Processor System Comparison

22311947 김서현
1140 컴퓨터구조

Ⅰ. 서론

프로세서는 컴퓨터의 두뇌라는 말을 자주 들어보았다. 그러나 단 한 번도 그 의미가 선뜻 와닿았던 적은 없었다. 하드웨어를 잘 모르는 내게 컴퓨터의 구조는 너무도 먼 세계의 이야기였다.

그러다 지난 학기 운영체제와 이번 학기 컴퓨터구조 수업을 들으며, 내겐 먼 이야기라고만 생각했던 개발자 관점의 하드웨어에 대해 배우게 되었다. 특히 이번 학기 ISA와 프로세서의 구조를 공부하며 '프로세서는 컴퓨터의 두뇌'라는 말의 의미를 어렴풋이나마 깨닫기 시작했다.

우리가 사용하는 컴퓨터는 1과 0의 신호를 처리하는 아주 작은 논리회로의 조합에서부터 출발한다. 이 기본적인 원리가 소규모 임베디드 시스템에서부터 대형 슈퍼컴퓨터까지 모든 종류의 컴퓨터를 구현해 낼 수 있다는 사실을 알게 되자, 소프트웨어 개발자 역시 하드웨어에 대한 깊이 있는 이해가 필요하다는 확신이 들었다.

해당 보고서는 이러한 흥미를 바탕으로 수업에서 배운 아키텍처 지식을 활용해 서로 다른 프로세서들을 이해하고 그 공통점과 차이점을 비교 분석하는 것을 목적으로 한다.

이를 위해 소규모 임베디드, 개인용 컴퓨터, 대형 슈퍼컴퓨터 시스템에서 사용되는 대표적인 프로세서를 하나씩 선정하였다. 프로세서 선정 기준은 각 분야에서 대표성을 가질 정도로 유명하며 아키텍처 및 파이프라인에 대한 기술 자료가 오픈되어 있고 그러면서도 비교적 최신에 개발되었는지이다.

보고서의 본론에서는 선정된 각 프로세서의 특징을 설명하고 MIPS 아키텍처와 비교하여 ISA 및 파이프라이닝을 분석한다. 최종적으로는 이를 통해 각 프로세서가 현재의 방식으로 발전되어 온 배경을 이해하며, 그 구조적 특징이 갖는 장단점을 고찰하는 것을 최종 목적으로 한다.

Ⅱ. 배경지식

1. 프로세서와 ISA

컴퓨터의 구조는 크게 ISA(Instruction Set Architecture)와 이를 실제로 구현한 마이크로아키텍처(Microarchitecture)로 나누어진다. 이 중 소프트웨어 개발자가 직접적으로 알아야 하는 것은 ISA이다.

ISA는 명령어 집합 구조로 소프트웨어와 하드웨어 사이의 규약이다. ISA는 프로그래머(혹은 컴파일러)의 관점에서 볼 수 있는 명령어 집합을 의미하며 명령어의 정의뿐 아니라 피연산자를 저장하는 레지스터의 정보와 메모리 접근 방식 등을 포함한다.

ISA를 설계하는 두 가지 대표적인 철학은 CISC와 RISC로 나뉜다. CISC(Complex Instruction Set Computer)는 복잡한 명령어 집합을 가진다. 하나의 명령어가 메모리 접근, 연산, 저장 등의 여러 작업을 한꺼번에 처리하는 것이다.

명령어 하나가 많은 작업을 처리하므로 과거 컴파일러 기술이 부족하던 시절 프로그래머의 부담을 줄여주었다. 또한 수백 가지가 넘는 명령어를 제공하므로 같은 프로그램을 컴파일하더라도 CISC로 작성한 파일이 더 적은 용량을 가진다.

그러나 명령어마다의 길이가 달라 명령어를 해독하는 디코드 과정이 복잡하다. 각 명령어마다 요구되는 클락 수가 달라 파이프라이닝이 힘들다는 단점 역시 존재한다. 즉, CISC는 프로그램의 실행에 필요한 총 명령어의 개수인 IC는 줄어들지만 명령어당 평균 클락 사이클 횟수인 CPI는 높아진다는 특징을 가진다.

