Dan Negrut

Primer:
Elements of Processor Architecture.
The Hardware/Software Interplay.

Supplemental material provided for
ECE/ME/EMA/CS 759

October 23, 2013

Table of Contents

Preface: Purpose and Structure

This primer is intended to help graduate students in disciplines such as Mechanical Engineering, Chemistry, Civil Engineering, or Biology with answering a simple question: "How can parallel computing augment my research and its impact?". These are individuals who took a C, C++, or Java programming class who understand the power of Data Analytics and Computational Science but lack the skills to leverage existing hardware to solve, for instance, a climate modeling problem at a resolution that offers unprecedented level of detail; providing the computational support to sequence the human genome and identify relevant mutations in matters of seconds; enable real time 3D visualization of the human heart; perform an electronic structure computation that backs up an accurate molecular dynamics simulation at the nanoscale; or resolve through direct numerical simulation two phase flows and spray dynamics for improved internal combustion engine design. A lot of emphasis has been recently placed in Science and Engineering on using modeling and computer simulation as alternatives to experimental and testing efforts. A validated computer program indeed can serve as a very effective and cost-conscious alternative to experimental tests in guiding and accelerating scientific discovery and engineering innovation. Moreover, recent advances in data sensing and collection has led to a deluge of raw data that needs to be processed, interpreted, and leveraged in fields as diverse as health care, social networks, and genetics.

The purpose of this primer is to help the student get a basic understanding of the architecture of modern processors and the hardware and software context within which they operate. This effort was motivated by the belief that in order to write good software a basic understanding of the hardware that will execute one's program will help. Understanding how the hardware works enables one to visualize the process of parallel execution. This skill takes time to master but it is good to possess since it allows for: (a) expeditious design of parallel programs; and (b) writing software that runs fast and correctly. In other words, it enables one to get it done and have it done right.

As the title suggests, this is a primer. The presentation follows a 10-80 approach: with 10% of the discussion required to exhaustively cover a topic the hope is that the most useful 80% of the concepts will be understood. Depth was not a priority, nor was breadth. The priority was in providing enough information to help with the process of designing fast and correct parallel code. As such, this text does not attempt to present a comprehensive landscape of the topics covered. My knowledge would not allow it, and you would probably not be interested on a treatise on the topic of instruction level parallelism, for instance.

The community revolving around computers approaches them from three perspectives. There are colleagues seeking answers to difficult issues, e.g., how to design a better instruction architecture set, how to improve pipelining and out of order execution, how a memory consistency model should be formulated. They will find this primer too basic and short on details. At the other end of the spectrum, there are colleagues for which computers provide a means to an end. The computer can be used to watch a movie, send email, or run an off-the-shelf commercial Computer Aided Engineering program to produce data that is subsequently processed to yield insights in engineering design. This textbook is meant to assist the group in between, colleagues who need a *basic understanding and intuition* into the inner workings of a multiprocessor computer to develop software that can solve larger domain specific problems faster. Recent trends suggest that the required discipline boundary crossing by engineers and scientists into a territory that traditionally has been identified with the Computer Science and/or Computer Engineering fields is increasingly desirable. This primer attempts to smoothen the discipline boundary crossing, a process as worthwhile as frustrating at times. The role of parallel computing is bound to increase and there is a clear need for engineers and scientists who are savvy computer users.

Document Structure

In its structure, this primer reflects a trend that obdurately held for a decade: sequential computing model has passed its high-water mark owing to realities in the microprocessor industry. Parallel computing is riding a rising tide owing to better use of power per amount of computation, better ability to hide memory latencies with useful computation, and a simpler microarchitecture that does not need to handle complexities associated with instruction level parallelism.

The selection of topics in this primer caters to students with no formal training in computer architecture or operating systems. The primer tries to cover these two topics at a pace that would take four weeks of class to go through the material. In-depth coverage of each of these two topics typically takes two semester courses and PhD theses continue to be inspired by problems in these two areas. As such, it should come as no surprise if after reading the section on buses (not the yellow things that take kids to school but the conduits for information passing in a computer system), one will be left with unanswered questions. Rather than providing exhaustive coverage of each topic, the goal is to introduce the concepts that combine to form a framework in which the process of parallel execution can be understood and visualized, the two ultimate goals of any engineer or scientist who makes the computer a means to an end.

Presently, the primer is organized in two chapters. The first provides an overview of critical building blocks that combine to make the computer work. The discussion first introduces the Von Neumann computing model and the concept of instruction and transistor. The machine instruction holds a work order that eventually will lead to a choreographed interplay of groups of transistors to fulfill the action spelled out by this instruction. The fetch-decode-execute cycle is analyzed, which requires the prior discussion of the concepts of control unit, arithmetic logic unit, and registers. Physical memory issues are covered next, which brings to the front the concepts of memory hierarchy, random access memory and cache along with two key attributes: latency and bandwidth. The interplay between hardware and software is the topic of a separate section, which discusses the virtual memory and a support cast that includes translation-lookaside-buffers, addressing techniques, and memory organization. A final section is dedicated to the laborious process of starting with source code and obtaining a program that might rely on third party libraries at link time.

The second chapter builds on the first one and provides an overview of one CPU processor; i.e., Intel’s Haswell, and one GPU card; i.e. Nvidia’s Fermi GTX480, in an exercise that places in context the concepts of the first chapter. The chapter sets off with a discussion of the three walls in sequential computing: memory, power, and instruction level parallelism, that are generally regarded as responsible for a gradual embracing of microarchitecture designs aimed at parallel computing. The steady and fast holding of Moore's law is what allowed the recent release of a 1.4 billion transistor four core CPU by Intel in June of 2013 and a seven billion GPU card by Nvidia in late 2012. The two architectures, code name Haswell and Fermi, respectively, are used as vehicles that will carry the reader towards a better understanding of three important components of a chip design: the front end; i.e., the decoder and scheduler, the memory ecosystem, and the back end; i.e., the execution/functional units.

This is version 0.1 of this primer. It has rough edges and it is very much work in progress, an effort that eventually will lead to a companion textbook for ME/CS/ECE/EMA 759. Any constructive feedback will be greatly appreciated and will carry an automatic acknowledgment in any future version of this primer and derived products that might include this material. You can post your comments online at http://sbel.wisc.edu/Forum/viewtopic.php?f=15&t=352 or you can reach me at negrut@wisc.edu.

Acknowledgments

I'm very thankful to all the students who have taken my "High Performance Computing for Engineering Applications" class, which was first time offered back in 2008 and taken by eight Mechanical Engineering students who would have not cared much hadn't been for me being their advisor. I learned along with each generation of these students, and to a very large extent this document represents a collection of things that were, and in some respects continue to be, rather esoteric to me.

In particular, I'd like to thank Andrew Seidl who provided the sys admin support that came in so handy so many times during all these past years; to Hammad Mazhar, a roaming spirit when it comes to computing, for his enthusiasm and passion – it's contagious; to Ang Li, for being a fun person to work with and also for taking the time to go through an early version of this document. A special thank you goes to Radu Serban, for carefully reading this document and providing very constructive comments.

Saving the best for last – I am very grateful to my best friend, who happens to have been married me, and our two crazy boys for allowing me to spend in my office the extra time required to put this document together.

1. Sequential Computing: Basic Architecture and Software Model

1.1. From high level language to executable instructions

Suppose you are interested in writing software that captures a new way of modeling turbulence in combustion. Or maybe you write software that implements a strategy that attempts to match two DNA sequences. Your code, saved in a source file, will combine with code stored in other source files to produce a program that is executed by a computer. To this end, chances are you will be writing C, C++, FORTRAN or maybe Java code. These are called high-level languages; they facilitate the process of conveying to the machine the steps that your algorithm needs to take in order to compare the two DNA sequences. High-level languages are a convenience. The computer does not need them and does not directly use them to compare the two DNA sequences. Instead, a compiler parses the high-level code and generates assembly code. Table 1 shows a simple C program and snippets of the assembly code generated by a compiler for the x86 architecture and by a different compiler for a reduced instruction set computing (RISC) architecture. These two computer architectures use different mnemonics to indicate a sequence of instructions that need to be carried out by the simple program. Typically, if the C source code is in a file with the extension "c" or "cpp", the assembly file would have the same name with the extension "s".

```c
int main(){
  const double fctr = 3.14/180.0;
  double a = 60.0;
  double b = 120.0;
  double c;
  c = fctr*(a + b);
  return 0;
}
```

```
   call     ___main
   fldl     LC0
   fstpl    -40(%ebp)
   fldl     LC1
   fstpl    -32(%ebp)
   fldl     LC2
   fstpl    -24(%ebp)
   fldl     -32(%ebp)
   faddl    -24(%ebp)
   fldl     LC0
   fmulp    %st, %st(1)
   fstpl    -16(%ebp)
   movl     $0, %eax
   addl     $36, %esp
   popl     %ecx
   popl     %ebp
   leal     -4(%ecx), %esp
   ret
LC0:
   .long 387883269
   .long 1066524452
   .align 8
LC1:
   .long 0
   .long 1078853632
   .align 8
LC2:
   .long 0
   .long 1079902208
```

```
main:
        .frame  $fp,48,$31
        #vars=32, regs=1/0, args=0, gp=8
        .mask   0x40000000,-4
        .fmask  0x00000000,0
        .set    noreorder
        .set    nomacro
        addiu   $sp,$sp,-48
        sw      $fp,44($sp)
        move    $fp,$sp
        lui     $2,%hi($LC0)
         …
        mul.d   $f0,$f2,$f0
        swc1    $f0,32($fp)
        swc1    $f1,36($fp)
        move    $2,$0
        move    $sp,$fp
        lw      $fp,44($sp)
        addiu   $sp,$sp,48
        j       $31
         …
$LC0:
        .word   3649767765
        .word   1066523892
        .align  3
$LC1:
        .word   0
        .word   1078853632
        .align  3
$LC2:
        .word   0
        .word   1079902208
        .ident  "GCC: (Gentoo 4.6.3
p1.6, pie-0.5.2) 4.6.3"
```

Table 1. C code (left) and assembly code associated with it. In the middle is a snippet of the assembly code generated by the gcc compiler when using the "–S" flag on an x86 machine. In the right column is a snippet of MIPS assembly code generated for the same C code.

A very qualified individual might bypass the high-level language and directly write assembly code. This is what used to be done in '60s and early '70 and it is still done today for critical parts of the code that require utmost optimization. Assembly code is however not portable because it is architecture specific. Essentially it is a low level programming language that uses mnemonics to instruct the processor what it needs to do, one instruction at a time. To this end, one line of assembly code translates into one processor instruction. This translation is carried out by the assembler whose output is a series of instructions, each of which stored in the main memory. An instruction is a cryptic thing. Specifically, take for instance an MIPS instruction in Table 1: `add $t0, $s1, $s2`. The 32-bit MIPS instruction associated with this assembly code is `00000010001100100100000000100000 [1]`. The assembly code that stated "store in the temporary register $t0 the sum of the two values stored in registers $s1 and $s2" ends up being represented as a series of architecture-specific 32 bits that embed all the information necessary to carry out the specific instruction. Note that the instruction that directs an x86 architecture to carry out the addition of two registers and storing of the result in a third one has a completely different bit pattern. Finally, a "register" represents a hardware asset capable of storing the value of a variable for a period of time and characterized by a very low latency associated with the process of storing (writing) and retrieving (reading) the value.

The "`add`" instruction above is one of several low level; i.e., elementary, instructions that the processor can be demanded to carry out. The collection of these low level instructions makes up the instruction set of the processor and along with it goes the concept of instruction set architecture (ISA). The ISA defines the instruction set but also a set of data types that go along with these instructions, a set of registers for data storage, the memory architecture, and protocols for interrupting the execution (interrupts and exception handling). Adopting an ISA is an important decision but it is a purely intellectual exercise, more like generating a wish list. Once adopted, the next hurdle is to provision for the hardware that implements the wish list, that is, the produce the processor's microarchitecture. Once a microarchitecture designed is agreed upon, one needs to ensure that a fabrication process is in place and capable of producing the microarchitecture that implements the ISA.

To gain an appreciation of what goes into designing hardware aware of what 00000010001100100100000000100000 means, one should start by understanding the format of an instruction. First, instructions can have different lengths in different ISAs. To keep things simple, the discussion here will pertain MIPS, which is a fixed length ISA, and focus on its 32-bit flavor [2].

The first six bits encode the basic operation; i.e., the **opcode**, that needs to be completed: adding two numbers (000000), subtracting two numbers (000001), dividing two numbers (000011), etc. The next group of five bits indicates in which register the first operand is stored, while the subsequent group of five bits indicates the register where the second operand is stored. Some instructions require an address or some constant offset. This information is stored in the last 16 bits. These would be the least significant bits; the most significant bits are associated with the **opcode**. This is the so called big endian convention, where the leftmost bits are the most significant ones, similar to the way digits make up the decimal numbers.

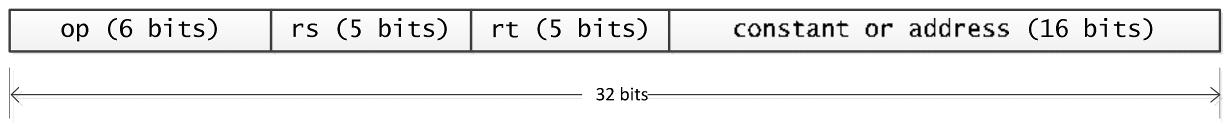


Figure 1. Format of a processor instruction of type I for the MIPS ISA. There are two more formats for MIPS instructions: type R from "register", and J-type from "jump". For instance, the R-type has the same first three fields op, rs, rt but packs three additional fields: the five bit rd field (register destination); the five bit shamt field (shift amount, something irrelevant here that assumes value 00000); and the six bit funct field, which is a function code that further qualifies the opcode [2].

Example 1.1
Consider the following line of C code:
`a[4] = delta + a[3];`
Assume that as a result of previous load operations the address of `a`, which is an array of integers that each takes four bytes of memory (one word), is stored in register `$s2` while the address of `delta` is stored in register `$s4`. The MIPS assembly for the line of code above would look like

```
lw  $t0, 12($s2)  # reg $t0 gets value stored 12 bytes from address in $s2
add $t0, $s4, $t0 # reg $t0 gets the sum of values stored in $s4 and $t0
sw  $t0, 16($s2)  # a[4] gets the sum delta + a[3]
```

The opcodes for the three instructions above are 100011 (`lw`), 000000 (`add`), and 101011 (`sw`) [2]. For the first instruction, using notation introduced in Figure 1, `rs=10010`, `rt=01000`, and the immediate address is 12, whose binary representation is `0000000000001100`. The second instruction is an R-type instruction for which `rs=10100` (first operand), `rt=01000` (second operand), `rd=01000` (destination of add operation), `shamt=00000` (irrelevant here), and `funct=100000`, the latter representing the code for `add`. Finally, for the third instruction `rs=10010`, `rt=01000`, and the immediate address is `0000000000010000`, which is the binary representation of 16.
For now, `rs` being 10100 or `rt` being 01000 should be interpreted as an encrypted way of saying that the operand for the `add` operation is stored at a location identified by 10100 or 01000, respectively. More context will provided when discussing registers, which these numbers are used to identify (see Example 1.5).

The instructions can now be pieced together to yield [3]

```
10001110010010000000000000001100
00000010100010000100000000100000
10101110010010000000000000010000
```

The sequence of three instructions above is what the processor will execute in order to fulfill the request embedded in the one line of C code above. This simple example makes an important point: there is a difference between the number of lines of code (C, C++, Java, etc.) and the number of instructions that are slated to be executed once that source code is compiled and turned into an executable program.

The task of translating assembly language into machine language is the task of the assembler. The outcome of this process is machine code, which is architecture specific. On reduced instruction set computing (RISC) architectures, the instruction is stored within one word, which, depending on the processor, can be 8, 16, 32 or 64 bits long. Although RISC architectures are encountered from cellular phone to supercomputers, the Scientific Computing community mostly uses the x86 architecture implemented by both Intel and AMD. It represents an example of a complex instruction set computing (CISC) architecture and in some cases uses several words to store one complex instruction.

In Example 1.1, the assembly instruction lw $t0, 12($s2) ended up encoded in the 32 bit sequence 10001110010010000000000000001100. Incidentally, when regarded as a base 2 number, this sequence of bits has the decimal representation 2387083276 and the hexadecimal representation 11C900018. The idea of representing instructions as numbers was promoted by Von Neumann, who proposed to store instructions in memory like any other numbers and subsequently read or even write them. This led to the concept of stored program, in which both instructions and data had the same representation and were stored similarly albeit in different regions of the memory. The computers used in Scientific Computing today draw on this model, sometimes known as the Princeton model. A schematic of the Von Neumann architecture that implements the Princeton model is provided in Figure 2.

