Part IV

Appendix

A
Case Studies of Real Processors

Let us now look at the design of some real processors such that we can put all the concepts that we have learned up till now in a practical perspective. We shall study embedded (for smaller mobile devices), and server processors of three major processor companies namely ARM, AMD, and Intel. Let us start with a disclaimer. The aim of this section is not to compare and contrast the design of processors across the three companies, or even between different models of the same company. Every processor is designed optimally for a certain market segment with certain key business decisions in mind. Hence, our focus in this section would be to study the designs from a technical perspective, and appreciate the nuances of the design.

A.1 ARM® Processors

Let us now describe the design of ARM processors. The most important point to note about the ARM processors (popularly referred as *ARM cores*) is that ARM designs the processors, and then licenses the design to customers. Unlike other vendors such as Intel or IBM, ARM does not manufacture silicon chips. Instead, vendors such as Texas Instruments and Qualcomm buy the license to use the design of ARM cores, and add additional components. They then give a contract to semiconductor manufacturing companies, or use their own manufacturing facilities to manufacture an entire SOC(System on Chip) in silicon.

ARM has three processor lines for its latest (as of 2012) ARMv8 architecture. The first line of processors is known as the ARM® Cortex® -M series. These processors are mainly designed to be used as micro-controllers in embedded applications such as medical devices, automobiles, and industrial electronics. The main focus behind the design of such processors is power efficiency, and cost. In this section, we shall describe the ARM Cortex-M3 processor that has a three stage pipeline.

The second line of processors is known as the ARM® Cortex® -R series. These processors are designed for real time applications. The main focus here is reliability, high speed and real time response. They are not meant to be used by consumer electronics devices such as smart phones. The Cortex-R series processors do not have support for running operating systems that use virtual memory.

The ARM® Cortex® -A series processors are designed to run regular user applications on smart phones, tablets, and a host of high end embedded devices. These ARM cores typically have complex pipelines, support for vector operations, and can run complex operating systems that require hardware support for virtual memory. We shall study the Cortex-A8 and Cortex-A15 processors in this section.

A.1.1 ARM® Cortex® -M3

System Design

Let us begin with the ARM Cortex-M series processors that have been designed primarily for the embedded processor market. For such embedded processors, energy efficiency and cost are more important than raw performance. Consequently, ARM engineers designed a 3 issue pipeline devoid of very complicated features.

The Cortex-M3 supports a basic version of the ARMv7-M instruction set as described in Chapter 4. It is typically attached to other components using the ARM® AMBA® bus as shown in Figure A.1.

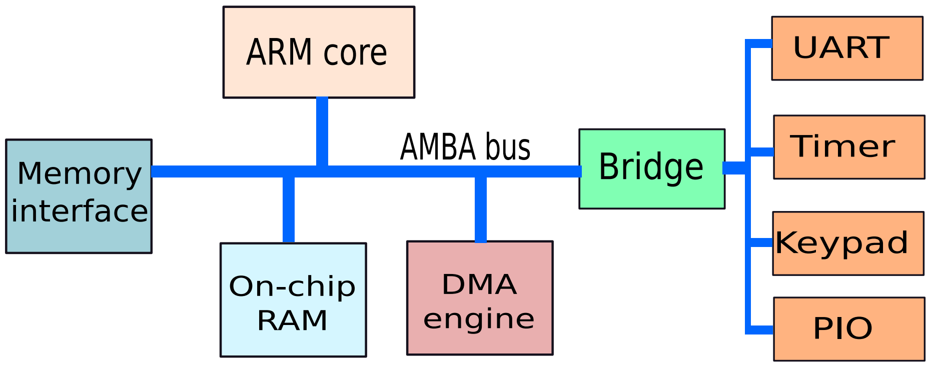

Figure A.1: The ARM Cortex-M3 connected to the AMBA bus along with other components , source [arm, b]. Reproduced with permission from ARM Limited. Copyright ©ARM Limited (or its affiliates)

AMBA (Advanced Microcontroller Bus Architecture) is a bus architecture designed by ARM. It is used to connect an ARM core with other components in SOC based systems. For example, most of the processors in smart phones and mobile devices use the AMBA bus to connect to high speed memory devices, DMA engines, and other external buses through bridge devices. One such external bus is the APB bus (Advanced Peripheral Bus) that is used to connect to peripherals such as the keyboard, UART controller (Universal Asynchronous Receiver/Transmitter Protocol), timer, and the PIO (parallel input output) interface.

Pipeline Design

Figure A.2 shows the pipeline of the ARM Cortex-M3. It has three stages namely fetch (F), decode (D), and execute (E). The fetch stage fetches the instruction from memory, and is the smallest stage of all the three stages.

The decode stage(D stage) has three different sub-units as shown in Figure A.2. The D stage has an instruction decode and register read unit, which is similar to the operand fetch unit in *SimpleRisc* . It decodes the instruction, and forms the instruction packet. Simultaneously, it reads the values of the operands that are embedded in the instruction, and also reads values from the register file. The AGU (address generation unit) extracts all the fields in the instruction, and schedules the execution of the load or store instruction in the next stage of the pipeline. It plays a special role while processing the *ldm* (load multiple) and *stm* (store multiple) instructions. Recall from our discussion in Section 4.3.2 that these instructions can read or write to multiple registers at the same time. The AGU creates multiple operations out of a single *ldm* or *stm* instruction in the pipeline. The *branch* unit is used for branch prediction. It predicts both the branch outcome, and the branch target.

The execute stage is fairly heavy in terms of functionality, and some instructions take 2 cycles to execute. Let us look at the regular ALU and branch instructions first. Recall from our discussion in Section 4.2.2

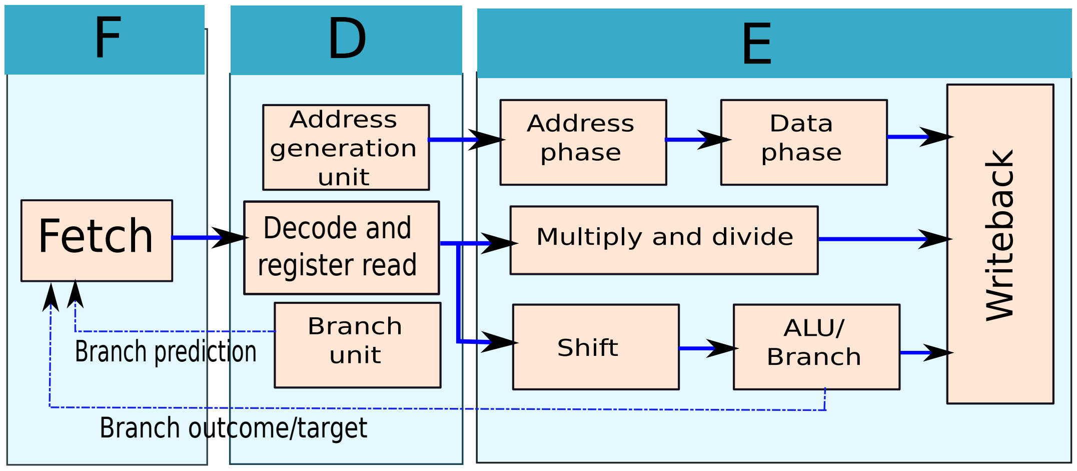

Figure A.2: The pipeline of the ARM Cortex-M3 , source [arm, b]. Reproduced with permission from ARM Limited. Copyright ©ARM Limited (or its affiliates).

that ARM instructions can have one shifter operand. Secondly, computing the value of the 32-bit immediate from its 12 bit encoding is essentially a shift (rotate is a type of shift) operation. Both of these operations are executed by the shift unit that has a hardware structure known as a *barrel shifter*. Once the operands are ready, they are passed to the ALU and branch unit that computes the branch outcome/target, and the ALU result.

ARM has two kinds of branches – direct and indirect. For direct branches, the offset of the branch target from the current PC is embedded in the instruction. For example, a branch to a label is an example of a direct branch. It is possible to compute the branch target of a direct branch in the decode stage. ARM also supports indirect branches, where the branch target is the result of an ALU or memory instruction. For example, the instruction *ldr pc, [r1, #10]* is an example of an indirect branch. Here, the value of the branch target is equal to the value loaded from memory by the load instruction. It is in general difficult to predict the target of indirect branches. In the Cortex-M3 processor, whenever there is a branch misprediction (either target or outcome), the two instructions fetched after the branch are cancelled. The processors starts fetching instructions from the correct branch target.

Along with the basic ALU, the Cortex-M3 has a multiply and divide unit that can perform both signed and unsigned, multiplication and division. The Cortex-M3 supports two instructions, *sdiv*, and *udiv* for signed and unsigned division respectively. Along with these instructions it has support for multiply, and multiply-accumulate operations as described in Section 4.2.1.

The load and store instructions typically take two cycles. They have an address generation phase, and a memory access phase. The load instructions takes 2 cycles to execute. Note that in the second cycle, it is not possible for other instructions to execute in the *E* stage. The pipeline is thus stalled for one cycle. This specific feature reduces the performance of the pipeline. ARM removed this restriction in its high performance processors. The store instruction also takes 2 cycles to execute. However, the second cycle that accesses memory does not stall the pipeline. The processor writes the value to a store buffer (similar to a write buffer as discussed in Section 10.3.3), and proceeds with its execution. It is further possible to issue back to back (consecutive cycles) store and load instructions, where the load reads the value written by the store. The pipeline does not need to stall for the load instruction because it reads the value written by the store from the store buffer.

A.1.2 ARM® Cortex® -A8

As compared to the Cortex-M3, which is an embedded processor, the Cortex-A8 was designed to be a full fledged processor that can run on sophisticated smart phones and tablet processors. Here, *A* stands for *application*, and ARM's intent was to use this processor to run regular applications on mobile devices. Secondly, these processors were designed to support virtual memory, and also contained dedicated floating point and SIMD units.

Overview of the Cortex-A8

The defining feature of the pipeline of the Cortex-A8 core is that it is a dual issue superscalar processor. However, it is not a full blown out-of-order processor. The issue logic is inorder. The Cortex-A8 has a 13-stage integer pipeline with sophisticated branch prediction logic. Since it uses a deep pipeline, it is possible to clock it at a higher frequency than other ARM processors that have shallower pipelines. The Cortex-A8 core is designed to be clocked between 500 MHz and 1GHz, which is a fairly fast clock speed in the embedded domain.

Along with the integer pipeline, the Cortex-A8 contains a dedicated floating point and SIMD unit. The floating point unit implements ARM's VFP (vector floating point) ISA extension, and the SIMD unit implements the ARM® NEON® instruction set. This unit is also pipelined and has 10 stages. Moreover, the ARM Cortex-A8 processor has a separate instruction and data cache, which can be optionally connected to a large shared L2 cache.

Design of the Pipeline

Figure A.3 shows the design of the pipeline of the ARM Cortex-A8 processor. The fetch unit is pipelined across two stages. Its primary purpose is to fetch an instruction, and update the PC. Additionally, it also has a built in instruction prefetcher, ITLB (instruction TLB), and branch predictor. The advanced features of the branch predictors of Cortex-A8 and Cortex A-15 are discussed in Section A.1.3. The instructions subsequently pass to the decode unit.

The decode unit is pipelined across 5 stages. The decode unit is more complicated in the Cortex-A8 processor than the Cortex-M3 because it has the additional responsibility of checking the dependences across instructions, and issuing two instructions together. The forwarding, stall, and interlock logic is thus much more complicated. Let us number the two instruction issue slots 0 and 1. If the decode stage finds two instructions that do not have any interdependences, then it fills both the issue slots with instructions, and sends them to the execution unit. Otherwise, the decode stage just fills one issue slot.

The execution unit is pipelined across 6 stages, and it contains 4 separate pipelines. It has two ALU pipelines that can be used by both the instructions. It has a multiply pipeline that can be used by the instruction issued in slot 0 only. Lastly, it has a load/store pipeline that can again be used by instructions issued in both the issue slots.