RISC(Reduced Instruction Set Computer)는 간단한 명령어 집합을 가진다. 하나의 명령어는 메모리 접근, 연산, 저장과 같은 가장 작은 단위의 일만을 수행한다. 모든 명령어가 같은 길이를 가지므로 디코드가 간단하며 명령어에서 요구하는 클락 횟수가 동일해 파이프라이닝에 용이하다.

다만 단순한 명령어들을 조합해 복잡한 작업을 구현하여야 하므로 동일한 CISC 코드에 비해 실행 파일의 크기가 늘어나며, 더 많은 레지스터를 요구하고, 더 높은 역량의 컴파일러를 더 필요로 한다는 단점 역시 존재한다. 즉, RISC는 IC는 높아지지만 CPI는 낮아지는 특징을 가진다.

해당 보고서에서 비교의 기준으로 사용될 MIPS의 경우 RISC 구조를 가진 프로세서이다. MIPS의 명령어 집합은 크게 계산 명령어, 메모리 접근 명령어(Load/Store), 분기 명령어(Branch/Jump)로 나뉜다. 특히 실행 시간이 가장 오래 걸리는 메모리 접근 명령어를 Load/Store라는 별개의 명령어로 분리함으로써 common case의 성능을 크게 향상시켰다.

MIPS는 R-type, I-type, J-type의 3가지 명령어 포멧을 가지며 모든 명령어가 32비트의 길이에, 가장 앞 6비트가 Opcode라는 점에서 동일하다. 또한 컴파일러가 사용할 수 있는 32개의 범용 레지스터를 제공한다.

❑ 3 Instruction Formats: all 32 bits wide

OP	rs	rt	rd	sa	funct

OP	rs	rt	immediate

OP	jump target

이처럼 MIPS의 ISA가 단순하고 규격화된 모습을 고집하는 이유는 프로세서의 성능을 끌어올리는 파이프라인 기법을 가장 효율적으로 구현하기 위해서이다.

2. 파이프라이닝

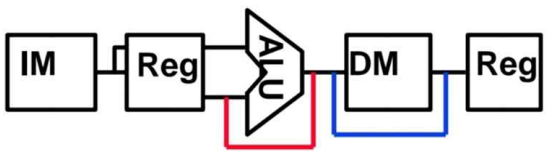
파이프라이닝은 현대 프로세서의 성능을 끌어올리는 가장 중요한 기법 중 하나이다. 프로세서의 총 실행 시간은 IC × CPI × CC(clock cycle time)로 계산되며 파이프라이닝을 사용할 시 명령어 처리를 여러 단계로 분리해 CC를 낮출 수 있다.

이때 중요한 점은 파이프라이닝 기술이 명령어 자체의 실행 시간(Latency)을 낮추는 기술이 아니며 시간 당 처리되는 명령어의 개수(Throughput)를 높이기 위한 기술이라는 것이다.

파이프라이닝이 없는 프로세서의 CC는 명령어의 임계 경로 시간이다. 대부분 프로세서의 경우, 메모리에 접근해 데이터를 불러오는 Load 명령어가 가장 시간이 오래 걸리므로 이를 CC로 사용하게 된다. 그러나 파이프라이닝은 프로세서의 명령어를 IF(Instruction Fetch), ID(Instruction Decode), EX(Execute), MEM(Memory Access), WB(Write Back)와 같이 여러 개의 작은 단계로 분할한다.

그 결과 파이프라이닝을 적용한 프로세서의 CC는 명령어의 임계 경로 시간이 아닌 각 단계의 임계 경로 시간에 의해 결정된다. 단계의 처리 시간이 명령어의 처리 시간보다 훨씬 짧으므로 CC를 획기적으로 단축할 수 있다.

위에서 언급한 다섯 단계(IF, ID, EX, MEM, WB)는 MIPS 파이프라이닝의 표준 단계이다. 파이프라이닝을 기준으로 MIPS의 구조를 살펴보면 아래 다섯 가지로 나뉘는 것을 확인할 수 있다.



IM(Instruction Memory)에서는 명령어 패치 단계가 이루어진다. 이때 PC가 가리키는 주소에서 32비트의 명령어를 읽어온다.

Reg(Register File)에서는 가져온 명령어 디코드 단계가 이루어진다. 가져온 명령어를 해독하고 명령어에 필요한 값들을 레지스터에서 읽어온다.

