Computer Hardware & Industry Perspective

커리큘럼

주차	날짜	이론	실습	선각 보고서 주제
1주차	4/20 (일)	오리엔테이션, 하드웨어 구성 요소 개요	데스크탑 전체 분해, 주요 부품 구조 관찰	컴퓨터를 이해한다는 건 무엇일까?
2주차	4/26 (토)	메모리 & 저장장치 구조	RAM 분리/장착, SSD 분해 관찰	저장 장치는 결국 사라질까?
3주차	5/3 (토)	발열과 냉각 구조	CPU 쿨러 제거 + 써멀 재도포, 팬 속도 측정	열 관리가 성능을 결정한다면 우리는 어떻게 설계해야 할까?
4주차	**5/17 (토)**	**CPU 구조 & 최신 기술 + 분해 실습**	**CPU 히트스프레더 제거**	**CPU의 성능 향상이 둔화된 이유와 기술적 대안**
5주차	미정	GPU 구조 & 최신 기술 + 분해 실습	GPU 분해, 방열판 구조 분석, 팬 곡선 실험	AI와 게임 성능을 동시에 잡으려면 GPU는 어떻게 진화해야 할까?
6주차	미정	입출력 장치 & HCI	키보드/마우스 분해 및 센서 원리 학습	사람과 기계의 경계는 어디까지 사라질 수 있을까?
7주차	미정	반도체 제작: 웨이퍼~패키징 전체 흐름	공정 흐름도 도식화, 칩 구조 스케치, 패키지 비교	성능은 결국 패키징으로 결정되는가?
8주차	미정	미래 컴퓨팅 구조 - 포스트 폰 노이만 등	아키텍처 비교, 미래 예측	우리는 어떤 컴퓨팅 구조 위에서 살아가게 될까?

※ 일정은 변동 가능하며, 팀 내 논의를 통해 유동적으로 조정될 수 있습니다.
※ 주차별로 배운 내용을 다음주까지 리포트(선각)를 작성해 발표합니다.

C.H.I.P

22:00

1교시 : 과제 발표

쉬는 시간 10분

2교시 : 이론

쉬는 시간 10분

3교시 : 실습

01:00?

C.H.I.P

이론

ISA
MicroArchitecture
Pipelining
Superscalar

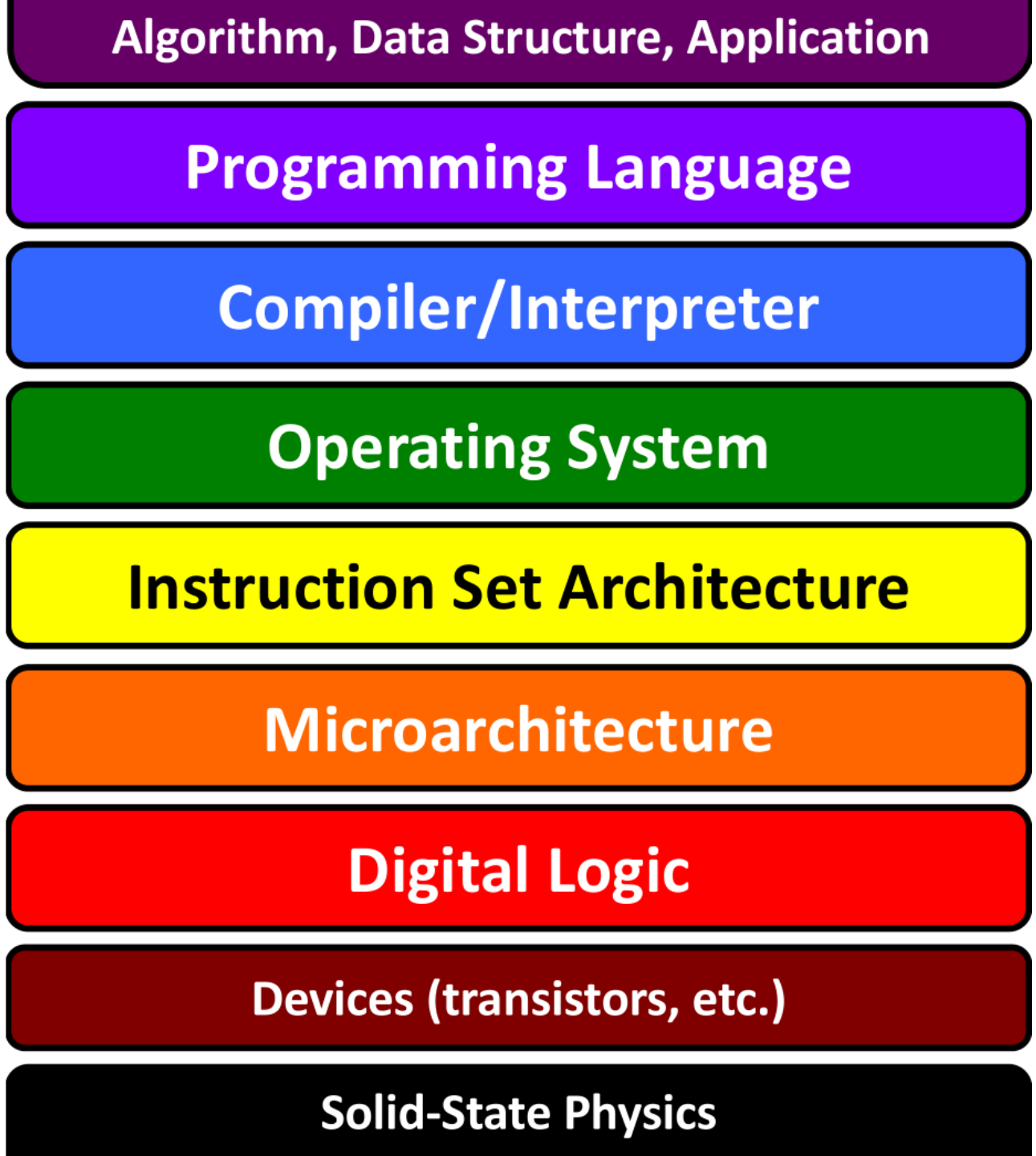
ISA

ISA (Instruction Set Architecture)

- 명령어 집합
- 하드웨어를 제어하는 명령어의 표준
- 소프트웨어가 하드웨어와 소통하기 위해 사용하는 공식 언어

위로 갈수록
추상화

>>>



C.H.I.P

구성 요소

- Instruction Format : 명령어의 비트 구조 (opcode, operand, immediate 등)
- Register Set : 사용할 수 있는 레지스터 종류 및 개수 (예: x86은 EAX, ARM은 X0~X30)
- Addressing Modes : 메모리 주소를 계산하는 방식 (직접, 간접, 오프셋 등)
- Data Types : 정수, 부동소수점, 벡터 등
- Privilege Levels : 사용자 모드 / 커널 모드
- Exception/Interrupt Handling : 예외 처리 방식

예시

- x86-64 : 복잡하고 오래된 ISA. CISC 계열. 다양한 연산과 addressing 지원
- ARM : 모바일, 저전력 중심. RISC 철학. 일관된 명령어 크기
- RISC-V : 모듈형 설계, 오픈소스 ISA. 교육용 + 차세대 SoC에서 주목받는 ISA

CISC vs RISC

항목	CISC (Complex Instruction Set Computer)	RISC (Reduced Instruction Set Computer)
의미	복잡한 명령어 집합 구조	간단하고 정형화된 명령어 집합 구조
철학	명령어 하나로 많은 일을 하자	명령어는 단순하고 빠르게 하자
명령어 길이	가변적 (1~15바이트 등)	고정형 (보통 4바이트)
명령어 개수	수백 개 이상, 다양함	상대적으로 적고 단순함
명령어 실행 시간	명령어에 따라 다름	대부분 1클럭 이내 실행
하드웨어 복잡도	디코더가 복잡, 실행기 설계도 어려움	디코더 단순, 파이프라인 설계 쉬움
예시 ISA	x86, x86-64 (인텔, AMD)	ARM, RISC-V, MIPS, SPARC

CISC – x86

MOV AX, [BX+SI]
→ 메모리에서 값을 읽고, 바로 레지스터로 이동 (한 줄에 여러 연산 포함)

RISC - ARM

LDR R0, [R1]
ADD R2, R0, R3

→ 메모리 접근과 연산을 명확히 분리, 하나의 명령어는 하나의 일만

왜 이런 차이가 생겼을까?

관점	CISC	RISC
초기 목적	메모리 비싸던 시절, 명령어를 짧게 압축	하드웨어 성능 향상, 파이프라인 최적화
개발 철학	컴파일러가 못하던 일을 CPU가 도와주자	CPU는 빠르고 단순하게, 컴파일러가 최적화하자

ISA가 뭐냐고 면접관이 물어본다면?

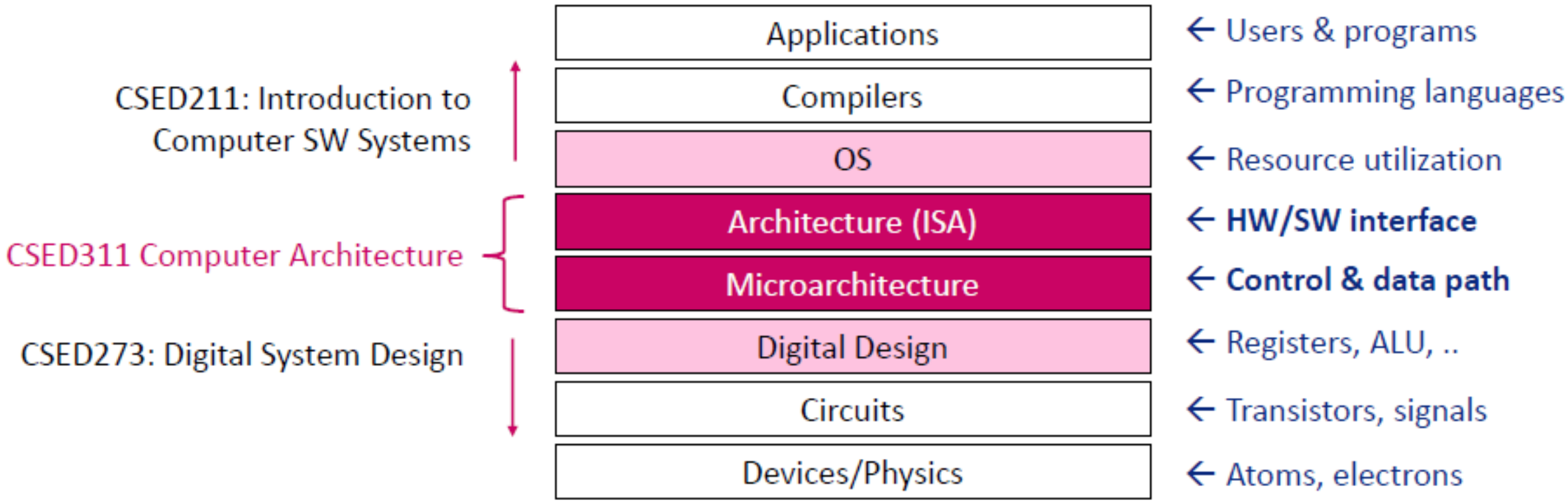
소프트웨어와 하드웨어가 소통할 수 있도록 정해진 명령어의 표준 규약입니다.
컴파일러가 코드를 기계어로 바꿀 때 이 ISA에 맞춰 변환하며,
어떤 연산이 가능하고, 어떤 레지스터를 쓸 수 있는지를 정의합니다.