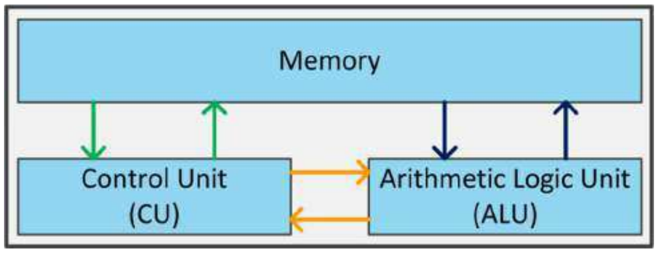


Figure 2. Schematic of the Von Neumann architecture

1.2. From transistors to the CPU

Casting Scientific Computing in the framework provided by the Von Neumann architecture yields a simple story: the control unit (CU) receives instructions from memory. Upon decoding these instructions, the CU relies on the arithmetic-logic unit (ALU) to carry out the encoded operations. In the process, the ALU needs to communicate with the memory for several purposes: to read; i.e., to get the operands, and write; i.e., to store the result. Arrows are depicted in both senses between any two of the three actors in this play: the CU, the ALU, and the memory. All these arrows depict information and data flow, which are both conveniently represented and stored as patterns of 1s and 0s in fixed-length (RISC) or variable-length (CISC) sequences of bits. These three actors are complex characters and this section will quickly sketch their roles in the overall play. The discussion will start with the CU, ALU, and the registers, the latter providing essential support for performing the duties of both the CU and ALU. Next, the fetch-decode-execute-store cycle associated with a typical instruction will be highlighted, along with an explanation of how the hardware supports this four-step process. The "bus" will be briefly discussed in its role of a conduit for information transfer between the play's actors. Finally, the ALU, CU, registers, and memory will be brought together to introduce the concept of execution pipeline and provide a description of its stages.

1.2.1. The CPU's control unit

The CU is the entity in charge of performing the machine cycle: fetch a machine instruction from memory; decode the instruction; execute the instruction; store the output. In each of these four steps, registers; i.e., very low latency memory storage available in hardware in the immediate vicinity of the CU, play a central role as they are the place holders of inputs and outputs associated with the instruction at hand. The collection of all registers is sometimes called the register file. The bigger this file is the better since it is the fastest form of memory that can store data and instructions on a computer. For instance, for the typical MIPS architecture, the size of the register file is 32 [2]. That is, there are 32 four byte registers used to store information critical to the execution of a program: pointers to the location in memory that stores the instruction being executed, temporary or persistent values needed to complete an instruction, arguments (inputs) passed to a function, output values passed back to the calling program by a function, values required by the operating system that hosts the executing program, etc.

The four stages of the machine cycle are managed by the CU. As soon as an instruction is available, the CU sets out to decode the 32-bit sequence; based on this pattern, it activates various gates on the chip to establish a data path. A data path can be envisioned as a sequence of steps that provided one or more inputs changes the state of the register file and/or produces some outputs.

In manufacturing, this is similar to a flexible production line at a car manufacturer that can receive orders for one of ten car models. First, an order for the sport model comes in, and the control unit decides which shops on the factory floor should participate in the process of assembling the sport car. Because there is a factory clock, the CU and each shop on the factory floor will be able to synchronize. The CU has the means to inform each shop exactly what components it needs to pick up, when, and where to deposit for the next shop to pick up and further process. To continue this analogy, the sequence of coordinated operations, both in space (the output of one should be fed into some other shop) and time (a shop should not pick up its input before a certain time since it might pick up leftover parts associated with the previously build pick-up truck), would combine to form an assembly line. This series of operations would establish a "car-production path", similar to the datapath that the CU manages. It is made up of datapath elements, which include at a minimum the instruction and data memories, the register file, and an arithmetic logic unit (ALU).

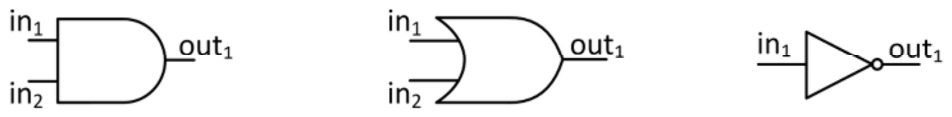


Figure 3. Standard representation of the AND, OR, and NOT (inverter) gates. The first two gates require two transistors to implement the expected functionality, while the inverter requires only one [2]. For the AND gate, when the voltage in_1 and voltage in_2 are nonzero, the voltage out_1 will be nonzero. For the OR gate, if the voltage in_1 or voltage in_2 is nonzero, the voltage out_1 will be nonzero. In reality, the voltage does not need to be zero; the digital elements operate with a high voltage and a low voltage. Abstractly, one of them is considered the logical TRUE signal and the other one logical FALSE signal. They map into the 1 and 0 states associated with a signal. The logic equation associated with these gates is $out_1 = in_1 \cdot in_2$, $out_1 = in_1 + in_2$, and $out_1 = \overline{in_1}$, respectively. The logic algebra is defined in Table 2.

For instance, the inverter gate that implements the NOT logical operation requires only one transistor. By configuring this transistor, when a positive voltage is applied to the base the transistor switches on and connects to the ground thus inverting the input applied at the base. Thus, a higher voltage in_1 (representing a bit with value 1) is converted into a low voltage out_1 (representing a bit with value 0).

Of essence here is how fast the transistors switch as well as their size. The former is important since the faster they switch the sooner the outcome of a logical operation becomes available. The latter is crucial since increasingly complex logic requires an increasingly larger number of transistors that combine to produce more complex functional units. It can be shown that any logic process, complex as it might be, can be characterized by a collection of the basic logic operations AND, OR, and NOT implemented in hardware by the three corresponding gates. In fact, any desired logic can be represented in the form of a logic equation expressed in terms of the algebra of the logic operations AND, OR, and NOT (Boolean algebra).

AND	in_2=0	in_2=1
in_1=0	0	0
in_1=1	0	1

OR	in_2=0	in_2=1
in_1=0	0	1
in_1=1	1	1

NOT	
in_1=0	1
in_1=1	0

Table 2. The three basic operations, or building blocks, of the Boolean algebra. The outcome of any basic operation is 0 or 1. In a logical equation in Boolean algebra AND is typically represented by a dot, $in_1 \text{AND}\, in_2 \equiv in_1 \cdot in_2$; OR is represented by +, $in_1 \text{OR}\, in_2 \equiv in_1 + in_2$; and NOT by an over-bar, $\text{NOT} in_1 \equiv \overline{in_1}$. AND and OR are commutative, associative, and, when combined, distributive. These three building blocks suffice to represent, through suitable combinations, any logic operation.

Example 1.2

Assume you are assigned the task of designing a digital logic block that receives three inputs via three bus wires and produces one signal that is 0 (low voltage) as soon as one of the three input signals is low voltage. In other words, it should return 1 if and only if all three inputs are 1. Effectively, you have to design a circuit that implements a specific "truth table"; i.e., a table that explicitly covers all possible scenarios given the three inputs. For the problem at hand, the truth table is captured in Table 3.

Based on the basic truth tables in Table 2 it can be shown that the required logic is captured by the following equation: $out = \overline{\overline{in_1} + \overline{in_2}} \cdot in_3$. Designing the digital logic block is now straightforward. One should negate the first two signals, perform an OR operation on them, negate again, and perform an AND operation of the intermediate result with the third signal. In order to simplify a logic block representation, shown in Figure 4, the inverter gate is typically represented as a small circle ○ on a bus line that enters a gate.

This solution is not the optimal one in terms of transistors used; in fact, several better designs exist. Also note that one could have implemented this block based on the $out = \overline{\overline{in_3} + \overline{in_2}} \cdot in_1$ logic equations. Nonetheless, the point is that a set of seven transistors: one AND gate, one OR gate, and three inverters implemented the required logic. That is, given three inputs via three bus wires, this circuit produces one signal that is low voltage as soon as one of the three input signals is low voltage.

in_1	in_2	in_3	out
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

Table 3. Truth table capturing the behavior expected of the digital circuit.

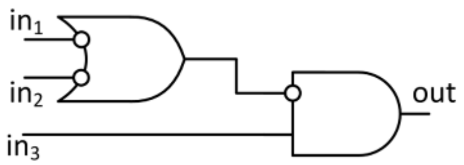


Figure 4. Logic block that implements the requested truth table. Inverters are represented by circles ○ applied to signals moving down the bus.

Example 1.3

Assume you need to design a logic block whose behavior can be changed based on the value of an input signal called "selector". The logic block is to output the value of the input in_1 if the selector is set to 0, and the value of the input in_2 if the selector is set to 1. This logic block is called a multiplexor, abbreviated as MUX, and is one of most basic logic blocks on which the Control Unit relies upon in controlling the datapath. A symbol used for MUXs in digital logic control and its hardware implementation are shown in Figure 5. The logic equation associated with a two input and one output MUX is $out = in_1 \cdot \overline{s} + in_2 \cdot s$.

in_1	in_2	s	out
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	1

Table 4. Truth table capturing the MUX behavior.

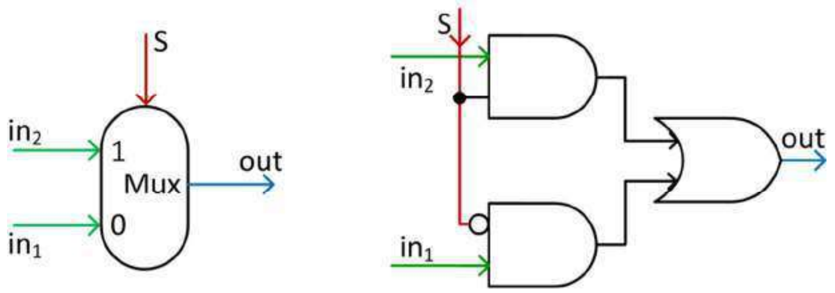


Figure 5. Typical representation of the MUX in digital logic control (left), and its hardware implementation in terms of gates (right) [2]. Based on the value of the selector s, the MUX can behave like an AND or OR gate.

1.2.2. The CPU's arithmetic logic unit

The arithmetic logic unit (ALU) is the component of the CPU that upon request is capable of adding, multiplying, or comparing two numbers. If the CU is in charge of defining and controlling the datapath, the ALU is the entity in charge of actually carrying out the math associated both with data and execution flow. The math part is obvious: numbers need to be added, subtracted, multiplied, etc. However, the ALU also plays an important role in the execution flow since it's called upon when, at an execution branching point, based on an offset, an address needs to be calculated to load the instruction that needs to be executed next.

To understand how the ALU is put together, let's focus first on a one bit ALU that adds up two bits. The process of adding the two bits with carryover is summarized in Figure 6. There are three inputs: the two signals to be added, along with a CarryIn bit or signal. There are two outputs: the sum and the value of the CarryOut. Based on the rules of the arithmetic of adding two binary numbers, the digital circuit that supports addition in the ALU should implement the truth table of Table 5. The task of producing the right logic block can be partitioned in two stages. First, one needs to identify the logic that given the three inputs produces the right CarryOut. Second, one should concentrate on the logic that yields the correct sum given the three inputs.

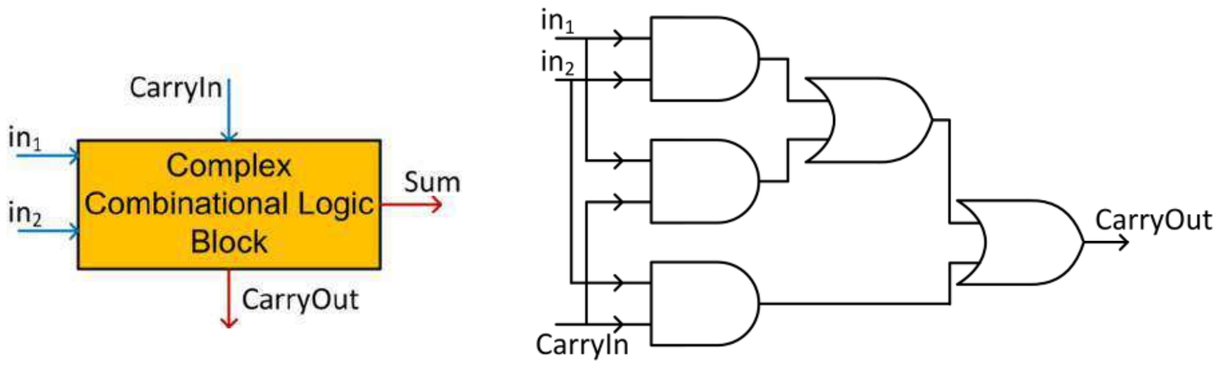


Figure 6. The set of inputs and outputs handled by a logic block whose purpose is to implement two bit addition with carryover (left). Hardware implementation in terms of gates that supports the logic associated with producing the CarryOut signal given the three inputs in_1, in_2, and CarryIn.

Inputs			Outputs		Comments
in_1	in_2	CarryIn	Sum	CarryOut	Sum is in base 2
0	0	0	0	0	
0	0	1	1	0	0+0 is 0; the CarryIn kicks in, makes the sum 1
0	1	0	1	0	
0	1	1	0	1	0+1 is 1, but CarryIn is 1; sum ends up being 0, CarryOut is 1.
1	0	0	1	0	
1	0	1	0	1	1+0 is 1, but CarryIn is 1; sum ends up being 0, CarryOut is 1.
1	1	0	0	1	1+1 is 0, carry 1.
1	1	1	1	1	1+1 is 0 and you CarryOut 1. Yet the CarryIn is 1, so the 0 in the sum becomes 1.

Table 5. Truth table for the one bit addition operation with carryover. The operations are associated with base 2 arithmetic.

Recalling the $+$ stands for OR and $\cdot$ stands for AND, the CarryOut logic is implemented by the following logic equation [2]:

$$\mathrm{CarryOut} = (\mathrm{in}_1 \cdot \mathrm{CarryIn}) + (\mathrm{in}_2 \cdot \mathrm{CarryIn}) + (\mathrm{in}_1 \cdot \mathrm{in}_2) + (\mathrm{in}_1 \cdot \mathrm{in}_2 \cdot \mathrm{CarryIn})$$

In other words, CarryOut is 1 as soon as two of the in_1, in_2, and CarryIn are 1. Based on this, CarryOut can actually be computed as

$$\mathrm{CarryOut} = (\mathrm{in}_1 \cdot \mathrm{CarryIn}) + (\mathrm{in}_2 \cdot \mathrm{CarryIn}) + (\mathrm{in}_1 \cdot \mathrm{in}_2),$$

since by dropping the last term we don't change the value of the quantity in the right-hand side of the logic equation.

The logic block that implements this logic equation is provided in Figure 6 (right): there are three AND gates corresponding to the three $\cdot$ operations in the logic equation along with two OR gates corresponding to the two $+$ operations. The logical block for the Sum is a bit more involved since it draws on a slightly more complex logic equation [2] to implement the fourth column of the table truth in Table 5:

$$\mathrm{Sum} = (\mathrm{in}_1 \cdot \overline{\mathrm{in}_2} \cdot \overline{\mathrm{CarryIn}}) + (\overline{\mathrm{in}_1} \cdot \mathrm{in}_2 \cdot \overline{\mathrm{CarryIn}}) + (\overline{\mathrm{in}_1} \cdot \overline{\mathrm{in}_2} \cdot \mathrm{CarryIn}) + (\mathrm{in}_1 \cdot \mathrm{in}_2 \cdot \mathrm{CarryIn})$$

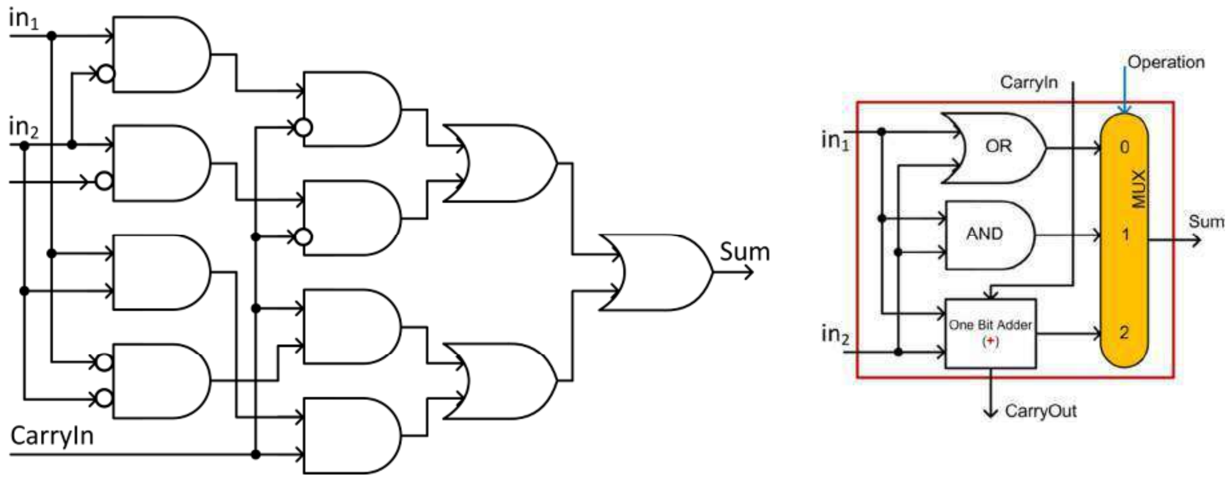


Figure 7. Hardware implementation in terms of gates to support the logic required to produce the Sum signal given the three inputs in_1, in_2, and CarryIn (Left). This logical unit, augmented with the logical unit that produces CarryOut (Figure 6, right), is packaged as in Figure 6, left, and then combined with a MUX to produce a one bit ALU complex combinational block (Right) capable of performing AND, OR, and addition.

The trek of increasingly complex hardware aggregation that supports increasingly sophisticated logic operations continues by bringing into the picture arrays of logic blocks. This is demonstrated through a somewhat simplified aggregation of a 32 bit ALU. Although in practice the ALU is typically implemented differently for improved performance and reducing the number of transistors used, Figure 8 illustrates a so-called ripple design, in which the output of one block becomes the input of the next block. In this case, the block ALUi, where i=0,…,31 is the 1 bit ALU of Figure 7 (right).