ALU(Arithmetic Logic Unit)에서는 실행 단계가 이루어진다. 앞서 읽어온 레지스터 값들을 ALU에 넣고 실제 명령어를 실행한다. 만약 실행하려는 명령어가 Load/Store 명령어라면 메모리 주소를 계산한다.

DM(Data Memory)에서는 메모리 접근 단계가 이루어진다. Load 명령어라면 DM에서 데이터를 읽어오고, Store 명령어라면 DM에 데이터를 쓰게 된다. 만약 메모리 접근 명령어가 아니라면 해당 부품에서는 아무런 일도 일어나지 않고 통과되게 된다.

마지막으로 Reg 부품을 다시 사용해 쓰기 단계가 이루어진다. ALU 실행 시의 계산 결과나 DM에서 읽어온 값을 최종 목적지에 저장한다.

그러나 이처럼 파이프라이닝을 도입하면 명령어를 순차 실행할 때는 없었던 복잡한 문제들이 발생한다. 여러 명령어가 한 프로세서의 안에서 실행되며 해저드(Hazard)가 발생하는 것이다. 파이프라이닝을 방해하는 해저드에는 크게 세 가지 종류가 있다.

구조적 해저드는 두 명령어가 동시에 같은 하드웨어 부품을 사용하려 할 때 발생한다.

데이터 해저드는 이전 명령어의 연산 결과가 나오기도 전에 다음 명령어가 그 결과를 사용하려 할 때 발생한다. 이 경우 파이프라인은 잠시 대기(Stall)해야만 한다.

제어 해저드는 브랜치 명령어에서 분기 유무가 확정되기도 전에 프로세서가 잘못된 명령어를 미리 가져왔을 때 발생한다. 이 경우 잘못 가져온 명령어들을 모두 버리고(Flush) 올바른 명령어를 다시 가져오는 과정이 요구된다.

해저드로 인한 손해는 파이프라인의 단계가 높아질수록 크게 증가한다. 다만 이러한 단점들에도 불구하고 파이프라이닝을 사용하는 이유는 파이프라이닝을 사용함으로써 프로세서의 성능을 압도적으로 높일 수 있기 때문이다.

만약 파이프라인의 단계를 MIPS의 5단계보다 더 늘린다면 제어 해저드와 데이터 해저드는 크게 악화될 것이다. 해저드의 처리 비용이 증가함으로써 오히려 프로세서의 전체 성능이 낮아질 수도 있다는 것이다.

결국 파이프라인을 몇 단계로 할지 정하는 것은 클락 속도의 향상과 해저드 처리 비용 사이의 Trade-off를 고려하여야 하며, 이것이 바로 현대의 프로세서들이 서로 다른 파이프라인 구조를 갖게 된 핵심적인 이유라고 할 수 있다.

Ⅲ. 소규모 임베디드 시스템

1. ARM Cortex-M4

소규모 임베디드 시스템에서 선정한 프로세서는 ARM Cortex-M4이다. Cortex-M4는 IoT, 웨어러블 기기, 소형 로봇, 드론 등 수많은 임베디드 시스템의 표준이다. 또한 ARM은 교육 및 개발자용으로 아키텍처 문서를 매우 상세하게 공개한다.

2. ISA 분석

Cortex-M4는 ARMv7-M 아키텍처라는 ISA를 사용한다. 이는 MIPS와 마찬가지로 RISC 계열이다. 그러나 일반 PC와는 다른 저전력, 저비용이라는 임베디드 환경에 맞추기 위해 여러 실용적인 변화를 거쳤다.

MIPS는 모든 명령어가 32비트의 고정적인 길이를 가진다. 이를 통해 패치, 디코드 단계를 단순화하였으나 일부 명령어에서는 필요하지 않은 비트까지 표현해야 한다는 단점이 존재했다.

그러나 Cortex-M4에서는 16비트와 32비트가 혼합된 가변 길이 ISA를 사용한다. MIPS의 add 명령어와 같은 단순한 작업은 32비트가 아닌 16비트로 처리함으로써 코드의 밀도를 높인다. 임베디드 시스템의 ROM의 용량이 매우 작고 비싸기 때문에 프로그램의 크기를 줄여주는 코드 밀도가 치명적인 요소로 작용한다.