예를 들어 x86, ARM, RISC-V 같은 것들이 대표적인 ISA이고,
같은 ISA라도 CPU 설계 방식에 따라 성능 차이가 날 수 있습니다.

MicroArchitecture

Microarchitecture — ISA의 구현체

- ISA를 실제로 실행하기 위한 CPU 내부 구조 설계


CSED211: Introduction to Computer SW Systems
CSED311 Computer Architecture
CSED273: Digital System Design
Applications
Compilers
OS
Architecture (ISA)
Microarchitecture
Digital Design
Circuits
Devices/Physics
← Users & programs
← Programming languages
← Resource utilization
← HW/SW interface
← Control & data path
← Registers, ALU, ..
← Transistors, signals
← Atoms, electrons

Same Architecture
Different Microarchitecture

AMD Phenom X4

- X86 Instruction Set
- Quad Core
- 125W
- Decode 3 Instructions/Cycle/Core
- 64KB L1 I Cache, 64KB L1 D Cache
- 512KB L2 Cache
- Out-of-order
- 2.6GHz

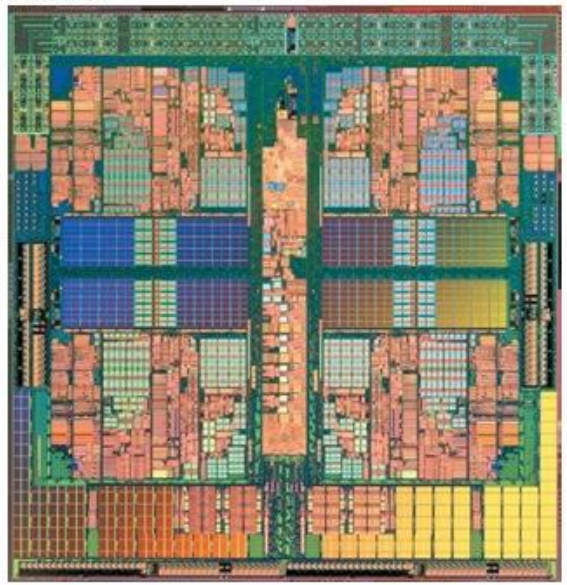


Image Credit: AMD

Intel Atom

- X86 Instruction Set
- Single Core
- 2W
- Decode 2 Instructions/Cycle/Core
- 32KB L1 I Cache, 24KB L1 D Cache
- 512KB L2 Cache
- In-order
- 1.6GHz

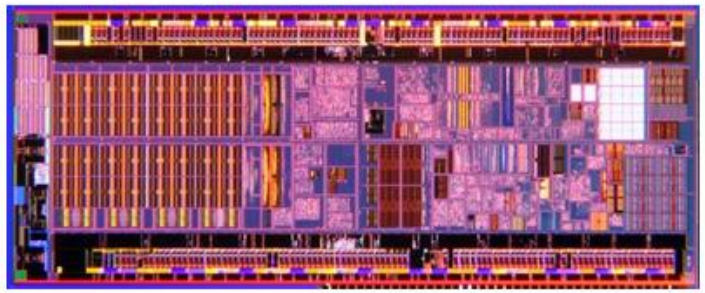


Image Credit: Intel

Different Architecture
Different Microarchitecture

AMD Phenom X4

- X86 Instruction Set
- Quad Core
- 125W
- Decode 3 Instructions/Cycle/Core
- 64KB L1 I Cache, 64KB L1 D Cache
- 512KB L2 Cache
- Out-of-order
- 2.6GHz

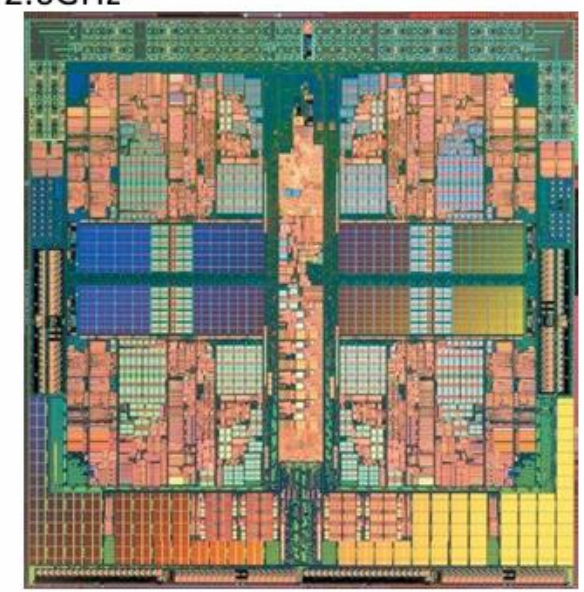


Image Credit: AMD

IBM POWER7

- Power Instruction Set
- Eight Core
- 200W
- Decode 6 Instructions/Cycle/Core
- 32KB L1 I Cache, 32KB L1 D Cache
- 256KB L2 Cache
- Out-of-order
- 4.25GHz

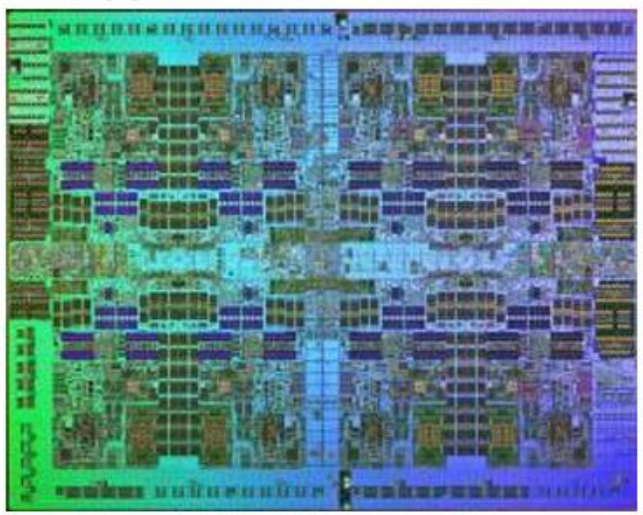


Image Credit: IBM
Courtesy of International Business Machines Corporation, © International Business Machines Corporation.

구성 요소

- Pipeline 설계 : 페치-디코드-실행-메모리-쓰기 단계를 몇 단계로 쪼갰는가?
- Out-of-Order Execution : 명령어 실행 순서를 재정렬해서 성능을 높임
- Branch Prediction : 조건 분기를 미리 예측하여 pipeline stall 방지
- Cache Hierarchy : L1/L2/L3 캐시, prefetcher, replacement policy 등
- Execution Units : ALU, FPU, Load/Store unit 등 다중 병렬 유닛 배치
- Speculative Execution : 미리 명령어 실행 → 성능 향상 but 보안 이슈(Spectre/Meltdown)

예시

- Intel Raptor Lake : x86 ISA 기반. 고성능/고효율 코어 혼합. 향상된 분기 예측기 사용
- AMD Zen4 : Ryzen 7000시리즈, x86 ISA 기반. 확장된 L2 캐시, SMT 지원
- Apple Firestorm : M1, A14 Bionic, ARM ISA 기반. 극단적인 캐시 최적화

Raptor Lake

Double the Efficient-Cores[1]

Wider

Up to 16 E-cores

4MB L2 per cluster

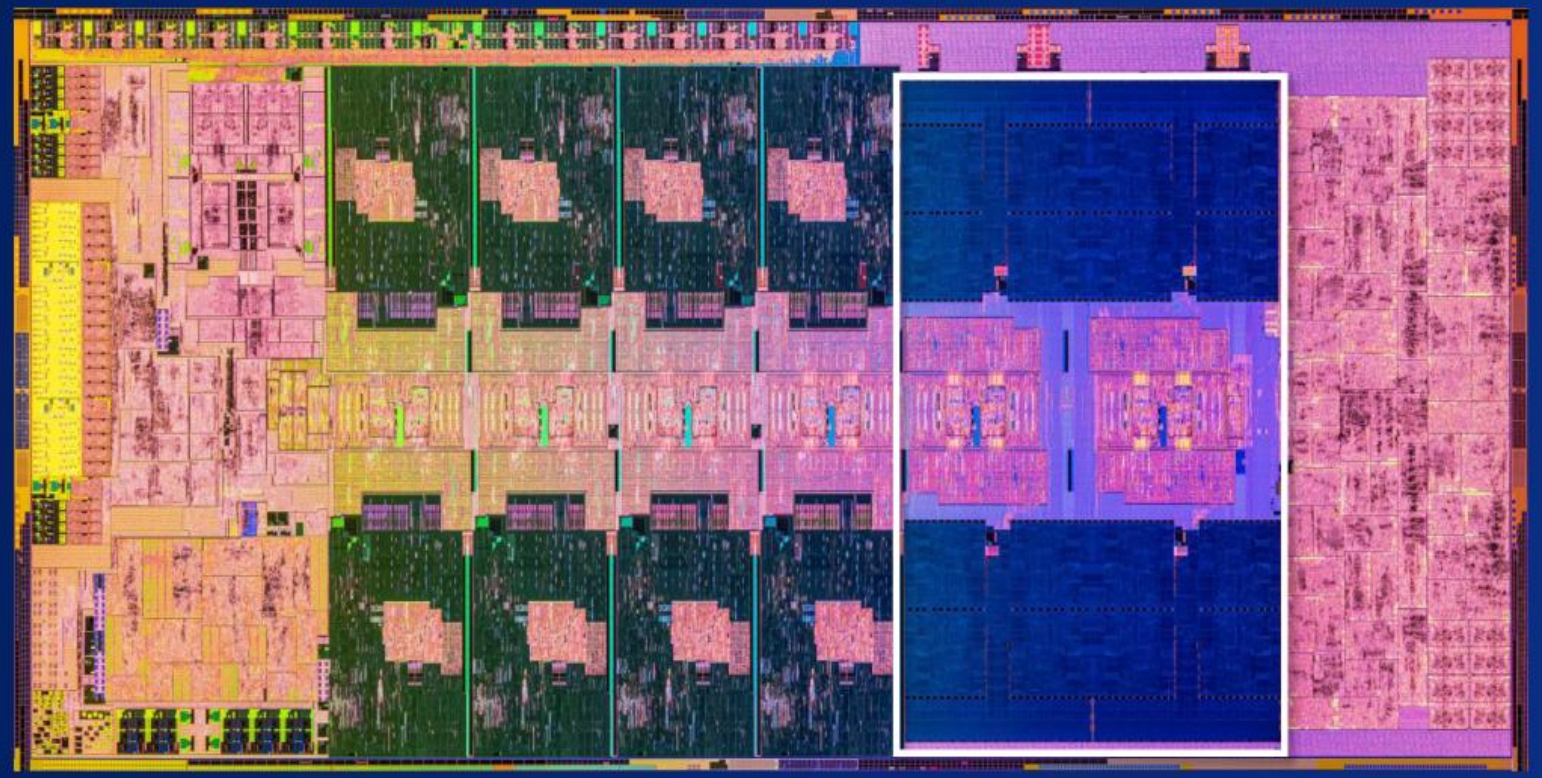
Faster

Up to 600MHz faster (ACT)