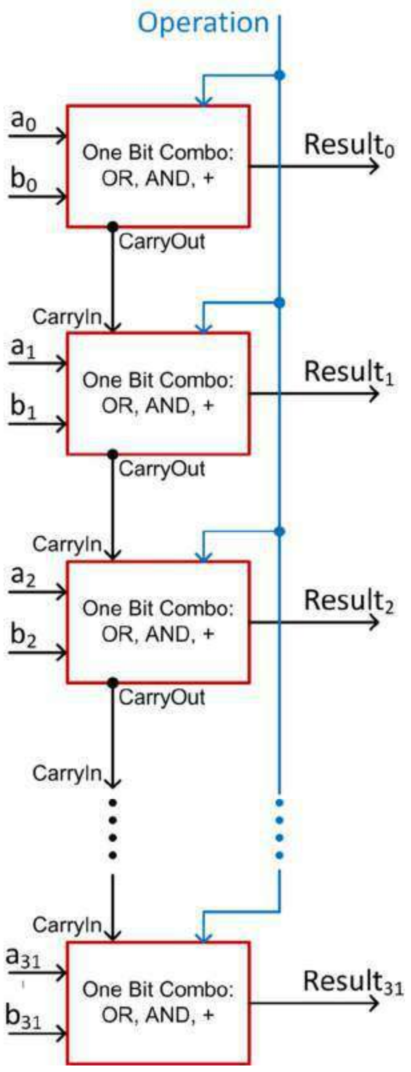


Figure 8. Ripple design using an array of logic elements: A set of 32 basic 1 bit ALU blocks are combined to produce a 32 bit ALU.

Although the discussion in this section focused on one possible way to support addition, AND, and OR operations, it is relatively straightforward to implement subtraction of two numbers by negating the second argument (the subtrahend) and then performing two additions: $\mathrm{in}_1 - \mathrm{in}_2 = \mathrm{in}_1 + \overline{\mathrm{in}_2} + 1$ [2]. Likewise, the logic blocks discussed so far can implement the NOR function: $\mathrm{NOT}(\mathrm{in}_1 \,\mathrm{OR}\, \mathrm{in}_2)$, by using DeMorgan's theorem: $\overline{\mathrm{in}_1 + \mathrm{in}_2} = \overline{\mathrm{in}_1} \cdot \overline{\mathrm{in}_2}$, which amounts to an AND operation on the inverted signals in_1 and in_2. Again, this is only one possible way, in reality more efficient ways are used to implement NOR and NAND.

There are many other basic operations that an actual 32 bit ALU needs to support. For instance, support for relational operators such as “set on less than” has not been discussed. The fundamental idea remains the same albeit implemented in a different way. Specifically, logical gates are combined to implement a logic equation. To summarize the message of this section, these transistors are combined into logic gates (AND, OR, and NOT), which are further combined into logical units. These logical units can subsequently be combined with “multiplexors”, or MUXs, which control the behavior of these logical units to produce complex combinational blocks. In this section we used an array of such identical blocks to illustrate how these complex hardware blocks provide the functionality required by basic operations in a 32 bit ALU. This increasingly more sophisticated process of aggregation, which requires a large number of transistors to implement a desired logic, is captured in Figure 9. It becomes apparent that it is desirable to have a large number of transistors per unit area. Historically, the number of transistors per unit area doubled roughly every 18 to 24 months. This is known as Moore's law, named after one of Intel's cofounders who postulated it in early 1970s. Moore's law has been accurate for four decades and took the microprocessor industry from 2300 transistors chip designs on a 12 mm^2 die in 1971, to the Intel 10-Core Xeon Westmere-EX chip with 2.6 billion transistors on 512 mm^2 in 2011. By mid-2013 it is anticipated that high end graphics cards will rely on chips that pack more than 7 billion transistors. Note that the current technology can be used to cram far more transistor per unit area, which of course will come at a price. In this context Moore's law concerns the number of transistors per unit area that makes the most sense from a return on investment point of view.

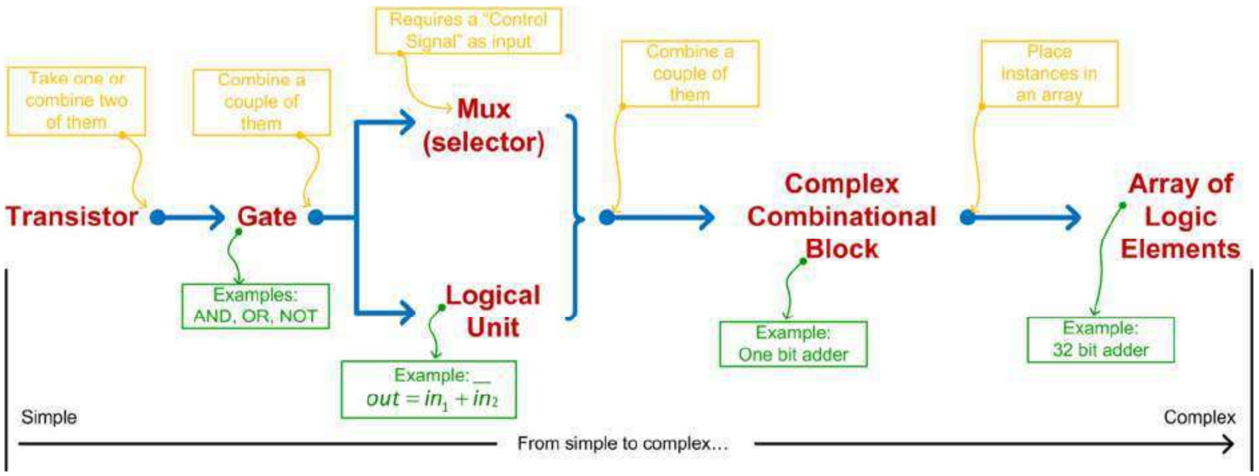


Figure 9. A schematic of how starting from transistors as building blocks, one can build up hardware entities capable of performing increasingly complex logical operations.

Sidebar 1.
In late 2011 Intel started manufacturing on a broad scale a radically new transistor design. It was called a three-gate, or 3D, transistor due to a narrow fin that came out of plane and rendered this design 3D. The benefit of the new design is an increase in the surface on which the electrons can travel. Before, the electrons travelled from the source to the drain in a fashion controlled by the gate. This happened by the small surface in the vicinity of the area pointed to by the arrow that shows the "High-K Dielectric". In 3D designs, the fin provides a larger area for the electrons to move from source to drain. This can be used in several ways: either decrease the size of the transistors since the area for electron movement got a boost, or keep a large area and shorten the time required by the electrons to move from the source to the drain. This means a faster on-off switch function provided by the transistor. A third option is to combine these trends to accomplish a compromise that still leads to a smaller but faster transistor. The amount of power required to control 3D transistors is also reduced.

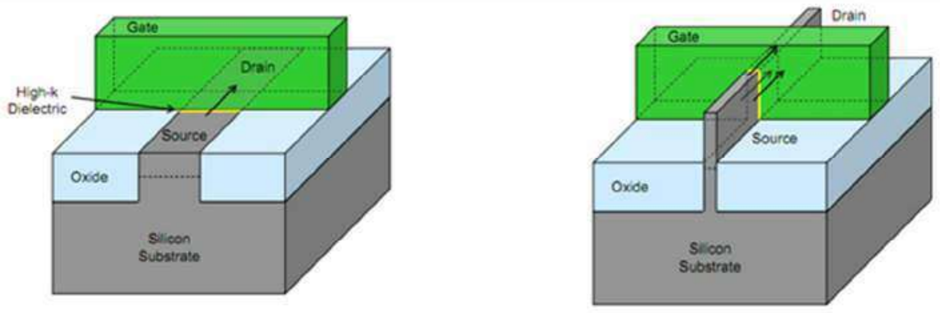


Figure 10. Traditional planar transistor (left) compared to new 3D design. The term "fin" is used sometimes in naming this type of transistors (like FinFET). Source: Intel.

1.2.3. Registers

In sequential computing, the CPU might be analyzing and handling several instructions at a time, an approach adopted to speed up execution through speculative execution and/or branch prediction and/or out of order execution. Even when this is done, the execution of the code proceeds sequentially following a deterministic path that ends up ignoring, for instance, all but one of the speculative paths adopted by the processor in anticipation of the actual execution path. The repeating mantra is that of bringing over an instruction from memory, decoding it, and executing it. This fetch-decode-execute, abbreviated FDX, draws on registers for fast memory that stores both information related to the instruction being executed and the data that is processed by this instruction. Registers are an extremely precious resource because they are the fastest storage units available on the chip and because typically there are not that many of them. For instance, on a chip that implements the 32 bit MIPS ISA, there are typically 32 registers. Both the number and utilization of registers differs from ISA to ISA. However, it is safe to assume that there are several types of registers that will be encountered on most ISAs.

- Current Instruction Register (`CIR`) – a register that holds the instruction that is currently executed. Recall that the instruction, as discussed in Example 1.1 looks like 10101110010010000000000000010000 and as such is perfectly suited to be stored in a register.
- Program Counter (`PC`) – a register that holds the address of the next instruction that will be executed. Unlike `IR`, `PC` contains an *address* of an instruction, not the actual instruction
- Memory Data Register (`MDR`) – register that holds data that has been read in from main memory or produced by the CPU and waiting to be stored in main memory
- Memory Address Register (`MAR`) – register that holds the address of the memory location in main memory (`RAM`) where input/output data is supposed to be read in/written out. Unlike `MDR`, `MAR` contains an *address* of a location in memory, not actual data
- Accumulator (`AC`) – a register that typically resides inside the ALU. Used to accumulate a result as the outcome of simple operations

There are also a number of registers tied to the task of handling function (subroutine) calls and their meaning will be further clarified when discussing the memory model and the scope of a variable declaration:

- Global Pointer (`gp`) – a register that holds an address that points to the middle of a block of memory in the static data segment. This block of memory holds data that is globally available to all functions; this might include variables that have file or global scope.
- Stack Pointer (`sp`) – a register that holds an address that points to the last location on the stack. More on this later.
- Frame Pointer (`fp`) - a register that holds an address that points to the beginning of the procedure frame. More on this later.
- Return Address (`ra`) – the address pointing to an instruction where upon finishing a sequence of instructions, the execution should return (jump) and commence the execution

Finally, there are some generic registers. In parenthesis is provided the number of such registers available on the typical 32 bit MIPS ISA. Since they come in larger numbers they don't have an acronym:

- Registers for Subroutine Arguments (4) – used to pass arguments upon the invocation of a function (subroutine)
- Registers for temporary variables (10) – used to store intermediate results that are used as intermediate results in performing a more complex computation. How they are used in a C program, for instance, is to a large extent controlled by the compiler

- Registers for saved temporary variables (8) – registers that are guaranteed to save the value they hold between the call and return of a function (subroutine)

Increasing the number of registers is desirable but hard. They provide the smallest latency memory due to their proximity to the CU and ALU. However, increasing their number requires a reworking of the chip to what amounts a complete new design. Large numbers of registers are typically encountered in graphics processing unit cards; the Nvidia GTX 580 card, for instance, among its 16 multiprocessors, has more than 500,000 32 bit temporary variable registers [4]. In the end, registers are nothing but very basic hardware entities capable of storing information, one word at a time, that can be retrieve at the highest speed that information can be retrieve from anywhere on the computer. A register can store a floating point number that needs to be added to a different number; or it can store an address in main memory that is about to be accessed through a fetch instruction; or it can store an offset that indicates how many instruction are about be skipped at the request of a jump instruction. They are a critical component in defining the data path through the fetch-decode-execute cycle that is at the cornerstone of the execution and any program.

1.2.4. The Fetch-Decode-Execute sequence

The fetch-decode-execute (FDX) sequence is made up of a relatively small sequence of tasks that are repeatedly executed in order to carry out the request embedded in an instruction. The particular tasks, their number, and the relative ordering are specific to each instruction. In this context, an instruction can be regarded as a request that is being honored through the above mentioned sequence of tasks, which rely on logistics provided by the CU and draws on the help of the ALU.

An instruction should not be confused with a line of code in a program that you put together. A simple example of the latter would be `a += foo(2);`, which increments the current value of a by whatever value the function call `foo` returns upon a call with argument 2. This line of code ends up translated by the compiler in a series of instructions. Honoring the request encapsulated in each of these instructions is the result of the FDX sequence. Clearly, at some point there will be an add instruction. However, prior to that, several other instructions will set the stage for the add instruction. Of these instructions, probably the more remarkable one is a jump to the beginning of a sequence of instructions that implement the function `foo`. Upon the execution of this sequence of instructions that implements the function `foo`, the execution jumps back to the function caller and the return value will eventually be added to `a`. To summarize, a program is a sequence of instructions, each of which is implemented as a sequence of tasks. This sequence of tasks is called the FDX sequence; in each stage of this sequence: the fetch, decode, and execute, the spotlight is on the registers, which are the actors that participate in a play directed by the CU.

The actors in the fetch stage are the `PC`, `MAR`, `MDR`, and `CIR`. The address of the next instruction to be executed is first transferred from `PC` into `MAR`. Next, since we have its address (stored in `MAR`), the actual instruction is fetched and placed in `MDR`. The address stored in `PC` is incremented to point to the address of the location in memory that holds the next instruction to be executed. Then, the instruction stored in `MDR` is transferred into `CIR`. So far, the journey of the instruction has been: memory to `MDR` to `CIR`. `PC` provided key information for locating the instruction in memory, and `MAR` was a middleman.

The decode stage counts on the fact that each instruction, represented in Example 1.1 as a 32 bit construct, has encrypted in it several pieces of information. For starters, it has an operation code that eventually dictates what needs to be done. What needs to be done requires operands, e.g., two numbers to add, an address of the instruction that needs to be executed next, etc. The decode stage represents the collection of tasks carried out to decipher the information encrypted in the instruction. The process of decoding generates a chain reaction controlled by the CU. Based on the bit pattern in the operation code, the CU sends the write "operation" signal to the MUXs that control the behavior of complex logical blocks. This can be related, for instance, to Figure 7 or Figure 8, where the decoding of the operation code is related to the type of input signal sent to the MUX by the CU. It is very insightful to take a step back and contemplate how the six bits in the opcode (see Example 1.1) define a chain reaction that touches upon thousands of transistors organized in complex logic blocks to implement an operation such as AND or add. The operands for such an operation are also baked into the instruction. It is this aspect that brings into the play the lesser acknowledged registers, or the supporting cast, the ones that are identified by temporary register 1, temporary register 2, etc. They either store the value of the operand or, for indirect access, their address. The fortuitous circumstance that these registers are holding the right operands at the right time is owed to the foresight of the compiler, which sometimes might insert one or more load instructions to set up the stage for the current instruction; i.e., by the time, for instance, an add instruction is encountered, by careful planning the required operands happen to already be one in the `AC` and the second one in a temporary register.

The collection of tasks carried out during the execution stage depends on the type of request encrypted in the instruction. The family of requests is not large; for instance, for the 32 bit MIPS ISA, since the `opcode` has six bits, there is a maximum of 64 different operations that can be represented. There are several other fields such as `funct`, for instance, which can be used to further qualify the opcode. In the end, anywhere from 50 to 250 operations are expected to be supported. This number is what differentiates between RISC and CISC architectures. The former supports a small number of instructions; relatively straightforward to implement, they are used to represent more complex operations as a sequence of basic instructions. CISC supports a very large family of instructions. Translation is challenging and so is the support in hardware for carrying them out. Typically, one of these CISC instructions corresponds to several RISC instructions.

In what follows we will discuss the FDX cycle of only two instructions to gain an intuition into what the execution stage amounts to. The first instruction implements the often encountered request for a jump operation, which is ubiquitous owing to its use whenever one calls a library function or the execution encounters an `if` statement. For instance, upon a library call, the program jumps to an address in memory that marks the beginning of the sequence of instructions that implement the functionality advertised by this library's function, a topic revisited in section 1.9. Conceptually, a jump request is straightforward to implement. First, the opcode for a jump is 000000. Next, once it's clear that what is dealt with is an R-type instruction, the function field value; i.e., the pattern of the last six bits, dictates the type of jump to be serviced. For instance, in the 32 bit MIPS ISA, a 001000 would indicate that the instruction is a "return from subroutine" instruction, and that the program should jump to the address stored in the register `ra`.

Example 1.4

For the 32 bit MIPS ISA, assume that the instruction in the `CIR` register reads (the x bits are irrelevant):

000000xxxxxxxxxxxxxxxxxxxx00100.

The assembly code associated with this machine code is

jr $ra

It requests a jump at the address stored in `ra`. As a result, in the execution stage of the FDX cycle the content of the `ra` register is copied into the `PC` register and the instruction is concluded. Consequently, when the next FDX cycle picks up the instruction address from `PC` and moves it into `CIR`, it will point exactly to the instruction that needs to be executed upon the return from the subroutine.

It is worthwhile to revisit briefly the encoding that goes on in an instruction. It turns out that there are three classes of instructions: R-type, I-type, and J-type. R-type instructions are sometimes called register type. Their defining attribute is the intense use of registers. One way to regard them is as instructions that are self-reliant; they reference registers, which store the information required to fulfill the associated request. The I-type instructions are called immediate execution instructions and carry less baggage. Finally, there are J-type instructions that encrypt in their body the information required to perform a jump operation associated, for instance, with calling a function, returning from a function call, or constructs such as `if-then-else`, `switch`, `while`, `for` loops.

