A

SimpleRisc ISA

In this book, all the examples that use assembly code have been written in the *SimpleRisc* assembly language. It is a toy assembly language that was originally introduced by Sarangi [Sarangi, 2015].

Inst.	Format	Inst.	Format
add	add rd, rs1, (rs2/imm)	lsl	lsl rd, rs1, (rs2/imm)
sub	sub rd, rs1, (rs2/imm)	lsr	lsr rd, rs1, (rs2/imm)
mul	mul rd, rs1, (rs2/imm)	asr	asr rd, rs1, (rs2/imm)
div	div rd, rs1, (rs2/imm)	nop	nop
mod	mod rd, rs1, (rs2/imm)	ld	ld rd, imm[rs1]
cmp	cmp rs1, (rs2/imm)	st	st rd, imm[rs1]
and	and rd, rs1, (rs2/imm)	beq	beq offset
or	or rd, rs1, (rs2/imm)	bgt	bgt offset
not	not rd, (rs2/imm)	b	b offset
mov	mov rd, (rs2/imm)	call	call offset
ret	ret		
$rd \rightarrow$ destination register id, $rs1 \rightarrow$ first source register $rs2 \rightarrow$ second source register, $imm \rightarrow$ immediate			

Table A.1: The *SimpleRisc* instruction set

Its main features are as follows.

1. It is a 32-bit ISA. It has 16 registers numbered $r0, r1, \ldots r15$.
2. The first 14 registers are general purpose registers. $r14$ is the stack pointer; it is also referred to as *sp*.
3. $r15$ is the return address register (also referred to as ra).
4. There is a special $flags$ register that is set by the *cmp* (compare) instruction. Later conditional branches use it to make their decisions.

5. It has 21 instructions. The format of the instructions is shown in Table A.1.

6. Most arithmetic and logical instructions are in the 3-address format. The destination comes first, and the sources come later.

An example program to compute the factorial of a number stored in $r0$ is as follows.

```
.factorial:
/* input stored in r0, output in r1 */
/* Assume that the value in r0 is greater than 0 */
mov r1, 1   /* prod = 1 */

/* loop */
.loop
        /* check the iterator */
        cmp r0, 0
        beq .exit

        /* multiply */
        mul r1, r1, r0  /* prod = prod * r0 */

        /* loop */
        sub r0, r0, 1 /* r0 = r0 - 1 */
        b .loop

.exit:
```

B

Tejas Architectural Simulator

B.1 Overview

For proposing and evaluating architectural features, designers and researchers typically use an architectural simulator. It is a large software program that simulates all the features of a processor including the memory system, on-chip network, and off-chip DRAM. We can think of it as a *virtual processor* that can run a full program including an operating system and the programs running on it. Along with providing overall execution statistics such as the total number of simulated cycles, cache miss rates, energy, and power consumption values, we can also use architectural simulators to implement new protocols and processor designs. We can accurately assess their advantages and overheads. Note that in this case, the main task is to just simulate the overheads in terms of time and power while ensuring that the program running on the processor executes correctly. Correctness of the program is not being verified here.

As compared to implementing novel features in a hardware description language, using an architectural simulator is much faster. Its simulation speed is typically 100 times more, does not require sophisticated software or FPGA boards, and can also be easily parallelized. There are four types of commonly used architectural simulation methods.

Cycle-accurate Simulation Such simulators are typically tightly coupled with the real hardware. They model latencies exactly. It is expected that the time a program will take to run on an architectural simulator will be the same as the corresponding hardware implementation (in terms of simulated execution cycles). Such simulators are typically very slow, and we also need access to a hardware implementation to calibrate the simulator.

Cycle-approximate Simulation Simulators in this category are not coupled with a specific hardware implementation. They assume a piece of generic hardware and provide numbers that are internally consistent. They are much faster and are the most popular as of today.

Sampled Simulation In this case, we do not simulate all the instructions. We separate the actual execution of the instructions from the simulation. We execute all the instructions; however, we only simulate small sequences of dynamic instructions. These sequences are periodically extracted from the running program. The final simulation results are obtained by extrapolating the results obtained by considering the size of the sequences and the total number of dynamic instructions in the program.

Statistical Simulation Such approaches typically extend sampled simulation to incorporate statistical and machine learning models. We can simulate small snippets of the execution or collect a few metrics from hardware performance counters, and then try to use a learned model to predict the rest of the outputs of the simulation. This method admits statistical approaches and machine learning based techniques that try to estimate the final execution statistics.

B.2 Tejas Architectural Simulator

Let us now describe the Tejas architectural simulator [Sarangi et al., 2015], which is a cycle-approximate simulator and can simulate regular programs, operating systems, Java programs, and CUDA programs. It can be freely downloaded from `http://www.cse.iitd.ac.in/tejas`.

B.2.1 Design of Tejas

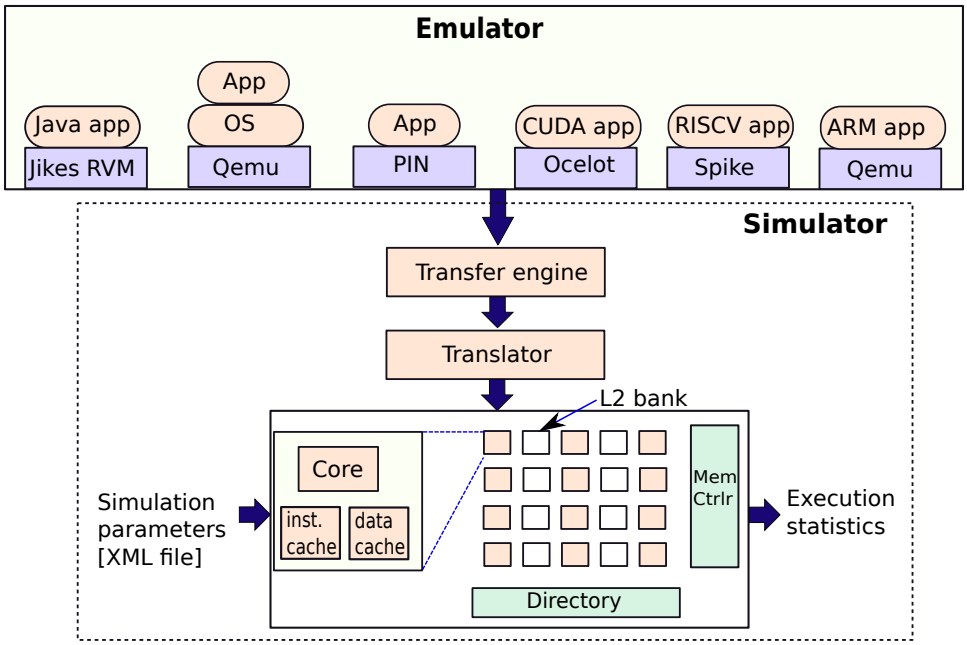

Figure B.1: Design of Tejas

While talking about an architectural simulator, we need to distinguish between two key concepts: *emulator* and *simulator*. The emulator executes the program instruction by instruction. This can either be a sequential program or a parallel program. In the latter case the emulator spawns parallel threads. The emulator is presumed to be always correct. It collects *instruction traces*, which include the PC of the instruction, its contents, the branch direction, and the load/store addresses. The instruction traces are sent to the simulator, which simulates the behavior of the processor including its timing and power. Specifically, the simulator is responsible for implementing the pipeline, NoC, caches, and the entire memory system.