Up to 4.3GHz turbo

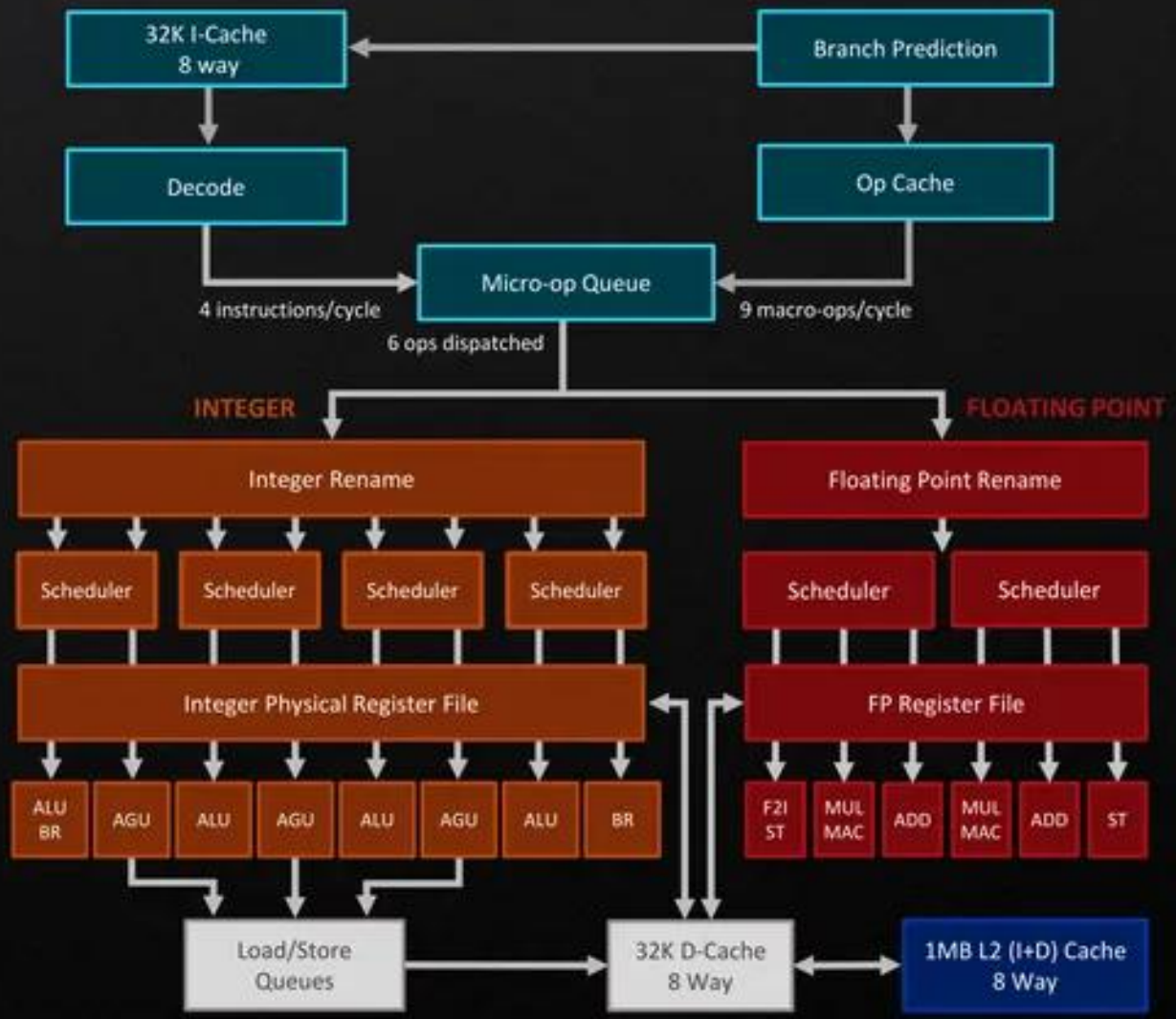
Smarter

Significantly optimized prefetcher algorithm



Microarchitecture Overview

- Branch Prediction Improvements
- Larger Op Cache
- Larger Instruction Retire Queue
- Larger Int/FP register file
- Deeper buffers throughout the core
- Power efficient AVX-512 support in the Floating-Point Unit
- Load/Store improvements
- L2 Cache 1M, 8-way



ISA와 Microarchitecture의 관계

구분	ISA	Microarchitecture
역할	무엇을 할 수 있느냐	어떻게 그것을 할 것이냐
정의	명령어와 그 형식의 표준	명령어를 실행하는 CPU의 설계
예시	x86, ARM, RISC-V	Zen4, Raptor Lake, Apple M1 Firestorm
영향	소프트웨어 호환성 결정	실제 성능, 전력 소모, 면적 등 결정
소프트웨어 영향	컴파일러, OS, ABI	거의 없음 (하지만 간접 성능 영향)
하드웨어 영향	기본 규칙만 명시	거의 모든 세부 구현 책임

요약

항목	설명
ISA	명령어의 '형식과 의미'를 정의한 소프트웨어 인터페이스의 규약
Microarchitecture	해당 ISA를 실제로 구현한 CPU 내부 설계의 상세한 방법
비유	ISA는 '프로그래밍 언어의 문법' / Microarchitecture는 '컴파일러의 구현'

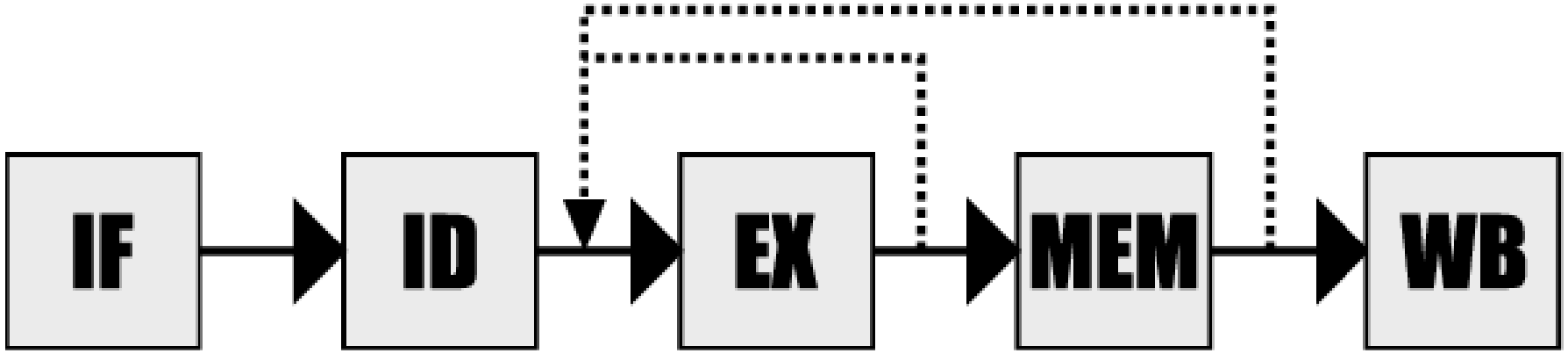
Pipelining

파이프라이닝 — 명령어를 쪼개서 겹쳐 실행

단계	설명
IF (Instruction Fetch)	명령어 가져오기
ID (Instruction Decode)	명령어 해석
EX (Execute)	연산 수행
MEM (Memory Access)	메모리 읽기/쓰기
WB (Write Back)	결과 레지스터에 저장

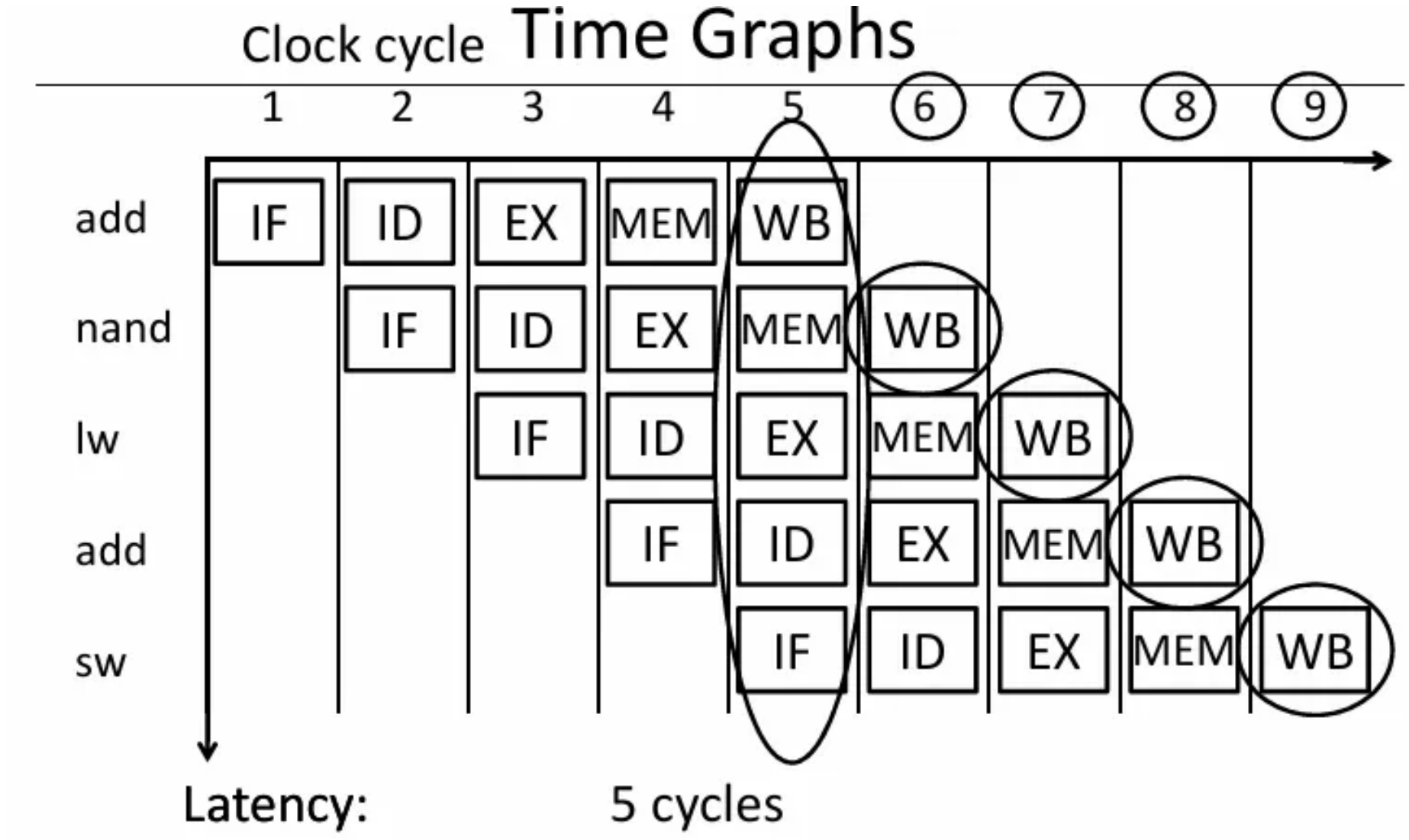
명령어 한 개를 끝내고 다음 걸 실행하는 게 아니라,
여러 명령어를 동시에 다른 단계에서 실행하는 방식