가변 길이 ISA를 사용함으로써 MIPS에 비해 디코드 단계가 복잡해진다는 단점 역시 존재한다. 프로세서가 명령어를 읽은 후 해당 명령어가 16비트인지 32비트인지를 판단하는 추가적인 로직이 필요하다. 이는 Cortex-M4의 파이프라인 설계를 복잡하게 만드는 주된 원인이 된다.

r0~r31까지 32개의 범용 레지스터를 가지는 MIPS와 달리 Cortex-M4는 16개의 레지스터를 가진다. MIPS에 비해 사용가능한 레지스터의 개수는 적지만 레지스터를 표현하는 비트 수를 줄임으로써 16비트 명령어의 공간을 효율적으로 사용할 수 있다.

또한 Cortex-M4는 MIPS에는 존재하지 않는 DSP 명령어와 FPU 명령어를 지원한다. 해당 명령어들을 사용함으로써 보다 더 빠른 속도로 연산을 진행할 수 있다.

3. 파이프라인 구조 분석

MIPS의 5단계 파이프라인의 목표가 Throughput의 증가라면 Cortex-M4의 파이프라인의 목표는 저전력과 빠른 응답 속도이다. Cortex-M4는 MIPS와 달리 기본적으로 3단계 파이프라인을 사용한다. MIPS의 EX, MEM, WB 단계를 EX라는 하나의 단계로 합치고 전체 명령어의 흐름을 패치, 디코드, 실행의 세 단계로 나눈다.

Cortex-M4는 파이프라인의 단계를 낮춤으로써 Interrupt Latency를 줄였다. 임베디드 시스템은 센서의 값이 도착하거나 버튼이 눌리는 등의 인터럽트가 발생하면 이에 빠른 속도로 반응해야 한다. 이때 파이프라인 단계가 높으면 현재 실행 중인 명령

어들이 모두 끝날 때까지 기다려야 인터럽트 처리를 시작할 수 있다.

그러나 파이프라인을 3단계로 줄임으로써 응답 속도를 높일 수 있다. 인터럽트가 발생했을 때 EX 중인 명령어를 끝내거나 중단함으로써 즉시 인터럽트 서비스 루틴(ISR)로 점프할 수 있게 되는 것이다.

또한 파이프라인의 단계가 깊어질수록 각 단계 사이 데이터를 임시 저장하는 파이프라인 레지스터의 개수 역시 증가한다. Cortex-M4는 3단계 파이프라이닝을 사용함으로 오직 2개의 레지스터만이 필요하다. 하드웨어 로직을 단순하게 바꿈으로써 저전력, 저비용의 프로세서를 구성할 수 있게 되는 것이다.

이에 더해 짧은 파이프라인은 해저드가 발생할 수 있는 경우의 수를 줄인다. EX 단계가 여러 사이클을 소모하는 Multi-cycle Execute 방식을 택함으로써 하드웨어 복잡도를 낮췄다.

즉, Cortex-M4는 32비트 고정 명령어를 버리는 대신 16비트 명령어를 혼합함으로써 코드의 밀도를 높인다. 5단계의 파이프라인을 버리고 3단계의 짧은 파이프라인을 택함으로써 빠른 응답 속도와 저전력의 이점을 볼 수 있다.

이는 모두 Cortex-M4가 저전력과 저비용을 추구하는 소규모 임베디드 시스템이기 때문이며, Cortex-M4는 임베디스 시스템의 목적에 맞는 방식으로 발전되어 왔다는 사실을 확인할 수 있다.

Ⅳ. 개인용 컴퓨터 시스템

1. Intel Core i-시리즈

개인용 컴퓨터 시스템에서 선정한 프로세서는 Intel Core i-시리즈, 그중에서도 스카이레이크(Skylake) 마이크로아키텍처이다. 스카이레이크는 현재의 인텔 CPU 아키텍처의 기본 골격이 되며 PC 프로세서의 표준이라 할 수 있다.

순수 RISC 방식의 MIPS와 임베디드 RISC 방식의 Cortex-M4와 비교했을 때 Intel Core i-시리즈는 최고의 단일 스레드 성능을 제공한다. 이를 위해 극도로 복잡한 로직의 하이브리드 구조를 따르게 된다.