NEON and VFP instructions are sent to the NEON/VFP unit. It takes three cycles to decode and schedule the NEON/VFP instructions. Subsequently, the NEON/VFP unit fetches the operands from the NEON register file that contains thirty two 64-bit registers. NEON instructions can also view the register file as sixteen 128 bit registers. The NEON/VFP unit has six 6 stage pipelines for arithmetic operations, and it has one 6 stage pipeline for load/store operations. As discussed in Section 11.5.2, loading vector data is a very performance critical operation in SIMD processors. Hence, ARM has a dedicated load queue in the NEON unit for populating the NEON register file by loading data from the L1 cache. For storing data, the NEON unit writes data directly back to the L1 cache.

Each L1 cache (instruction/data) has a 64 byte block size, has an associativity of 4, and can either be 16 KB of 32 KB. Secondly, each L1 cache has two ports, and can provide 4 words per cycle for NEON and

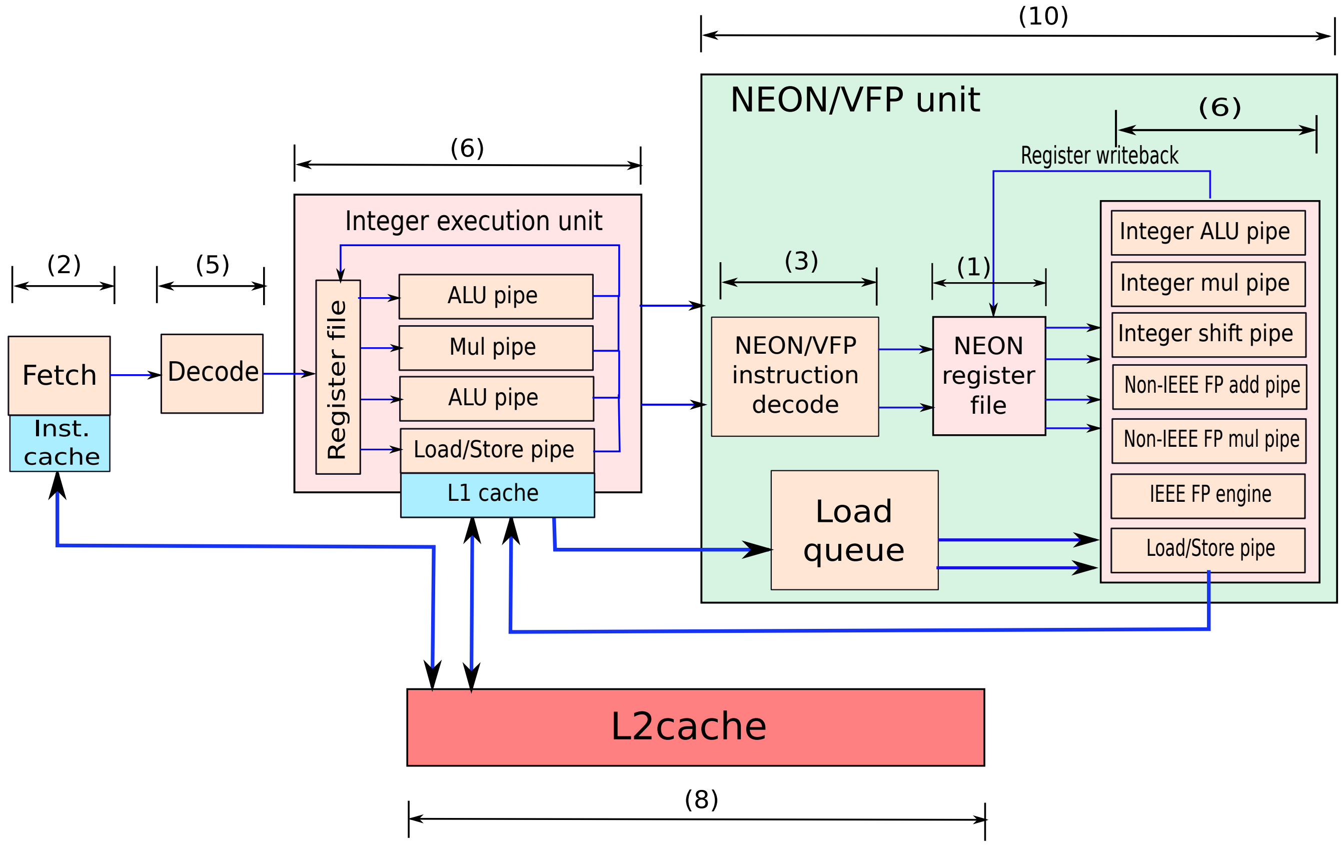

Figure A.3: The pipeline of the ARM Cortex-A8 processor , source [arm, a]. Reproduced with permission from ARM Limited. Copyright ©ARM Limited (or its affiliates).

floating point operations. The point to note here is that the NEON/VFP unit and the integer pipelines share the L1 data cache. The L1 caches are optionally connected to a large L2 cache. It has a block size of 64 bytes, is 8 way set associative, and can be as large as 1 MB. The L2 cache is split into multiple banks. We can lookup two tags at the same time, and the data array accesses proceed in parallel.

A.1.3 ARM® Cortex® -A15

The ARM Cortex-A15 is the latest ARM processor to be released as of early 2013. This processor is targeted towards high performance applications.

Overview

The Cortex-A15 processor is much more complicated, and much more powerful than the Cortex-M3 and Cortex-A8. Instead of using an inorder core, it uses a 3-issue superscalar out-of-order core. It also has a deeper pipeline. Specifically, it has a 15 stage integer pipeline, and a 17-25 stage floating point pipeline. The deeper pipeline allows it to run at a significantly higher frequency (1.5 – 2.5 GHz). Additionally, it fully integrates VFP and NEON units on the core instead of having them as separate execution units. Like server processors, it is designed to access a large amount of memory. It can support a 40 bit physical address, which means that it can address up to 1 TB of memory using the latest AMBA bus protocol that supports system level coherence. The Cortex-A15 is designed to run modern operating systems, and virtual machines. Virtual machines are special programs that can help run multiple operating systems concurrently on the same processor. They are used in server and cloud computing environments to support users with varying software requirements. The Cortex-A15 incorporates sophisticated power management techniques that dynamically shut off parts of the processor when they are not being used.

Another iconic feature of the Cortex-A15 processor is that it is a multicore processor. It organises 4 cores per cluster, and we can have multiple clusters per chip. The Snoop Control Unit provides coherency within a cluster. The AMBA4 specification defines the protocol for supporting cache and system level coherence across clusters. Additionally, the AMBA4 bus also supports synchronisation operations. The memory system is also faster and more reliable. The Cortex-A15's memory system uses SECDED (single error correct, double error detect) error control codes.

Design of the Pipeline

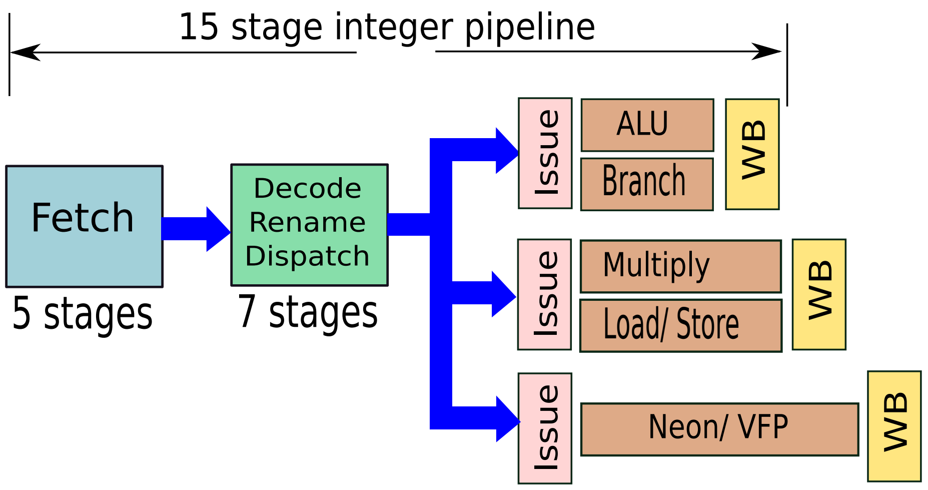

Figure A.4: Overview of the ARM Cortex A-15 processor, source [arm, d]. Reproduced with permission from ARM Limited. Copyright ©ARM Limited (or its affiliates).

Figure A.4 shows an overview of the pipeline of a Cortex-A15 core. We have 5 fetch stages. Here, fetch is more complicated because, the Cortex-A15 has a sophisticated branch predictor that can handle many types of branch instructions. The decode, rename, and instruction dispatch units are pipelined across 7 stages. Recall from our discussion in Section 9.11.4 that the register rename unit, and instruction window are critical to the performance of out-of-order processors. Their role is to find sets of instructions that are ready to execute in a given cycle.

The Cortex-A15 has several execution pipelines. The integer ALU, and branch pipelines require 3 cycles each. However, the multiply, and load/store pipelines are longer. Unlike other ARM processors that treat the NEON/VFP units as a physically separate unit, the Cortex-A15 integrates it on the core. It is a part of the out-of-order pipeline. Let us now look at the pipeline in some more detail (refer to Figure A.5).

The Cortex-A15 core's (the same features are available in the branch predictor of the Cortex-A8 also) branch predictor contains a predictor for direct branches, a predictor for indirect branches, and a predictor to predict the return address. The indirect branch predictor tries to predict the branch target based on the PC of the branch instruction. It has a 256 entry buffer that is indexed by the history of a given branch,

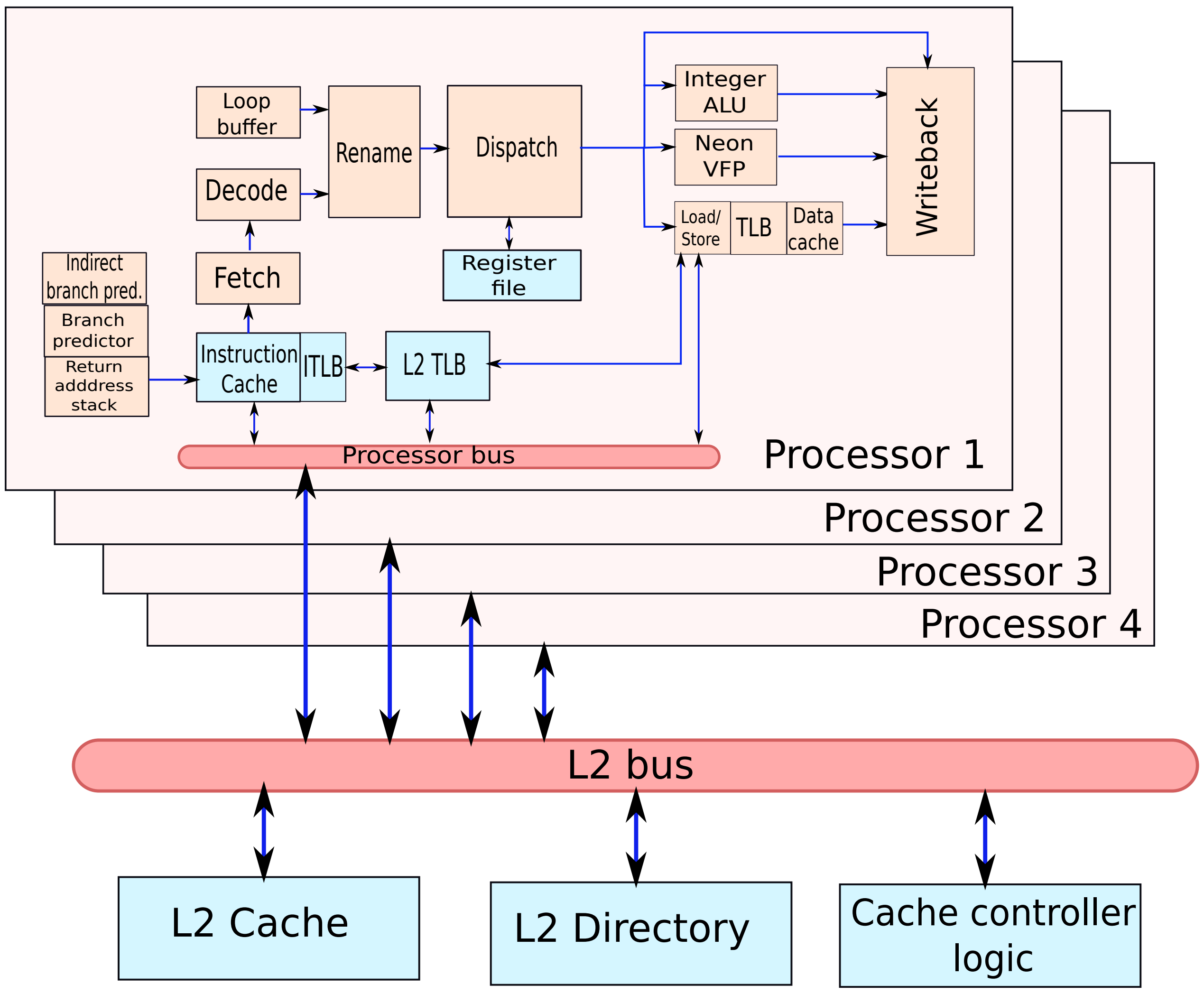

Figure A.5: The pipeline of the ARM Cortex A-15 processor , source [arm, c]. Reproduced with permission from ARM Limited. Copyright ©ARM Limited (or its affiliates).

and its PC. We do not actually need sophisticated branch prediction logic to predict the target of a return instruction. A simpler method is to record the return address whenever we call a function, and push it on a stack (referred to as the *return address stack*(RAS)). Since function calls exhibit last in-first out behaviour, we need to simply pop the RAS and get the value of the return address while returning from a function. Lastly, to support wider issue widths the fetch unit is designed to fetch 128 bits at once from the instruction cache.