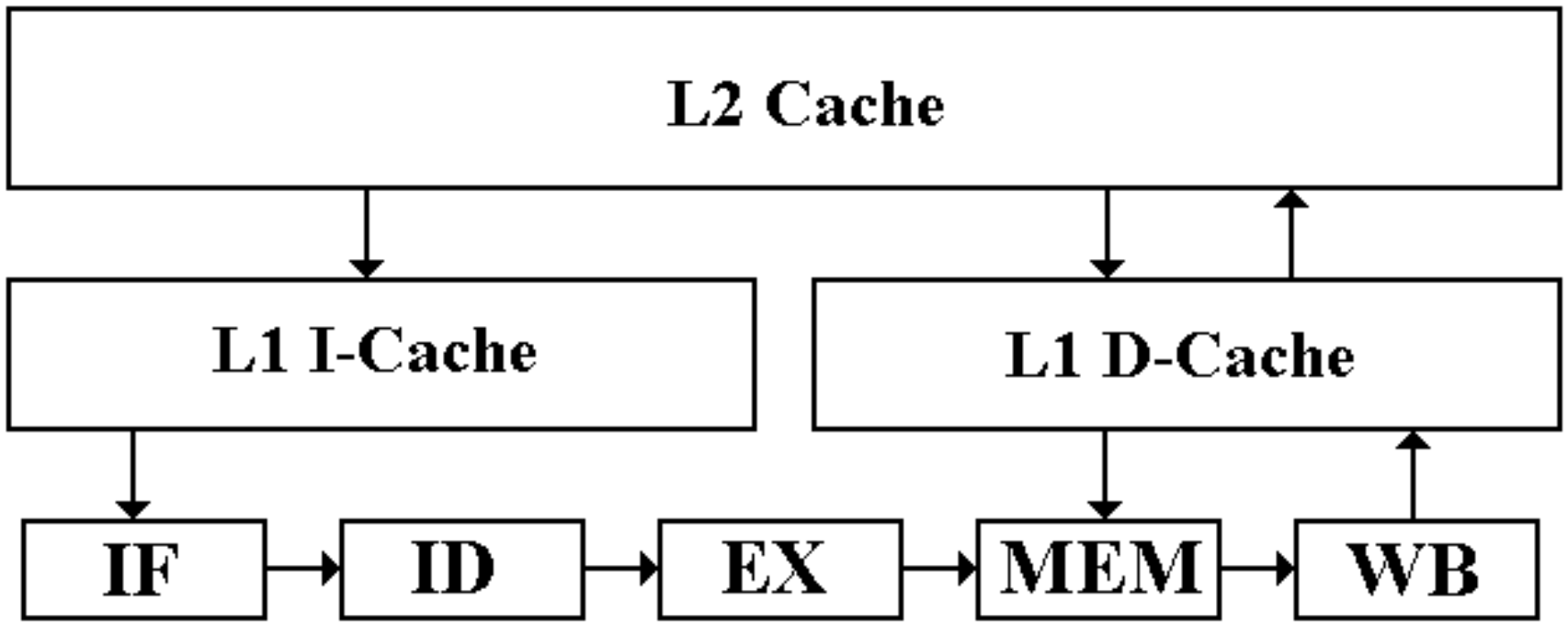
→ 마치 세탁 공정처럼, 여러 옷을 동시에 각기 다른 단계에서 처리
→ 성능을 높이기 위한 병렬성 확보 수단

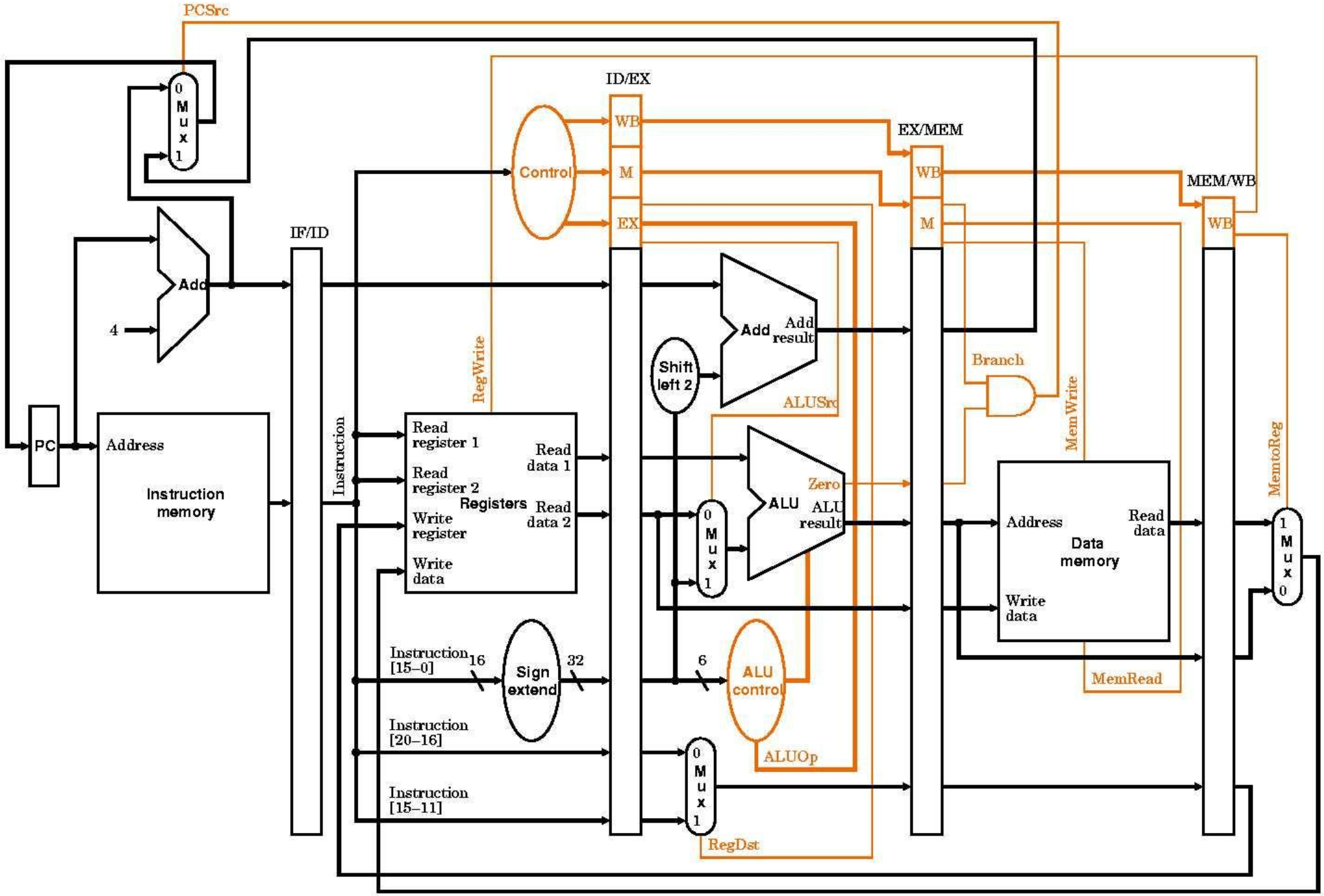


IF: Instruction fetch
ID: Instruction decode and register read
EX: Execute, address generation
MEM: Memory access
WB: Write back to registers

Time Graphs


Clock cycle
1
2
3
4
5
6
7
8
9
add
nand
lw
add
sw
IF
ID
EX
MEM
WB
Latency:
5 cycles


L2 Cache
L1 I-Cache
L1 D-Cache
IF
ID
EX
MEM
WB


PCSrc
M
u
x
0
1
ID/EX
EX/MEM
MEM/WB
WB
M
EX
Control
Add
4
IF/ID
PC
Address
Instruction memory
Instruction
RegWrite
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
Shift left 2
Add
Add result
ALUSrc
Zero
ALU
ALU result
Branch
MemWrite
MemtoReg
Address
Data memory
Read data
Write data
MemRead
Instruction [15–0]
16
Sign extend
32
6
ALU control
ALUOp
Instruction [20–16]
Instruction [15–11]
RegDst

병목(Bottleneck) — 파이프라인이 막히는 이유들

- 메모리 지연 (Memory Latency)
  - 캐시 미스 발생 시 메모리 접근이 느려 명령어 정지
    - → 해결책: 캐시 계층 구조, prefetching
- 데이터 의존성 (Data Hazard)
  - 이전 명령어 결과가 아직 안 나왔는데 다음 명령어가 그 값을 쓰려고 할 때
  - 예: add r1, r2, r3 다음에 mul r4, r1, r5
    - → 해결책: 레지스터 리네이밍, 포워딩(Forwarding)
- 구조적 충돌 (Structural Hazard)
  - 하드웨어 자원 부족 (예: ALU가 하나뿐이라 둘 다 쓸 수 없음)
    - → 해결책: 슈퍼스칼라 구조로 여러 유닛 추가