2. ISA 분석

스카이레이크는 앞선 두 프로세서와 다르게 x86-64라는 CISC 방식의 ISA를 사용한다. x86-64의 명령어는 가변 길이로 어떠한 명령어는 1바이트일 수도 있고 어떠한 명령어는 15바이트에 달할 수도 있다. 명령어의 길이가 고정되어 있지 않으므로 패치/디코드 단계가 매우 복잡하다. 또한 명령어의 바이트 수를 실시간으로 파악하는 것은 파이프라인에 엄청난 부담을 준다.

x86-64는 CISC 기반의 ISA이므로 명령어 하나가 여러 가지의 일을 수행할 수 있다. 예를 들어 LOAD로 메모리값을 레지스터로 가져오고, ADD로 연산하고, STORE로 저장하는 3단계를 “ADD [메모리 주소], 레지스터”와 같은 한 줄의 형태로 표현할 수 있다. 이는 코드의 밀도를 높여 적은 수의 명령어로 많은 일을 처리하는 것을 가능하게 한다. 그러나 명령어마다 요구되는 클락 사이클 수가 달라지고, 기본 사이클의 길이가 길어져 파이프라이닝을 더 복잡하게 만든다.

x86-64는 MIPS와 달리 64비트 모드에서 오직 16개의 범용 레지스터만을 제공한다. 프로그래머(또는 컴파일러)가 사용할 수 있는 레지스터의 개수가 한정되어 있으므로 파이프라인에서 데이터의 병목현상을 일으킬 수 있다.

x86-64는 이전의 명령어들을 모두 포용하여야 하므로 높은 하위 호환성을 제공하여야 한다. 이 때문에 x86-64는 수십 년간 누적된 수천 개의 명령어를 지원하며 디코더를 더욱 복잡하게 만드는 원인이 된다.

3. 파이프라인 구조 분석

스카이레이크의 파이프라인 구조는 CISC 기반의 ISA를 내부적으로는 RISC처럼 바꿔 실행하는 것이다. 이것은 스카이레이크를 포함한 모든 현대의 인텔, AMD 프로세서의 핵심 설계 철학이며 이를 하이브리드 구조라 부른다.

MIPS와 Cortex-M4가 순차적 파이프라인을 사용하는 것과 달리 스카이레이크는 비순차적 실행과 슈퍼스칼라(Superscalar) 구조를 사용한다. MIPS는 프로세서에 들어온 명령어를 순차적으로 실행한다. 그러나 스카이레이크는 명령어의 순서를 무시하고

데이터가 준비된 명령어부터 처리하여 실행 효율을 최대한으로 끌어올린다.
또한 한 클락에 하나의 명령어만을 처리하는 MIPS와 달리 스카이레이크는 한 클락에 여러 개의 명령어(4~8개)를 동시에 처리하는 슈퍼스칼라 구조를 채택하였다. 스카이레이크의 파이프라인은 MIPS의 5단계보다 훨씬 깊고 복잡하며(약 14~19단계) 크게 아래 3가지의 엔진으로 나뉜다.

1) 프론트엔드

프론트엔드는 L1 캐시에서 x86 명령어를 가져와서 내부 RISC 명령어인 마이크로옵(Micro-operations)으로 번역한다. 이때 파이프라이닝이 최대 19단계로 깊기 때문에 분기예측에 실패해 제어 해저드가 발생하면 앞의 18단계를 모두 버려야 하는 치명적인 손해가 발생한다. 이를 위해 MIPS보다 더 정확한 분기예측이 필요하다.
디코더는 가져온 x86 명령어를 해독하여 고정 길이의 마이크로옵으로 변환한다. 해당 디코드 작업은 매우 오랜 시간이 걸리므로 스카이레이크는 한 번 번역한 마이크로옵을 L0 캐시에 저장해두었다가 다음번에 같은 코드가 실행되면 디코드를 건너띄고 L0 캐시에서 마이크로옵을 바로 가져온다.

2) 실행 엔진

실행 엔진은 번역된 마이크로옵을 슈퍼스칼라 구조를 통해 순서에 상관없이 가장 빠른 방식으로 처리한다. 16개의 범용 레지스터를 가진 x86의 한계를 극복하기 위해 레지스터 개명기를 사용해 내부에 존재하는 180개의 물리 레지스터에 매핑한다.
매핑이 끝난 마이크로옵은 예약 스테이션(RS)라는 대기실 공간으로 보내진다. RS는 스케줄러를 통해 지속적으로 관리받다가 필요한 데이터가 준비된 순서대로 실행 유닛에 보내진다. 실제 명령어의 처리는 슈퍼스칼라 구조 아래 한 클락에 최대 8개의 마이크로옵을 동시에 실행된다.