The *loop buffer* (present in the Cortex-A8 also) is a very interesting addition to the decode stage. Let us assume that we are executing a set of instructions in a loop, which is most often the case. In any other processor, we need to fetch the instructions in a loop repeatedly, and decode them. This process wastes energy, and memory bandwidth. We can optimise this process by saving all the decoded instruction packets in a loop buffer such that we can bypass the fetch and decode units altogether while executing a loop. The register rename stage can thus get instructions from the decode unit or the loop buffer.

The core maintains a reorder buffer (ROB) (see Section 9.11.4) that contains the results of all the instructions. Recall that entries in the ROB are allocated in program order. The rename stage maps operands to entries in the ROB (referred to as the result queue in ARM's documentation). For example, if instruction 3 needs a value that is going to be produced by instruction 1, then the corresponding operand is mapped to the ROB entry of instruction 1. All the instructions subsequently enter the instruction window and wait for their source operands to be ready. Once they are ready, they are dispatched to the corresponding pipelines. The Cortex-A15 has 2 integer ALUs, 1 branch unit, 1 multiply unit, and 2 load/store units. The NEON/VFP unit can accept 2 instructions per cycle.

The load/store unit has a 4 stage pipeline. For ensuring precise exceptions stores are only issued to the memory system, when the instruction reaches the head of the ROB (there are no earlier instructions in the pipeline). Meanwhile, any load operation that has a store operation to the same address in the pipeline gets its value through a forwarding path. Both the L1 caches (instruction and data) are typically 32 KB each.

The Cortex-A15 processor supports a large L2 cache (up to 4 MB). It is a 16 way set associative cache with an aggressive prefetcher. The L1 caches, and the L2 cache are a part of the cache coherence protocol. The Cortex-A15 uses a directory based MESI protocol. The L2 cache contains a *snoop tag array* that maintains a copy of all the directories at the L1 level. If an I/O operation wishes to modify some line, then the L2 cache uses the snoop tag array to find if the line resides in any L1 cache. If any L1 cache contains a copy of the line, then this copy is invalidated. Likewise, if there is a DMA read operation, then the L2 controller fetches the line from the L1 cache that contains a copy of it. It is additionally possible to extend this protocol to support L3 caches, and a host of peripherals.

A.2 AMD® Processors

Let us now study the design of AMD processors. Recall that AMD processors implement the x86 instruction set, and AMD manufactures processors for mobile devices, netbooks, laptops, desktops, and servers. In this section, we shall look at two processors at both the ends of the design spectrum. The AMD Bobcat processor is meant for mobile devices, tablets, and netbooks. It implements a subset of the x86 instruction set, and the main objectives of its design are power efficiency, and an acceptable level of performance. The AMD Bulldozer processor is at the other end of the spectrum, and is meant for high end servers. It is optimised for performance and instruction throughput. It is also AMD's first multithreaded processor, which uses a novel type of core known as a *conjoined core* for implementing multithreading.

A.2.1 AMD Bobcat

Overview

The Bobcat processor (original paper [Burgess et al., 2011]) was designed to operate within a 10-15W power budget. Within this power budget, the designers of Bobcat were able to implement a large number of complex architectural features in the processor. For example, Bobcat uses a fairly elaborate 2 issue out-of-order pipeline. Bobcat's pipeline uses a sophisticated branch predictor, and is designed to fetch 2 instructions in the same cycle. It can subsequently decode them at that rate and convert them to complex micro-ops (Cops). A complex micro-op (Cop) in AMD's terminology is a CISC like instruction that can read and write to memory. The set of Cops are subsequently sent to the instruction queue, renaming engine, and scheduler.

The scheduler selects instructions out-of-order and dispatches them to the ALU, memory address generation units, and the load/store units. The load/store unit also sends requests to the memory system out-of-order in the interest of performance. Hence, we can readily conclude that Bobcat supports a weak memory model. Along with sophisticated micro-architectural features, the Bobcat processor also supports

SIMD instruction sets (up to SSE 4), methods to automatically save the processor state in memory, and 64-bit instructions. To ensure that the power consumption of the processor is within limits, Bobcat contains a large number of power saving optimisations. One such prominent mechanism is known as *clock gating*. Here, the clock signal is set to a logical 0 for units that are unused. This ensures that there are no signal transitions in the unused units, and consequently there is no dissipation of dynamic power. The Bobcat processor also uses pointers to data as much as possible, and tries to minimise copying data across different locations in the processor. Let us now look at the design of the pipeline in some more detail.

Design of the Pipeline

Figure A.6 shows a block diagram of the pipeline of the AMD Bobcat processor. A distinguishing feature of the Bobcat processor is the fairly sophisticated branch predictor. We need to first predict if an instruction is a branch or not. This is because, there is no way of finding this out quickly in the x86 ISA. If an instruction is predicted to be a branch, we need to compute its outcome (taken/not taken), and the target. AMD uses an advanced pattern matching based proprietary algorithm for branch prediction. After branch prediction, the fetch engine fetches 32 bytes from the I cache at once, and sends it to an instruction buffer.

The decoder considers 22 instruction bytes at a time, and tries to demarcate instruction boundaries. This is a slow and computationally intensive process because x86 instruction lengths can have a lot of variability. Larger processors typically cache this information such that decoding an instruction for the second time is easier. Since the decode throughput of Bobcat is only limited to 2 instructions, it does not have this feature. Now, most pairs of x86 instructions fit within 22 bytes, and thus the decoder can most of the time extract the contents of both the x86 instructions. The decoder typically converts each x86 instruction to 1-2 Cops. For some infrequently used instructions, it replaces the instruction with a microcode sequence.

Subsequently, the Cops are added to a 56 entry reorder buffer (ROB). Bobcat has two schedulers. The integer scheduler has 16 entries, and the floating point scheduler has 18 entries. The integer scheduler selects two instructions for execution every cycle. The integer pipeline has two ALUs, and two address generation units (1 for load, and 1 for store). The floating point pipeline can also execute two Cops per cycle with some restrictions.

The load-store unit in the processor forwards values from store to load instructions in the pipeline whenever possible. Bobcat has 32 KB (8 way associative) L1 D and I caches. They are connected to a 512 KB L2 cache (16 way set associative). The bus interface connects the L2 cache to the main memory, and system bus.

Let us now consider the timing of the pipeline. The Bobcat integer pipeline is divided into 16 stages. Because of the deep pipeline, it is possible to clock the core at frequencies between 1-2 GHz. The Bobcat pipeline has 6 fetch cycles, and 3 decode cycles. The last 3 fetch cycles are overlapped with the decode cycles. The renaming engine, and scheduler take 4 more cycles. For most integer instructions, we require 1 cycle to read the register file, 1 cycle to access the ALU, and 1 more cycle to write the results back to the register file. The floating point pipeline has 7 additional stages, and the load-store unit requires 3 additional stages for address generation, and data cache access.

A.2.2 AMD Bulldozer

As the name suggests, the Bulldozer core (original paper [Butler et al., 2011]) is at the other end of the spectrum, and is primarily meant for high end desktops, workstations, and servers. Along with being an aggressive out-of-order machine, it also has multithreading capabilities. The Bulldozer is actually a combination of a multicore, fine grained multithreaded processor and an SMT. The Bulldozer core is actually a “conjoined core”, which consists of two smaller cores that share functional units.

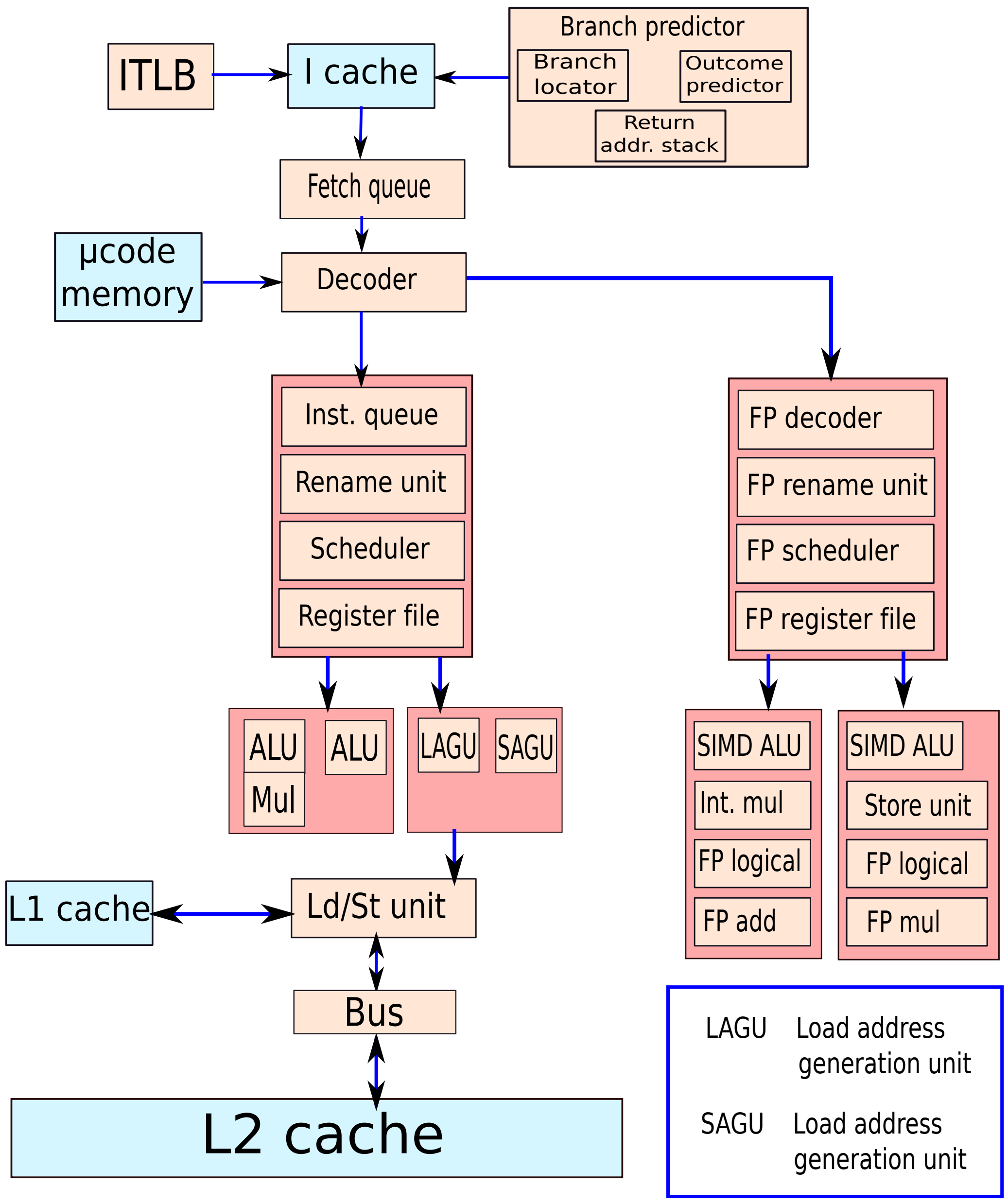

Figure A.6: The pipeline of the AMD Bobcat processor. ©[2011] IEEE. Adapted and reprinted, with permission. Source [Burgess et al., 2011]