To make things more interesting, there are both R-type and I-type `add` instructions. The former encrypts information about the registers holding the numbers that need to be added. The latter has already one of the operands in the `AC` register or increments an operand by a constant and as such requires less information encrypted in its body. The common denominator here is that the decoding stage of the FDX cycle undoes the slicing process carried out by the compiler. It dissects the instruction and provides the CU several bit packages based on which the CU choreographs the execution process essentially providing the right MUX controls and ensuring that data moves along the datapath.

Example 1.5

This example discusses the FDX cycle associated with a simple instruction. In the fetch stage, the content of `PC` is flushed into `MAR`. The string of bits in `MAR` represents an address of a memory location. The content of that memory location is next flushed into `MDR`. From `MDR` it is further flushed into `CIR`, marking the end of the fetch stage: an instruction is stored in `CIR`.
Assume that what's stored in `CIR` is the following sequence of bits:
`00000010100010000100000000100000`
Incidentally, this is the machine code instruction representation of the following assembly code instruction (see Example 1.1):
`add $t0, $s4, $t0`
This string of bits is what the decoder is presented with, marking the beginning of the decode stage. Upon dissecting this string of bits, since the first six bits are 000000, the decoder figures out that it's dealing with an R-type instruction. A quick look at the last six digits will indicate that the type of register operation to be carried out is addition. Based on the pattern of 1 or 0 associated with these two groups

of six bits, the CU is sending a set of signals to the MUXs associated with the ALU to prime it for an addition. Dealing with an R-type instruction means that the operands are already stored in registers. Just like 007 is used to identify James Bond, 01000 identifies the temporary register $t0. Likewise, 10100 identifies $s4.
At this point the execution stage kicks off. The content of the register $t0 is flushed into `AC`. The ALU, using the logic block in Figure 8 and being primed as far as the MUXs are concerned with the right command input, carries the add overwriting the `AC` memory. The new value in the `AC` is flushed out and the output is directed towards register $t0 based on information provided by the decoder. At this point the address stored in `PC` is incremented by 1 to point to the next instruction and the FDX cycle concludes.

1.2.5. The Bus

The FDX cycle relies on the existence of a means to move information between the CPU and main memory. For instance, using information stored in `PC`, an instruction is fetched from main memory. First of all, an address needs to transit form the CPU to the main memory. The main memory should be nice enough to provide what is stored at the said address and send this back to the CPU. The situation is similar no matter whether what is requested is data or instructions. The pattern is that an address is provided and information that needs to be stored moves one way or the other, depending whether we are dealing with a read or a write operation. As such, there are three channels for moving information: one for moving addresses (the address bus), one for moving data and/or instructions (the data bus), and one for control purposes (the control bus). The three buses, address, data, and control, combine to form the system bus.

In general, a bus can be regarded as a communication pathway that transfers information from a source to a destination inside the computer. The source/destination can be the CPU, main memory, hard-disk, GPU, or other peripheral devices such as the keyboard, printer, joystick, etc. Physically, the bus is one or more sets of connectors (wires). While the purpose of the address and data bus is clear, the role of control bus is less intuitive: it provides the medium for the devices present in the system to coordinate and work as a team. For instance, assume that at each time step of a simulation your program saves to a file data associated with the state of a dynamic system. Without getting into details, the CPU communicates with the secondary storage unit (hard-disk) every once in a while to output the requested information. When/how the CPU and secondary storage unit interact is coordinated through a system of interrupts and requests that move through the control bus.

The most important attributes of the data and address bus are the size of the bus and the frequency at which they operate. The size dictates how much information one can expect to receive at destination upon a clock tick once a steady state communication stream has been established between source and destination. The clock tick is inversely related to clock frequency. If the width of the bus is 32, then upon reaching steady state the bus can deliver 32 bits of information upon each tick of the clock. The sizes of the address and data buses do not have to be identical. In modern architectures, the address bus determines the number of memory locations accessible. If there are 32 bits, there are 2^{32} = 4294967296 memory addresses. If each address identifies a memory location that has the size of one byte, this suggests that one can reference 4GB of data using 32 bits. Nowadays an address bus might be 48 lanes wide. Going beyond that would push extra bits around which are not used reference actual memory locations since the memories in current use are not that large. Also, if very large addresses are required, for instance a 52 bit wide address might simply be sent over two clock cycles. There is an analogy with a highway that has 10 lanes one way. If one wants to send 12 cars down those lanes, 10 cars go first and the two more follow shortly thereafter.

Finally, the most relevant bus as far as the parallel computing programmer is concerned is the data bus. Its width varies significantly from eight to 256 lanes and beyond, and it is what differentiates between entry level and high performance architectures. All other factors being identical, wider data buses are capable of pushing more data into the CPU per clock tick. In turn, this can sometime prevent the CPU from becoming data starved and idling waiting for data to become available. This idling is the most likely reason for CPU underutilization in high performance computing. High end graphics cards today have data buses that are hundreds of lanes wide. For instance, an Nvidia GTX 480 card has memory that sends/receives data through a 384 lane connection.

1.2.6. Execution Pipelining

Around 1908, Henry Ford started perfecting one simple idea: using an assembly line to manufacture cars. This allowed him to speed up production and reduce costs eventually establishing his company as forefront of innovation in manufacturing. Basically, at each tick of the assembly line out came a Ford Model T. This idea has a close analog in instruction processing: at each cycle, the CPU typically completes the processing of an instruction since any modern architecture nowadays has a pipelined mode of operation.

The motivation behind this approach is very intuitive: take a very complex task; i.e. producing a car or processing an instruction, and generate a sequence of simple operations that take an input from upstream and provide an output downstream. At the end of this sequence of simple operations one gets a new car or completes the processing of an instruction. It becomes apparent that having the complex task mapped into a long sequence of operations is advantageous. On an assembly line at which one worker or possibly robot performs one simple operation, it follows that having a long assembly line will allow multiple works to work at the same time on a complex task. This translates into parallel execution. Moreover, since each of these basic operations is basic, it means that it is performed quickly. Globally, this translates into moving the assembly line very often; in other words, the tick is short. And as such, with each tick a new product is finished. At the factory in Piquette, the first full month of traditional assembly yielded 11 Ford T cars. In 1910, one T car came out every three minutes.

This analogy highlighting the importance of a deep assembly line carries over to instruction processing. This is the reason why the FDX cycle is not broken into three stages that correspond immediately to the fetch, decode, execute operations associated with instruction cycle. Instead, the assembly line, which is replaced by what is called a CPU pipeline, is broken into more stages. For instance, a RISC architecture pipeline has at least five stages [2]: fetch an instruction, decode the instruction while reading registers, execute the operation (might be request to calculate a new address), memory access for operand, and write-back into register.

There is one aspect that makes pipeline design challenging: instruction heterogeneity. Designing and improving a T model assembly line was relatively straightforward as the line served one product. Moreover, over the years, Ford became notorious[1] for resisting any substantial change to the vehicle, which in turn allowed the industrial engineers plenty of time to optimize the assembly line. The CPU faces a relatively broad set of instructions that it must be capable to execute. The basic five-stage pipeline mentioned above is not a perfect match for the entire family of instruction that a CPU might process. However, it can accommodate all of them in the sense that some stages of the pipeline will idle for some instructions. To understand why, think of an assembly line capable to build on demand one vehicle that might be one of three types of trucks, or four types of SUVs, or two types of lawn-mower tractors. It is likely that there will be stations on the assembly line where the lawn-mower will pass through these stations untouched.

Example 1.6. A very simplified illustration of how pipelining helps.

Table 6 lists several MIPS instructions along with the amount of time, reported in square brackets in picoseconds in the table header, required by each pipeline stage to tender its service. Some instructions do not require the attention of each pipeline stage [2]. This is marked by N; a Y indicates the instruction is processed during that stage.

Instruction class	Fetch [200 ps]	Register Read [100 ps]	ALU Operation [200 ps]	Data Access [200 ps]	Register write [100 ps]	**Total time**
Load word (`lw`)	Y	Y	Y	Y	Y	800 ps
Store word (`sw`)	Y	Y	Y	Y	N	700 ps
R-Format (`add`, `sub`)	Y	Y	Y	N	Y	600 ps
Branch (`beq`)	Y	Y	Y	N	N	500 ps

Table 6. Several representative instructions, along with the total amount of time they would require for completion in an ideal setup. Since in a pipeline each stage must take the same amount of time, the passing of any instruction through the pipeline under normal circumstances requires the same amount of time, in this case 800.

Consider now a set of three store instructions:

```
sw   $t0,  0($s2)
sw   $t1, 32($s2)
sw   $t2, 64($s2)
```

In the absence of pipelining, based on information provided in the "Store word" row of the table, the amount of time required to process these three instructions is 3×700=2100 ps. One instruction is not picked up for processing before the previous one is finished, which leads to the timeline shown in Figure 10. Figure 11 shows the pipelined scenario, which requires 1400 ps to process the three instructions.

For the five-stage pipeline discussed here and outlined in Table 6 the speedup upper limit is five; i.e., equal to the number of stages.

[1] Ford went so far as stating that "Any customer can have a car painted any color that he wants so long as it is black."

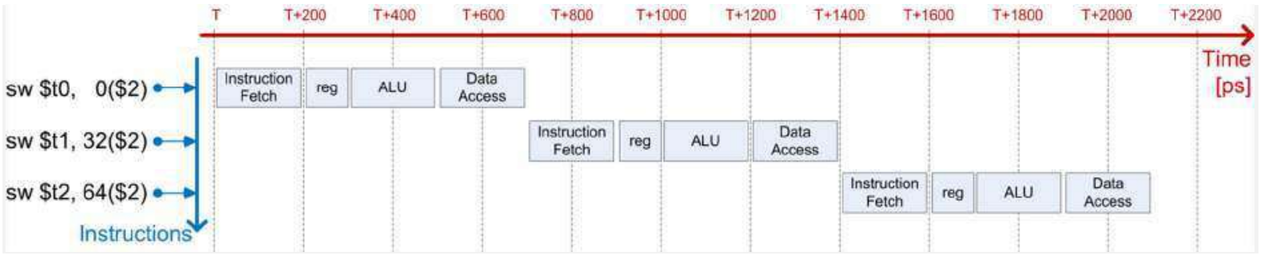


Figure 10. Timeline for processing of three sw instructions without pipelining. The value T is the reference time at which the first store is picked up for processing.

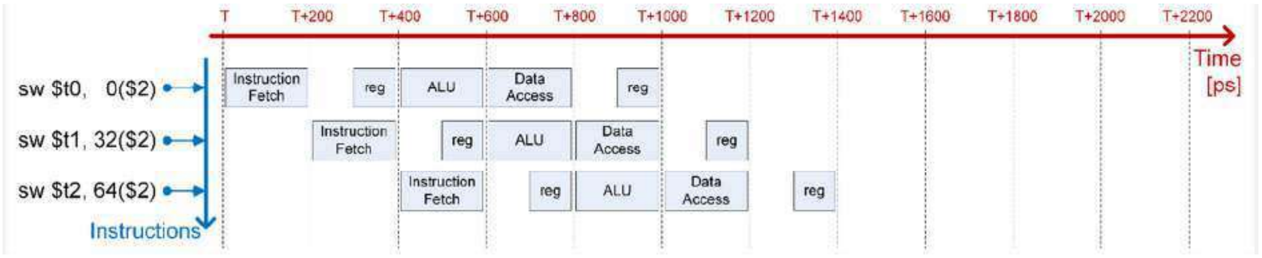


Figure 11. Timeline for processing of three sw instructions in pipelined fashion. The value T is the reference time at which the first store is picked up for processing.

Using a pipelined processor does not reduce the time it takes to process one instruction. Quite the contrary: there are some instructions that end up moving down the pipeline in more cycles than necessary if that instructions would have been processed in isolation. This is because the pipeline is general enough to handle a wide spectrum of instructions and some of them do not require all of its stages. Where is the speedup coming from then? It goes back to simultaneous processing of five instructions, as opposed to one in a nonpipelined architecture. This makes the point that throughput rather than instruction completion time is what gets boosted in pipelining. It is important to emphasize that one car gets assembled at each cycle of the clock; i.e., each time the assembly line moves a tick. Pipelining allows the ticks to come faster, since the processing was split in five stages and the work, supposedly, was equally split in five.

Designing a pipeline is complex; it is part science, part intuition, and part experience. It is an optimization problem with many constraints. Adding more stages to a pipeline would seem like the obvious thing to do; a deeper pipeline would decrease the amount of effort in each stage and as such the ticks could be ramped up improving throughput. However, deep pipelines have their problems: just like in the analogy with the lawn mower, the SUV, and the truck, if the pipeline is deep and does a myriad of things, processing many lawn mowers will actually slow down execution as the assembly line would move half-assembled lawn mowers around with few stations actually making meaningful build-up contributions and even those necessarily modest. This can be resolved by understanding well the type of problems run on the processor and by designing the pipeline to align well with the scenarios that the processor sees most often.

A deep pipeline adds several processing hazards. A "control hazard" is associated with the handling of a branch; for instance, in an `if` statement. A branch instruction, like any other instruction, takes its time moving through the pipeline. Since we do not know the outcome of the branching instruction, we don't know what the next instruction should be. The question is this: should we stop feeding the pipeline and let the branch instruction move its way through the pipeline to completion so that we know what needs to be done next? Stalling the pipeline is detrimental and the situation is addressed through what is called branch prediction: a hopefully good guess is made in relation to the outcome of the branch/no-branch decision, and the `PC` register is set right away to an instruction that will immediately enter the pipeline. If the prediction was wrong, the instructions associated with the wrong branch are evicted and the right instructions are processed. The simplest form of branch prediction is static; based on heuristics, the compiler makes choices at compile time. For instance, for a do-while loop, it is likely that the loop if executed once will be executed again; so it makes sense to predict accordingly. Similar reasoning can be done for `if`-statements, `for`-loops, etc. Dynamic branch prediction requires the storing of additional information to support a more intelligent branch/no-branch decision. The same instruction encountered twice in the life of a program can lead to different branch decision based on recent history. Dynamic branch prediction is reported to reach 90% accuracy in some cases [2]. High rates are desirable since the branch misprediction is costly: the entire pipeline will get flushed thus wasting a large number of instructions that have been executed or are in flight.

Data hazards represent the second class of hazards associated with pipelining. They occur when a result generated in one instruction is reused too soon by a subsequent instruction causing a stall of the pipeline. The following sequence of two consecutive instructions is used to make this point:

```
add  $t0, $t2, $t4      #  $t0 = $t2 + $t4
addi $t3, $t0, 16       #  $t3 = $t0 + 16 ("add immediate")
```

The two MIPS assembly language instructions are "translated" at the right. The register `$t0` is written by the first instruction and its content immediately used in the second instruction to produce the value stored in the register `$t3`. Yet the value of `$t0` becomes available at the end of the fifth pipeline stage and waiting for it to be available would stall the pipeline due to the second instruction depending on the first one. There are two often used ways in which this pipeline stall is avoided. The first relies on the compiler to identify this data access pattern and attempt to change the instruction order by implementing a couple of other instructions that must be executed in between the two troublesome instructions. The second approach requires hardware support. Essentially there is a conduit in the pipeline in which the ALU execute stage of the pipeline feeds data upstream back in the pipeline. In our example, as soon as the results that will eventually find its way in register `$t0` becomes available at the end of the ALU execute stage of the pipeline, it will be rushed into an upstream stage of the pipeline to prevent or minimize stall. This "cheating" process, where a backdoor conduit is used to take a shortcut is called bypassing or forwarding.

A third class of pipelining hazards, called structural hazards, is relatively straightforward to introduce using our vehicle assembly analogy. Structural hazards are situations in which there is contention for the resources required to properly operate the pipeline. If there was only one type of vehicle manufactured on an assembly line, structural hazard would be completely eliminated in the design phase of the pipeline. However, when the assembly line handles a variety of vehicle types: trucks, SUVs, and lawn-mower tractors, and a variety of models for each vehicle type, situations might arise where the assembly of the wheel of a truck requires a precision measurement instrument that happens by an unfortunate circumstance to be needed upstream in the pipeline by a lawn mower for an engine alignment measurement. Several courses of action can be taken: over-design the pipeline and buy all the tools/machines that will cover any possible pipeline scenario; this is expensive. Stall the entire pipeline for the truck assembly stage to finish using the precision measurement instrument. This might slow down the pipeline significantly if it was designed on a budget and these stalls happen too often. Finally, try to have the plant manager plan ahead to avoid situations like this by reordering the work order to have a couple of SUVs in between the lawn-mower batch and the last truck, for instance. When it comes to instruction processing in a CPU, the approach used is a combination of all of the above: provision somewhat generously the hardware to eliminate as many hazards as possible; be open to the idea of stalling the pipeline once in a while so that you keep the price of the hardware down by not over-provisioning; rely on the compiler to rearrange the order of the instructions to further mitigate some potential structural hazards.

1.3. The Clock and Execution Performance Metrics. Superscalar Processors

The main modules in a computer have access to a system clock whose ticks can be monitored on a line in the control bus. The clock is not used to indicate absolute time; it is a very precise circuit that provides a continuous stream of equal length high and low pulses. The length of time between to ticks is called clock cycle or clock or cycle and it is typically a constant value dictated by the frequency at which the processor operates. For instance, a modern 2 GHz processor has a clock cycle of 500 picoseconds.