Tejas can use different emulators as shown in Figure B.1. By default, it uses Intel PIN [Luk et al., 2005], which runs x86 binaries and collects traces. Tejas supports other emulators as well such as the Jikes virtual machine [Alpern et al., 2005] for Java programs, the Qemu [Bellard, 2005] virtual machine

for full fledged operating systems, Ocelot [Farooqui et al., 2011] for CUDA programs, Spike for RISC-V programs, and Qemu's ARM version for ARM programs. The traces have the same high-level format.

Subsequently, the transfer engine is used to transfer traces to the simulator, which is written in Java. It is a separate process. Standard IPC (inter-process communication) mechanisms such as shared memory, sockets, and pipes can be used. Once the traces reach Tejas, it simulates the synchronization behavior of threads, and once a thread is unblocked, its traces are transferred to the Translation Engine. The Translation Engine has separate modules for each ISA. Tejas defines a virtual instruction set known as VISA (virtual ISA). Regardless of the original ISA, its traces are internally converted to the VISA ISA. The instructions are sent to the pipeline simulator for the corresponding core. This allows us to design a generic core that is ISA-independent.

Tejas simulates the pipeline within each core, its caches, the NoC, L2/L3 banks, the directories, and the memory controllers that send messages to off-chip DRAM modules. The simulator is fully configurable. Its input is in the form of an XML file, which includes the configurations of all the hardware structures and the number of instructions that need to be simulated. The final output includes detailed statistics for each hardware structure, the number of simulated cycles, the details of stalls, and power consumption statistics. Tejas includes the Cacti [Muralimanohar et al., 2009] and McPat [Li et al., 2009] tools to simulate power consumption.

B.2.2 Semi-event Driven Simulation

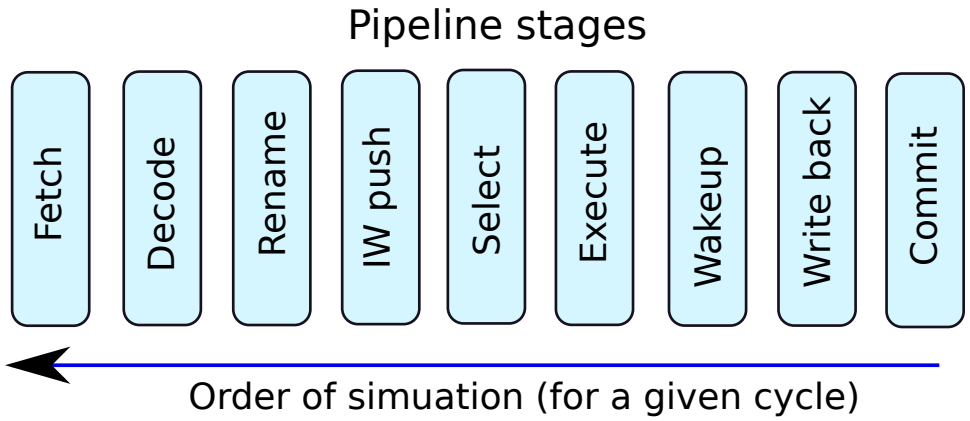

Figure B.2: Sequence of pipeline stages that are simulated

There are two ways of performing architectural simulations. The first is an *iterative approach*, which is primarily used to simulate in-order processors. For a given cycle i, we start from the last stage (write-back stage), find instructions that are ready to execute on it in cycle i, and simulate their execution. Then we move to the second last stage and do the same. This process continues till we reach the first stage. This process can be extended to out-of-order processors; however, in this case, we need to deal with large non-deterministic delays and thus the bookkeeping overhead is substantial. Hence, this approach is in general considered to be a fast scheme when it comes to simulating the pipeline only.

Consequently, to simulate the memory system and NoC of OOO processors, we typically use an event queue (event driven model), which is a priority queue ordered according to the time at which events get *activated*. Consider an example. Let's say that the response to a memory request is expected 10 cycles later. If the request was issued in cycle 100, the response needs to be processed in cycle 110. We then add an event to the event queue with its timestamp set to 110.

The general idea is that in cycle i, we fetch all the events from the event queue that have a timestamp equal to i. The assumption is that in previous cycles, all the events for this cycle would have been added

by all the event producing units. While processing an event, we may insert new events into the event queue. Once all the events for cycle i have been processed, we move to the next event in the event queue (in increasing order of timestamps). This approach is more flexible than the iterative approach, yet it is far slower.

Tejas thus uses a hybrid approach known as a semi-event driven model. It uses the iterative approach to simulate the traditional OOO pipeline as shown in Figure B.2. Here also, we start from the commit stage and work our way back to the fetch stage. For the memory system, NoC, directory, and memory controllers that have rather non-deterministic delays, the event queue based mechanism is used. This design strategy provides the best of both worlds.

B.2.3 Optimizations and Corner Cases

To sustain a simulation throughput of 100 KIPS (100 kilo instructions per second), we need a very efficient memory manager. For every instruction and every operand, we cannot allocate memory on the heap. Given that Java uses a garbage collector, the overhead of managing so many objects will become prohibitive. Hence, Tejas uses pools – set of pre-allocated objects. Whenever we need an object of a certain type, we just fetch it from its pool, and for de-allocating an object, we return it to its pool.

Since an architectural simulation approximates a real execution, it also needs to take care of corner cases and race conditions. Simulating relaxed memory models, managing MSHRs, and simulating cache coherence are particularly complex because of the large number of intermediate states. For the purposes of architectural simulation, we need to focus on the common case primarily because we are simulating for timing and power. Let's say, if a case arises once every 100,000 cycles, we need not handle it in a special way. We can just use a brute force solution, where for example we can disable parallel updates to the same hardware structure by using locks or just forcibly fix the global state.

Tejas tries to reuse event objects as much as possible. A core event can lead to a cache event, which can lead to an event destined to the memory controllers. Instead of creating separate events, Tejas uses the same event and keeps on changing its type and its fields. The same event is reused for completing subtasks of a request in the core, caches, and the memory system.

B.2.4 Parallelization

Tejas has been parallelized by assigning different cores to different threads [Malhotra et al., 2017]. In this case, we cannot have a global notion of time nor afford a global event queue. Threads have a local notion of time, where the timeline is viewed as an array of slots. Consider an example. Let's say that if a message is sent from unit i to unit j at time 10 (local time of i), it would have been processed at $t = 15$ at unit j, if they shared a global clock. In this case, we search for free time slots at or after $t = 15$ at unit j (as per its local time). The request is processed at the earliest such time slot. We thus need to maintain a slot array for each thread that can be updated in parallel. The authors use lock-free data structures to implement such a fast, parallel, and scalable slot array.

Another approach is as follows. Emulation is typically 1-2 orders of magnitude faster than simulation. If we need to simulate a billion instructions, we can start 10 emulators, and move each one of them very quickly to a point after 100 million, 200 million instructions and so on. Then we pair a simulation thread with each emulator and begin simulating. In this case, we just simulate 100 million instructions per thread and finally combine the results. Disregarding the time it takes the emulators to reach the starting points, we can obtain a roughly 10X speedup here. The problem is that other than the first thread, the rest of the threads will not be starting from the correct architectural state. This can be solved by including a small *warm up* phase – simulate 10-25 million instructions before the starting point. The ideal speedup for this example is 10X, in practice it is much lower because of the overheads of parallel execution, memory contention, overhead of threading, and the time spent in warm-up phases.