Overview

Both the Bulldozer threads share the fetch engine (refer to Figure A.7), and decode logic. This part of the pipeline (known as the *front end*) switches between the two threads once every cycle, or few cycles. The integer, load-store, and branch instructions, are then dispatched to one of the two cores. Each core contains an instruction scheduler, register file, integer execution units, L1 caches, and a load-store unit. We can think of each core as a self sufficient core without instruction fetch and decode capabilities. Both the cores share the floating point unit that runs in SMT mode. It has its dedicated scheduler, and execution units. The

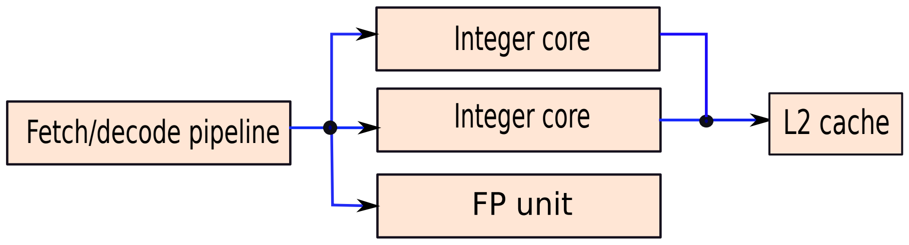

Figure A.7: Overview of the Bulldozer processor

Bulldozer processor is designed to run server as well as numerical workloads at 3-4 GHz. The maximum power dissipation is limited to 125-140W.

Detailed Design

Let us now consider a more detailed view of the processor in Figure A.8.

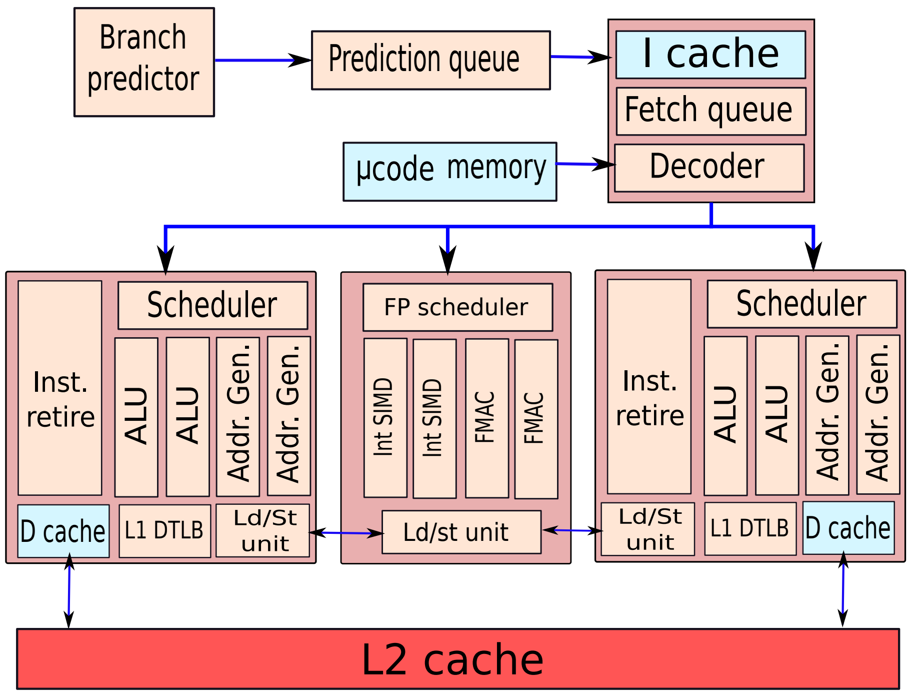

Figure A.8: Detailed view of the Bulldozer processor. ©[2011] IEEE. Adapted and reprinted, with permission. Source [Butler et al., 2011]

The Bulldozer processor has twice the fetch width of the Bobcat processor. It can fetch and decode up to 4 x86 instructions per cycle. Akin to Bobcat, the Bulldozer processor has sophisticated branch prediction logic that predicts whether an instruction is a branch, the branch outcome, and the branch target. It has a multilevel branch target buffer that saves the predicted branch targets of roughly 5500 branch instructions. The decode engine converts x86 instructions into Cops. One Cop in AMD is a CISC instruction albeit sometimes simpler than the original x86 instruction. Most x86 instructions get converted to just one Cop. However, some instructions get translated to more than one Cop, and it is sometimes necessary to use the microcode memory for instruction translation. An interesting aspect of the decode engine is that it can dynamically merge instructions to make a larger instruction. For example, it can merge a compare instruction, and a subsequent branch instruction, into one Cop. This is known as *macro-instruction fusion*.

Subsequently, the integer instructions are dispatched to the cores for execution. Each core has a rename engine, instruction scheduler (40 entries), a register file, and a 128 entry ROB. The core's execution unit consists of 4 separate pipelines. Two pipelines have ALUs, and two other pipelines are dedicated to memory address generation. The load-store unit co-ordinates access to memory, forwards data between stores to loads, and performs aggressive prefetching using stride prefetchers. Recall that stride prefetchers can automatically deduce array accesses, and fetch from array indices that are most likely to be accessed in the future.

Both the cores share a 64 KB instruction cache. However, each core has a 16 KB write-through L1 cache, where each load access takes 4 cycles. The L1 caches are connected to an L2 cache that comes in various sizes (ranging from 1-2 MB in the design presented in [Butler et al., 2011]). It is shared across the cores, and has a 18 cycle latency.

The floating point unit is shared between both the cores. It is more than a mere functional unit. We can think of it as an SMT processor that schedules and executes instructions for two threads simultaneously. It has its own instruction window, register file, rename, and wakeup-select (out of order scheduling) logic. Bulldozer's floating point unit has 4 pipelines that process SIMD instructions (both integer and floating point), and regular floating point instructions. The first two pipelines have 128 bit floating point ALUs called FMAC units. An FMAC (floating point multiply accumulate) unit can perform an operation of the form ($a \leftarrow a + b \times c$), along with regular floating point operations. The last two pipelines have 128 bit integer SIMD units, and additionally the last pipeline is also used to store results to memory. Lastly, the floating point unit has a dedicated load-store unit to access the caches present in the cores.

A.3 Intel® Processors

Let us now discuss the design of Intel processors. As of writing this book (2012-13) Intel processors dominate the laptop and desktop markets. In this section, we shall present the design of two Intel processors that have very different designs. The first processor is the Intel® Atom™, which has been designed for mobile phones, tablets, and embedded computers. At the other end of the spectrum lies the Sandy Bridge multicore, which is a part of the Intel® Core™i7 line of processors. These processors are meant to be used by high end desktops and servers. Both of these processors have very different business requirements. This has translated to two very different designs

A.3.1 Intel® Atom™

Overview

The Intel Atom processor started out with a unique set of requirements (see [Halfhill, 2008]). The designers had to design a core that was extremely power efficient, had enough features to run commercial operating systems and web browsers, and was fully x86 compatible. A naive approach to reduce power would have been to implement a subset of the x86 ISA. This approach would have led to a simpler and more power efficient decoder. Since the decoding logic is known to be power hungry in x86 processors, reducing its complexity is one of the simplest methods to reduce power. However, full x86 compatibility precluded this option.

Hence, the designers were forced to consider novel designs that are extremely power efficient, and do not compromise on performance. Consequently, they decided to simplify the pipeline, and consider 2-issue inorder pipelines only. Recall from our discussion in Section 9.11.4 that out-of-order pipelines have complicated structures for finding the dependences between instructions, and for executing them out of order. Some of these structures are the instruction window, renaming logic, scheduler, and wakeup-select logic. These structures add to the complexity of the processor, and increase its power dissipation.

Secondly, most Intel processors typically translate CISC instructions into RISC like micro-ops. These micro-ops execute like normal RISC instructions in the pipeline. The process of instruction translation consumes a lot of power. Hence, the designers of the Intel Atom decided to discard instruction translation. The Atom pipeline processes CISC instructions directly. For some instructions that are very complicated, the Atom processor does use a microcode ROM to translate them into simpler CISC instructions. However, this is more of an exception that the norm.

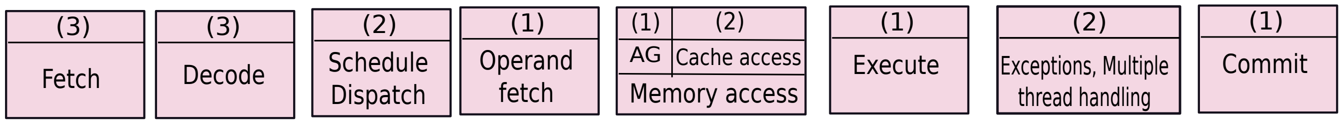

Figure A.9: The pipeline of the Intel Atom processor. (AG → address generation) ©[2008] The Linley Group. Adapted and reprinted, with permission. (Originally published in the Microprocessor Report. source [Halfhill, 2008])

As compared to RISC processors, the fetch and decode stages are more complicated in CISC processors. This is because instructions have variable lengths, and demarcating instruction boundaries is a tedious process. Secondly, the process of decoding is also more complicated. Hence, Atom dedicates 6 stages out of its 16-stage pipeline to instruction fetch and decoding as shown in Figure A.9. The remaining stages perform the traditional functions of register file access, data cache access, and instruction execution. Along with the simpler pipeline, another hallmark feature of the Intel Atom processor is that it supports 2-way multithreading. Modern mobile devices typically run multitasking operating systems, and users run multiple programs at the same time. Multithreading can support this requirement, enable additional parallelism, and reduce idle time in processor pipelines. The last 3 stages in the pipeline are dedicated to handling exceptions, handling multithreading related events, and writing data back to register or memory. Like all modern processors, store instructions are not on the critical path. In general, processors that do not obey sequential consistency write their store values to a write buffer and proceed with executing subsequent instructions.