3) 백앤드

비순차적 실행으로 인해 섞여버린 결과를 원래 프로그램의 순서대로 재조립한다. 실행 엔진이 계산을 끝내면 그 결과는 재정렬 버퍼에 임시적으로 저장된다. 은퇴 유닛이 재정렬 버퍼를 감시하며 원래 명령어의 순서대로 실행 결과를 확정(Commit)한다.

즉, Skylake는 단일 스레드 내에서 최고의 성능을 지원하기 위해 CISC 기반의 명령어, 하이브리드 구조, 비순차적 슈퍼스칼라 실행 엔진을 사용한다. 이를 통해 MIPS, Cortex-M4는 비교할 수 없는 압도적인 성능을 제공할 수 있다. 그러나 깊어진 파이프라인 단계만큼 높은 하드웨어 복잡도와 막대한 전력 소모량을 가진다.

Ⅴ. 대형 슈퍼컴퓨터 시스템

1. NVIDIA A100 Tensor Core GPU

대형 슈퍼컴퓨터 시스템에서 선정한 프로세서는 NVIDIA A100 Tensor Core GPU이다. A100은 앞서 설명한 MIPS, Skylake와 같은 CPU가 아니라 범용 목적 GPU, GPGPU(General-Purpose GPU)이다. 즉, A100의 설계 목적은 스카이레이크처럼 하나의 스레드를 빨리 처리하는 것이 아니라 수만 개의 작업을 동시에 처리하는 것이다.

이러한 목적의 차이는 ISA와 파이프라인 구조에서 CPU와는 다른 근본적인 차이를 만들어낸다. A100은 암페어 GA100 아키텍처를 기반으로 한다.

2. ISA 분석

A100의 ISA는 MIPS나 x86과는 완전히 다른 계층 구조를 가진다. PTX(Parallel Thread Excution)는 프로그래머(혹은 컴파일러)가 바라보는 가상의 ISA이다. PTX 코드는 NVIDIA 드라이버에 의해 A100 하드웨어가 직접 실행할 수 있는 네이티브(Native) ISA로 다시 컴파일된다. 이를 위해 사용되는 SASS(Shader ASSembler)는 인텔의 마이크로옵처럼 내부적으로 처리되며 아키텍처 세대마다 바뀌고 외부에 공개되지 않는다.

A100의 가장 핵심적인 특징은 SIMT(Single Instruction, Multiple Thread)이다. MIPS의 경우 하나의 명령어는 반드시 하나의 데이터만을 처리한다. Skylake의 경우 하나의 명령어가 여러 개의 데이터 조각을 한 번에 처리한다. 반면 A100은 32개의 스레드를 하나의 Warp로 묶어 이 단위로 스레드를 관리한다.

즉, 32개의 스레드는 모두 같은 PC(프로그램 카운터)를 공유하지만 각자 다른 데이터를 가지고 연산을 수행한다. SIMT는 스레드별로 브랜치를 다르게 처리할 수 있어 SIMD(Single Instruction Multiple Data)보다 유연한 실행이 가능하다.

A100의 SM(Streaming Multiprocessor) 하나는 MIPS나 Skylake와는 비교할 수 없는 많은 수의 레지스터를 가진다. MIPS가 32개의 32비트 레지스터를 가지고 Skylake가 16개의 64비트 레지스터와 180개의 물리 레지스터를 가지는 것과 달리, A100은 SM 당 65,536개의 32비트 레지스터(256KB)를 가진다.

Skylake의 물리 레지스터가 하나의 스레드가 비순차적 실행을 할 수 있도록 돕는다면 A100의 거대한 레지스터 파일은 수천 개의 스레드가 동시에 SM에 상주하며 컨텍스트 스위칭(Warp Switching)을 할 수 있도록 지원한다. 이 부분에 대해서는 아래 파이프라인 구조 분석 파트에서 자세한 설명이 이어진다.