The ticks play an important role in synchronization; they provide a repeating pattern that is used to schedule all the operations in a computer: interrupts, loads, stores, pushing of data from a register into a different one, etc. If a module needs to carry out tasks on a different clock cycle, the module cycle will use a second circuit to over- or under-sample the system clock. In terms of nomenclature, ticks are also called cycles and the length of a cycle is usually provided by specifying the frequency associated with the clock. For instance, old Pentium II computers used to operate the CPU at 266 MHz, the system bus at 66 MHz, the Peripheral Component Interconnect (PCI) bus at 33 MHz, and the Industry Standard Architecture (ISA) bus at 8.3 MHz. The 66 MHz system bus in this architecture was called the front-side bus and it was the reference bus for the entire system. All these frequencies have been increasing; what has not changed is the role these components are typically playing: the PCI continues to provide a high speed bus that, for instance, connects the GPU to the rest of the hardware, or, in cluster-type supercomputers, connects a computer to another computer through a fast interconnect using a Host Channel Adapter (HCA) card. Likewise, the role of ISA was superseded by similar protocols whose role was that of providing a bus that connected the slower components of a system (mouse, keyboard, hard-disks, etc.)

Going back to the CPU and the issue of performance, what a user perceives when running a program is the "program execution time". For a program, this is sometimes called wall clock time; it represents the amount of time from the beginning of the program until its end. As such, it includes the time associated with all the housekeeping chores that the CPU has to handle: operating system issues, peripherals asking for support through interrupts and requests, operating system tasks, running other programs such as a web browser, an email client and a video streaming application. The "CPU execution time" provides a finer comb, in that it accounts for the time effectively dedicated to the execution of a program. This quantity is not straightforward to discern – it requires a profiling tool. This will be discussed in later sections both for GPU and CPU parallel computing. On a quiet machine that is dedicated to the execution of one program, such as real-time systems, the difference between program execution time and CPU execution time is very small.

The CPU execution time can be further qualified in user time and system time. The former is time spent processing instructions that are compiled out of user code or instructions in third party libraries that are directly invoked by user code. The latter is time spent in other libraries that most often the user doesn't even know they exist: support for writing/reading files, support for event handling, exceptions, etc. It is not always simple to draw a clear line between user and system times; sometimes the same function call, for instance a call for dynamic memory allocation, can be added into the user time category if the call is explicitly made by the user, or it can be regarded as system time if made deep in a library that was indirectly invoked by a user instruction. A formula can be used in the end to estimate the CPU execution time and it relies on three ingredients: the number of instructions that the program executes, the average number of clock cycles per instructions, and the clock cycle time. The number of instructions executed includes user and system instructions; they combine to provide the Instruction Count (IC). The number of clock cycles per instructions (CPI) is an average value since several tens of distinct instructions are recombined in a myriad of ways to produce programs that have millions of instructions[2]. An average is computed, and then used to predict the CPU Time [2]

$$\text{CPU Time} = \text{Instruction Count} \times \text{CPI} \times \text{Clock Cycle Time}.$$

Since the clock cycle time is the inverse of the clock rate, the above formula can be equivalently expressed as [2]

$$\text{CPU Time} = \frac{\text{Instruction Count} \times \text{CPI}}{\text{Clock Rate}}$$

Reducing the CPU Time calls for reducing the product of the three factors that enter its equation. For two good decades starting in mid-1980s the path of least resistance was cranking up the Clock Rate, or equivalently decreasing the Clock Cycle Time. In mid 2000s', the path of least resistance ran into the wall. This is a consequence of the observation that dynamic power dissipation in a chip is linearly proportional to the frequency. Attempts at reducing CPU Time are now focused on decreasing the product IC$\times$CPI and represents an area of active research. Effectively, the IC goes back to the ISA: how do we choose the represent in instructions the lines of code that we write in a C program? Are these instructions very simple, represented using a constant number of bits, etc.? We touched upon these issues in the context of the RISC vs. CISC discussion. Increasingly, the consensus is that the IC, and as such an ISA choice, is playing a secondary role and that in fact, it is the microarchitecture of the chip that

[2] This is analog to the DNA: there are four "instructions", A, G, C, T. They combine in a number of ways to result in various sequences of instructions that become "programs" that produce very different outcomes.

ultimately dictates the performance since it directly influences the CPI. The chip microarchitecture in this context is the layer that takes an abstract ISA and provides the support for its implementation in silicon. It is the outcome of a process of grouping/organizing transistors on a chip to perform certain tasks and cooperate for the end goal of executing instructions fast. In some respects, an ISA is like a standard; it depends on how you implement the standard in the microarchitecture that makes all the difference. Billions of dollars go into designing the microarchitecture of a chip since it poses tremendous analytical and technological challenges. For the former, one can cite the questions of pipelining: how many stages, how bypassing should be implemented, how one should support fast floating point operations, etc. For the latter, one should be able to cram all these transistors and connect them to implement whatever solution was deemed optimal in the design stage.

A need emerged in time for guidance in the process of designing or revamping a microarchitecture. To make good use of the transistors available on a chip, one should have an idea about what people use the chip for. SPEC, the Standard Performance Evaluation Corporation, is a non-profit corporation formed to "establish, maintain and endorse a standardized set of relevant benchmarks" that can used to "give valuable insight into expected real performance" [5]. There is a benchmark called CINT2006 that compiled a collection of programs that rely heavily on integer arithmetic; CFL2006 is a similar suite of programs that invoke a large number of floating point operations. The programs in CINT2006 and CFL2006 are real-world programs that are used extensively. Considered representative for a large class of programs out there, they define a proving ground for new ideas in ISA and microarchitecture design and an unbiased judge that assesses the performance of new computer systems.

Example 1.7

The purpose of this example is to demonstrate the type of information that makes possible a performance assessment of a new chip, in this case AMD's Opteron X4 – 2346, code name Barcelona. For each program in the CINT2006 benchmark, the table includes information on the three metrics used to assess the performance of a chip: IC, CPI, and clock cycle time [2].

CINT2006 Programs		**AMD Opteron X4 – 2356 (Barcelona)**			
Description	**Name**	**Instruction count [$\times 10^9$]**	**CPI**	**Clock Cycle Time [seconds $\times 10^{-9}$]**	**Execution Time [seconds]**
Interpreted string processing	perl	2118	0.75	0.4	637
Block-sorting compression	bzip2	2389	0.85	0.4	817
GNU C compiler	gcc	1050	1.72	0.4	724
Combinational optimization	mcf	336	10.00	0.4	1,345
Go game (AI)	go	1658	1.09	0.4	721
Search gene sequence	hmmer	2783	0.80	0.4	890

Chess game (AI)	sjeng	2176	0.96	0.4	837
Quantum computer simulation	libquantum	1623	1.61	0.4	1,047
Video compression	h264avc	3102	0.80	0.4	993
Discrete event simulation library	omnitpp	587	2.94	0.4	690
Games/path finding	aster	1082	1.79	0.4	773
XML parsing	xatancbmk	1058	2.70	0.4	1,143

Table 7. Results for the SPEC CPU Benchmark CINT2006, which is a collection of programs heavy on integer arithmetic. The table reports metrics obtained for the quad core AMD Opteron X4 – 2356 chip: Instruction Count (IC), average Cycles Per Instruction, and clock cycle time. The names of the programs that make up the benchmark along with their description are provided in the first two columns.

The CPI values reported in Table 7 are somewhat surprising: Since this processor is pipelined, why wouldn't this number be 1? Then if it is not one, how can it be smaller than 1? The answer to the first question is straightforward: the value reported represents an average value – it captures the fact that some instructions, for instance a jump to a subroutine, are more complex than others. Moreover, some instructions wait on memory. A cache miss, a concept explained in section 1.7, results in a system memory transaction that requires a large number of cycles.

The answer to the second question is more involved. More and more often, a processor has the ability to execute two or more instructions in the same clock cycle. This is called multiple issue in reference to the issuing of two or more instructions during a clock cycle. This has the potential to build on top of the pipelining concept to lead to architectures that have CPIs less than 1: for a double issue core, one has CPI=0.5. Consider the C code snippet below:

```c
int a, b, c, d;
//some code here that sets values of a and b
c=a+b;
d=a-b;
```

There is no dependency between these two lines of code; they can be translated into a collection of instructions that can be processed in parallel. For multiple issue architectures, the compilers are on the lookout for situations like this that can be leveraged at compile time. A similar situation arises in parallel computing in the Single Instruction Multiple Thread (SIMT) execution model adopted by CUDA for GPU computing. SIMT discussion will be deferred for later, but the take-away point is that it is possibly to know ahead of time of opportunities for parallel execution and capitalize on them; this is called static multiple-issue. The dynamic version tries at run time, by simply looking at the instruction stream which instructions can be executed in parallel. The defining attributes of a dynamic multiple-issue architecture are (a) instructions are issued from one instruction stream; (b) more than one instruction is processed by the same core in the same clock cycle; and (c) the data dependencies check between instructions being processed takes place at run time. Dynamic multiple-issue architectures are sometimes called superscalar architectures and have been implemented by both Intel and AMD. Both pipelining and multiple-issue are representations of what in industry is called Instruction-Level Parallelism (ILP) and have been responsible in 1990s and first half of 2000s for significant speed improvements that required no effort on the part of the programmer.

Sidebar 2.
A research lab in China designed a chip with an RISC microarchitecture similar to that of the DEC Alpha 21164. The chip, ShenWei SW1600, ended up outfitting the Sunway BlueLight MPP Supercomputer. The supercomputer ranked 14 on the November 2011 TOP500 list available on a website ranking every six months the fastest supercomputers in the world [6]. The four-issue superscalar SW1600 chip, which had two integer and two floating point execution units, had a seven stage integer pipeline and ten stage floating-point pipeline. The L1 cache size was 8 KB for instructions and 8 KB for data, with an L2 cache of 96 KB. The chip had a 128-bit system bus and supported up to 1 TB of physical memory and 8 TB of virtual memory. The concepts of cache and virtual memory will be discussed in sections 1.7 and 1.8, respectively.

1.4. Bandwidth and latency issues in data movement

The data reported in Table 7 suggests that the same processor can sometimes have a CPI of less than 1.0, see for instance the video compressing application **`h264avc`**, while other times it has a CPI of 10, see for instance the program **`mcf`**. This difference can be traced back to where the processor accesses the data that it processes. When data is not in the proximity of the processor, numerous trips to system memory, sometimes called main memory or RAM, incur large latencies. This causes the CPU's pipeline to stall waiting on data that unfortunately is not stored in registers, not even in cache memory, but rather far from the processor; i.e., off-chip, in main memory. It is said that the slowdown is due to the memory access latency. Yet latency is not unique to memory access operations. The latency and bandwidth are attributes that are characteristic to any process in which requests for information transfer are made and then honored.

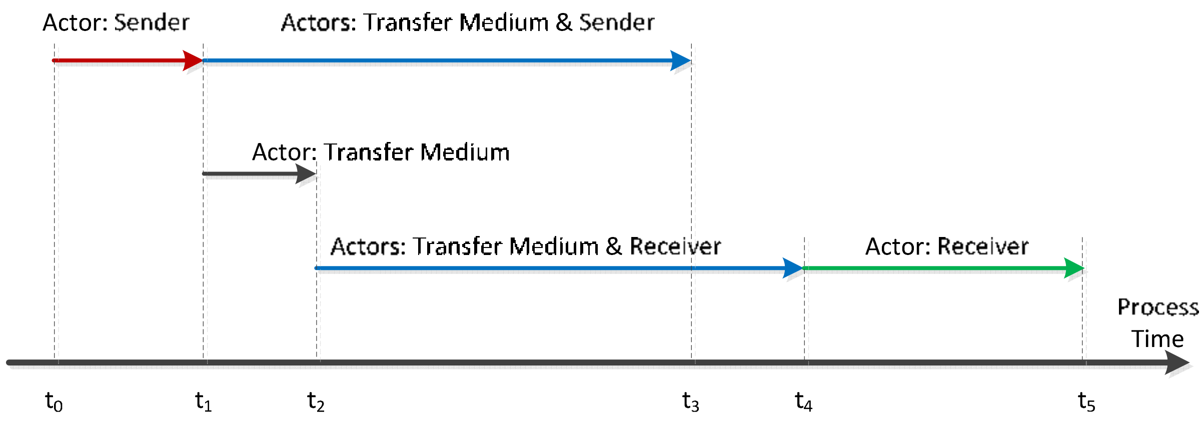


Figure 12. Timeline, Send-Receive process with a summary of the actors involved in carrying out the process.

Data Transmission Latency is the amount of time required by data to be transferred from its origin or sender to its destination or receiver. It accounts for the time associated with the sender's overhead, the flight time, the transmission time, and the receiver's overhead. The overhead on the sender side represents the amount of time required to initiate the transmission, which effectively means injecting the data into the network or pushing it into the bus; i.e., the transfer medium. The flight time represents the amount of time required by the first bit of information to move across the transfer medium. The transmission time is the time necessary for the entire chunk of data to move over the transfer medium and it is related to the size of the message and the transfer medium bandwidth. It can be regarded as the amount of time the sender keeps injecting data into the transfer medium. The overhead on the receiver side is associated with the process of extracting the data from the transfer medium.

1.5. Data Storage

Results in Table 7 revealed that for some programs the CPI count was far from 1.0; i.e., an desirable upper value that one would expect for a multiple issue pipelined processor. The programs **`omnitpp`** and **`mcf`** have CPI counts of three or higher. Briefly, it was stated that cache misses are responsible for the stalling of the pipeline while waiting for data to become available. This section elaborates on this issue by presenting a more in-depth view of the memories that come into play when running a program.

Figure 13 presents an overview of the types of memory common in Scientific Computing. The amount of memory and access time goes up from the top to the bottom. Registers were discussed in section 1.2.3, the focus next is on Cache memory and the Main Memory. We will not discuss here how data is stored on tape; this is most often the duty of a system administrator and is carried out for data backup purposes. It suffices to say that accessing data stored on tape is six to eight orders of magnitude slower than reading data from cache, for instance. However, the amount of data that can be stored on tape also six to eight orders of magnitude higher than what can be stored in cache. Also, the discussion on Hard Disk as a storage device will be brief and carried out when elaborating on the concept of virtual memory. It is in general three to four orders of magnitude slower than main memory accesses and typically stores three to four order of magnitude more data.

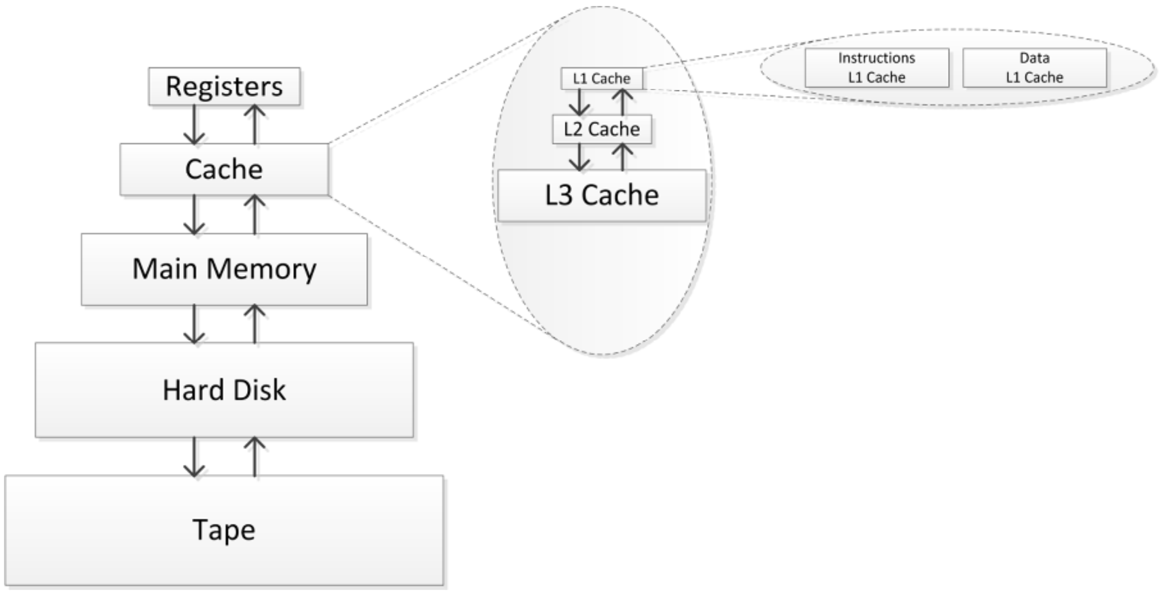


Figure 13. The memory hierarchy. From up to down: more memory, longer access times. In general, this is a true hierarchy: data stored at a higher level is also stored at a lower level. Of these storage spaces, only the registers and caches are on the CPU. Of any two blocks in this figure, the bigger one suggests a larger amount of memory associated with that level. The relative size of blocks though is not proportional to the relative size of the actual memories. For instance, a relatively recent Intel processor, Intel Core i3-2100, has a processor with two cores; each

core has 32 KB of Instruction L1 Cache, 32 KB of Data L1 Cache, and 256 KB L2 Cache. Additionally, the two cores share 3 MB of L3 Cache. The main memory typically goes up to 128 GB. Hard disks store in the neighborhood of 1 TB of information.