B.2.5 Evaluation

Tejas has been validated against native hardware, and the error is limited to 1-11% for the sequential SPEC CPU2006 (`http://www.spec.org`) benchmarks and 4-33% for parallel benchmarks (Splash2 suite [Woo et al., 1995]). The errors for architectural simulation are typically in that range. The main aim is to ensure that the numbers are internally consistent.

Also note that we typically simulate the single-threaded SPEC benchmarks either individually or as an ensemble (bag of tasks: one thread mapped to each core), when we wish to simulate a set of sequential workloads. For parallel workloads, we normally use the Splash 2 [Woo et al., 1995] and the Parsec benchmark [Bienia et al., 2008] suites. For getting stable results, it is a good idea to simulate at least a billion instructions from each distinct program phase. Program's typically exhibit phase behavior, where their behavior remains stable for a period of time, and then as they move to a different region of code, their behavior changes, yet remains stable for some time. We need to ensure that our simulation captures all the phases and the results attain their steady state values.

C

Intel Processors

C.1 Sunny Cove Microarchitecture

Intel released the details of the Sunny Cove microarchitecture in December 2018. It is meant to be fabricated using a 10 nm process and will be the architecture of its state-of-the-art server chips for the next few years (as of 2020). The aim was to increase the IPC by increasing the fetch and issue widths, and improve the performance of cryptographic applications by adding a set of custom ISA extensions.

C.1.1 ISA Extensions

Intel added a few new instructions to the 512-bit SIMD AVX instruction set, which primarily focused on the performance of cryptographic operations. The reason is that gradually, security is becoming a first-order concern in the design of processors, and thus it is necessary to support a wide variety of cryptographic operations such as AES and primitives that use Galois field arithmetic.

Improvements in the memory system target large shared memory systems and NVM memory such as Intel's 3D XPoint memory. It was necessary to add support for large physical and virtual address spaces because now the physical memory space can be extended by adding NVM modules. Additionally, the architecture has better support for cloud computing by having dedicated storage for storing up to 32K encryption keys; these can be distributed to individual virtual machines or applications.

C.1.2 Processor Design

Front End

The design of the front end of the processor, which includes the fetch logic, branch predictors, micro-op caches and the decoder has not been disclosed publicly. Let us thus explain the design of the Intel Skylake architecture, which precedes the Sunny Cove architecture.

Intel Skylake has an 8-way 32 KB L1 i-cache that provides 16 bytes per cycle. After pre-decoding, the instructions are stored in a 50-entry instruction queue. Subsequently, five macro-instructions (variable length CISC instructions) are sent to the decode unit that contains five decoders. There are four simple decoders and one complex decoder. A complex decoder typically produces multiple micro-ops (μOPs) for a single macro-instruction. Overall, the decoders can supply 5 μOPs per cycle. There are other sources of μOPs as well. Intel has a μOP cache that is connected to the branch predictor. It can supply

6 μOPs per cycle, and the microcode ROM can supply 4 μOPs per cycle (refer to Figure C.1). The microcode ROM is used to translate very long and complex CISC instructions. These μOPs then enter the decode queue. Every cycle, the decode queue can send 6 μOPs to the rename table and ROB.

These μOPs are subsequently renamed, added to the ROB, and physical registers are allocated to hold their results. In total, Intel Skylake can send up to 8 such μOPs to the scheduler (instruction window along with the wake-up/select logic). An astute reader will not that the dispatch bandwidth is more than the decode bandwidth. This is a common feature in advanced processors where internal stages have a higher bandwidth to sustain peaks in ILP. This however does necessitate additional buffers between the decode and dispatch stages.

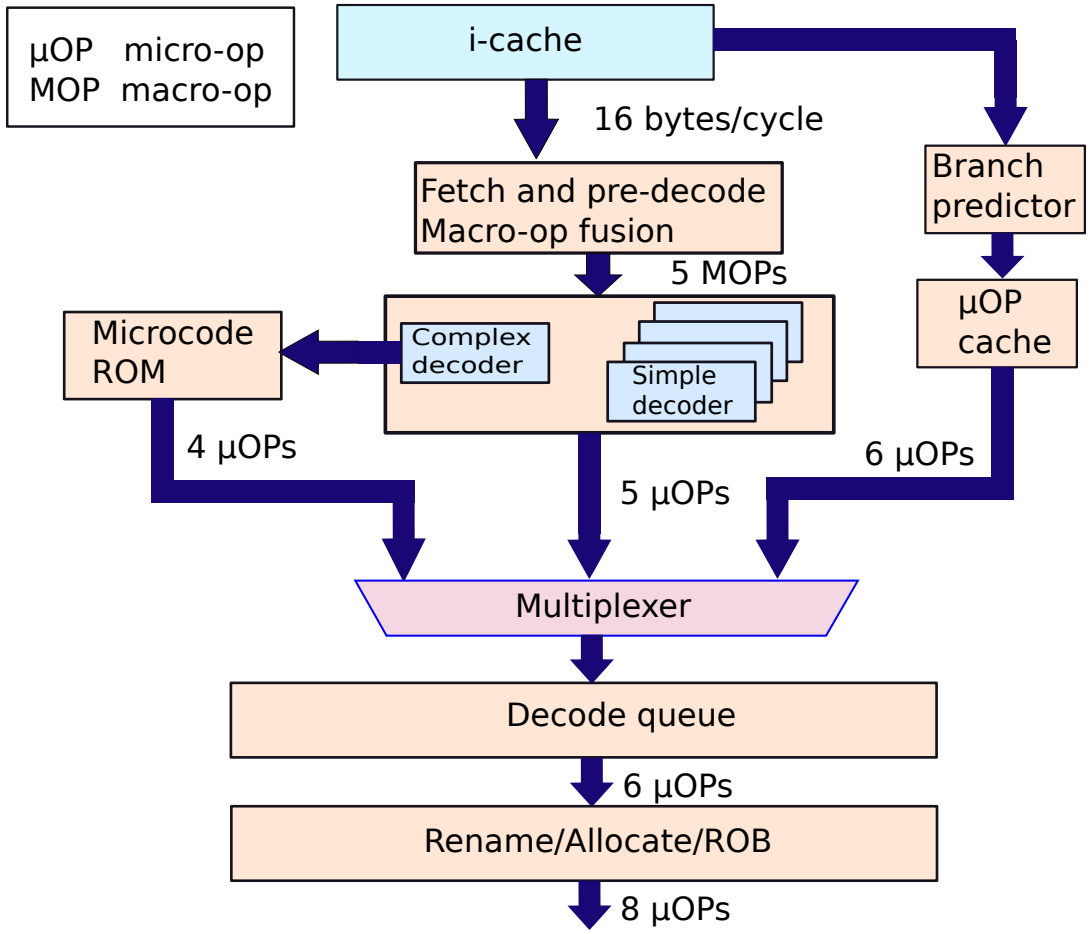

Figure C.1: Front end of the Skylake microarchitecture

Back End

Figure C.2 shows the backend architecture [Kanter, 2019] of Intel Sunny Cove. Sunny Cove's scheduler is split into multiple reservation stations. The reservation stations are connected to a set of execution ports that are in turn connected to a set of functional units. For ALU operations, there are two clusters of functional units: one integer cluster and one vector/floating point cluster. A key feature of this class of architectures is the functional unit for the *LEA* (load effective address) instruction, which transfers the computed memory address to a register instead of the memory contents. It is needed while making