Detailed Design

Let us now describe the design in some more detail. Let us start with the fetch, and decode stages (see Figure A.10). In the fetch stage, the Atom processor predicts the direction and target of branches, and fetches a stream of bytes into the instruction prefetch buffers. The next task is to demarcate instructions in the fetched stream of bytes. Finding, the boundaries of x86 instructions is one of the most complicated tasks performed by this part of the pipeline. Consequently, the Atom processor has a 2 stage *pre-decode* step that adds 1 bit markers between instructions, after it decodes them for the first time. This step is performed by the ILD (instruction length decoder) unit It then saves the instructions in the I cache. Subsequently, pre-decoded instructions fetched from the I cache can bypass the pre-decoding step, and directly proceed to the decoding step because its length is already known. Saving these additional markers, reduces the effective size of the I cache. The size of the I cache is 36 KB; however, after adding the markers, it is effectively 32 KB. The decoder does not convert most CISC instructions into RISC like micro-ops. However, for some complicated x86 instructions, it is necessary to translate them into simpler micro-ops by accessing

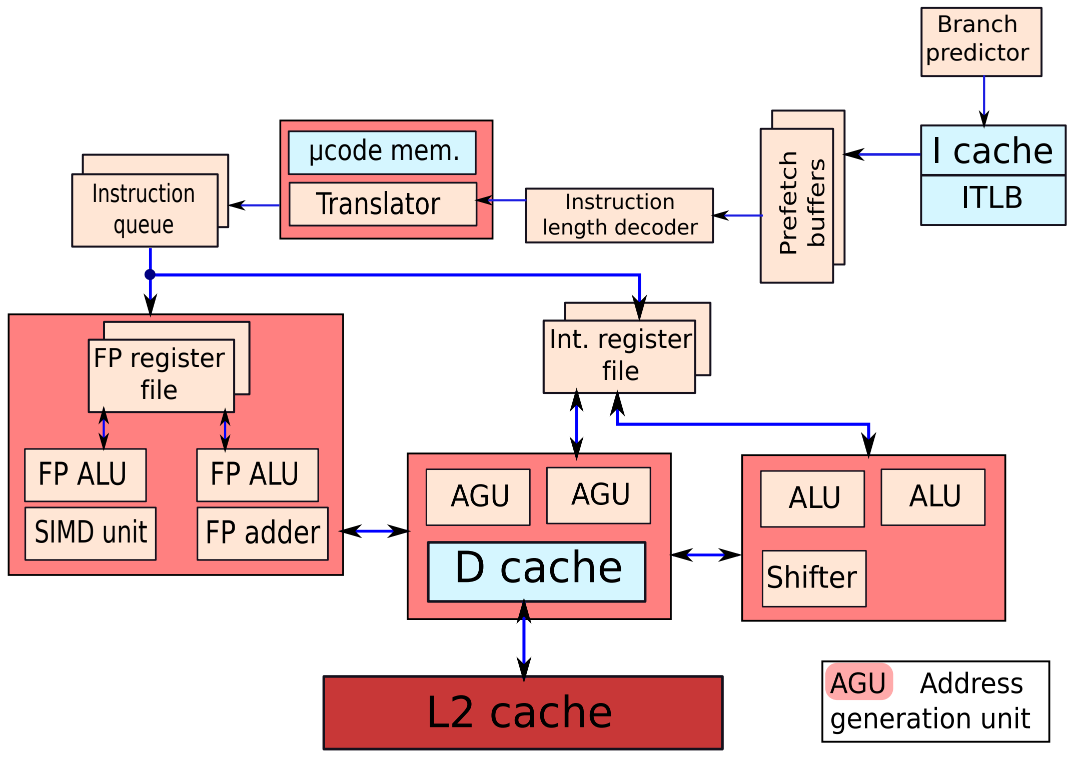

Figure A.10: A block diagram of the Intel Atom processor ©[2008] The Linley Group. Adapted and reprinted, with permission. (Originally published in the Microprocessor Report. source [Halfhill, 2008])

the microcode memory.

Subsequently, integer instructions are dispatched to the integer execution units, and the FP instructions are dispatched to the FP execution units. Atom has two integer ALUs, two FP ALUs, and two address generation units for memory operations. For supporting multithreading, it is necessary to have two copies of the instruction queue (1 per thread), and two copies of the integer and FP register files. Instead of creating a copy of a hardware structure like an instruction queue, Intel follows a different approach. For example, in the Atom processor, the 32 entry instruction queue is split into two parts (with 16 entries each). Each thread uses its part of the instruction queue.

Let us now discuss a general point about multithreading. Multithreading increases the utilisation of resources on a chip by decreasing the time that they remain idle. Thus, a multithreaded processor is ideally expected to have a higher power overhead (because of higher activity), and also have better instruction throughput. It is important to note that unless a processor is designed wisely, the throughput might not predictably increase. Multithreading increases the contention in shared resources such as the caches, the TLBs, and the instruction schedule/dispatch logic. Especially, the caches get partitioned between the threads, and we expect the miss rates to increase. Similar is the case for the TLBs also. On the other hand, the pipeline need not remain idle in the shadow of an L2 miss or in low ILP (instruction level parallelism) phases of a program. Hence, there are pros and cons of multithreading, and we have performance benefits only when the good effects (performance increasing effects) outweigh the bad effects (contention increasing effects).

A.3.2 Intel Sandy Bridge

Overview

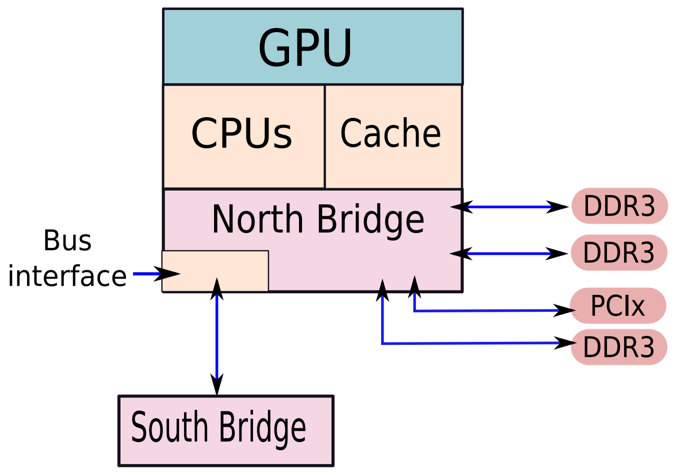

Figure A.11: Overview of the Sandy Bridge processor ©[2010] The Linley Group. Adapted and reprinted, with permission. (Originally published in the Microprocessor Report. Source [Gwennap, 2010])

Let us now discuss the design of a high performance Intel processor called the Sandy Bridge processor, which is a part of some of the latest (as of 2012) Intel Core i7 processors in the market. The main aims [Gwennap, 2010] while designing the Sandy Bridge processor was to support emerging multimedia workloads, numerically intensive applications, and multicore friendly parallel applications.

The most distinguishing features of the Sandy Bridge processor is that it contains an on-chip graphics processor. The graphics processor is loaded with specialised units for performing image rendering, video encoding/decoding, and custom image processing. The CPU and GPU communicate through a large shared on chip L3 cache. An overview of the Sandy Bridge processor is shown in Figure A.11.

Along with the addition of more components on chip, a lot of modifications to the CPU were also made. Sandy Bridge processor has full support for the new AVX instruction set, which is a 256 bit SIMD instruction set. For each SIMD unit, it is possible to perform 4 double precision operations or 8 single precision operations simultaneously. Since so many high performance features were being added it was necessary to add many power saving features also. Nowadays, techniques such as DVFS (dynamic voltage frequency scaling), clock gating (shutting off the clock) and power gating (shutting off the power for a set of functional units) for unused units are common. Additionally, the Sandy Bridge designers modified the design of the core to minimise copying values between units as much as possible (recall that a similar design decision was also taken by the designers of AMD Bobcat also), and also made basic changes to the design of some core structures such as the branch predictor and branch target buffer for power efficiency.

An important point to note here is that processors such as the Intel Sandy Bridge are designed to support multiple cores (4-8). Secondly, each core supports 2-way multithreading. Hence, we can run 16 threads on an 8 core machine. These threads can actively communicate between each other, the L3 cache banks, the GPU, and the on chip North Bridge controller. With so many communicating entities we need to design flexible on chip networks that facilitate high bandwidth and low latency communication. The designers of the Sandy Bridge have opted for a ring based interconnect over the traditional bus.

The Sandy Bridge processor was designed for a 32 nm [1] semiconductor process. Its successor is the Intel Ivy Bridge processor, which has the same design, but is designed for a 22 nm process.

Detailed Design

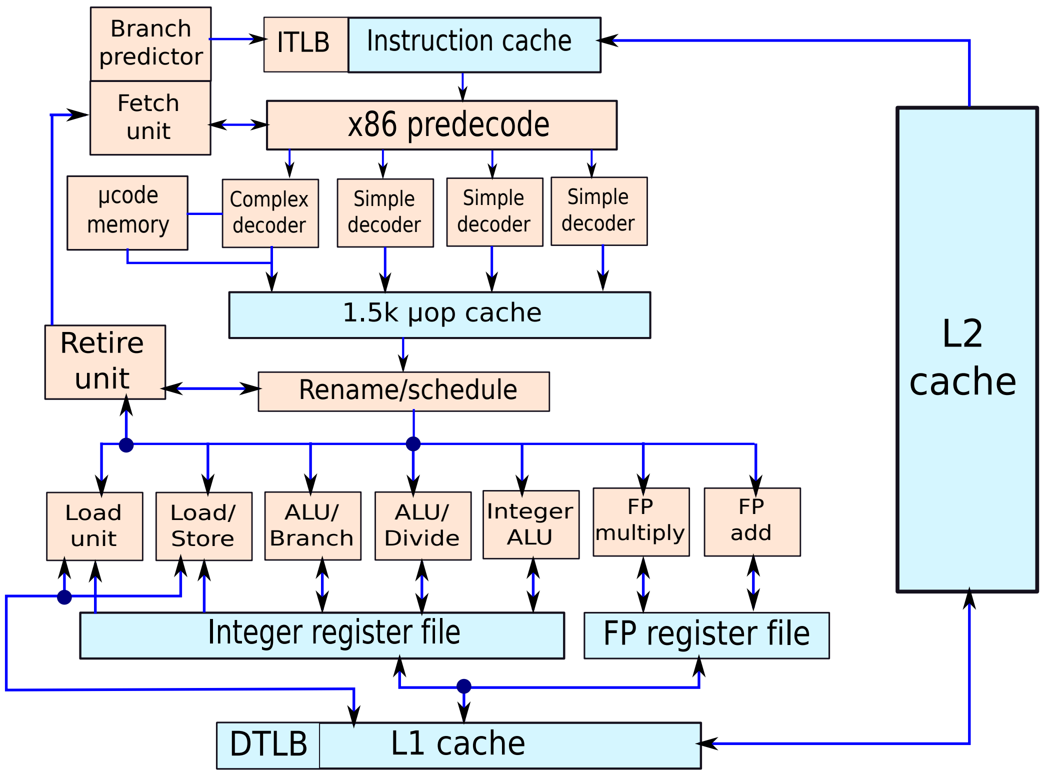

Figure A.12: Detailed view of the Sandy Bridge processor ©[2010] The Linley Group. Adapted and reprinted, with permission. (Originally published in the Microprocessor Report. Source [Gwennap, 2010])

Let us now consider the detailed design of a Sandy Bridge Core in Figure A.12 (also refer to [Gwennap, 2010]). The Sandy Bridge processor has a 32 KB instruction cache, that can provide 4 x86 instructions every cycle. The first step in decoding a stream of x86 instructions, is to demarcate their boundaries (known as predecoding). Once, 4 instructions are predecoded, they are sent to the decoders. Sandy Bridge has 4 decoders. Three of them are simple decoders, and one decoder is known as a complex decoder that uses the microprogram memory. All the decoders convert CISC instructions into RISC like micro-ops. Sandy Bridge has a L0 cache for micro-ops that can store roughly 1500 micro-ops. The L0 micro-op cache has performance benefits in terms of both performance and power. It reduces the branch misprediction latency if the instruction at the branch target is available in the L0 cache. Since most of the branches in programs are near branches, we expect to have a good hit rate at the L0 cache. Secondly, we can also save power. If an instruction's micro-ops are available in the L0 cache, then we do not need to fetch, predecode, and decode the instruction once again. We thus avoid these power hungry operations.

We need to point out an interesting design decision (see [Gwennap, 2010]) that was taken by the designers with respect to the branch predictor. This design decision is representative of many similar problems in

[1] The smallest possible structure that can be fabricated reliably has a dimension of 32 nm

computer architecture. One such problem is whether we should design a small structure with complicated entries, or should we design a large structure with simple entries? For example, should we have a 4-way associative 16 KB cache, or a 2-way associative 32 KB cache? In general, there is no definite answer to questions of this nature. They are highly dependent on the nature of the target workloads. For the Sandy Bridge processors, the designers had a choice. They could have either chosen a branch predictor with 2-bit saturating counters, or a predictor with more entries, and a 1 bit saturating counter. The power and performance tradeoffs of the latter design was found to be better. Hence, they chose to have 1 bit counters.

Subsequently, 4 micro-ops are sent to the rename and dispatch units that perform out-of-order scheduling. In earlier processors such as the Nehalem processor, temporary results of instructions that were in flight were saved in the ROB. Once the instructions finished, they were copied to the register file. This operation involves copying data, and is thus not efficient from the point of view of power. Hence, Sandy Bridge avoids this, and saves results directly in the physical register file similar to high performance RISC processors. When an instruction reaches the rename stage, we check the mappings in the rename table, and find the ids of the physical registers that contain, or are supposed to contain at a future point of time, the values of source operands. We subsequently either read the physical register file, or wait for their values to be generated. In your author's view, using physical register files is a better approach than using other approaches that save the results of unfinished instructions in the ROB, and later copy the results back to the register files. Using physical register files is fast, simple, and power efficient. By using this approach in the Sandy Bridge processor, the ROB got simplified, and it was possible to have 168 in-flight instructions at any point of time.