Superscalar

슈퍼스칼라 — 한 사이클에 여러 명령어 실행

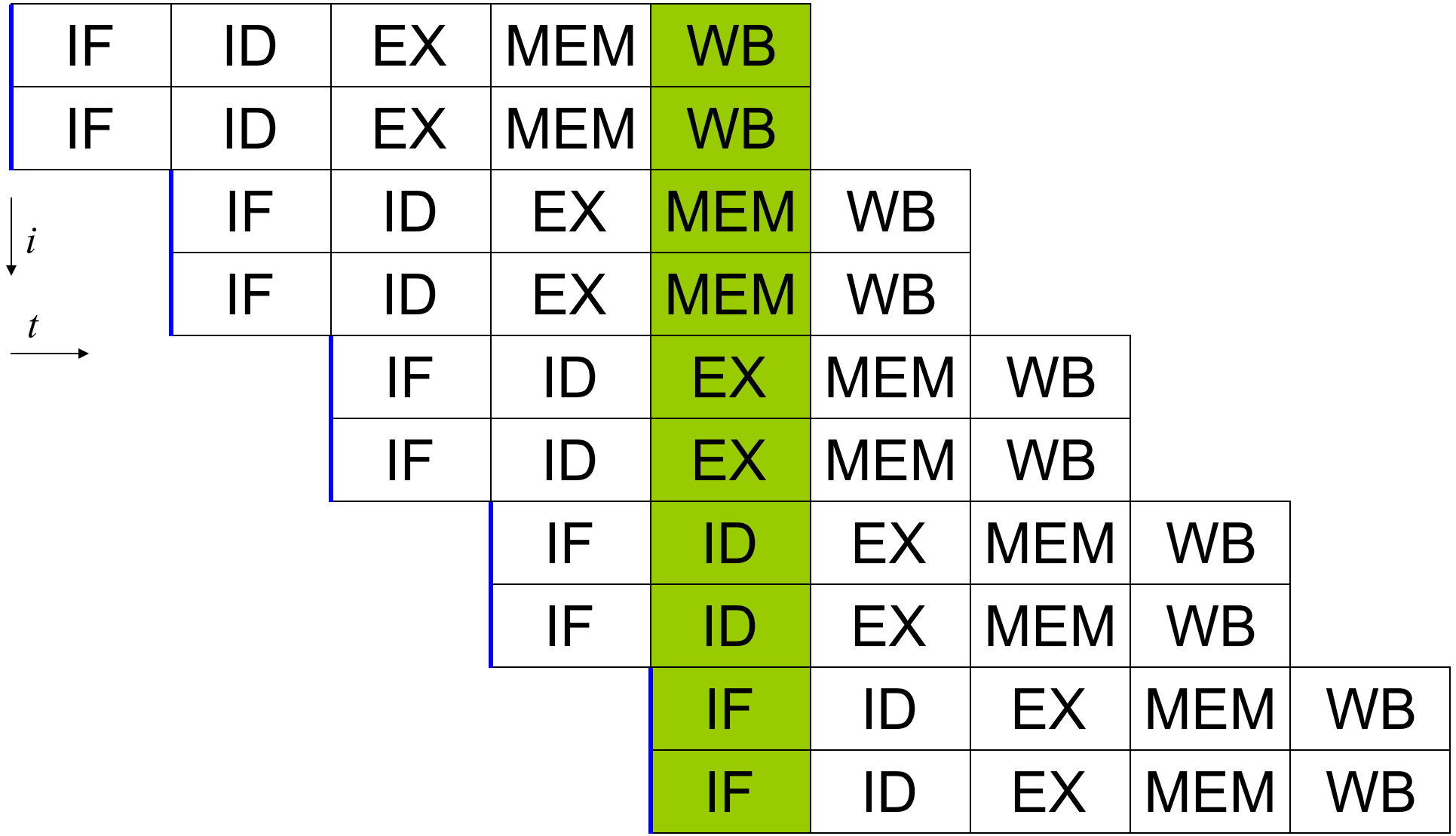
CPU가 한 번에 여러 명령어를 받아 다수의 실행 유닛에 병렬로 배분

특징

- 명령어를 Fetch/Decode 단계에서 여러 개 동시에 읽고 분석
- ALU, FPU, Load/Store 유닛 등을 여러 개 사용
- → 이상적인 경우, 1클럭에 2개 이상 명령어를 실행할 수 있음

대표적 CPU 설계

- Intel Core, AMD Ryzen, Apple M1 등 대부분의 현대 CPU는 슈퍼스칼라 구조 사용
- ARM Cortex-A 시리즈, RISC-V 고성능 코어 등도 다수 사용


IF ID EX MEM WB
IF ID EX MEM WB
IF ID EX MEM WB
IF ID EX MEM WB
IF ID EX MEM WB
IF ID EX MEM WB
IF ID EX MEM WB
IF ID EX MEM WB
IF ID EX MEM WB
IF ID EX MEM WB
i
t

명령어 사이에
데이터, 자원
의존성이
존재하는 경우에는
동시실행에 제한이 있다.

ADD와 DIV 두 가지 명령어를 사용하여 데이터 의존성을 이해해보자

- ADD R1, R2, R3 : R2와 R3를 더해서 R1에 저장한다.
- DIV R4, R1, R5 : R1을 R5로 나누어서 R4에 저장한다.

다음과 같은 값이 주어질 때

R1 = 8
R2 = 1
R3 = 3
R5 = 2

두 명령어를 순차적으로 실행하면?

명령a = ADD R1, R2, R3
명령b = DIV R4, R1, R5
R1 = 8, R2 = 1, R3 = 3, R5 = 2

R1 = ?
R4 = ?

두 명령어를 순차적으로 실행하면?

명령a = ADD R1, R2, R3
명령b = DIV R4, R1, R5
R1 = 8, R2 = 1, R3 = 3, R5 = 2

R1 = R2 + R3 = 1 + 3 = 4
R4 = R1 / R5 = 4 / 2 = 2

두 명령어를 실행 순서를 바꿔 실행하면?

명령b = DIV R4, R1, R5
명령a = ADD R1, R2, R3
R1 = 8, R2 = 1, R3 = 3, R5 = 2

R1 = ?
R4 = ?

두 명령어를 실행 순서를 바꿔 실행하면?

명령b = DIV R4, R1, R5
명령a = ADD R1, R2, R3
R1 = 8, R2 = 1, R3 = 3, R5 = 2

R4 = R1 / R5 = 8 / 2 = 4
R1 = R2 + R3 = 1 + 3 = 4

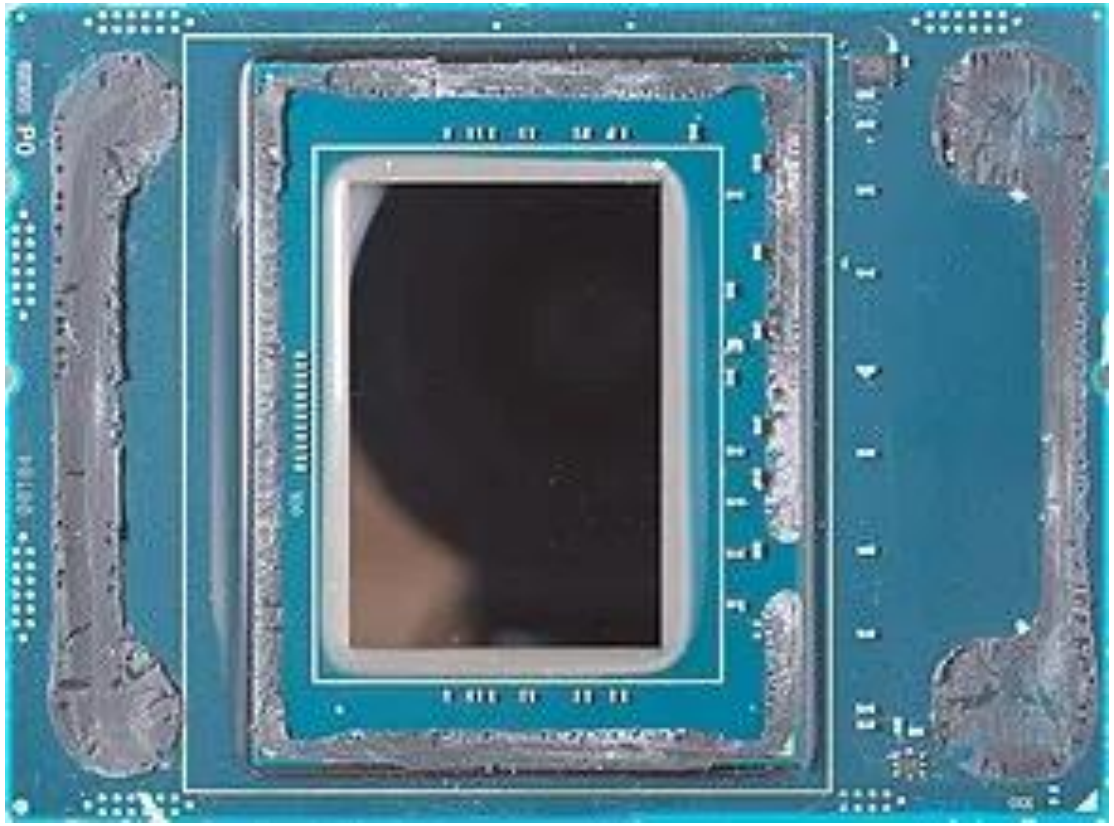
첫 번째 명령어에 의해 값이 정해지는 데이터를
두 번째 명령어에서 읽게 되는 경우에는
두 명령어의 실행 순서가 변경되어서는 안 되며,

이것을 'READ AFTER WRITE 의존성'이라고 한다.

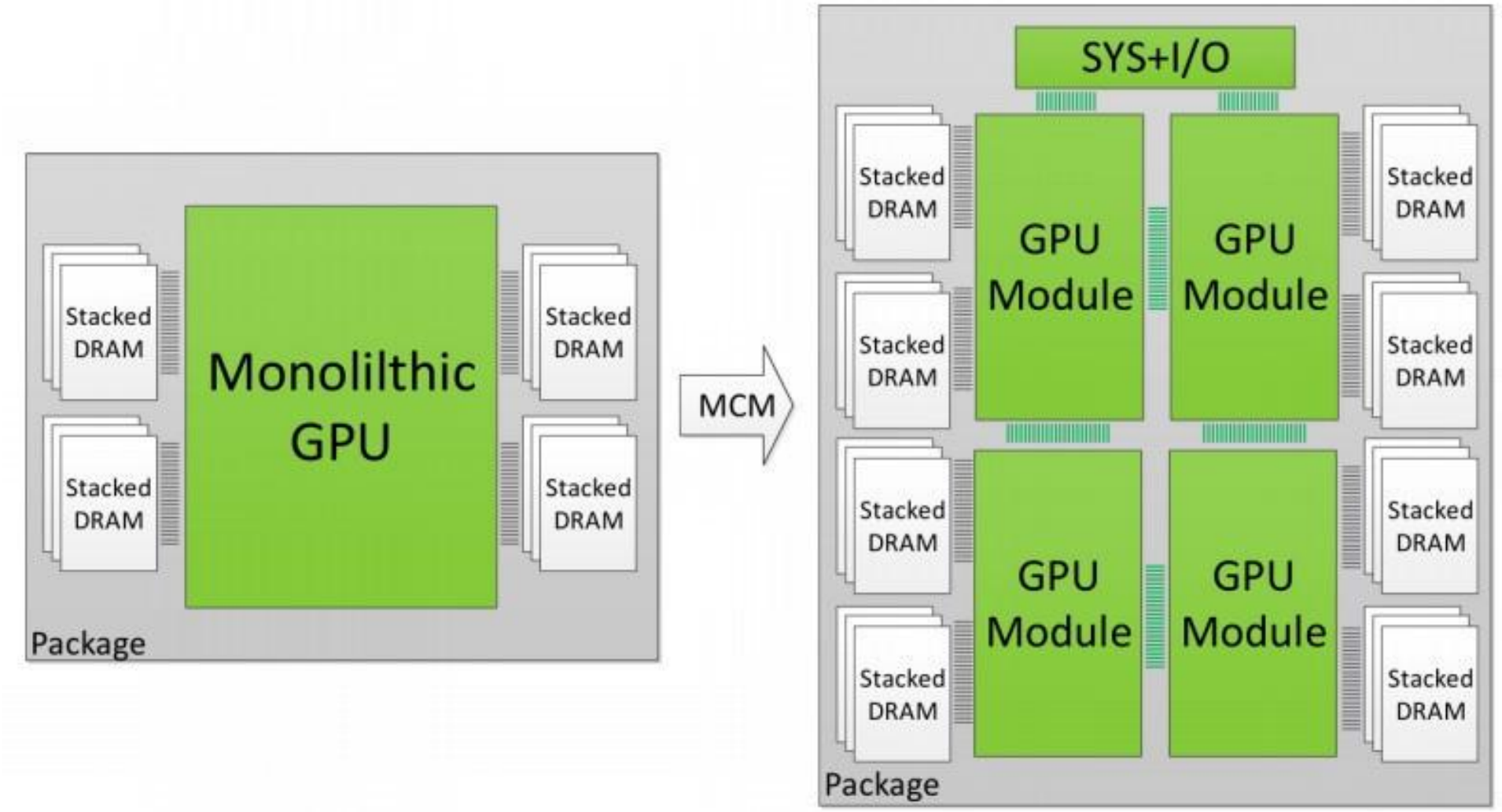
CPU 패키징 & 집적 구조의 주요 유형

구조명	설명	예시
모놀리식 (Monolithic)	하나의 큰 다이에 모든 것을 담은 구조	Intel i7-8700K, Apple A12
칩렛 (Chiplet)	여러 다이를 패키지에서 조합	AMD Ryzen, Intel Meteor Lake
3D 스태킹 (3D Stacking)	칩을 수직으로 쌓음	AMD 3D V-Cache, Intel Foveros
EMIB	칩렛 사이에 실리콘 브릿지를 삽입	Intel Kaby Lake-G
SoC (System on Chip)	CPU + GPU + NPU 등 모든 기능을 한 칩에	Apple M1, Qualcomm Snapdragon
MCM (Multi-Chip Module)	여러 칩을 하나의 기판 위에 배치 → 칩렛의 원형	구형 서버용 칩, 초기 EPYC
Heterogeneous Integration	이기종 코어를 한 패키지에 (CPU + FPGA 등)	Apple M1, Intel Lakefield