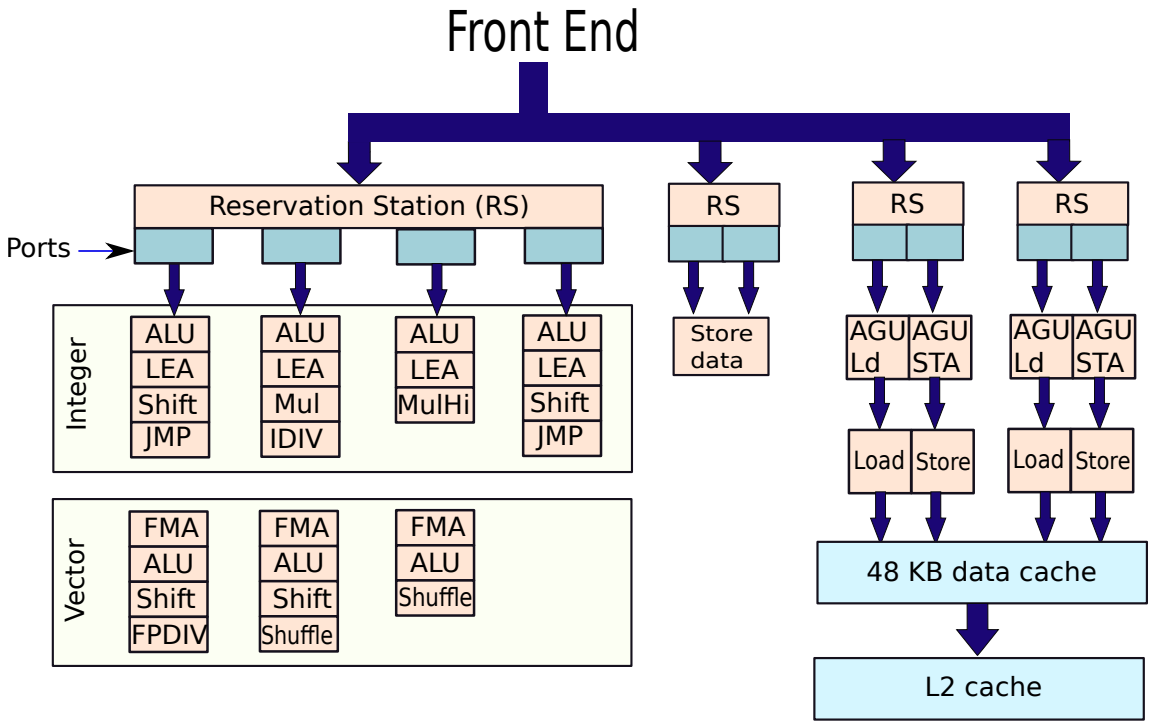

Figure C.2: Back end of the Sunny Cove microarchitecture

indirect accesses via pointers. To get a better understanding of the instruction types shown in the figure, the reader is requested to go through Intel's x86 programming manuals.

The floating point (FP) cluster supports regular arithmetic operations. Also note the functional unit for the shuffle operation that allows us to permute words in a 512-bit SIMD (SSE or AVX) register.

The architecture has six ports for load and store operations that have different functions. The aim was to be able to perform two loads or two stores per cycle. Hence, Intel added 4 AGUs (address generation units): two for loads and two for stores. The role of an AGU is to simply generate the memory address of a load or a store (it is basically an adder). Additionally, there are two ports for storing data in the write buffers or the L1 cache. The L1 cache is a 48-KB cache that is connected to a large L2 cache, which is smaller for desktop processors and much larger for server chips. Finally, note that most processors in this class use a 2-level TLB: an L1 TLB and an L2 TLB. This minimizes the TLB miss rate.

It is important to note that most processor vendors typically do not disclose the exact sizes of the units and the bit widths of the ports. Nevertheless, in an architecture such as Sunny Cove it is expected that the SIMD functional units will be able to handle 512-bit data in one go, and furthermore the processor should be able to execute at least a few vector (512-bit) loads and stores in a cycle.

C.2 Tremont Microarchitecture

We have discussed the Intel "Cove" series of microarchitectures that are predominantly aimed towards the server market. Let us now discuss the "Mont" series, which are processors that are aimed for the laptop and mobile markets. Here, of course, the main aim is power efficiency and getting the maximum amount of performance within a limited area and power budget.

Figure C.3 shows the microarchitecture of the Tremont core [Halfhill, 2019]. It is an out-of-order