The Sandy Bridge processor has 3 integer ALUs, 1 load unit, and 1 load/store unit. The integer units read and write their operands from a 160 entry register file. For supporting floating point operations, it has one FP add unit, and 1 FP multiply unit. They support the AVX SIMD instruction set (256-bit operations on sets of single and double precision numbers). Moreover, to support 256 bit operations, Intel added new 256-bit vector registers (YMM registers) in the x86 AVX ISA.

To implement the AVX instruction set, it is necessary to support 256-bit transfers from the 32 KB data cache. The Sandy Bridge processor can perform two 128 bit loads, and one 128 bit store per cycle. In the case of loading a YMM (256 bit) register, both the 128 bit load operations are fused into one (256 bit) load operation. Sandy Bridge has a 256 KB L2 cache, and a large (1-8 MB) L3 cache that is divided into banks. The L3 banks, cores, GPU, and North Bridge controllers are connected using a unidirectional ring based interconnect. Note that the diameter of an unidirectional ring is $(N-1)$ because we can send messages in only one direction. To overcome this restriction, each node is actually connected to two points on the ring. These points are diametrically opposite to each other. Hence, the effective diameter is close to $N/2$.

Let us conclude by describing a unique feature of the Sandy Bridge processor called *turbo mode*. The idea is as follows. Assume that a processor has a period of quiescence (less activity). In this case, the temperature of all the cores will remain relatively low. Now, assume that the user decides to perform a computationally intensive activity such as sharpen an image, and remove "red eyes." This requires numerically intensive computations that are also power intensive. Every processor has a rated thermal design power (TDP), which is the maximum amount of power a processor is allowed to dissipate. In the turbo mode, the processor is permitted to dissipate more power than the TDP for short durations of time (20-25s). This allows the processor to run all the units at a frequency higher than the nominal value. Once the temperature reaches a certain threshold, the turbo mode is automatically switched off, and the processor resumes normal operation. The main point to note is that dissipating a large amount of power for a short duration is not a problem. However, having very high temperatures for even a short duration is not allowed. This is because high temperature can permanently damage a chip. For example, if a wire melts, then the entire chip is destroyed. Since it takes several seconds for a processor to heat up, we can take advantage of this effect to have high frequency and high power phases to quickly complete sporadic jobs. Note that the turbo mode is not useful for long running jobs that take hours.

B Graphics Processors

B.1 Overview

High intensity graphics is a hallmark of contemporary computer systems. Today's computers from smart phones to high end desktops use a variety of sophisticated visual effects to enhance user experience. Additionally, users use computers to play graphics intensive games, watch high definition videos, and for computer aided engineering design. All of these applications require a significant amount of graphics processing.

In the early days, graphics support in computers was very rudimentary. The programmer needed to specify the co-ordinates of every single shape that was drawn on the screen. For example, to draw a line the programmer needed to explicitly provide the co-ordinates of the line, and specify its colour. The range of colours was very limited, and there was almost no hardware for off loading graphics intensive tasks. Since each line or circle drawn on the screen required several assembly statements, the process of creating, and using computer graphics was very slow. Gradually, a need arose to have some support for graphics in hardware.

Let us discuss a little bit of background before delving into hardware accelerated graphics. A typical computer monitor contains a matrix of pixels. A pixel is a small point on the screen. For example a 1920×1080 monitor has 1920 pixels in each row, and 1080 pixels in each column. Each pixel has a colour at a certain point of time. Modern computer systems can set 16 million colours for each pixel. A picture on a computer screen is essentially an array of coloured pixels, and a video is essentially a sequence of pictures. In a video, we typically show 50-100 pictures every second (known as the *refresh rate*), where one picture is marginally different from the previous one. The human eye cannot figure out the fact that the pictures on the computer screen are changing in rapid succession. The brain creates an illusion of a continuous animation.

B.1.1 Graphics Applications

We can divide modern graphics applications into two types. The first type is automatic image synthesis. For example, let us consider a complex scene in a game, where a character is running with a machine gun in a moonlit night. In this case, the programmer is not manually setting the value of every pixel to a given colour. This process is too slow and time consuming. None of our interactive games would work, if this method was used. Instead, the programmer writes a program at the level of high level objects. For example, she can define a scene with a set of objects like roads, plants, and obstacles. She can define a character, and the artefacts that he carries such as a machine gun, a knife, and a cloak. The programmer writes a program in terms of these objects. Additionally, she specifies a set of rules that define the interactions of such objects. For example, she might specify that if a character collides with a wall, then the character turns around and runs in the other direction. Along with defining objects, and the semantics of objects, it is essential to define the sources of light in a scene. In this case, the programmer needs to specify the intensity of light in a moonlit night. The illumination of the character and the background is then automatically calculated by dedicated graphics software and hardware.

Sadly, the graphics hardware does not understand the language of complex objects, and characters. Hence, most graphics toolkits have graphics libraries to break down complex structures into a set of basic shapes. Most shapes in computer graphics applications are broken down into a set of triangles. All operations such as object collision, movement, lighting, shadows, and illumination, are converted to basic operations on triangles. However, graphics libraries do not use the regular processor to process these triangles and ultimately create an array of pixels to be displayed on a computer screen. Once, the programmer's intent has been translated to operations on basic shapes, the graphics libraries send the code to *a dedicated graphics processor* that does the rest of the processing. The graphics processor generates a complex scene from data and rules supplied by the user. It operates on shapes specified by edges and vertices. Most of the time these shapes are triangles in 2D space, or tetrahedra in 3D space. The graphics processors also calculate the effect of lighting, object position, depth, and perspectives while generating the final image. Once, the graphics processor has generated the final image, it sends it to the display device. If we are playing a computer game, then this process needs to be done at least 50-100 times every second.

To summarise, we observe that since generating complex graphics scenes is difficult, and slow, programmers operate on a high-level description of objects. Subsequently, graphics libraries convert the programmer's directives into operations on basic shapes, and send the set of shapes, and rules for operating on them to the graphics processor. The graphics processor generates the final scene by operating on the basic shapes, and then converting them to an array of pixels.

The second important application of graphics processors is to display animated content such as videos. A high definition video has millions of pixels per scene. To reduce the storage requirements, most high definition videos are severely compressed (encoded). Hence, a computer needs to decode or decompress a video, compute the array of pixels 50-100 times per second, and display them on screen. This is a very compute intensive process, and can tie up the resources of a CPU. Hence, video decoding is also typically offloaded to the graphics processor that contains specialised units for handling videos.

Almost all modern computer systems contain a graphics processor, and it is referred to as the GPU (Graphics Processing Unit). A modern GPU contains more than 64-128 cores, and thus is designed for extensive parallel processing.

B.1.2 Graphics Pipeline

Let us now look at the pipeline of a typical graphics processor in Figure B.1.

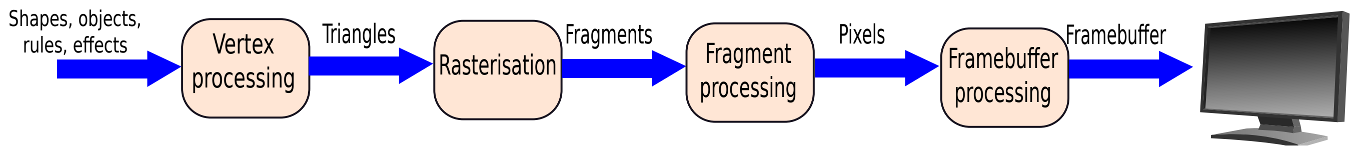

Figure B.1: A graphics pipeline [Blythe, 2008]

The first stage is called *vertex processing*. In this stage, the set of vertices, shapes, and triangles, are processed. The GPU performs complex operations such as object rotation, and translation. The programmer might specify that she wants a given object to move at a certain velocity towards another object. It is thus necessary to translate the position of a shape at a given rate. Such operations are also performed in this stage. The output of this stage is a set of simple triangles in a 2D plane.

The second stage is known as *rasterisation.* The process of rasterisation converts each of the triangles into a set of pixels, known as *fragments.* Moreover, it associates each pixel in a fragment with a set of parameters. These parameters are later used to interpolate the value of the colour. The third stage does *fragment processing.* This stage either colours the pixels of a fragment according to a fixed set of rules using the intermediate results computed in the previous stage, or, it maps a given texture to the fragment. For example, if a fragment represents the surface of a wooden table, then this stage maps the texture of wood to the colours of the pixels. This stage is also used to incorporate effects such as shadows and illumination.

Note that up till now we have computed the colours of the fragments for all the objects in a scene. However, it is possible that one object might be in front of another object, and thus a part of the second object might be hidden. The fourth stage aggregates all the fragments from the third stage, and performs an operation called *frame buffer* processing. The frame buffer is a large array that contains the colour values for each pixel. The graphics card passes the frame buffer to the display device 50-100 times per second. One of the operations performed in this stage is known as *depth buffering.* Here, it computes a 2D view of the 3D space at a certain angle by hiding parts of objects. Once the final scene is created, the graphics pipeline transfers the image to the frame buffer.

This is precisely the way complex games, or even standard operations such as minimising or maximising a window are rendered by a graphics processor. *Rendering* is defined as the process of generating a scene in pixels, by processing the high level description of a scene in terms of objects, rules, and visual effects. Discussing the exact details of rendering is beyond the scope of this book. The interested reader can refer to a book on computer graphics [Hughes et al., 2013]. The only point that the user should appreciate is that the process of rendering essentially involves a lot of linear algebra operations. Readers familiar with linear algebra will quickly appreciate the fact that object rotation or translation are all matrix operations. These operations process a large number of floating point values, and are inherently parallel.

B.1.3 Fusion of High Performance Computing and Graphics Computing

Towards the late nineties, there was a rapid growth in the field of computer graphics. There was a surge in computer gaming, desktop visual effects, and advanced engineering software that required sophisticated hardware accelerators for computer graphics. Hence, designers increasingly faced the need to create more vivid scenes, and more life like objects. Readers can compare the animation movies produced in the late eighties and today's Hollywood movies. Animated movies today have very life like characters with very detailed facial expressions. Thanks to graphics hardware, all of this is possible. To create such immersive experiences it became necessary to add a great degree of flexibility in graphics processors to incorporate different kinds of visual effects. Hence, graphics processor designers exposed a lot of internals of the processor to low level software and allowed the programmers to use the processor more flexibly. A set of programs called shaders were born in the early years of 2000. They allowed the programmer to create flexible fragment, and pixel processing routines.

By 2006, major GPU vendors had the realisation that the graphics pipeline can be used for general purpose computations also. For example, heavily numerical scientific code is conceptually similar to fragment, or pixel processing operations. Hence, if we allow regular user programs to access the graphics processor to perform their tasks then we can run a host of scientific programs on graphics processors. In response to this requirement, NVIDIA released the CUDA API that allowed C programmers to write code in C, and run it on a graphics processor. Hence, the term GPGPU (General Purpose GPU) was born.

Definition 164
GPGPU stands for General Purpose Graphics Processing Unit. It is essentially a graphics processor that allows regular users to write and run their code on it. Users typically use dedicated languages, or extensions to standard languages for producing code that is compatible with GPGPUs.

In this book we shall discuss the design of the NVIDIA® Tesla® GPU architecture. Specifically, we will discuss the design of the GeForce® 8800 GPU. The fastest parts of the GPU (the cores) typically operate at 1.5 GHz. Other parts operate at 600 MHz, or 750 MHz.

B.2 NVIDIA Tesla Architecture

Figure B.2 shows the Tesla architecture. Let us start explaining from the top of the figure. The host CPU sends sequences of commands and data to the graphics processor through a dedicated bus. The dedicated bus then transfers the set of commands and data to buffers on the GPU. Subsequently, the units of the GPU process the information. In Figure B.2 the work flows from top to bottom. Before we start discussing the details of the GPU, the reader needs to understand that the GPU is essentially a set of very simple inorder cores. Additionally, it has a large amount of extra hardware to co-ordinate the execution of complex tasks and allocate work to the set of cores. The GPU also supports a multilevel memory hierarchy, and has specialised units that exclusively perform a few graphics specific operations.