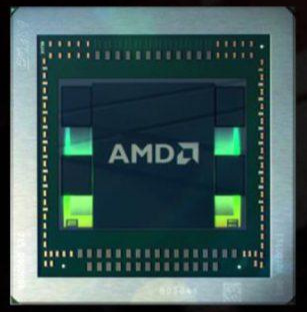
Monolithic



Monolithic -> MCM

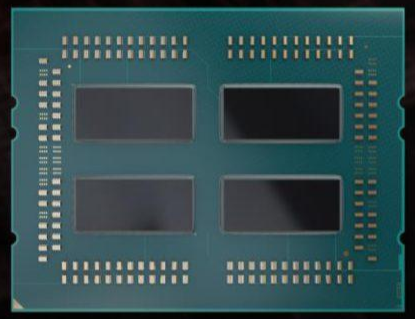


AMD LEADERSHIP PACKAGING



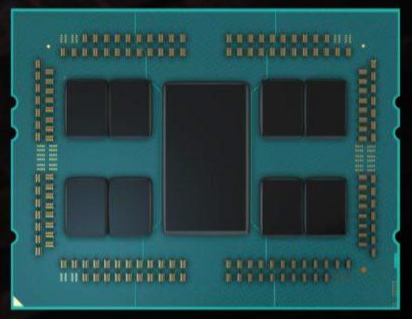
2.5D HBM

2015



MULTICHIP MODULE

2017

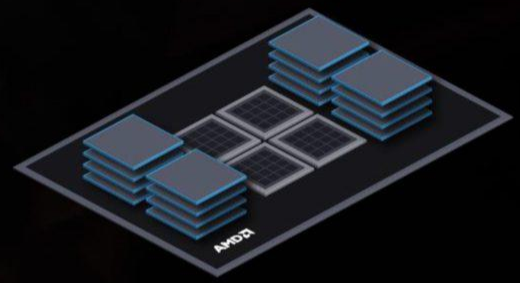


CHIPLETS

2019

>10x

BANDWIDTH DENSITY



"X3D" PACKAGING

(Hybrid 2.5 & 3D)