1.5.1. Static RAM (SRAM) and Dynamic RAM (DRAM)

SRAM is the memory of choice for caches, which provide a small amount of fast memory that sits inside a processor. Physically, it is an integrated circuit whose elements combine to make up memory arrays. This "element" is called a flip-flop cell and is typically made by combining six transistors that can together store one bit of information; i.e., a 0 state or 1 state [2]. On the plus side, the flip-flop has very short access time, of the order of 1 ns. When they combine, these elements have all the same access time independent of their position in the memory array. The "static" attribute comes from the fact that once a flip-flop is set, the element does not need to be revisited to refresh the value of the state as long as the element is powered. On the down side, flip-flops are bulky and expensive.

Figure 14 provides a schematic of a RAM chip. In this example it stores a collection four million (2^{22}) instances of eight bit data. A set of 22 pins are associated with the chip to provide the address of the eight bit storage entry where data is written to or read from. A set of eight pins carries data to the chip for storage; a set of eight pins are used to carry data away from the chip. A set of three control signals are used to initiate/prime a memory transaction (Chip select signal), indicate that the operation is a write (Write enable signal), or a read (Output enable signal) [2]. A memory transaction involves all these actors. For instance, a write operation requires a set-up time for the address line, a hold-time for the data line, and "Write enable" pulse width. The overhead associated with a memory transactions is the combination of these three times.

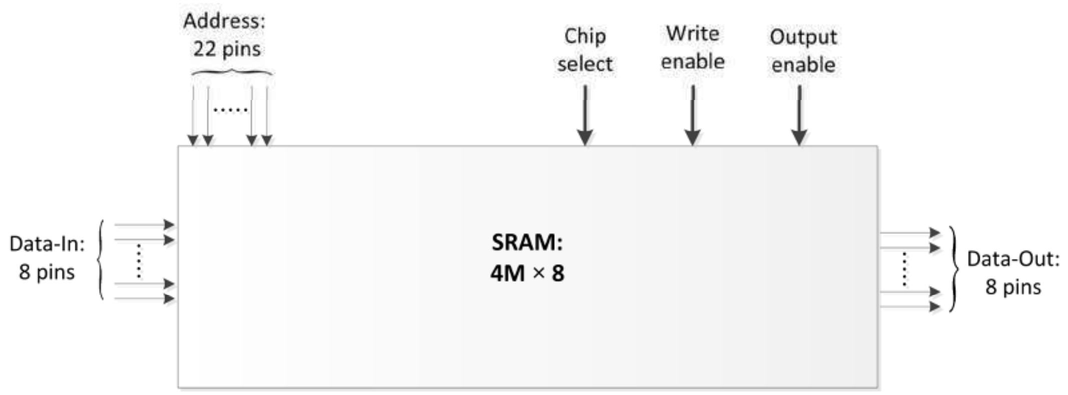


Figure 14. A schematic of a SRAM chip. In this example it stores 4 million instances of 8 bit data.

DRAM memory is most often used to make up the main memory. In many respects it is the opposite of SRAM memory: it is not bulky and it is relatively cheap; while SRAM is highly stable and has low charge leakage, DRAM demands periodic refreshing. The latter aspect goes back to the use of a capacitor to store information: presence of charge indicates a state, while lack of charge indicates the complementary state. A leak of charge equates to permanent loss of data. This is prevented by periodic refresh of the charge state, a process that happens at a frequency 1 KHz. During the refresh cycle, the corresponding memory cell cannot be accessed. This leads to a 99% availability since the memory is queried for read/write operations at MHz frequency as discussed shortly.

SRAM and DRAM are also different in terms of access speed. SRAM provided an address bus (22 pins in Figure 14) for immediate access. DRAM is organized in rows and columns of memory cells: each cell belongs to a wordline and to a bitline within that wordline. Access of a cell relies on a two level decoder that first selects the wordline and then the bitline. Locating a memory cell is typically slower since selecting the wordline and then bitline uses only one address line. The use of a Row Access Strobe (RAS) signal flags the use of the address line for wordline selection. The use of a Column Access Strobe (CAS) signal flags its use for a bitline selection. The accepted rule of thumb is that DRAM is approximately two orders of magnitude slower than SRAM leading to access times of 50-100 ns. While this latency is something that is difficult to ebb down, both SRAM and DRAM bandwidths have gone up owing to the use of a new access strategy that streams information stored in consecutive cells. The key idea was to slightly alter the hardware to provide one additional piece of information: how many consecutive cells are involved in a transaction. Rather than providing several individual transactions, one big transaction is carried out in one burst. This technology was called Synchronous SRAM or DRAM (SSRAM and SDRAM).

1.6. The CPU – Main Memory Communication

The Von Neumann computing model builds around the idea that the set of instructions that make up an executable are stored in main memory just like data that is processed by these instructions. In the FDX cycle discussion in section 1.2.4 we saw that the `PC` register held on to the memory location that stored the next instruction that will be executed by the processor. The information stored by this memory location should be brought over to the processor to initiate the FDX cycle. By the same token, data should also be brought over and possibly stored in local registers to be processed according to the work order encoded in the instruction. Bringing instructions over and shuttling data back and forth between the processor and main memory requires coordination. A memory controller is in charge of this coordination as it serves as the middle man between the processor and the main memory. Recently, this memory controller has been assimilated by the processor. In the past, it was a standalone entity hosted by a portion of the motherboard known as the northbridge. Specifically, most commodity workstations on the market between mid 1990s and early 2000s had a single core CPU that was connected to the memory controller hub on the northbridge of the motherboard's chipset through the front-side bus (FSB). The front-side bus operates at a frequency controlled by the clock generator and typically much lower than the internal frequency that the CPU operates at. The ratio between the two frequencies is typically in the range 5 to 15 and is called the CPU clock multiplier, or the bus-core ratio. For instance, a system with a FSB operating at 200 MHz and a 10X clock multiplier would operate on an internal clock of 2 GHz. A memory bus is also connected to the northbridge of the chipset and works on a frequency usually identical to that of the FSB.

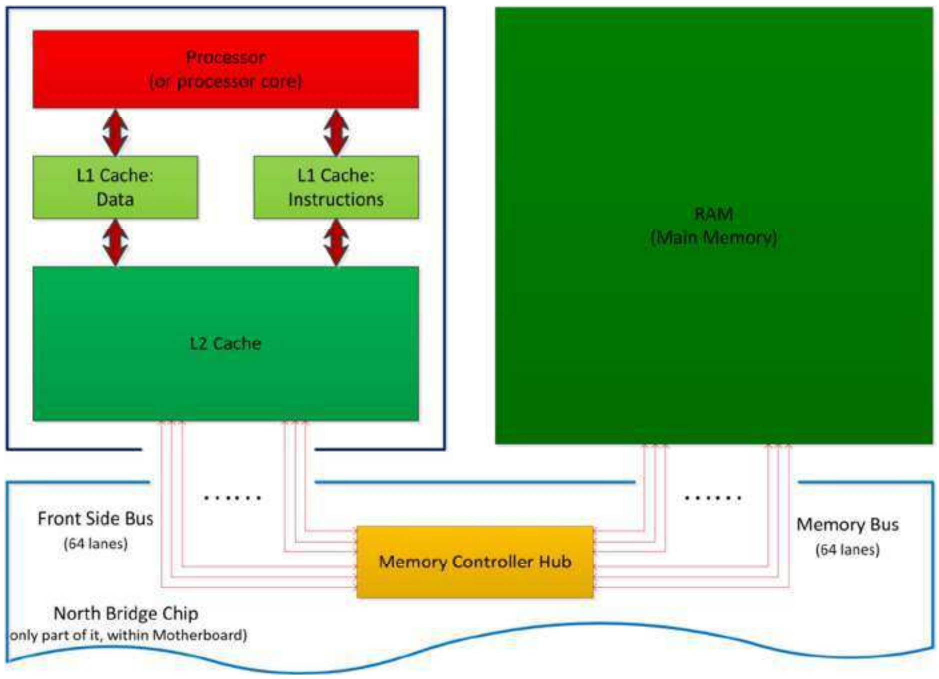


Figure 15. A schematic of a one core processor with L1 and L2 caches. The CPU is connected to the northbridge through the Front-Side Bus (FSB). The northbridge, which in the figure is showed to host the memory controller, was also connected to the main memory through the memory bus. FSB and the memory bus usually operate at the same frequency. Note that HyperTransport and QuickPath technology relocated the memory controller by placing it into the processor. While the northbridge still exists, its role decreased in relevance. In fact, recent chip designs have integrated the entire north bridge into the CPU chip.

The main memory is connected to the memory bus using a standard double data rate type synchronous dynamic random access memory (DDR2 SDRAM) interface. The commodity memory used in workstations and laptops today – DDR, DDR2, and DDR3 memory, is a type of SDRAM memory. Without getting into the technical details, the DDR technology allowed a doubling of the data transfer rate compared to its predecessor. This rate was doubled with DDR2, and yet again doubled by DDR3. DDR technology enabled a memory transfer rate (bandwidth) B_M [MB/s]

$$B_M = \frac{M_{CR} \times N \times 2 \times 64}{8} ,$$

where M_{CR} is the memory clock rate in MHz, N is the bus clock multiplier, the factor 2 reflects the dual rate attribute (the first D in DDR), 64 comes from the number of bits transferred. The result is scaled by 8 to produce a result in MB/s. The first generation DDR, widely adopted by 2000, had N=1; for DDR2, widely adopted in 2004, N=2; for DDR3, widely adopted in 2007, N=4. DDR4 support is slated for 2013. If, for instance, the frequency at which the memory arrays work is 200 MHz, B_M=3200 MB/s with DDR, B_M=6400 MB/s with DDR2, and B_M=12800 MB/s with DDR3.

The question is, to what extent can the FSB carry this much information that the memory is ready to pump into the bus? The answer depends on the technology used by the motherboard manufacturer to enable bit transfers, by the width of the bus (number of lanes), and the frequency at which the FSB operates. Typical FSB frequencies F_{FSB} are in the range 50 to 400 MHz. At the low end of the spectrum come old chips such as the Intel Pentium, Intel Pentium II, or AMD K5. At the high end of the spectrum come recent chips such as Intel Xeon. Almost without exceptions, the FSB width is 64 bits. Late 1990s saw one bit of data transferred per cycle (T_C): as the voltage went up and down, there was one bit of information that was transferred. As of early 2000s the technology advanced to the point where the number of transfers per cycle went up to $T_C = 4$. The FSB bandwidth, calculated with the relation

$$B_{FSB} = \frac{F_{FSB} \times T_C \times 64}{8} ,$$

becomes a perfect match for DDR2 memory, which was in used in mid 2000s. One interesting question is why the interest in DDR3, if one sticks with a FSB to memory bus ratio of 1:1. Then, the memory would be overdesigned unless a different mechanism would be considered to feed data from system memory to the CPU. This is exactly what started to happen in late 2000s, when two new technologies, HyperTransport promoted by AMD and Intel's QuickPath Interconnect, replaced the concept of FSB and enabled roughly a doubling of the bandwidths associated with the traditional FSB-based layout. Specifically, if a mid-2000 high end system worked with DDR2 and a FBS clocked at 200 MHz to reach a bandwidth of 6400 MB/s, current architectures that draw on QuickPath Interconnect technology reach 25560 MB/s, a bandwidth currently supported by DDR3 memory.

Note that critical as it is, bandwidth is only one of the aspects that control main memory latency. As indicated in Figure 12, there is overhead associated with issuing and answering the requests for data access. For memory accesses, the more important latency is the Column Address Strobe (CAS) latency, or CL. It represents the delay time between the moment a memory controller tells the memory module to access a particular memory column on a RAM memory module, and the moment the data from the array location is available on the module's output pins. Typical values are in the range of 10 to 20 ns.

1.7. Cache Memory

A single issue pipelined processor that operates at 2 GHz could in theory execute one instruction every 0.5 nanoseconds. SRAM has access times in that range; DRAM memory has access times about 50 times slower. In an ideal world, one would have all data available in SRAM so that the processor can work at capacity. Since SRAM is expensive and bulky only a small amount is available in the processor to store both data and instructions. This small amount of memory is called cache memory. As shown in Figure 15, it typically comes in two flavors: instruction cache and data cache. Their purpose is the same: provide memory that can feed the FDX cycle with information, both instruction and data, that keeps the processor working at nominal speed.

Although the amount of cache memory available to a processor or core is relatively small, locality of data and instructions helps us to populate the cache memories to increase our chances of the processor requesting information that is already cached. Locality is of two types: spatial and temporal. Spatial locality in program execution is an observed tendency of the majority of the programs to access information that occupies neighboring locations in memory. For instance, if one instruction is selected for execution, chances are that the next instruction that will be executed is stored in the very next memory location. This is certainly the case as long as there are no subroutine calls or execution jumps. Likewise, it is relatively often that the execution accesses sequentially elements in an array of floats, for instance. There are exceptions to the rule, such as integer programming or sparse linear algebra operations, but more often than not, there is a spatial locality present in the execution of a program. Finally, the temporal locality goes back to the observation that a variable used recently has a better chance of being used again in the immediate future than a dormant variable that has not been access in a while. For instance, think of a loop that is executed 100 times. For starters, the loop index is accessed heavily, as are some variables that get referenced inside that loop. There might be a variable that stores the gravitational acceleration that gets invoked, or some other variable whose value keeps being updated inside the loop.

Temporal locality justifies storing the value of a recently used variable in cache. Spatial locality justifies storing not only that specific variable, but also the data stored nearby in the lower level memory. "Lower level" memory can be the main memory, or for a system with cache memory at multiple levels, see Figure 15, it can be the L2 cache. This yields a hierarchical structure, which starts with the registers, continues with L1 cache, L2 cache, L3 cache, main memory, disk, tape. Not all of these memory spaces are present on each system – this is the most general scenario, see Figure 13. Going back to the FDX cycle, data requested by an instruction is going to be available fast provided it is stored in the upper part of this hierarchy. A L1 hit is a situation in which an instruction/data request is fulfilled based on information stored in L1 cache. If the required information is not found there, the request led to a L1 miss. The next best scenario is for a L2 hit. If this is not the case an L3 hit is what one can hope for, etc. Different programs have different hit/miss rates. For instance, the L1 hit rate represents the average number of memory transactions serviced by information stored in L1 cache. Having a program reach a high L1 hit rate depends on at least three factors: the nature of the problem being solved, the algorithm used to solve the problem, the implementation of the algorithm in software. Specifically, there are certain problems which are bound to have a lot of spatial locality, such as adding two long arrays. At the same time, there are problems, such as solving sparse linear systems, for which the memory access pattern is very irregular. When it comes to algorithms, the same problem often times can be solved by different algorithms. For instance, a sparse linear system can be solved with a direct factorization method or using an iterative algorithm. Chances are that the second approach is associated with improved memory locality, which in turn increases the hit rate. The very process of implementing an algorithm can have a big impact on the overall performance of the program.

1.7.1. Caching Strategies

Owing to the spatial locality attribute of memory transactions during the execution of a program, it became apparent that it made sense to cache not only one entry in the main memory but rather a block of memory around entry that has been accessed. The ensuing caching strategy is as follows: a block of memory that includes an entry just accessed is mapped into a cache line in the immediately higher memory level. The key words are "memory block" and "cache line".

The memory hierarchy implemented in a computer system ensures that data stored at a higher memory level is also stored at a lower memory level. For simplicity, imagine that you have a system with only one level of cache. Since the size of the L1 cache is much smaller than the size of the main memory, only a subset of the main memory can be stored in L1. Where does a block of the main memory get stored in a L1 cache line? At one end of the spectrum, one has the direct mapping strategy, which ensures that several main memory blocks, if end up in L1 cache, are always deposited in the same location no matter what. At the other end of the spectrum are approaches where a memory block can land in any position in the L1 cache. Example 1.8 intuitively explains the advantages and disadvantages of this approach, which is called the "fully associative cache" model. Like many things in life, the solution is in a compromise called the "set-associative cache" model. In this model, a main memory block can be mapped only in a handful of cache lines in the higher level memory.

The main advantage of the associative models is flexibility in old data eviction. Since in the "fully associative cache" one memory block can land anywhere, some criterion can be established for deciding which cache line will be evicted to make the necessary room to store the main memory block. Eviction criteria used in the past include the LRU (least recently used) rule, the LFU (least frequently used) rule, and the random rule. Carrying out tests to implement the LRU or LFU rules adversely impact the memory access time. That's why the random rule has been given consideration: with it, there is no overhead associated with the eviction part. However, note that the time penalty for reaching an eviction decision is not the only one inferred in relation to cache transactions. When the FDX cycle requires data in main memory, one should be able to quickly start looking into the cache to understand whether the value is stored in cache or a trip to main memory is in order. This process also adds to the information access overhead. In conclusions, there are two sources of overhead: one is associated with reaching an eviction decision using a rule such as LFU prior to depositing the new data into the cache. The second one is associated with finding if data exists in the cache or not. In the examples below former overhead will be referred to by T_e , from "eviction", and the latter by T_s , from "seek".

Example 1.8

Imagine a game in which you are shown an object and you have to generate from the 26 letters of the English alphabet the word that stands for that object. In this game there is a large letter warehouse far away that stores 10,000 As, 10,000 Bs, etc. Since the average length of English words is about 5.1 letters, very close to the place where the words are generated; i.e., in cache, there is room for only 6 letters stored in bin 1 through 6. When the object is shown, you would first try to use the letters in the bins to make up the word. If the letters needed are not present in the bin, you can run to the warehouse and bring them from there. Since you have to put these letters somewhere before you use them, some letters in the bins would have to be evicted. The goal is to assemble the word fast.