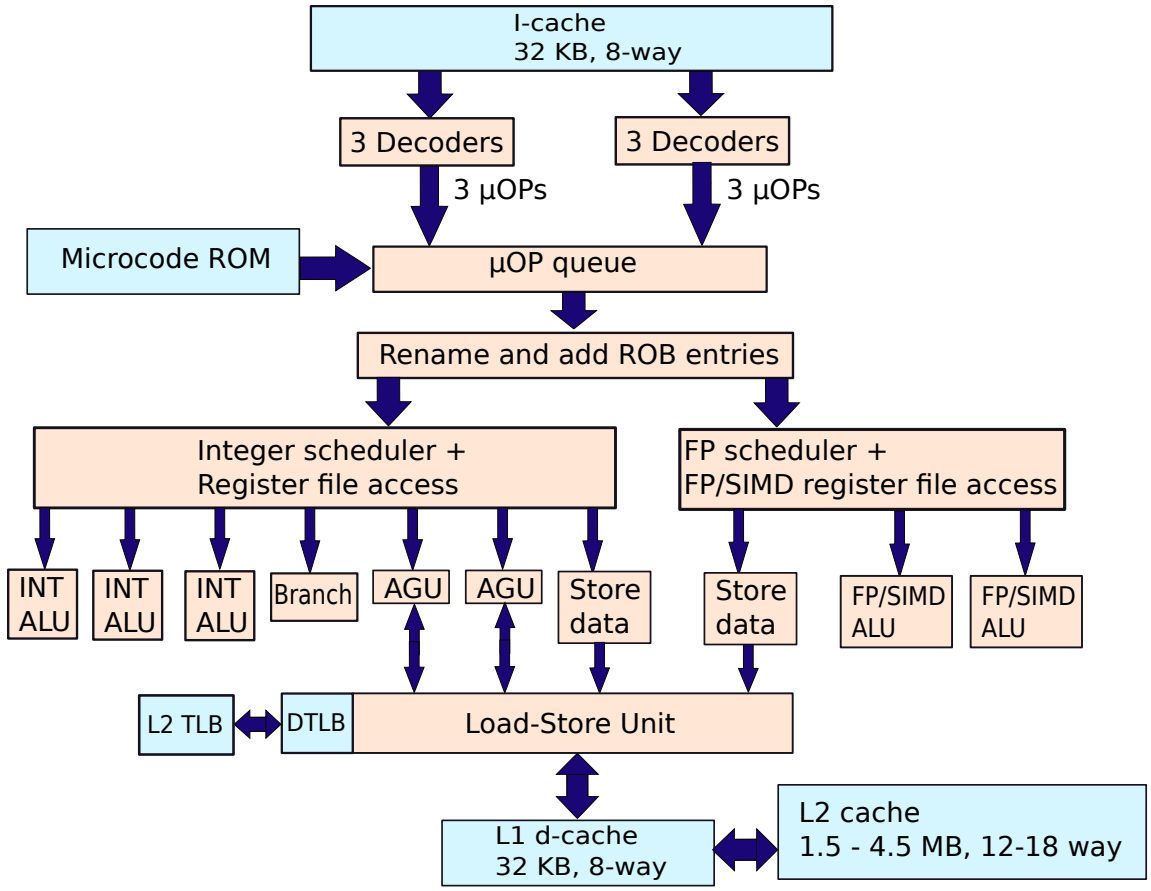

Figure C.3: The microarchitecture of the Tremont core

architecture that supports a single thread at a time. In comparison, most high-end server microarchitectures such as Sunny Cove support multithreading at the hardware level and are able to partition the resources among different threads running in parallel. However, this is not very important in the market segment that Tremont targets.

Now let us explain the microarchitecture of the Tremont core. We start out with a 32-KB 8-way instruction cache that feeds two sets of decoders. Each set contains three simple decoders; these two sets of decoders run in parallel. This is a problem for a complex instruction set such as x86 because instruction boundaries are not known in advance; we need to sequentially read all the instructions. There are several standard ways of solving this problem. The first is that we can do pre-decoding and store the instruction boundaries within the i-cache lines. This will allow such parallel decoders to quickly move to the right starting point. The other approach is that we start from a safe point such as a branch target or try to guess the beginning of an instruction using speculative techniques.

These two decoders feed 6 μOPs to the ROB and rename tables. After renaming, they enter one of the eight schedulers (instruction window + wakeup/select). The schedulers are connected to the register files and a set of 10 functional units. The processor has three integer ALUs, one branch unit, two address generation units (AGUs), and one store unit. The load store unit (LSU) contains the LSQ and also interfaces with the level 1 and level 2 TLBs.

This architecture also has a floating point (FP) scheduler. It is connected to two FP/SIMD ALUs and a store unit.

The L1 cache in this design is an 8-way 32 KB cache and the L2 cache can vary from 1.5-4.5 MB (12-18 ways).

As compared to the Sunny Cove microarchitecture, in this design, the size of the fetch unit is reduced, there are fewer ALU units, and we have fewer functional units that are involved in memory accesses – fewer AGUs and store units. This is thus a smaller and more power efficient design.

C.3 Lakefield Processor

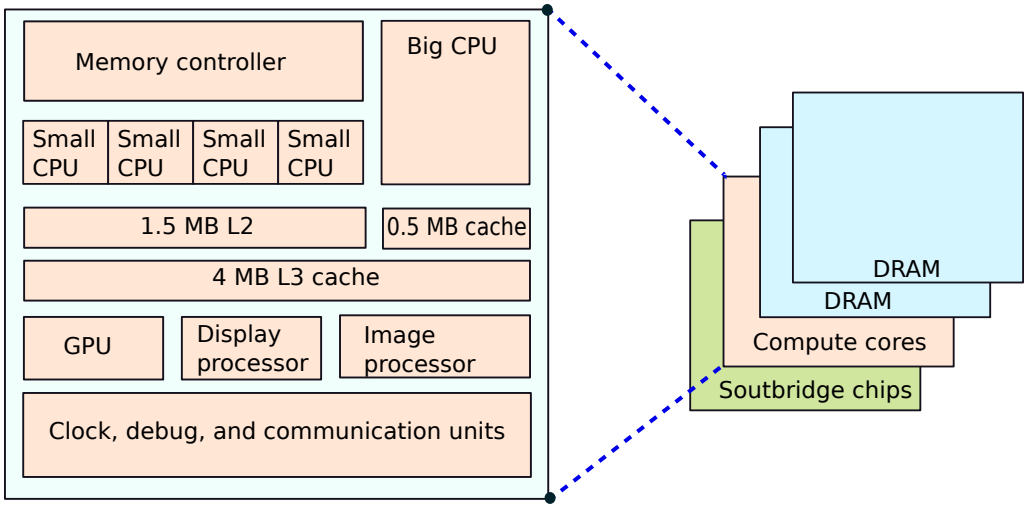

Figure C.4: The design of the Lakefield processor

Intel introduced the Lakefield processor in 2019 that combines a big Sunny Cove core and four small Tremont cores. It has been designed for mobile phones and small computing devices that are expected to run heterogeneous workloads. Its architecture is shown in Figure C.4.

Along with a heterogeneous design, this processor is revolutionary in many other ways. It uses the Foveros 3D stacking technology to stack four layers and create a 3D chip. The bottommost layer encapsulates the functionality of the erstwhile Southbridge chip that contains all the chips in the chipset that control the storage and I/O devices. For example, it has modules to control the USB devices, hard disks, audio controllers, PCI devices, and it also contains dedicated accelerators for cryptographic operations. The next layer contains the cores and the GPU. The top two layers are dedicated to the DRAM modules. The advantage of 3D stacking is that we can realize very fast and high bandwidth connections between the layers. Second, we need less space on the motherboard and this allows the processor to be used in devices with small form factors.

The Foveros technology allows us to connect two adjoining layers with a large array of microbumps (see Section 10.5.6). The layers themselves can be fabricated using different processes and different feature sizes. To connect two such layers that are fabricated using incompatible silicon processes, all that we need to do is vertically integrate them in a 3D package and align the microbumps.

Let us now focus on the layer that contains the cores. There are four small Tremont cores and one big Sunny Cove core. The L1 caches are within the cores, and the small cores are connected to a shared 1.5 MB L2 cache. The big core additionally has a 0.5 MB private cache for itself. All of these caches are connected to a shared LLC (4 MB L3). Intel also placed a wide variety of graphics and vision chips in this layer. This includes a standard GPU, a display processor that can support multiple displays, and an image processor for processing the inputs captured by cameras. We additionally have clock, debug,

and communication units in this layer.

The Lakefield processor is a one-of-a-kind design that combines heterogeneous computing, 3D stacking, and DRAM modules embedded in the package. Previously, such designs were not feasible primarily because of power and temperature issues; however, now with improvements in process technology and reduced feature sizes, it is possible to realize such designs. In the future it is expected that in the processor landscape, we shall have many such designs that have a very high degree of 3D integration and diverse computing devices.

D

AMD Processors

Zen 2 is AMD's latest microarchitecture for desktop and server processors. It is fabricated using TSMC's latest 7 nm process. Zen 2 cores are used in the Ryzen 3000 processors meant for desktops, Threadripper processors for high-end desktops, and Epyc processors for servers. We will first discuss the microarchitecture of a Zen 2 core, then discuss the AMD Ryzen 3000 series processors (codenamed "Matisse") that uses such Zen 2 cores to implement an SoC, and finally conclude with a discussion on the AMD Epyc™ 7742 server processor (codenamed "Rome") [Suggs and Bouvier, 2019, Gwennap, 2019b, Advanced Micro Devices, 2017]. Note that an SoC (system-on-chip) contains multiple computing or memory elements within a single chip. These can include cores, caches banks, memory controllers, and GPUs.