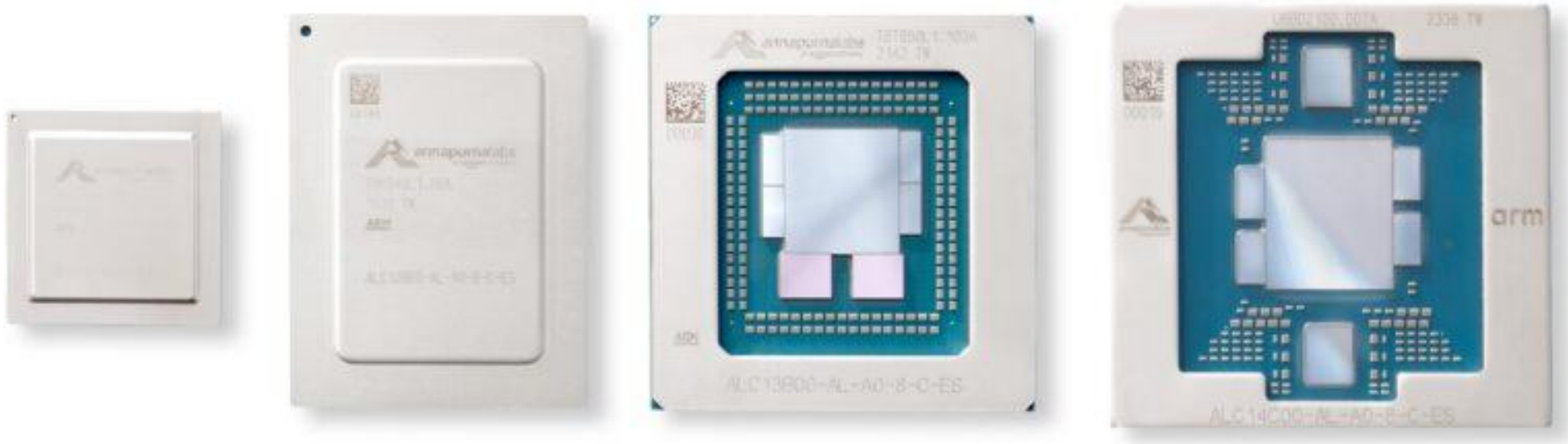
Future

Led Industry in 2.5D & Chiplet Architecture

Aggressive Roadmap for Chiplet & 3D Integration

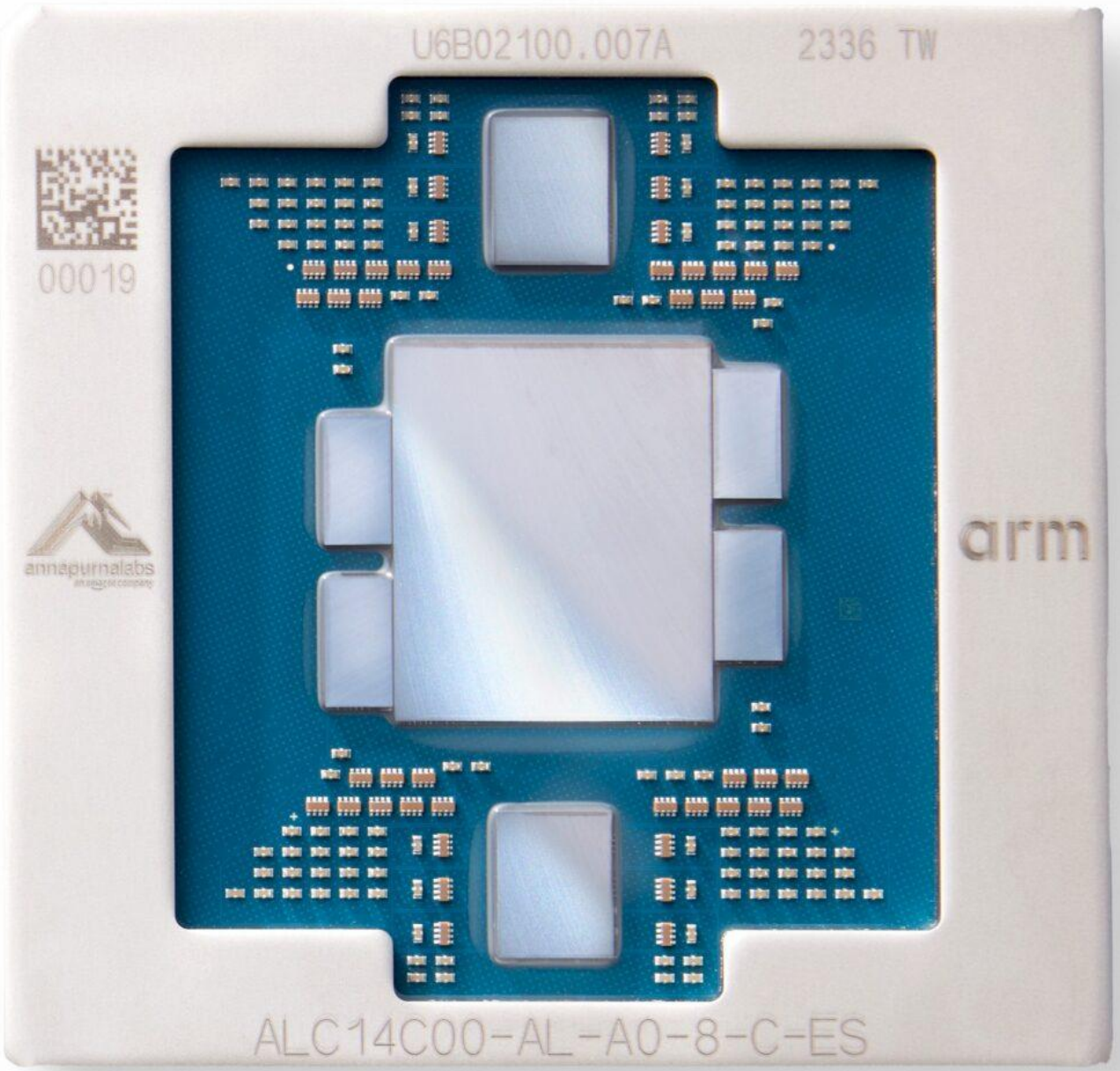
Future Bandwidth Number AMD Estimate

Chiplet



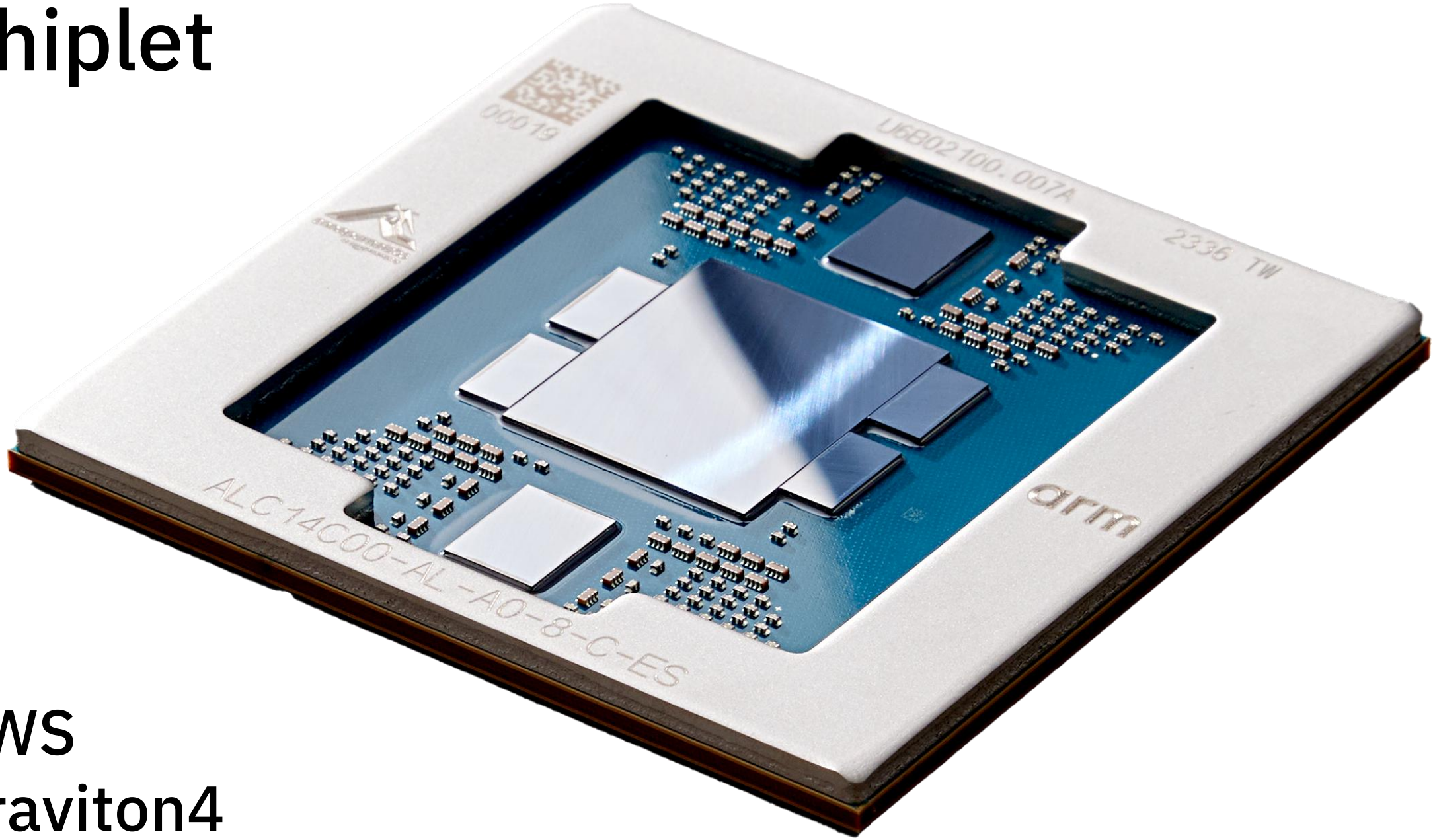
AWS
Graviton4

Chiplet



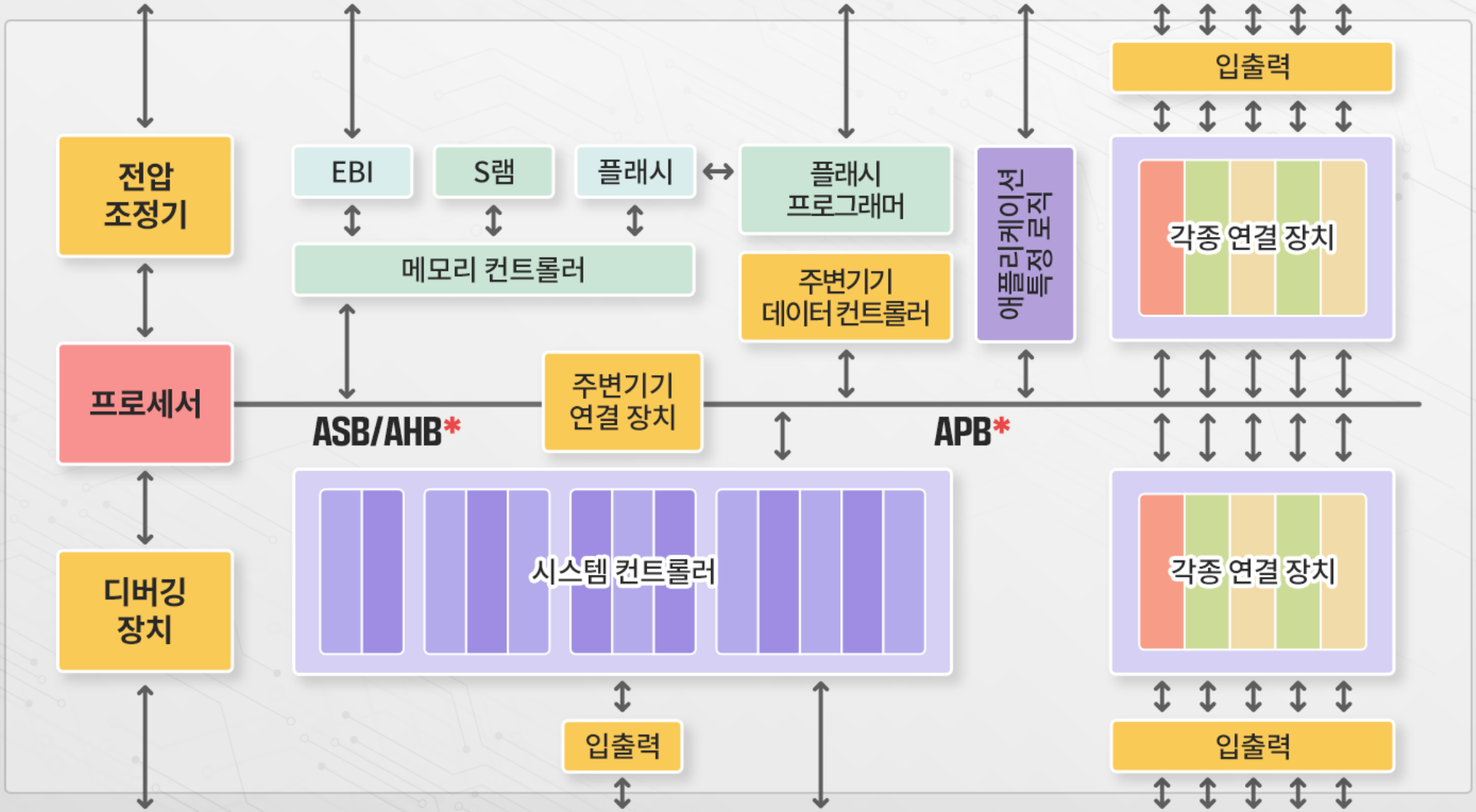
AWS
Graviton4

Chiplet

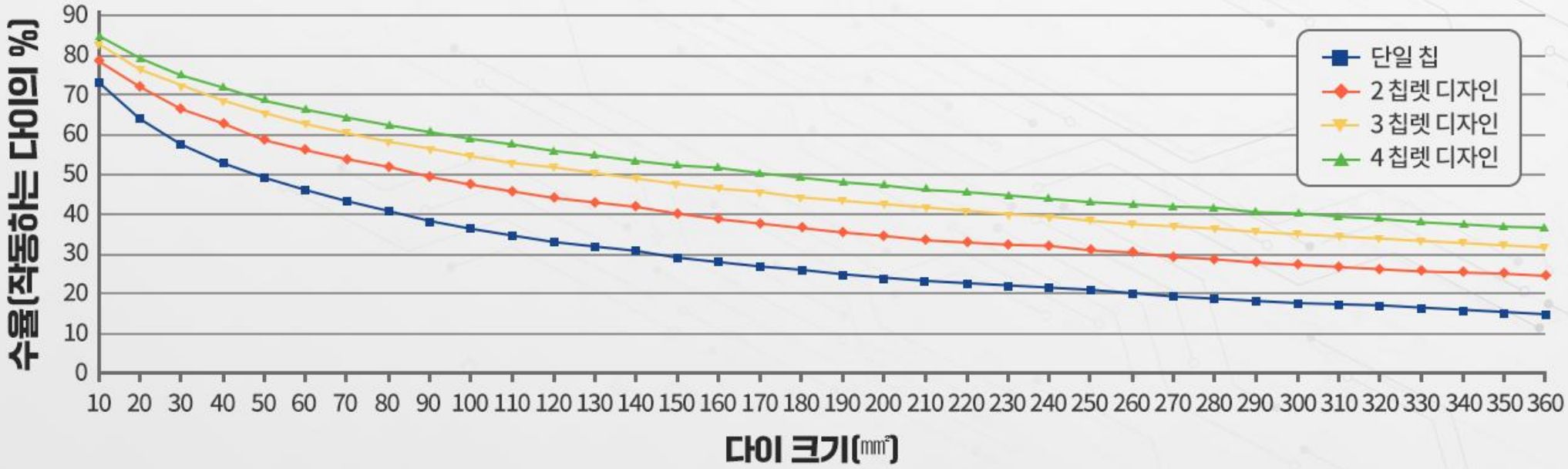


AWS
Graviton4

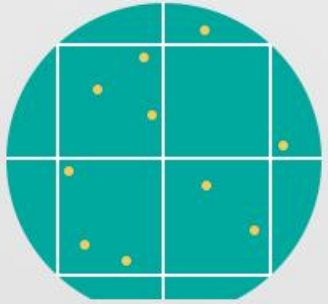
단일 칩 시스템(SoC)의 내부 구성



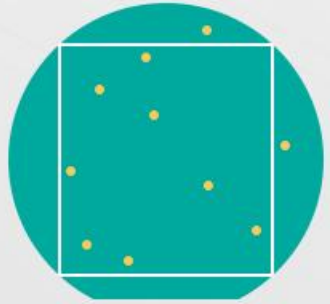
SK hynix NEWSROOM



총 16개의 다이 중
살아 있는 다이 8개
- 수율 50%



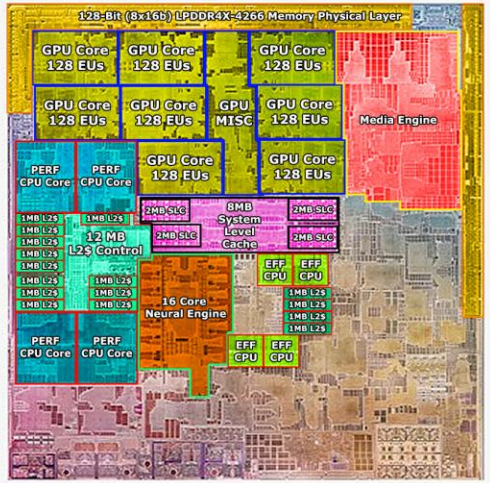
총 4개의 다이 중
살아 있는 다이 1개
- 수율 25%



총 1개의 다이 중
살아 있는 다이 0개
- 수율 0%

SK hynix
NEWSROOM

Heterogeneous integration

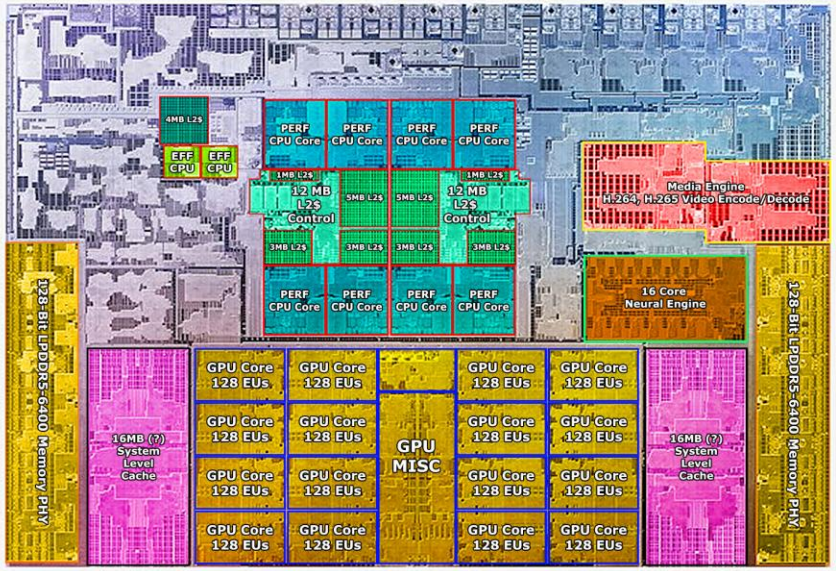


M1

| CPU: 4x Performance Cores, ≤3.2 GHz
+ 4x Efficiency Cores, ≤2.06 GHz

| GPU: 8x GPU Cores (1024 EUs)
24.576 „Threads in-flight"
~1.27 GHz
2.6 FP32 TeraFLOPs
82 GigaTexel/s (64 TMUs)
41 GigaPixel/s (32 ROPs)