B.2.1 Work Distribution

Three kinds of work can be assigned to the GPU – vertex processing, pixel processing, and regular computing jobs. The GPU defines its own assembly code, which uses the PTX and SASS instruction sets. Each instruction in these instruction sets defines a basic operation on the GPU. It uses either register operands, or memory operands. Unlike CPUs the structure of the register file in a GPU is typically not exposed to software. The programmer is expected to use an unlimited number of virtual registers. The GPU or the device driver map them to real registers.

Now, for processing vertices, low level graphics software sends a sequence of assembly instructions to the GPU. The GPU has a hardware assembler that produces binary code, and sends it to a dedicated vertex processing unit that co-ordinates and distributes the work among the cores in the GPU. Alternatively, the CPU can send pixel processing operations to the GPU. The GPU does the process of rasterisation, fragment processing, and depth buffering. A dedicated unit in the GPU generates code snippets for these operations, and sends them to a pixel processing unit that distributes the work items among the set of GPU cores. The third unit is a compute work distributor that accepts regular computational tasks from the CPU such as adding two matrices, or computing a dot product of two vectors. The programmer specifies a set of subtasks. The role of the compute work distribution engine is to send these set of subtasks to cores in the GPU.

Beyond this stage, the GPU is more or less oblivious of the source of the instructions. Note that this piece of engineering is the key contribution behind making GPUs successful. Designers have successfully split the functionality of a GPU into two layers. The first layer is specific to the type of operation (graphics or general purpose). The role of each pipeline in this stage is to transform the specific sequence of operations into a generic set of actions such that irrespective of the nature of the high level operation, the same set of hardware units can be used. Let us now take a look at the second half of the GPGPU that contains the compute engines.

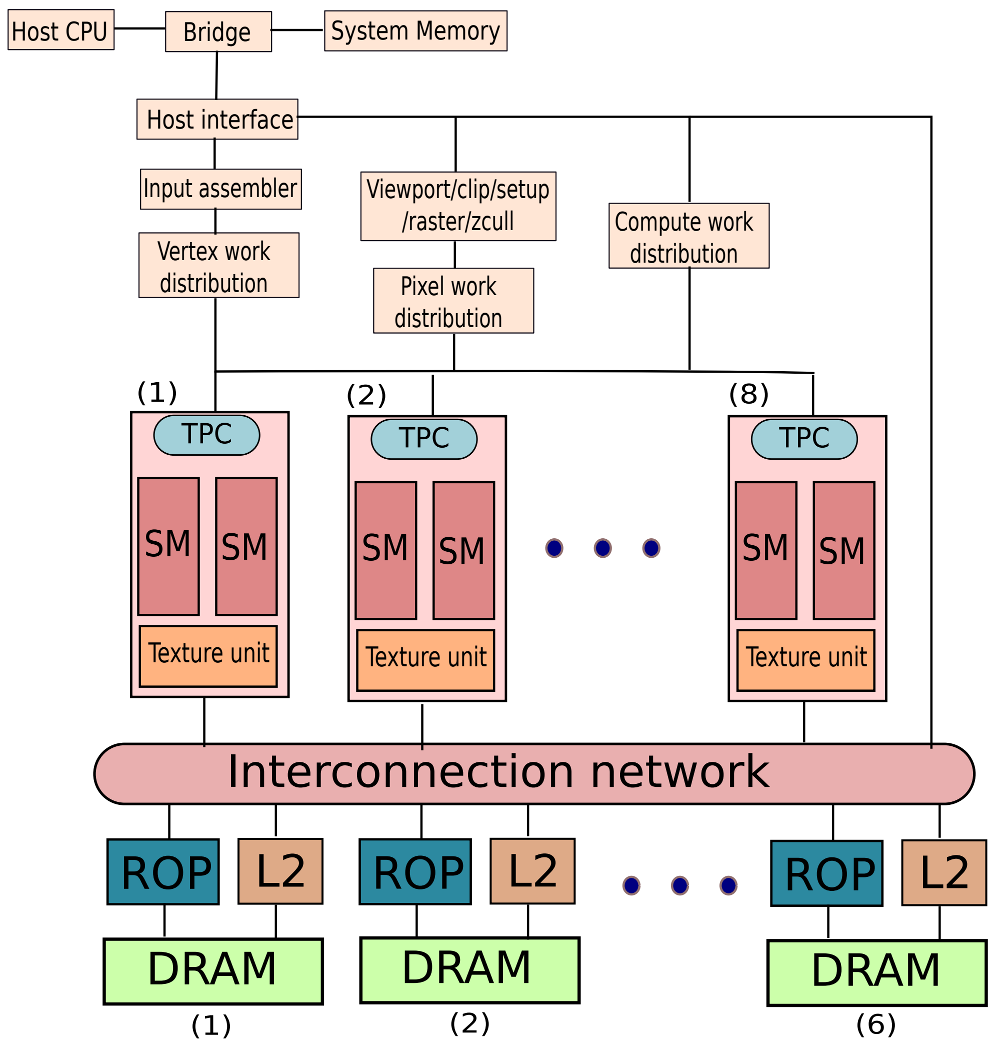

Figure B.2: NVIDIA Tesla Architecture ©[2008] IEEE. Reprinted, with permission. Source [Lindholm et al., 2008]

B.2.2 GPU Compute Engines

The GeForce 8800 GPU has 128 cores. The Cores are organised into 8 groups. Each group is known as a TPC (texture/ processor cluster). Each TPC contains two SMs (Streaming Multiprocessors). Moreover, each SM contains 8 cores known as streaming processors (SPs). Each SP is a simple inorder core that has an IEEE 754 compliant floating point ALU, branch and memory access units. Along with the set of simple cores, each SM contains some dedicated memory structures. These memory structures contain constants, texture data, and GPU instructions. All the SPs can execute a set of instructions in parallel, and are tightly synchronised with each other.

B.2.3 Interconnection Network, DRAM Modules, L2 Caches, and ROPs

The 8 TPCs are connected via an interconnection network to a set of caches, DRAM modules, and ROPs (raster operation processors). The SMs contain the first level caches. Upon a cache miss, the SP cores access the relevant L2 cache bank via the NOC. In the case of GPUs, the L2 cache is a shared cache split at the level of banks. Beneath the L2 caches, GPUs have a large external DRAM memory. The GeForce 8800 had 384 pins to connect to external DRAM modules. The set of pins are divided into 6 groups, where each group contains 64 pins. The physical memory space is also split into 6 parts, across the 6 groups. Rasterisation operations typically require some specialised processing routines. Unfortunately, these routines run inefficiently on TPCs. Hence, the GeForce 8800 chip has 6 ROPs. Each ROP processor can process at the most 4 pixels per cycle. It mostly interpolates the colour of pixels, and performs colour blending operations.

We are primarily interested in the design of the SMs. Hence, let us look at them in slightly greater detail.

B.3 Streaming Multiprocessors (SMs)

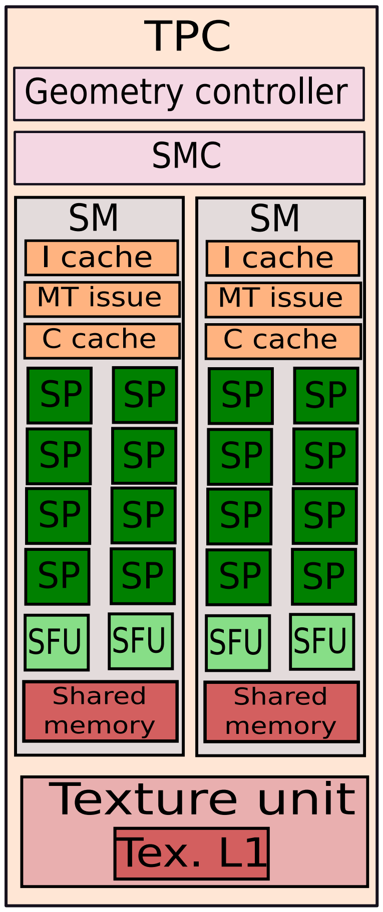

Figure B.3: A texture/processor cluster ©[2008] IEEE. Reprinted, with permission. Source [Lindholm et al., 2008]

Figure B.3 shows the structure of a TPC with two SMs. The geometry controller orchestrates vertex and shape processing on the individual cores. It brings in vertex data from the memory hierarchy, directs the cores to process them, and then co-ordinates the process of storing the output in the memory hierarchy. Moreover, it also helps in forwarding the output to the next stage of processing. The SMC (SM controller) schedules the requests for external resources. For example, it is possible that multiple cores in an SM might want to write to the DRAM memory, or access the texture unit. In this case, the SMC arbitrates the requests. Let us now look at the structure of an SM.

Each SM has an I Cache (instruction cache), a C cache (constant cache), and a built in thread scheduler for multithreaded workloads (MT Issue Unit). The 8 SP cores can access the shared memory unit embedded in the SM for communication between themselves. An SP core has an IEEE 754 compliant floating point ALU. It can perform regular floating point operations such as add, subtract, and multiply. It also supports a special instruction called multiply-add, which is required very frequently in graphics computations. This instruction computes the value of the expression: $a * b + c$. Along with the FP ALU each SP has an integer ALU that can execute regular integer instructions, and logical instructions. Moreover, the SP core can execute memory instructions, and branch instructions. Similar to vector processors, SP cores implement predicated instructions. This means that they dedicate issue slots to instructions in the wrong path; though, they are replaced with *nop* instructions. The SPs are optimised for speed, and are the fastest units in the entire GPU. This is due to the fact that they implement a very simple RISC like instruction set, which consists mostly of basic instructions.

For computing more sophisticated mathematical functions such as transcendental functions or trigonometric functions there are two special function units (SFUs) in each SM. The SFUs also have specialised units for interpolating the value of colours inside a fragment. GPUs use this functionality for colouring the inside of each triangular fragment. Along with specialised units, the SFUs have regular integer/ floating point ALUs also that are used to run general purpose codes.

The two SMs in a TPC share a texture unit. The texture unit can simultaneously process four threads, and fill the triangles produced after rasterisation with the texture of the surface associated with the triangles. The texture information is stored in a small cache within the texture unit. Upon a cache miss, the texture unit can fetch data from the relevant L2 cache, or from main DRAM memory. Now that we have described the different parts of a GPU, let us discuss how to perform a computation on a GPU.

Each thread in an SM (mapped to an SP) can either access per thread local memory (saved on external DRAM), or shared memory (shared across all the threads in an SM, and saved on chip), or global DRAM memory. Programmers can explicitly direct the GPU to use a certain kind of memory.

B.4 Computation on a GPU

The graphics processing model is actually a combination of multi-threading, multi-programming, and SIMD execution. NVIDIA calls its model SIMT (Single Instruction, Multi-threaded). Let us look at NVIDIA's SIMT execution model.

The programmer starts out by writing code in the CUDA programming language. CUDA stands for Compute Unified Device Architecture. It is a custom extension to C/C++ that is compiled by NVIDIA's *nvcc* compiler to generate code in both the CPU's ISA (for the CPU), and in the PTX instruction set (for the GPU). A CUDA program contains a set of *kernels* that run on the GPU and a set of functions that run on the host CPU. The functions on the host CPU transfer data to and from the GPU, initialise the variables, and co-ordinate the execution of kernels on the GPU. A *kernel* is defined as a function that executes in parallel on the GPU. The graphics hardware creates multiple copies of each CUDA kernel, and each copy executes on a separate thread.