D.1 Zen2 Microarchitecture

D.1.1 Fetch and Decode Logic

Figure D.1 shows the microarchitecture of the Zen 2 core. Zen 2 uses an advanced variant of the TAGE branch predictor [Seznec, 2007]. Some features of this predictor are as follows. Along with hashing entries, the predictor also stores tags in the entries to eliminate aliasing. Second, it uses variable-sized histories: small histories for easy-to-predict branches and long histories for difficult-to-predict histories.

Zen 2 uses an 8-way 32-KB i-cache that can supply 32 bytes per cycle to the decode unit. The decode unit has four decoders that can process 4 instructions per cycle. Additionally, the core contains a micro-op cache that has 2K to 4K entries. It reduces the need to decode every instruction. Frequently executed instructions hit in this cache, and this saves decode energy. Additionally, the pipeline containing the micro-op cache is shorter, and thus there is an improvement in the overall latency as well. The decoded μOPs (micro-instructions) are stored in a μOP queue.

The fetch logic also contains a three-level BTB (branch target buffer) and a 32-entry return address stack. Most codes have some indirect branches, where the target of the branch changes (it is typically the value of a register). Zen 2 contains an indirect branch predictor that predicts the target based on the history of the branch. We typically use such indirect branches while using function pointers in C and virtual functions in C++.

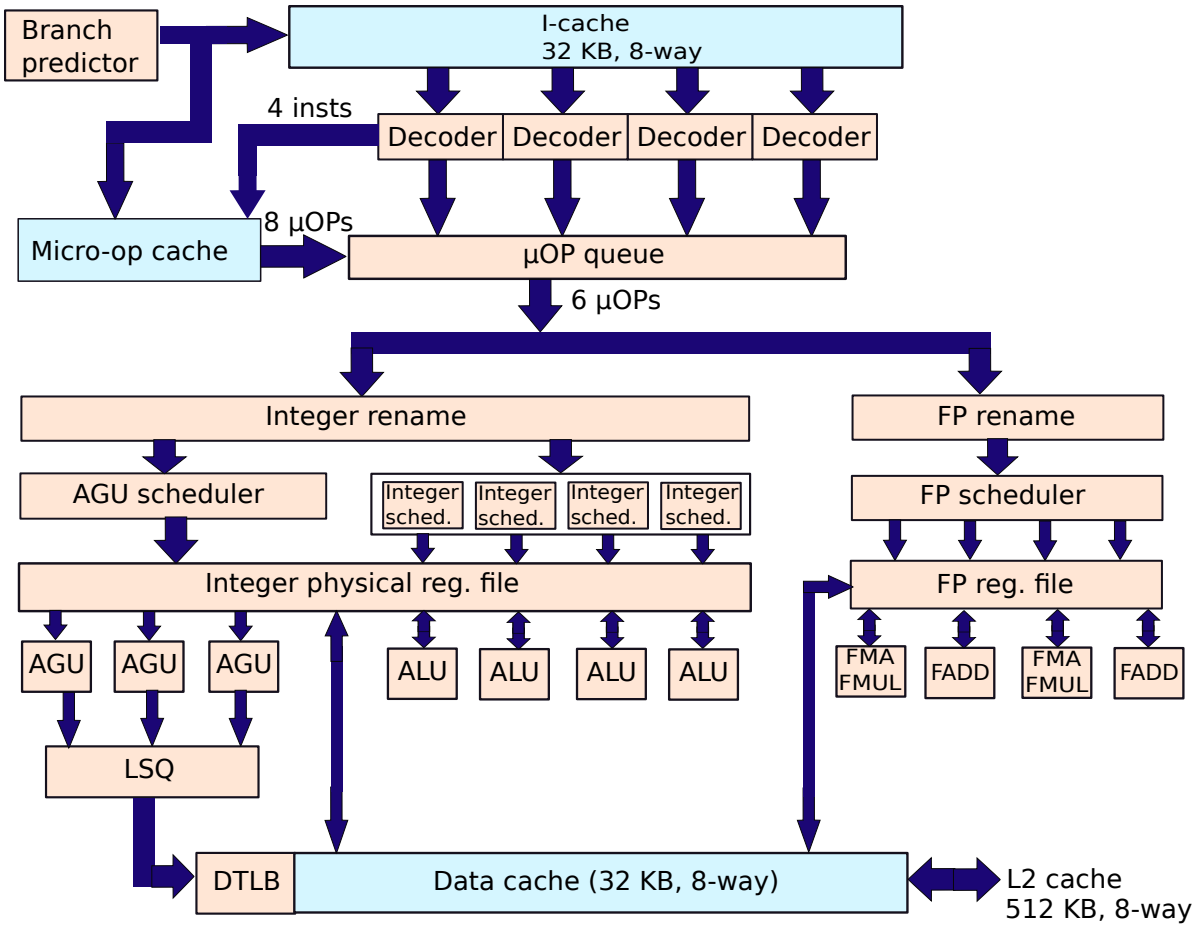

Figure D.1: The microarchitecture of the AMD Zen2 processor (source [Suggs and Bouvier, 2019])

D.1.2 Scheduling and Execution

After register renaming, the instructions are sent to the corresponding schedulers (instruction window + wakeup/select logic). The schedulers can receive up to 6 micro-ops per cycle (the dispatch width is 6). There are four ALU schedulers, one AGU (for loads and stores) scheduler, and one FP (floating point) scheduler.

The AGU scheduler is connected to two load AGUs (address generation units) and one store AGU. Each such AGU can process one μOP per cycle. The floating point (FP) scheduler is similarly connected to two FP multiply and two FP add units. Each FP multiply unit can also process the FMA instruction (fused multiply and add). The SIMD data path is 256 bits in the Zen 2 core. This means that if we have packed single precision floating point numbers, then we can operate on 8 pairs of such numbers in parallel.

Once an instruction is scheduled, it reads the values from the corresponding register file and proceeds towards the execution units. The Zen2 core has an elaborate load-store unit that contains an LSQ that can forward values from earlier stores to later loads, track in-flight cache misses, and support x86's complex addressing modes.

The Zen 2 core has an 180-entry general purpose physical register file for storing integers and memory addresses, and an 160-entry vector register file in the floating point unit. The four integer ALUs and

the three AGUs access the general-purpose register file. It should be noted that each core supports 2-way SMT (simultaneous multithreading). The schedulers incorporate a notion of fairness such that the threads make similar rates of progress.

D.1.3 Data Caches

Each core has an 8-way 32-KB data cache that can support two 256-bit loads and one 256-bit store per cycle. It uses a variant of way prediction 7.4.4 to hide the time it takes to convert virtual addresses into physical addresses. It computes a hash of the last virtual address that was used to access a cache line, and stores the hash in the cache line itself. Before the physical address is available, the core speculatively accesses the line that has a matching hash.

The data cache is connected to a 512-KB, 8-way L2 cache. This is connected to an L3 cache (4 MB or 16 MB). Like all processors in its class, the Zen 2 core has a 2-level TLB. The first level has an I-TLB (for instructions) and a D-TLB (for data). The level 1 TLBs are connected to level 2 TLBs. To handle TLB misses, the Zen 2 core has two hardware page walkers to service TLB misses. The Zen 2 core supports huge pages (2 MB or 1 GB). A 1 GB page is stored as a set of 2 MB pages in the TLB – this is known as *smashing*.

