – the latest and greatest Intel x86 processor design, Core i*4 "Haswell", largely based on the previous "Sandy Bridge" design.
- Intel's Sandy Bridge Microarchitecture – the previous Intel x86 processor design, Core i*2 "Sandy Bridge", representing a blending of the Pentium Pro and Pentium 4 design styles.
- AMD's Bulldozer Microarchitecture – the novel resource-sharing approach used in AMD's latest processor designs, blurring the lines between SMT and multi-core.
- Intel's Next Generation Microarchitecture Unveiled – Intel's revival of the venerable P6 core from the Pentium Pro/II/III/M to produce the Core microarchitecture.
- The Pentium 4 and the PowerPC G4e (and Part II) – a comparison of the very different designs of two extremely popular and successful, if somewhat maligned, processors.
- Into the K7 (and Part II) – the AMD Athlon, the only competitor to ever really challenge Intel's dominance in the world of x86 processors.
- The AMD Opteron Microprocessor (video) – a 1-hour presentation covering both the Opteron/Athlon 64 processor and AMD's 64-bit extensions to the x86 architecture.
- Niagara II: The Hydra Returns – Sun's innovative UltraSPARC T Niagara processor, revised for a second generation and taking thread-level parallelism to the extreme.
- Crusoe Explored – the Transmeta Crusoe processor and its software-based approach to x86 compatibility.

And here are some articles not specifically related to any particular processor, but still very interesting...

- Designing an Alpha Microprocessor – a fascinating look at what really goes on in the various stages of a project to make a new processor.
- Things CPU Architects Need To Think About (video) – an interesting 80-minute presentation given by Bob Colwell, one of the principle architects of the Pentium Pro/II/III.

And if you want to keep up with the latest news in the world of microprocessors...

- Ars Technica
- AnandTech
- Microprocessor Report
- Real World Tech

That should keep you busy!