The GPU maps each such thread to an SP core. It is possible to seamlessly create and execute hundreds of threads for a single CUDA kernel. An astute reader might argue that if the code is the same for multiple

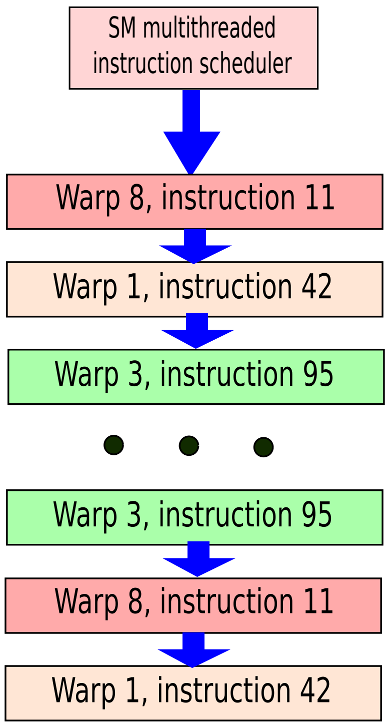

Figure B.4: Scheduling of warps ©[2008] IEEE. Reprinted, with permission. Source [Lindholm et al., 2008]

copies then what is the point of running multiple copies. Well, the answer is that the code is not exactly the same. The code implicitly takes the id of the thread as an input. For example, if we generate 100 threads for each CUDA kernel, then each thread has an unique id in the set $[0 \dots 99]$. Based on the id of the thread, the code in the CUDA kernel performs appropriate processing. Recall that we had seen a very similar example, when we had written OpenMP programs (see Example 121). Now, it is possible that the threads of many separate applications might be running at the same time. The MT issue logic of each SM schedules the threads and co-ordinates their execution. An SM in this architecture can handle up to 768 threads.

If we are running multiple applications in parallel then the GPU as a whole will need to schedule thousands of threads. The scheduling overhead is prohibitive. Hence, to make the task of scheduling simpler, the GeForce 8800 GPU groups a set of 32 threads into a *warp*. Each SM can manage 24 warps. A warp is an atomic unit of threads, and either all the threads of a warp are scheduled, or no thread in the warp is scheduled. Moreover, all the threads in a warp belong to the same kernel, and start at exactly the same address. However, after they have started they can have different program counters.

Each SM maps the threads of a warp to SP cores. It executes the warp instruction by instruction. This is similar to classic SIMD execution, where we execute one instruction on multiple data streams, and then move to the next instruction. The SM executes an instruction for each thread in the warp, and after all the threads have completed the instruction, it executes the next instruction. If the kernel has a branch that is data or thread dependent, then the SM executes instructions for only those threads that have instructions in the correct branch path. The GeForce GPU uses predicated instructions. For the instructions on the wrong path, the predicated condition is false. Hence, these instructions are dynamically replaced with *nop* instructions. Once the branch paths (taken, and not taken) reconverge, all the threads in a warp become active again. The main difference from the SIMD model is that in a SIMD processor, the same thread handles multiple data streams in the same instruction. Whereas, in this case, the same instruction is executed in multiple threads, and each instruction operates on different data streams. After executing an instruction in a warp the MT issue unit might schedule the same warp, another warp from the same application, or a warp from another application. The GPU essentially implements fine grained multithreading at the level of warps. Figure B.4 shows an example.

For executing, a 32 thread warp, an SM typically uses 4 cycles. In the first cycle, it issues 8 threads to each of the 8 SP cores. In the second cycle, it issues 8 more threads to the SFUs. Since the two SFUs have 4 functional units each, they can process 8 instructions in parallel without any structural hazards. In the third cycle, 8 more threads, are sent to the SP cores, and finally in the fourth cycle, 8 threads are sent to the two SFU cores. This strategy of switching between using SFUs, and SP cores, ensures that both the units are kept busy. Since a warp is an atomic unit, it cannot be split across SMs, and each instruction of the warp must finish executing for all the active threads, before we can execute the next instruction in the warp. We can conceptually equate the concept of warps to a 32 lane wide SIMD machine. Multiple warps in the same application can execute independently. To synchronise between warps we need to use global memory, or sophisticated synchronisation primitives available in modern GPUs.

B.5 CUDA Programs

A CUDA program naturally maps to the structure of a GPU. We first write a kernel in CUDA that performs a set of operations depending on the thread id that it is assigned at runtime. A dynamic instance of a kernel is a *thread* (similar to a thread in the context of a CPU). We group a set of threads into a *block*, or a *CTA* (co-operative thread array). A block or a CTA corresponds to a warp. We can have 1–512 threads in a block, and each SM can buffer the state of at most 8 blocks at any point of time. Each thread in a block has a unique thread id. Similarly, blocks are grouped together in a *grid*. The grid contains all the threads for an application. Different blocks (or warps) may execute independently of each other, unless we explicitly enforce some form of synchronisation. In our simple example, we consider a block to be a linear array of threads, and a grid to be a linear array of blocks. Additionally, we can define a block to be a 2D or 3D array of threads, or a grid to be a 2D or 3D array of blocks.

Let us now look at a small CUDA program to add two n element arrays. Let us consider the CUDA program in parts. In the following code snippet, we initialise three arrays a, b, and c. We wish to add a and b element wise and save the results in c.

```
#define N 1024

void main() {
        /* Declare three arrays a, b, and c */
        int a[N], b[N], c[N];

        /* Declare the corresponding arrays in the GPU */
        int size = N * sizeof(int);
        int *gpu_a, *gpu_b, *gpu_c;

        /* allocate space for the arrays in the GPU */
        cudaMalloc((void**) &gpu_a, size);
        cudaMalloc((void**) &gpu_b, size);
        cudaMalloc((void**) &gpu_c, size);

        /* initialise arrays, a and b */
        ...

        /* copy the arrays to the GPU */
        cudaMemcpy (gpu_a, a, size, cudaMemcpyHostToDevice);
        cudaMemcpy (gpu_b, b, size, cudaMemcpyHostToDevice);
```

In this code snippet we declare three arrays (a, b, and c) with N elements in Line 5. Subsequently, in Line 9, we define their corresponding storage locations (*gpu_a*, *gpu_b* and *gpu_c*) in the GPU. We then allocate space for them in the GPU by using the *cudaMalloc* call. Next, we initialise arrays a and b with values (code not shown), and then copy these arrays to the corresponding locations (*gpu_a*, and *gpu_b*) in the GPU using the CUDA function *cudaMemcpy*. It uses a flag called *cudaMemcpyHostToDevice*. In this case the *host* is the CPU and the *device* is the GPU.

The next operation is to add the vectors *gpu_a*, and *gpu_b* in the GPU. For this purpose, we need to write a *vectorAdd* function that can add the vectors. This function should take three arguments consisting of two input vectors, and an output vector. Let us for the time being assume that we have such a function with us. Let us show the code to invoke this function.

```
vectorAdd <<< N/32, 32 >>> (gpu_a, gpu_b, gpu_c);
```

We invoke the vectorAdd function with three arguments: *gpu_a*, *gpu_b* and *gpu_c*. Let us now look at the expression: $<<< N/32, 32 >>>$. This piece of code indicates to the GPU that we have $N/32$ blocks, and each block contains 32 threads. Let us now assume that the GPU magically adds the two arrays and saves the results in the array *gpu_c* in its physical memory space. The last step in the *main* function is to fetch the results from the GPU, and free space in the GPU. The code for it is as follows.

```
/* Copy from the GPU to the CPU */
cudaMemcpy (c, gpu_c, size, cudaMemcpyDeviceToHost);

/* free space in the GPU */
cudaFree (gpu_a);
cudaFree (gpu_b);
cudaFree (gpu_c);

} /* end of the main function */
```

Now, let us define the function *vectorAdd*, which needs to be executed on the GPU.

```
/* The GPU kernel */
__global__ void  vectorAdd ( int *gpu_a, int *gpu_b, int *gpu_c) {
        /* compute the index */
        int idx =  threadIdx.x + blockIdx.x * blockDim.x;

        /* perform the addition */
        gpu_c[idx] = gpu_a[idx] + gpu_b[idx];
}
```

Here, we access some built in variables that are populated by the CUDA runtime. In general, a grid and a block have three axes (x, y, and z). Since we assume only one axis in the blocks and the grid in this example, we only use the x axis. The variable $blockDim.x$ is equal to the number of threads in a block. If we would have considered 2D grids, then the dimension of a block would have been $blockDim.x \times blockDim.y$. $blockIdx.x$ is the index of the block, and $threadIdx.x$ is the index of the thread in the block. Thus, the expression $threadIdx.x + blockIdx.x * blockDim.x$ represents the index of the thread. Note that in this example, we associate each element of the arrays with a thread. Since the overhead of creation, initialisation, and switching of threads is small, we can adopt this approach in the case of a GPU. In the case of a CPU that has large overheads with creating and managing threads, this approach is not feasible. Once, we compute the index of the thread, we perform the addition in Line 7.

The GPU creates N copies of this kernel, and distributes it among N threads. Each of the kernels computes a different index in Line 4, and proceeds to perform the addition in Line 7. We showed a simple example. However, it is possible to write extremely complicated programs using the CUDA extensions to C/C++ replete with synchronisation statements, and conditional branch statements. The reader can consult the book by Farber [Farber, 2011] for an in-depth coverage of CUDA programming.

To summarise, let us show the entire GPU program. Note that we club the kernel of the GPU along with the code that is executed by the CPU into a single program. NVIDIA's compiler splits the single file into two binaries. One binary runs on the CPU and uses the CPU's instruction set, and the other binary runs on the GPU and uses the PTX instruction set. This is a classical example of a MPMD style of execution where we have different programs in different instruction sets, and multiple streams of data. Thus, we can think of the GPU's parallel programming model as a combination of SIMD, MPMD, and fine grained multithreading at the level of warps. We leave the readers with an artist's impression of a GPU (see Figure B.5).

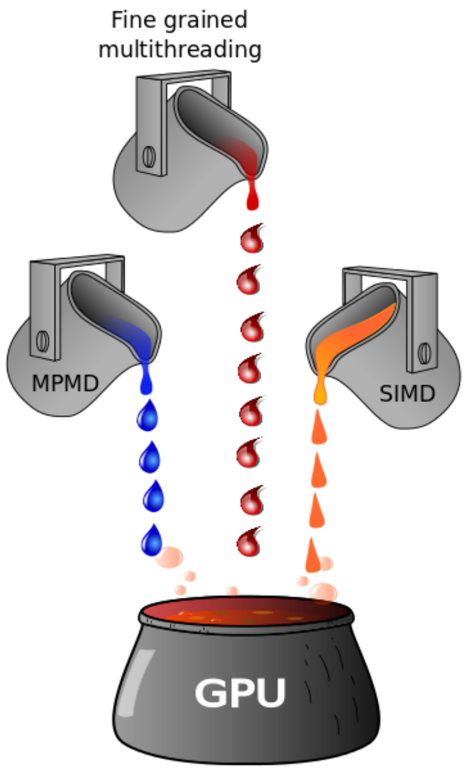

Figure B.5: Artist's impression of a GPU

```
#define N 1024

/* The GPU kernel */
__global__ void  vectorAdd ( int *gpu_a, int *gpu_b, int *gpu_c) {
    /* compute the index */
    int idx =  threadIdx.x + blockIdx.x * blockDim.x;

    /* perform the addition */
    gpu_c[idx] = gpu_a[idx] + gpu_b[idx];
}
void main() {
    /* Declare three arrays a, b, and c */
    int a[N], b[N], c[N];

    /* Declare the corresponding arrays in the GPU */
    int size = N * sizeof(int);
    int *gpu_a, *gpu_b, *gpu_c;

    /* allocate space for the arrays in the GPU */
    cudaMalloc((void**) &gpu_a, size);
    cudaMalloc((void**) &gpu_b, size);
    cudaMalloc((void**) &gpu_c, size);

    /* initialise arrays, a and b */
    ...

    /* copy the arrays to the GPU */
    cudaMemcpy (gpu_a, a, size, cudaMemcpyHostToDevice);
    cudaMemcpy (gpu_b, b, size, cudaMemcpyHostToDevice);

    /* invoke the vector add operation in the GPU */
    vectorAdd <<< N/32, 32 >>> (gpu_a, gpu_b, gpu_c);

    /* Copy from the GPU to the CPU */
    cudaMemcpy (c, gpu_c, size, cudaMemcpyDeviceToHost);

    /* free space in the GPU */
    cudaFree (gpu_a);
    cudaFree (gpu_b);
    cudaFree (gpu_c);

} /* end of the main function */
```