| Memory: 128-bit LPDDR4X-4266
68.3 GB/s bandwidth

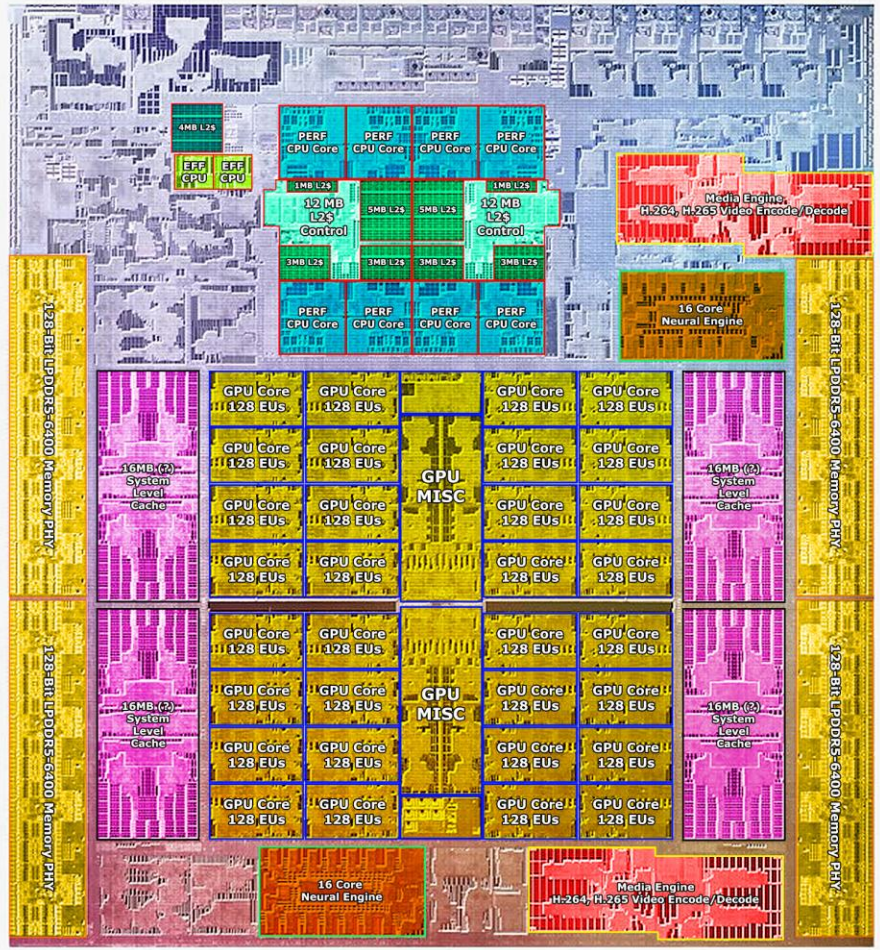


M1 Pro

| CPU: 8x Performance Cores
+ 2x Efficiency Cores

| GPU: 16x GPU Cores (2048 Execution Units)
49.152 „Threads in-flight"
~1.27 GHz
5.2 FP32 TeraFLOPs
164 GigaTexel/s (128 TMUs)
82 GigaPixel/s (64 ROPs)

| Memory: 256-bit LPDDR5-6400
204.8 GB/s bandwidth



M1 Max

| CPU: 8x Performance Cores
+ 2x Efficiency Cores

| GPU: 32x GPU Cores (4096 Execution Units)
98.304 „Threads in-flight"
~1.27 GHz
10.4 FP32 TeraFLOPs
327 GigaTexel/s (256 TMUs)
164 GigaPixel/s (128 ROPs)

| Memory: 512-bit LPDDR5-6400
409.6 GB/s bandwidth

PS5

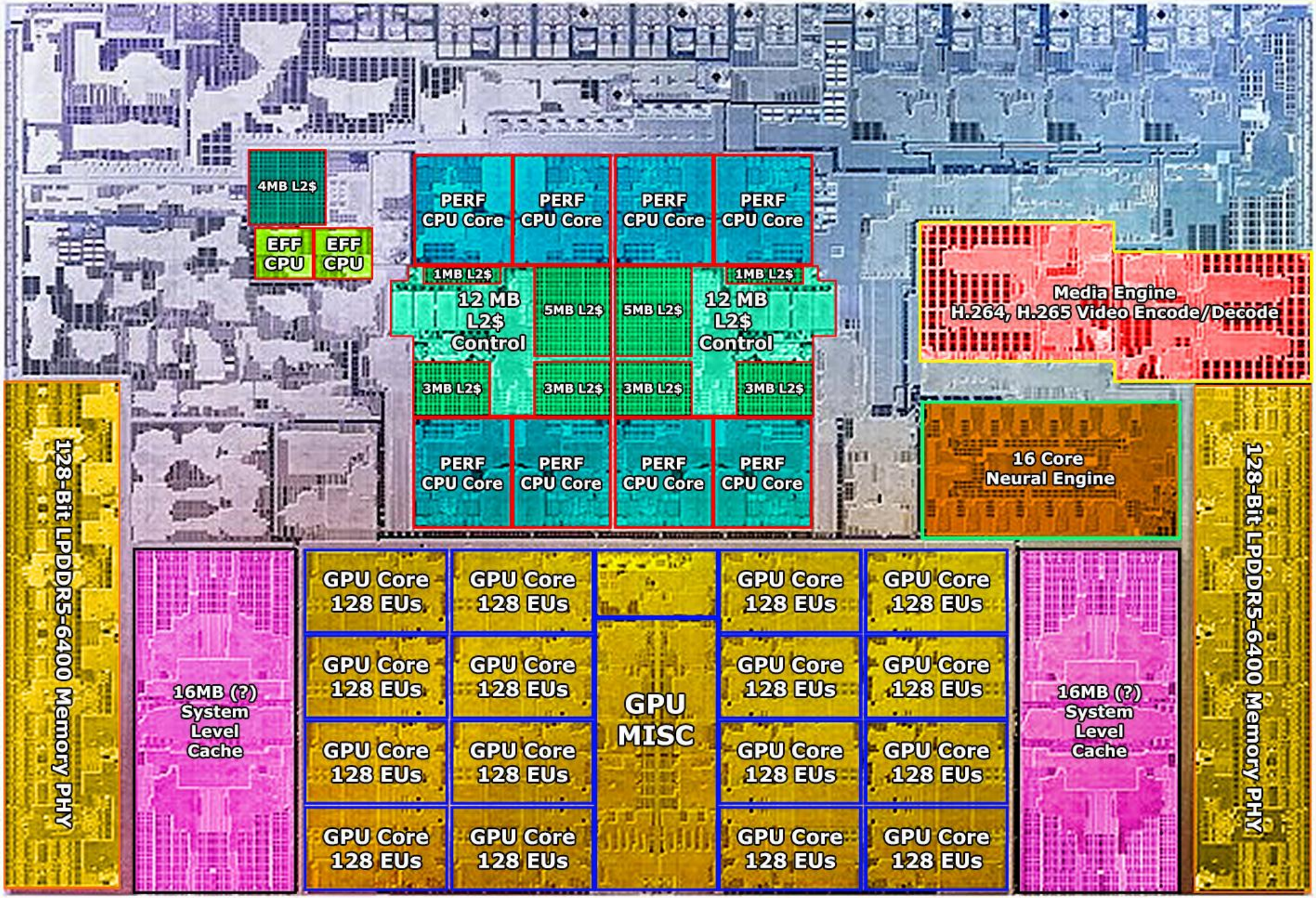
8x Zen2, ≤3.5 GHz

18x Cores (2304 EUs)
92.160 „Threads"
~2.23 GHz
10.28 FP32 TeraFLOPs
321 GTex/s (144 TMUs)
143 GPix/s (64 ROPs)

256-bit GDDR6-14000
448 GB/s bandwidth

Sources: M1 CPU clock frequency from Anandtech, chip images from Apple via Anandtech

Compiled and annotated by Locuza, October 2021


4MB L2$
EFF CPU
EFF CPU
PERF CPU Core
PERF CPU Core
PERF CPU Core
PERF CPU Core
1MB L2$
1MB L2$
12 MB L2$ Control
5MB L2$
5MB L2$
12 MB L2$ Control
3MB L2$
3MB L2$
3MB L2$
3MB L2$
PERF CPU Core
PERF CPU Core
PERF CPU Core
PERF CPU Core
Media Engine
H.264, H.265 Video Encode/Decode
16 Core Neural Engine
128-Bit LPDDR5-6400 Memory PHY
128-Bit LPDDR5-6400 Memory PHY
16MB (?) System Level Cache
16MB (?) System Level Cache
GPU Core 128 EUs
GPU MISC

비유로 기억해보자

구조	비유
모놀리식	한 덩어리 도시락
칩렛	모듈형 반찬 세트
3D 스태킹	도시락을 층층이 쌓음
SoC	올인원 도시락 (밥+반찬+디저트 한 칸에)
EMIB	도시락 속 미세 파이프 연결
이기종 통합	여러 나라 음식이 한 도시락에

실습

CPU 히트스프레더 제거

[과제 안내] 4주차 선각 리포트

주제 : CPU의 성능 향상이 둔화된 이유와 기술적 대안

주제 해설 : 예전엔 "무어의 법칙"에 따라 매년 CPU가 두 배씩 빨라졌습니다.
하지만 요즘은 10% 성능 향상도 어렵습니다.왜 CPU는 더 이상 빨라지지 않는 걸까요?

제출 방법

- 파일명: `홍길동_4주차_선각리포트.pdf`
- 형식: PDF 변환 후 슬랙 `CHIP 톡방`에 제출
- 마감: 다음 모임 전까지
- 다음 모임 : 5월 24일 토요일 22시

다음 시간엔?

우리는 GPU 속 수천 개의 연산 유닛이 어떻게 대규모 병렬 계산을 수행하고,그래픽을 어떻게 실시간으로 처리하며,
그 구조가 어떻게 AI와 게임,
고속 연산의 미래를 이끄는지 탐구합니다.

- GPU 구조와 병렬 아키텍처의 세계로 -