In this game, if direct mapping is enforced, then it could go like this: letters A, B, C, and D when brought from the warehouse should always be placed in bin 1. Letters E, F, G, H, and I are to be stored always in bin 2, etc. Direct mapping is not unique: one might say that letters E, A, R, I and O should always be places in bin 1; J, Q, Z should always be placed in bin 2, etc. This second direct mapping choice is particularly bad since E, A, R, I and O are the most common letters in the English words [7]. You would force them to compete for bin 1, at a time when the three least frequent letters: J, Q, and Z occupy a bin by themselves. The advantage of direct mapping is simplicity: whenever you need an A you know right away that bin 1 is the place to go; in other words, T_s would be small. The disadvantage becomes clear when having to assemble a word like “mirror”. In this case, M would be picked up from its bin (assume 3), then if I happens not to be in the bin 1, a trip to the warehouse should bring and place it in bin 1. Another trip is in order to bring R and place in bin 1. It can then be recycled once, evicted due to another trip to the warehouse to bring O, which is followed by another trip to bring back R and place in the first bin. Note that this last trip might be saved if a “look-ahead” strategy is employed and the very last R is used when it is available in bin 1 in relation to the use of the first R and second R. Imagine that you are aware that, according to the Oxford dictionary, the least frequent letter in its entries is Q: E is 57 times more likely to show up in dictionary entry, A 43 times more likely, R and I about 38 times, etc. Then it makes sense to place them in separate bins to decrease the likelihood of having them evict each other.

Assume now that the fully associative model with an LFU strategy is used. For the LFU to work, some information is needed in relation to how frequent the letters are in English words. This information is compiled by the Oxford dictionary, which states that the relative frequency or letters in words is as follows: E-56.88; A-43.31; R-38.64; I-38.45; O-36.51; T-35.43; N-33.92; S-29.23; L-27.98; C-23.13; U-18.51; D-17.25; P-16.14; M-15.36; H-15.31; G-12.59; B-10.56; F-9.24; Y-9.06; W-6.57; K-5.61; V-5.13; X-1.48; Z-1.39; J-1.00; Q-1.00 [7]. As already mentioned, E is about 57 times more likely to be encountered in a word than Q. In this example, assume also that the letters in the six bins currently are, from bin 1 to bin 6: Y, K, C, U, B, W. Spelling “MIRROR” with an LFU strategy would go like this: get M from warehouse and place in bin 2 by evicting K. Get I from the warehouse and place in bin 6 by evicting W. Get R from the warehouse and place in bin 1 by evicting Y. Recycle R from bin 1. Get O from the warehouse and place in bin 5 by evicting B. Recycle R from bin 1. The sequence of 26 letter-value pairs is called a dictionary. It can be static, like in this example, or dynamic. In the latter case, the dictionary would be continuously updated to reflect the nature of the words encountered while playing the game. This can be advantageous if, for instance, the game concentrates on words from biochemistry only, where the letter frequency is quite different than what’s compiled by the Oxford dictionary, which attempts to include all English words.

The LFU strategy with a full associative map is nontrivial: one should have a dictionary, which has to be stored in a handy location. Second, searching around to figure out who needs to be evicted is costly if you have a large number of bins; in other words, T_e can be large. The LRU strategy addresses the former aspect as there is no need of a dictionary. Instead, each bin has a time stamp that registers the moment when a word assembly process request started; in this context, the bin time stamp should be that of the most recent word that used its letter. In our example, assume that the bins, form 1 to 6, store: Y(3), K(2), C(2), U(3), B(3), W(1). In parenthesis is listed the most recent time of issue for word assembly in which a bin participated. Spelling “MIRROR” at time 4 goes like this: M evicts W, bin 6 gets a time stamp (4). I evicts K, bin 2 gets a time stamp (4). R evicts C, bin 3 gets stamp (4). R is recycled from bin 3. O evicts Y, bin 1 gets time stamp (4). R is recycled from bin R. The state of the cache would be, from bin 1 to 6: O(4), I(4), R(4), U(3), B(3), M(4). Additional rules might be needed to choose a bin when several are available with the same time stamp. This came up when choosing where to place I and O.

It was stated above that in set associative cache can store a main memory block in one of several cache lines. For instance, current Intel cores implement an 8-way associative L1 cache; in other words, a memory block can land in one of eight cache lines in L1. The mechanism for selecting which of the eight locations will be selected and how is not explained here. This mechanism is simpler for direct mapping, and it is intuitively explained through a simple discussion in Example 1.9.

Example 1.9

Assume that the system discussed here only has on-chip L1 cache and off-chip main memory (RAM), and caching relies on direct mapping. We assume that the cache can hold four bytes; $C = 4$, and the size of the memory is $M = 128$ bytes. In most direct mapping, the destination $0 \le dest \le 3$ in cache of a variable that is stored in main memory at location $0 \le loc \le 99$ is determined by a modulo (%) operation: $dest = loc \ \ \% \ C$. For our example, the location with address 17 in main memory would land in the cache location of index 1, since 17 % 4 = 1. In practice, finding the location in cache is simple: recall that the addresses are represented in binary form. Thus, $(98)_{10} = (1100010)_2$. The last two bits of the binary representation of loc ; i.e., 10, provide the $dest$. Likewise, since the last two digits of the binary representation of 67 are 11, and since $(11)_2 = (3)_{10}$, it means that the address 67 in main memory gets mapped into location 3 in the cache.

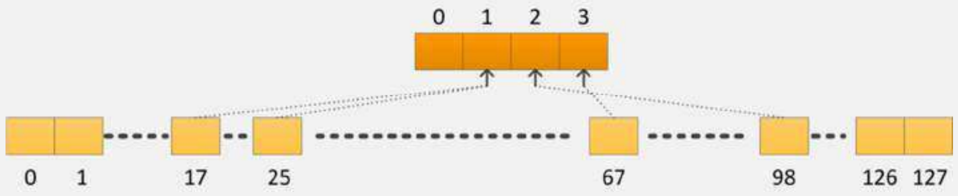


Figure 16. Main memory has 128 locations, while the cache has four locations. The value stored in main memory at location 98 is stored in cache at location 2 since 98 % 4 = 2.

It is thus easy to go from main memory to cache. The question is, how does one go back? In other words, if the Control Unit indicates that the memory location $(98)_{10} = (1100010)_2$ is needed, how can we know whether (a) what is cached at $dest = (10)_2$, which are the last two digits of address $(98)_{10} = (1100010)_2$, corresponds to $(98)_{10}$ or to $(74)_{10} = (1001010)_2$? Note that the last two digits in the binary representation of 74 are the same. And (b), if we figure out somehow that what's in there is what corresponds to location 98 in main memory, how do we know whether that value is supposed to be used or not? For instance, at the beginning of the program, all cache values are dirty, so you cannot use them. Also, in relation to cache coherence, a concept discussed in detail in [2, 8], the cached value might be stale since the main memory has been changed by a different process running in parallel with the execution of your program.

The concern (a) is addressed as follows: for each cache entry, there is a collection of C tags. Their purpose is to store all the bits of the loc that have been up to this point ignored. For instance, if the content of $loc = (98)_{10} = (1100010)_2$ is stored at $dest = (2)_{10} = (10)_2$, the first six bits of loc are stored in the tag field. Thus, in hardware, four collections of six bits are set aside to store the part of the loc that will be subsequently used in a splice operation to reconstitute the full address in the main memory. Going back to the request of the Control Unit, assume that the third tag field contains the following six bits: 110000 . As soon as the CU requested location (98) , since the third tag is 110000, it is clear that what's at location $loc = (10)_2$ in cache is exactly the quantity that is needed. Moving now on to (b), it is clear that another bit, called the "valid bit", is needed to indicate whether the cached value is to be used or not.

It is straightforward now to assess the memory requirements associated with this simplified direct mapping cache system. Assume that the memory space that you want to cache has 2^M locations. Assume that you settle for a cache that has 2^N memory locations. Then a number of $M-N$ bits are stored as tag for each location in the cache; additionally, you have one valid bit for a total of $M-N+1$ bits. In conclusion, in addition to the 2^N memory locations used for the actual caching, you have to store $2^N(M-N+1)$ bits.

The simplified direct mapping mechanism outlined in Example 1.9 glosses over the fact that in reality, what gets cached is not one memory location but rather a memory block. At the beginning of this subsection it was pointed out that the spatial locality affinity for memory access of real life programs when provided a strong motivation for caching not only one memory location, but rather a block that includes the desired value. Each of these memory blocks gets mapped to a cache line. The size of a cache line is equal to the size of the memory block but this value varies from system to system; currently, a good compromise is to have cache lines that can store 64 bytes. A system with a cache line too small might lead to multiple trips to a lower memory. A system with a cache line too large increases the chance of lines being evicted often since large cache lines translate into fewer memory blocks being mapped into cache.

Sidebar 3.
Under Linux, the command getconf provides information about the memory hierarchy of the processor used. The command
```
>> getconf –a | grep –i cache
```
yields on an Intel® Xeon® CPU E5520 the following output (where applicable, the numbers report figures in bytes):
```
LEVEL1_ICACHE_SIZE                  32768
LEVEL1_ICACHE_ASSOC                 4
LEVEL1_ICACHE_LINESIZE              32
LEVEL1_DCACHE_SIZE                  32768
LEVEL1_DCACHE_ASSOC                 8
LEVEL1_DCACHE_LINESIZE              64
LEVEL2_CACHE_SIZE                   262144
LEVEL2_CACHE_ASSOC                  8
LEVEL2_CACHE_LINESIZE               64
LEVEL3_CACHE_SIZE                   8388608
LEVEL3_CACHE_ASSOC                  16
LEVEL3_CACHE_LINESIZE               64
```
For this model the memory hierarchy has a three level deep cache structure. The L1 cache is split in two: 32KB of L1 data cache (DCACHE) and 32 of L1 instruction cache (ICACHE). The L1 instruction cache is four way associative while the data cache is eight way associative. The cache line size for all three levels is 64 bytes. The amount of L2 cache is 256 KB per core; the amount of L3 cache is 8192 KB per processor. The L2 cache is eight way associative while the L3 cache is 16 way associative. Note that since there is significantly more predictability in the sequence of instructions that need to be fetched than in the

sequence of data that will be used, the associativity of the instruction cache is lower. In other words, additional flexibility is provided for the data cache in deciding by allowing for a larger pool of locations from which a block of memory can choose.

1.7.2. Reading and Writing Data in the Presence of Caches

Example 1.9 outlined a mechanism that allows the CU to assess whether a variable, stored at a certain address *adrs* in the memory, is available in cache or not. Moreover, if the variable is available, the CU can find out based on the "valid bit" whether the value stored there is good to be used or is stale. If all that the CU needs to do is to read the value stored at *adrs* the situation is pretty simple. If the value stored at *adrs* is cached and its valid bit allows it, the value is moved into a register for further use. This is the fortunate situation, called a cache hit. A cache miss occurs when the value stored at *adrs* is not in cache or when the valid bit indicates the value to be stale. Upon a cache miss, a trip to the actual memory location *adrs* is made, the block of memory that contains the variable of interest is mapped into a cache line, the valid bits for the line are set to "fresh", and the CU request is serviced from cache. The salient point is that the CU always gets serviced from cache since if the data is not in cache, the CU is asked to wait for the variable to be first brought into cache. Two concepts, applicable for multi-level cache hierarchies, can be defined in conjunction with cache hit/miss scenarios. A cache "hit time" is the amount of time required to move down the cache hierarchy and figure out whether a memory request is a hit or miss. A "miss penalty" is the amount of time spent to bring a block of memory into L1 cache. If the required data is in L2, the penalty is relatively small. If it's not in L2 but is in L3, the penalty increases since data if first moved into L2 and from there into L1. The penalty is large if the data is not found in L3 either. The data would then be brought from main memory into L3; from L3 into L2; and finally from L2 into L1. Note that an even worse situation is when the data is not in main memory and it would have to be fetched from the hard disk. To conclude, a fortunate CU read request incurs an overhead equal to the hit time. Upon a cache miss, the overhead is the sum of the hit time and miss penalty.

A CU write request is slightly more involved. Suppose a variable `foo` is stored in a memory block *B* that is mapped into a L1 cache line *C*. For now, assume only L1 cache is present in the system. If the executable modifies the value of the variable `foo`, where and how is the new value of `foo` stored by the system? The answer to this question depends on the "write policy" adopted by the processor. Regardless of the write policy, most solutions first update the value of `foo` stored in C. At this point, in the "write-through policy", the memory controller initiates an update of the value of `foo` stored in *B* right away. A write buffer is typically used to ease the write latency by allowing the CU to initiate the write request and move on in the execution sequence as soon as the memory controller confirmed that `foo` was buffered for writing. In other words, before the new value of foo made it to *B* the CU can return to continue work on the FDX cycle. The larger the size of the write buffer is the more likely the CU will be able to deposit the new `foo` value and return to continue execution. However, one can imagine programs where, for instance, the variable `foo` is changed inside a short `for` loop at a rate that overcomes the rate at which writes out of the buffer to *B* can be carried out. In this case, once the buffer fills up, the processor will have to idle for the *C*-to-*B* write task to catch up.

The "write-back policy" is an alternate strategy in which the memory controller opts for a delayed update of the value of foo stored in *B*. The reason for this delay is as follows: maybe in the program executed by the processor there is a burst of activity that keeps updating the value of foo. Rather than making long trips to update foo in *B*, the controller decides to postpone this effort until the point when it becomes compulsory to follow through with the update of *B*. An update of *B* is absolutely necessary when, for instance, the value of *B* is to be sent to a file to be saved on the hard disk. Then, the most recent value of foo needs to be flushed from *C* to *B* and from there on towards the hard disk. However, the more likely scenario under which *B* should be updated with the value in *C* is in multi-core CPU systems for memory coherence reasons.

1.8. Virtual memory

1.8.1. Introduction, Overview, and Nomenclature

The discussion in the previous section revolved around how data is moved back and forth between the processor and memory. While very fast, caches can only store a relatively small amount of data. Data that cannot fit in caches is stored in the main memory; this is off-chip memory: significantly more of it, but relatively slow to access. If the amount of a process' data exceeds the capacity of the main memory, the data ends up stored on a hard disk or solid state disk. A process here is defined as an instance of a program executed by the CPU.

A common sense question is this: when a program references a memory location with address 0x0010aa29, where does this physical location exist in reality? Since there are several and different physical entities used to store data in the memory hierarchy the answer to this question is not immediately obvious. Along the same line, a second related question is this: can two programs, which start roughly at the same instant of time and end at roughly the same instant in time, hold in their program counter the address 0x0010aa29 without the danger of colliding; i.e., stepping on each other? This latter question is relevant since while multitasking, a one core CPU jumps around executing slices of programs much like a juggler juggles multiple balls at the same time.

The amount of main memory is relatively small yet in multitasking it needs to enable the execution of several processes. The concept of virtual memory plays an important role in facilitating the execution of these processes. What virtual memory accomplishes is an abstracting of the memory hierarchy by enabling each process executed by the CPU to have its own perspective on memory size and logical organization. The outcome of memory virtualization is the ability to hide the peculiarities of each computer's memory and present the processor a simple albeit fictitious memory space that the processor relies upon when executing a program. An analogy would be like this: imagine a client needs to store goods in warehouse A. There is no reason for the client to be familiar with all the corners of warehouse A and understand how goods will be stored in it. This though takes time, which might be wasted since when dealing with warehouse B the client would have to learn that one too. Obviously, this would not scale. Alternatively, the middleman running the storage provides the client a nice picture of the layout of a generic warehouse: the storage bins are all aligned, the same size, etc. The middleman provides a nice picture of the warehouse: they are all rectangular, have 100 bins per row and go basically so deep that you don't have to worry about it. The dialogue between the client and middleman takes place using locations in this nicely laid out abstract warehouse. When the client needs to retrieve something from location 0x0010aa29, the middleman will get the goods stored in a bin that it told the customer that has location 0x0010aa29. Where that bin actually is in warehouse A is irrelevant as far as the customer is concerned, and is a matter that the middleman is left to deal with. Virtualization is the process of abstracting the concept of warehouse: the warehouse client provided a nice, clean, simple and fictitious image of it. In reality, things are different: the warehouse A has a certain capacity and shape, can store goods from many clients, etc. However, the client doesn't care about these details –the middleman handles it. Note that the hardware component that plays among other roles the middleman is the memory management unit (MMU). For Intel and AMD processors it lives on the same die with the CPU.

Going back to the second question above, two different processes executed by a one core CPU could indeed have their program counters point to the address 0x0010aa29. This is a virtual memory address, or logical address. When the CPU goes back to executing each one of them for a short time slice, each process will execute different instructions since in this process' universe the logical address 0x0010aa29 is translated by the MMU to different locations in the physical memory.

There are several reasons that make the use of virtual memory necessary. First, it allows the operating system to run programs whose size requirements (data and/or instructions) cannot be accommodated by the amount of physical main memory present in the system. Second, it enables the operating system to multitask; i.e., it allows multiple programs to be active at the same time. Third, in multitasking, virtual memory allows for a gracious handling of segmentation of the main memory. Fourth, it enables a transparent and unitary way of handling memory and secondary storage (hard disks or solid state drives), in that there is practically no distinction between the two. In fact, the main memory can be regarded as a cache for the secondary memory. Lastly, it obviates the need to relocate program code or to access memory with relative addressing when, for instance, linking against a library. The program relocation issue is resolved at link time when the linker has the freedom to adopt a "single process memory use" mode. The linker will have a lot of freedom to place the instructions in a library at an address that is oblivious of details pertaining to the physical memory organization. This applies as well to the compiler when generating machine code based on a high level language such as C. It will be the responsibility of the MMU to convert addresses in the universe in which the linker operated to addresses into the physical system, which is expected to store instructions and data from several other programs.