Cortex-M4에 DSP/FPU라는 특수한 명령어가 존재했다면 A100에는 AI 연산을 위한 텐서 코어라는 하드웨어와 이를 위한 전용 명령어가 존재한다. “mma.sync” 명령어는 A100 PTX ISA의 핵심이다. 해당 명령어 하나는 4x4 행렬 곱셈 및 누적 연산(AI의 핵심 연산)을 단 한 번에 처리한다. 만약 MIPS였다면 이 연산을 처리하기 위해

수십에서 수백 개의 Load, Multiply, Add 명령어를 반복 실행해야 했을 것이다.

Skylake의 L1, L2, L3 캐시가 하드웨어의 관리를 받는 것과 달리 A100의 ISA는 프로그래머가 명시적으로 메모리 공간을 지정해야 한다. 프로그래머는 프로그램의 성능을 높이기 위해 크고 느린 글로벌 메모리(DRAM)에 접근할 것인지 작지만 빠른 SM 내부의 공유 메모리(SRAM)에 접근할 것인지를 “ld.global”, “ld.shared” 같은 명령어로 직접 결정해야 한다. 이를 통해 프로그래머의 부담이 증가하는 대신 프로그램의 성능을 극한까지 끌어올릴 수 있다.

3. 파이프라인 구조 분석

A100의 파이프라인은 MIPS나 Skylake의 단일 파이프라인과는 완전히 다른 개념을 가지고 있다. A100의 핵심 실행 유닛은 프로세서 자체가 아닌 SM이며, GA100 GPU는 이 SM을 108개나 가지고 있다.

SM의 개념 자체는 MIPS나 Skylake에 존재하는 코어의 존재와 비슷하다. 그러나 내부의 구조는 완전히 다른 모습을 하고 있다. A100의 SM은 거대한 하드웨어 멀티스레딩 머신이다. 1개의 SM은 4개의 파티션으로 나뉘며 각 파티션은 다음과 같은 실행 유닛을 가진다.

16개의 FP32 CUDA 코어 (ALU), 8개의 FP64 CUDA 코어 (ALU), 16개의 INT32 CUDA 코어 (ALU), 2개의 3세대 텐서 코어 (Matrix ALU), Load/Store 유닛, Special Function Unit (SFU) 등이 그것이다.

Skylake가 한 클락에 8개의 마이크로옵을 실행하는 슈퍼스칼라 구조를 가진다면 A100의 SM은 한 클락에 수십, 수백 개의 스레드에서 나온 연산을 동시에 처리하는 Massively Parallel 구조이다.

Skylake는 파이프라인을 멈추지 않기 위해 비순차적 구조의 복잡한 로직으로 데이터 해저드를 피해 갔다. 반면 A100은 완전히 다른 방식으로 이를 해결한다. 바로 하드웨어 멀티스레딩을 통한 지연 시간 숨기기(Latency Hiding)이다.

A100의 SM은 수십 개의 Warp, 즉 수천 개의 스레드를 동시에 상주시킬 수 있다. SM의 Warp 스케줄러가 이들 중 실행 가능한 Warp [A]를 선택해 파이프라인에 투입한다. 투입된 Warp [A]에서 수백 클락이 걸리는 글로벌 메모리 로드 명령어를 실행될 차례가 왔다면 어떻게 할까.

만약 A100이 MIPS나 Cortex-M4처럼 순차실행 프로세서였다면 해당 데이터가 도착할 때까지 파이프라인 전체가 스톨된다. Skylake와 같은 비순차적 실행 프로세서였다면 Load와 상관없는 다른 명령어들을 찾아 먼저 실행할 것이다.

그러나 A100의 경우 Warp [A]를 즉시 대기 상태로 바꾼다. 파이프라인은 비우지 않는다. Warp 스케줄러는 단 1클락 만에 SM에 상주해 있던 다른 Warp(메모리 접근이 필요 없는 명령어 실행 차례)를 파이프라인에 투입하여 실행한다.

해당 명령어 역시 메모리 접근을 시도하면 스케줄러는 또 다른 Warp를 투입한다. 수백 클락이 지나 Warp [A]의 메모리 접근이 끝나면 Warp [A]는 다시 준비 상태가

되어 스케줄러의 선택을 기다리게 된다.

결론적으로 보았을 때 A100의 개별 스레드 파이프라인 자체는 MIPS와 같이 단순하며 순차적이다. 하지만 파이프라인이 비는 틈을 다른 스레드(Warp)로 즉시 채워버림으로써 실행 유닛을 쉬지 않게 작동하게 한다. 이를 통해 스카이레리크의 ILP(명령어 수준의 병렬성)이 아닌 TLP(스레드 수준의 병렬성)을 극대화하며 전체 시스템의 Throughput을 극한까지 증가시킬 수 있다.