D.2 AMD Ryzen 3000 Series Processor (Codenamed Matisse)

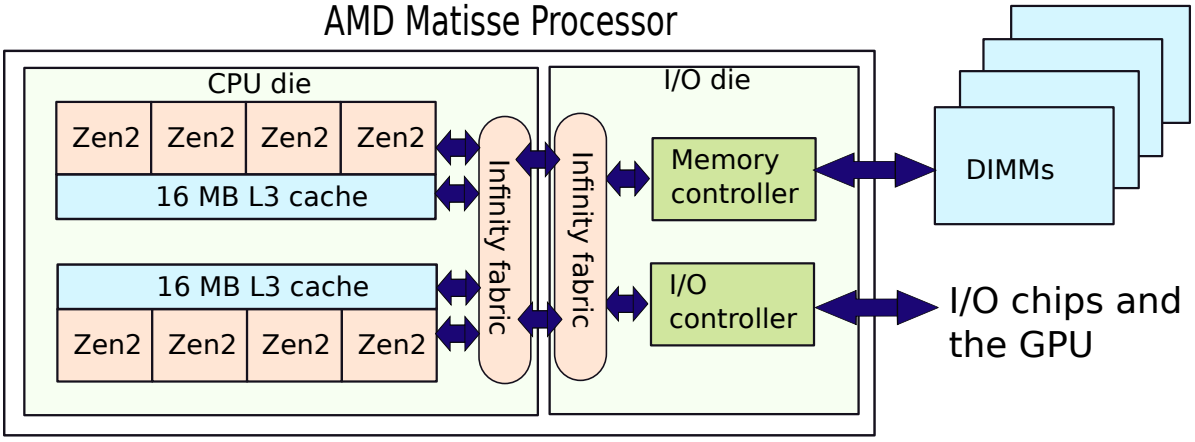

Figure D.2: The layout of the AMD Matisse package (source [Gwennap, 2019b])

Figure D.2 shows the architecture of an SoC targeted at client platforms such as laptop and desktop computers; it has eight such Zen2 cores (the Matisse chip). The cores are grouped into two clusters; each cluster has a shared 16-MB L3 cache. This is a chiplet-based design, where multiple dies (fabricated with different technologies) are integrated into the same package. Figure D.2 shows two chiplets: a core chiplet with 8 cores and 32 MB of L3 cache, and an I/O chiplet. The latter contains the memory controller and the I/O controllers. Both are connected with a low-latency and high-bandwidth interconnect, which AMD calls the *Infinity Fabric*.

D.3 AMD EPYC^{TM} 7742 Processor (Codenamed Rome)

Let us now discuss AMD's latest server processor (as of 2020) that uses the Zen 2 core. It is the AMD EPYC 7742 processor (codenamed "Rome"). As shown in Figure D.3(a), the chip can be viewed as

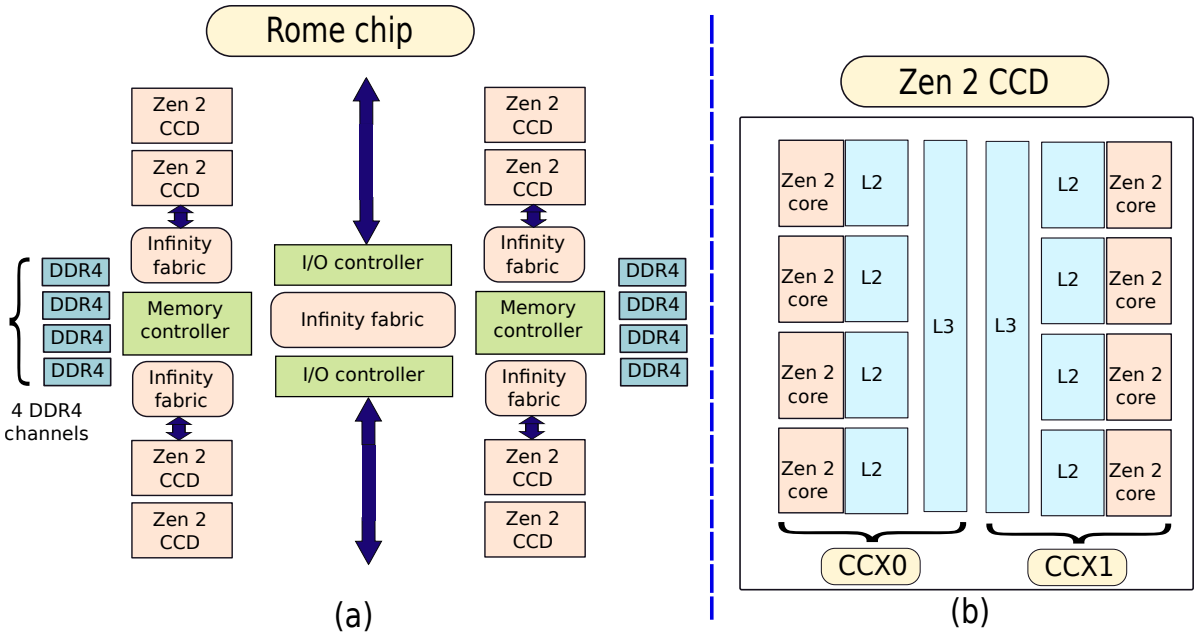

Figure D.3: The Rome chip (source [Suggs and Bouvier, 2019])

comprising four quadrants with two Core-Complex Dies (CCDs) and two memory channels per quadrant. Each CCD in turn consists of a pair of two core complexes: CCX0 and CCX1 (see Figure D.3(b)). Each such core complex consists of 4 cores sharing a 16 MB L3 cache. Each core supports 2-way Simultaneous Multi Threading (SMT).

Hence, the entire processor consists of 64 cores (128 hardware threads) with a 256 MB distributed L3 cache and 8 memory channels, where each memory channel can support the DDR4-3200 protocol. The four quadrants can be configured to expose different Non-Uniform Memory Access (NUMA) topologies to the operating system, designated by the Nodes Per Socket (NPS) parameter. NUMA is conceptually similar to NUCA (see Section 8.5), albeit it is at the level of main memory.

The reason for grouping channels together and creating a *NUMA domain* is as follows. Let's say we want to provide high bandwidth to a core. Then we would like the core to be able to access all the memory channels simultaneously and read or write data to the attached DIMMs. Using this technique, we can realize a very high bandwidth connection to memory. However, this technique might not work very well because the latencies to different memory banks are different. Some memory controllers are close to the core; they can be accessed quickly; however, many memory controllers are on the other side of the chip, and it is necessary to traverse the on-chip interconnect. Hence, we might not want to interleave memory accesses across all the channels. We might instead want to create groups comprising 2 or 4 channels and assign them to a core. It can then access these channels in parallel and interleave its memory accesses to maximize the available bandwidth. The NPS parameter allows us to control this behavior.

E

Qualcomm Processors

Qualcomm® is famous for its Snapdragon® processors, which are primarily designed for mobile phones. These processors are SoCs (systems on chip) That contain many other elements as well such as GPUs and custom accelerators. In this appendix, we will describe the latest Snapdragon 865 processor [Gwennap, 2019a, Hachman, 2019].