VI. 결론

해당 보고서는 MIPS의 ISA와 파이프라인 구조를 기준으로 하여 세 가지 서로 다른 시스템의 프로세서를 비교 분석하였다. 임베디드, PC, 슈퍼컴퓨터에서 선정한 3가지 프로세서와 MIPS의 비교는 아래의 표로 간단하게 요약할 수 있다.

특징	MIPS	ARM Cortex-M4	Intel Skylake	NVIDIA A100
주요 목적	RISC의 표준	저전력, 저비용, 빠른 응답 속도	최고의 단일 스레드 속도	최고의 병렬 처리량
ISA	RISC (32비트 고정)	RISC (16/32비트 가변)	CISC (1~15비트 가변)	SIMT (PTX 가상 ISA)
ISA 특징	Load/Store	높은 코드 밀도, DSP	복잡한 명령어, 높은 하위 호환성	SIMT, 텐서 코어(AI)
파이프라인	5단계 순차실행	3단계 순차실행	14~19단계 비순차 실행	수백 단계 순차 실행
핵심 구조	단순하고 규격화	빠른 인터럽트	슈퍼스칼라, ILP 극대화	Latency Hiding, TLP 극대화

이 표는 시스템의 목적에 따라 각 프로세서가 선택한 발전의 방향성을 명확히 보여준다.

ARM Cortex-M4는 저전력, 저비용, 빠른 응답 속도라는 임베디드 시스템의 목적을 위해 Throughput을 포기하였다. 16/32비트 가변 ISA로 코드의 밀도를 높여 메모리 비용을 절감했고 3단계의 짧은 파이프라인으로 빠른 인터럽트 응답 속도를 확보했다. 이는 MIPS와 같은 고성능을 포기한 대신 임베디드 환경에서의 핵심적인 장점들을 선택한 합리적인 발전이다.

Intel Skylake는 최고의 단일 스레드 성능이라는 개인 PC의 목적을 달성하기 위해 단순성을 버리고 복잡성을 선택했다. x86이라는 CISC 구조를 사용하기 위해 내부에서 이를 RISC(마이크로옵)로 변환하는 하이브리드 구조를 채택했다. 최대 19단계에 달하는 깊은 파이프라인과 비순차적 실행, 슈퍼스칼라 구조는 단일 스레드의 성능을 최고로 극대화하였다. 그러나 이를 위해 막대한 전력 소모와 높은 하드웨어 복잡도를 감수해야만 한다.

NVIDIA A100은 최고의 병렬 처리량이라는 슈퍼컴퓨터의 목적을 위해 Skylake와는 정반대의 발전을 선택했다. A100은 개별 스레드의 속도를 의도적으로 포기했다. 대신 수만 개의 스레드를 SIMT 구조로 묶고 파이프라인이 비는 틈을 다른 스레드(Warp)로 채워 넣는 Latency Hiding 전략을 사용했다. 이는 단일 작업에서 매우 낮은 속도를 보이지만 대규모의 작업은 Skylake와 비교가 불가능할 정도로 빠르게 처리하는 것이 가능하다.

프로세서는 저마다의 목적을 위해 진화한다. 만약 단일 스레드의 성능이 중요한 시스템에서 NVIDIA A100을 사용한다면 프로세서의 효율은 급격하게 떨어질 것이다. 반대로 낮은 예산을 가진 임베디드 시스템에서 Intel Skylake를 사용한다면 아무리

뛰어난 성능을 보이더라도 그 시스템은 실패한 시스템으로 여겨질 것이다.

모든 분야를 통틀어 가장 뛰어난 성능을 내는 절대적인 프로세서라는 것은 존재하지 않는다. 다만 프로그래머들은 각 프로세서의 구조적 특징을 이해하고 해당 시스템에 그 프로세서가 사용된 이유를 추론하며 각 시스템의 목적에 맞게 소프트웨어의 성능을 최적화하여야 할 것이다.

그리고 그것이 프로그래머인 우리가 프로그래머 관점의 하드웨어를 공부하여야 하는 가장 큰 이유이다.