Snapdragon has been released in 2020. It is a futuristic mobile chip where the main focus is on artificial intelligence(AI), high-intensity gaming, and 5G communication. This requires a diversity of cores and accelerators. We cannot have a single kind of core for so many diverse applications. Hence, the designers have opted for what is known as a big.LITTLE™ architecture. This was invented by ARM® Limited. Such an architecture contains a set of *big* cores to provide good single thread performance, and a set of *little* cores that are extremely power efficient. The advantage of such an architecture is that depending upon the workload we can dispatch it to either the big cores or the little cores to achieve an equitable trade-off between power and performance. Most multicore mobile platforms as of today use such an architecture.

Let us split our discussion into two parts. We shall first discuss the general computer architecture, and then move to the application-specific accelerators.

E.1 Compute Cores

The organization of the cores and caches is shown in Figure E.1. This is an octa-core architecture. It contains three types of cores. It has one large ARM® Cortex®-A77 core, which implements the 64-bit ARM v8 instruction set. This is a fairly wide OOO core with a decode width of 4 and a fetch width of 6. The frequency is set to 2.84 GHz and the L1 I/D cache sizes are set to 64 KB. This core is connected to a large 512-KB L2 cache.

Then it has three other Cortex-A77 cores that run at a lower frequency, 2.4 GHz. The L2 cache allocated for these three cores is limited to 3×256 KB.

Then we come to the third types of cores (little cores): four Cortex-A55 cores. They also support the same version of the ARM ISA; however, these are far weaker superscalar in-order cores with a decode width of 2. Such in-order cores are typically very useful when coupled with accelerators. The in-order cores run the general-purpose code, and offload custom computations to accelerators that provide the speed-up. In comparison to the big cores, we provision only 128 KB of L2 cache for each little core, and also run them at a much lower frequency, 1.8 GHz.

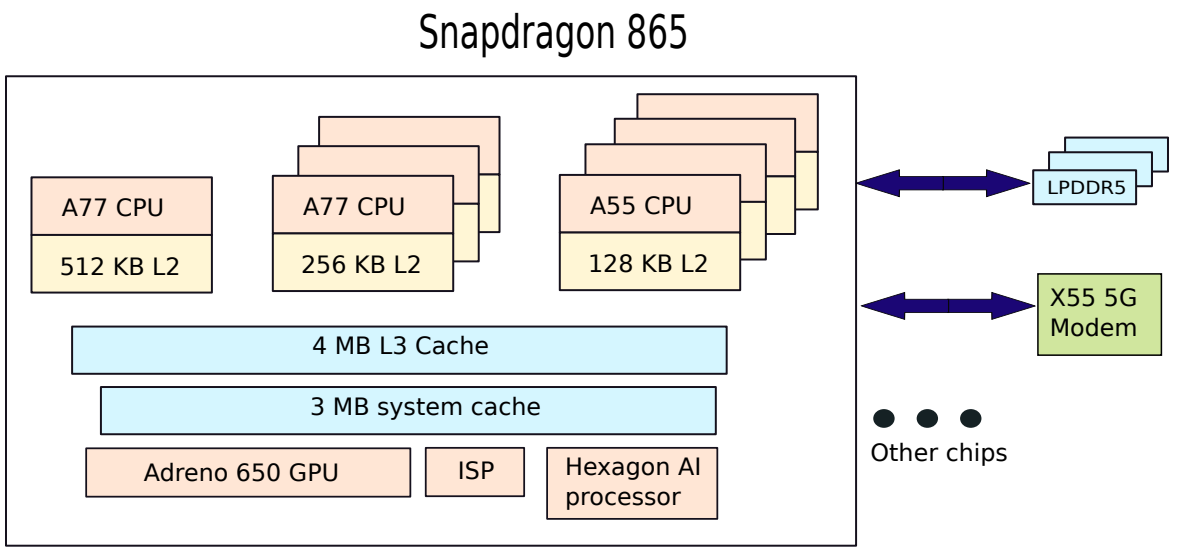

Figure E.1: The Qualcomm Snapdragon 865 processor (many of its components are not shown in the figure)

The cores share a 4 MB L3 cache, and use the low power LPDDR5 protocol to connect to off-chip memory modules. Here, "LP" stands for "low power". Some of the key features that make such protocols more power efficient include a narrow channel width (16 or 32 bits), reduced supply voltage, partial DRAM refresh modes, low power memory states, multiplexed control and address lines, and avoiding transmitting data if it is all zeros or all ones.

Lastly note that whenever we have a set of cores, a GPU, and a set of accelerators, we often need a dedicated memory structure that can be used to transfer data between them. This can either be the last level cache such as the L2/L3 caches, or we can add a separate memory structure for effecting such transfers. This is a standard design technique, and this has been used in the Snapdragon 865 processor as well. It adds a 3 MB system cache for such kind of communication. Think of this as a bespoke L4 cache.

E.2 Accelerators

To support model workloads such as AI, 5G, and advanced video processing, it is necessary to add custom accelerators.

Qualcomm adds an Adreno® 650 GPU, which can support modern 4K displays and high intensity graphics. The key design decision here was to support modern immersive gaming environments. Such gaming environments support a wide diversity of colors and also their screen refresh rate is set to 90-144Hz, which is far more than the typical 50 to 60 Hz refresh rates of modern monitors. To bring complex scenes to life, it is necessary to add a lot of depth information to the image and also support a wide variety of graphics effects. Keeping all of this in mind, futuristic GPUs such as the Adreno 650 have been designed.

Snapdragon 865 also has a dedicated AI processor (Hexagon 698), which is primarily a tensor processing accelerator. Modern smartphones use all kinds of AI technologies such as speech recognition, gesture recognition, and integrate data from all kinds of sensors that include gyroscopes, accelerators, and multiple cameras. They need a sophisticated AI engine to search for patterns in the data, and to effectively analyze it. This necessitates the need for a dedicated accelerator. Using this accelerator,

Snapdragon 865 offers a throughput of 15 trillion AI operations per second.

Qualcomm has a dedicated image signal processor (ISP) in the chip for processing images captured by the cameras. It can process billions of pixels per second. This is typically necessary to generate autofocus points, perform an optical zoom, capture still images and video, and capture slow motion video.

As of 2020, we are entering the era of 5G. 5G can in principle support a peak data rate of 10 Gbps. Most 5G systems are expected to use millimeter waves to transmit data at a very high frequency. As of mid 2020, there are limited 5G deployments in the world; however, the technology is showing a lot of promise. Hence, mobile chipmakers need to ensure that their SoCs are 5G ready. Qualcomm has thus added an X55 5G modem into the Snapdragon motherboard (note that it is a separate chip). Along with that, Qualcomm has also integrated other chips along with the Snapdragon processor on the motherboard, which include fast wireless and Bluetooth chips.

While designing a large SoC, designers are typically faced with a dilemma: should they have certain components within the SoC or place them outside it on the motherboard? The former design decision can provide us with greater performance as long as we can ensure that we can fabricate such a large chip with acceptable yield rates, and keep on-chip power dissipation below a certain threshold. If power consumption is an issue or there is a possibility that the chip will become too big and the yield rate will go down, the components may be fabricated as separate chips and placed close by on the motherboard.