The discussion in this subsection has brought up several terms that must be defined more formally. Physical address represents an address that points to a memory location in main memory. Virtual address or logical address is an address in the virtual memory that the processor operates in. Memory frame is a chunk of physical memory that belongs to what we called the main memory (RAM); it is also called physical memory page. Virtual memory page is a chunk of virtual memory that the processor operates with. Address translation is the process of translating a virtual or logical address, which lives in some virtual memory page, into a physical address, which points to a memory location in a memory frame. As discussed shortly, this translation is based on a page table (PT), which stores all the information needed to track down a memory frame given a virtual memory page. The party responsible for creating this fictitious world in which a process operates and perceives to enjoy undisputed access to memory is the operating system (OS). It is the OS that partitions the main memory in frames and maintains the PT. Note that the OS is a program just like many other programs running on a computer. The processor starts executing this special program so that all our other programs can be executed in multitasking mode. In this mode, each program gets a very small time slice in which it has the undivided attention of the processor. Just like a juggler, the processor handles several “balls” at the same time. The time slice when the program is active is when the juggler touches the ball before it throws it yet again up in the air where the ball actually resides most of the time. In fact, the processor (or one of its cores, if a multicore processor) only works on one program at each time and quickly switches from program to program to attend to all their needs. This management of this process is run by the OS, which just like any other program, is executed by the processor once in a while. Note that the OS has to establish a protection mechanism that prevents programs that the processor is multitasking on from intentionally or unintentionally stepping on each other’s data and/or instructions.

1.8.2. Virtual Memory as an Enabler of Caching for Secondary Memory

Recall that using a small amount of on-chip memory, caching was a mechanism that made available very efficiently data/instructions that would otherwise require costly trips to the main memory. Thus caching replaced the vast majority of off-chip with on-chip data/instruction accesses. Underlying this trick was the data locality trait of most programs. Virtual memory facilitates a similar trick: creating the impression that a very large amount of exclusive use memory is readily available to the processor. Specifically, virtual memory attempts and is often successful in creating the illusion that almost no memory transaction is costlier than a transaction involving the main memory. Note that calling the system memory a hierarchy is further justified by the following observation: the virtual memory enables the main memory to play for secondary storage the same role that the cache played for the main memory. Taking this parallel between caching and virtual memory one step further, note that in caching one block of main memory gets mapped into a cache line. When handling the virtual memory, one chunk of secondary memory is mapped into a memory frame, which has a virtual memory correspondent in the form of a virtual page. While a cache line is typically 64 bytes, the size of a virtual page is often times 4096 bytes. Not finding data in cache leads to a cache miss; not finding data in a frame leads to a page fault. Both mishaps are costly, yet a page fault has a miss time several orders of magnitude larger. This is because the physical proximity of the main memory to the cache and to the fact that the management of the virtual memory is done in software by the OS.

1.8.3. The Anatomy of a Virtual Address. Address Translation.

A virtual address is obtained by joining two sets of bits. The first set occupies the positions associated with the least significant bits in an address and is used to indicate in each page the offset of an address. As mentioned above, the typical size of the page is 4096; i.e., one can reference 4096 addresses in a page of virtual memory or a frame of physical memory. If the operating system is a 32 bit OS, since $4096=2^{12}$, this means that 12 out of 32 bits of the logical address are used to indicate the offset of the memory location in the page. Note that the “unit of address resolution” on modern CPUs from Intel and AMD is eight bits (one byte); i.e., the 4096 addresses are used to reference memory spaces that each stores one byte, from where it follows that the amount of memory stored by a page is 4 KB. The remaining 20 bits left in the virtual address are used to index into a table to get the frame number, see Figure 17. The most significant 20 bits of the logical address are changed based on information stored in the page table to become a set of N bits that are spliced to the least significant 12 bits to generate a physical address. The size of N depends on the size of the physical memory. For a laptop with 2 GB of RAM, N=19, since 2 GB = 2^{19+12} bytes: 19 bites for frame index and 12 bits for offset in the page.

Sidebar 4: More on the concept of "unit of address resolution".
There are example of systems with an address resolution of 16 bits (a Texas Instruments processor introduced in mid-1970s), or 36 bits (the DPD-10, a very successful Digital Equipment Corporation mainframe computer introduced in mid-1960s).

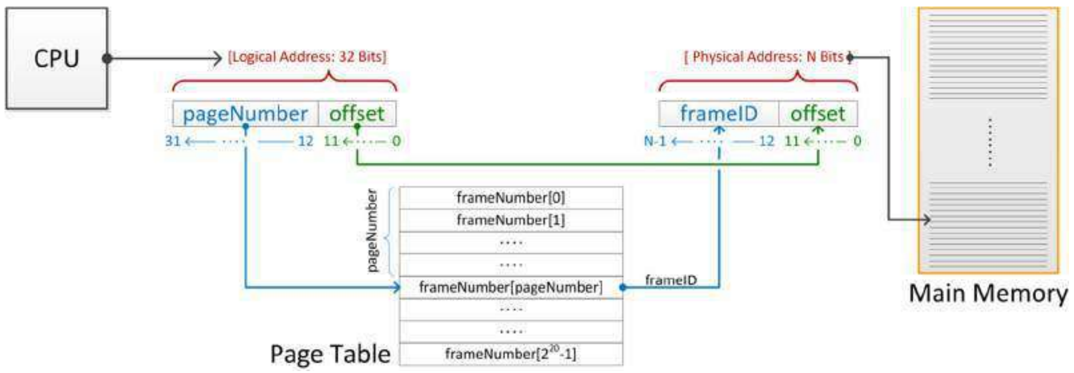


Figure 17. The anatomy of a virtual address on a 32 bit operating system and the process of address translation.

Example 1.10

Assume that a system runs a 32 bit OS. As such, the OS can reference 2^{32} addresses, each pointing to a one byte memory slot. This translates into 4 GB of virtual memory. Assume that the size of a page/frame is 4 KB. That means that the virtual memory can hold on to 10^6 pages. In other words, to cover the most demanding setup, enough memory should be set aside for the PT to store information about 1 million entries. For a 32 bit OS, storing an address typically requires 4 bytes. Consequently, each PT requires 4 MB of memory. This value of 4 MB is multiplied by the number of applications active at any given time, since each one of them has its own PT. Thus, if there are 30 applications active, the amount of main memory required would be 120 MB. The schematic in Figure 18 illustrates the translation process for a situation when the physical memory has 2 GB of RAM. In this process, the virtual address, which is represented using 32 bits ends up being mapped into a 31 bit physical address. This translation is successful since the required page is already in physical memory; i.e., the desired physical page number is in the Page Table. Had it not been, the page would have been brought over from secondary memory into RAM.

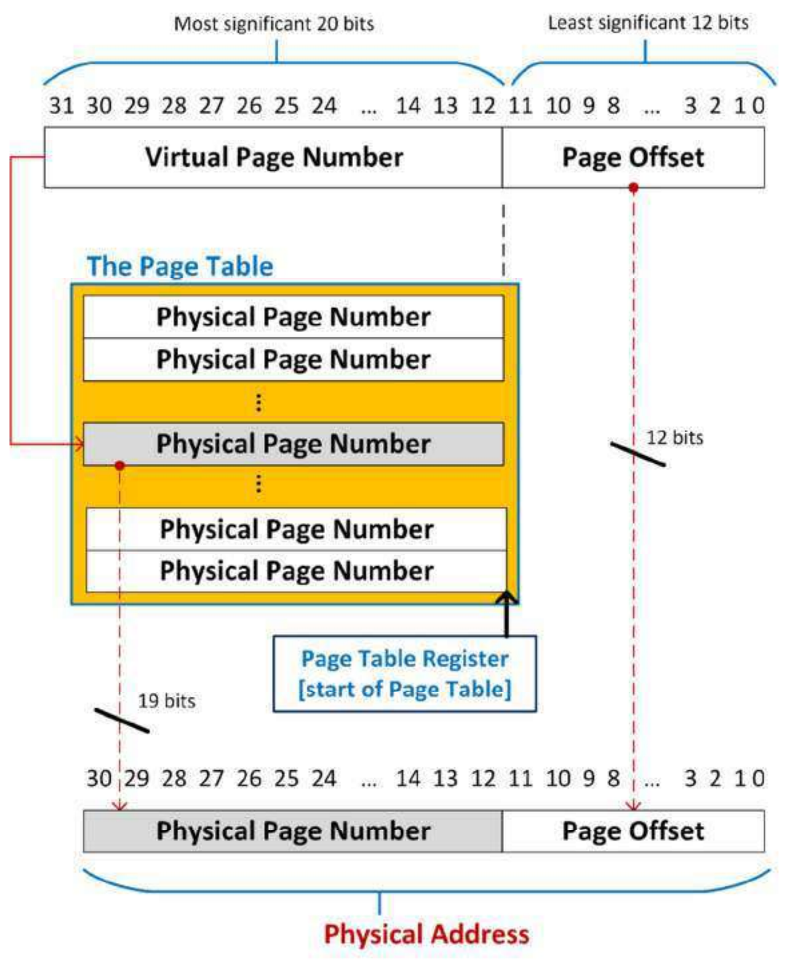


Figure 18. How the PT is used for address translation. Note that the PTR points to the beginning of the PT.

Sidebar 5.
This discussion segues naturally into the need for 64 bit operating systems. Up until recently, most commodity systems had 32 bit operating systems. In other words, the size of the virtual memory that the processor perceived as associated with each process was up to 4 GB. There were various tricks to go beyond this value, but they were awkward, convoluted, and most often deteriorated system performance. As of late 1990s, it became apparent that the pace at which the memory increased on commodity systems would lead to situations where the physical memory was larger than the virtual memory. While not necessarily a show stopper, it was deemed necessary to migrate to a 64 bit OS which again place the amount of virtual memory in a situation that accommodated all reasonable computing needs of the immediate future programs. The amount of virtual memory perceived by the processor after transitioning from 32 to 64 bits was more than 4 billion times larger than what a 32 bit OS could offer.

1.8.4. The Anatomy of the Virtual Memory

Assuming a 32 bit OS, the logical layout that the processor maintains of the virtual memory is that of a homogeneous space made up of 2^{20} logical memory pages. An active process executed by the processor at a certain instance in time partitions this logical memory space in several zones as shown in Figure 19.

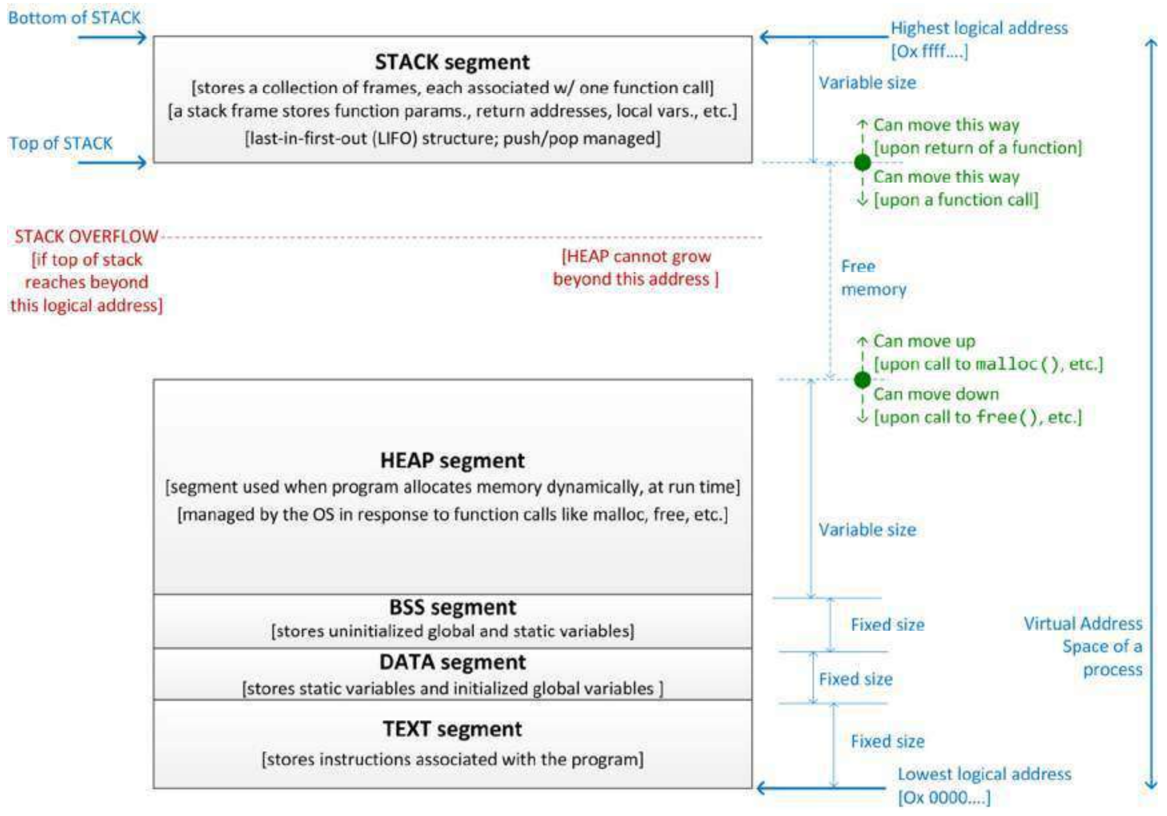
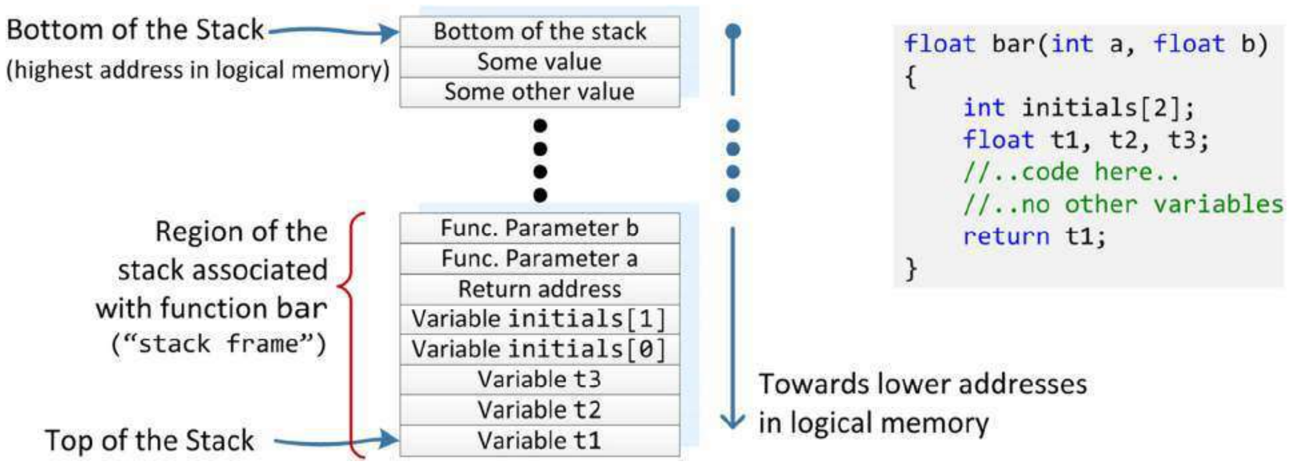

Bottom of STACK
STACK segment
[stores a collection of frames, each associated w/ one function call]
[a stack frame stores function params., return addresses, local vars., etc.]
[last-in-first-out (LIFO) structure; push/pop managed]
Top of STACK
Highest logical address
[Ox ffff....]
Variable size
Can move this way
[upon return of a function]
Can move this way
[upon a function call]
STACK OVERFLOW
[if top of stack reaches beyond this logical address]
[HEAP cannot grow beyond this address]
Free memory
Can move up
[upon call to malloc(), etc.]
Can move down
[upon call to free(), etc.]
HEAP segment
[segment used when program allocates memory dynamically, at run time]
[managed by the OS in response to function calls like malloc, free, etc.]
Variable size
BSS segment
[stores uninitialized global and static variables]
Fixed size
Virtual Address Space of a process
DATA segment
[stores static variables and initialized global variables]
Fixed size
TEXT segment
[stores instructions associated with the program]
Fixed size
Lowest logical address
[Ox 0000....]

Figure 19. The anatomy of the logical memory.

At the low end of the address spectrum comes a chunk of constant size virtual memory called the "TEXT segment" used to store the instructions associated with the program. It includes instructions associated with code compiled in libraries and which the program is linked against. The next chunk of constant size, called the "DATA segment", is used to store the initialized global and static variables in a program. For instance, in C programming, this is where variables declared like `static int i=0;` or `const char str[]="Hello!";` where the latter is declared outside the body of any function, would be placed. Next, the "BSS segment" is another fixed-size region of the memory, which stores at slightly higher addresses and end of the data segment global and static variables that do not have explicit initialization in the source code. At a yet higher address starts the "HEAP segment", which unlike the previous virtual memory regions is of variable size. It is a memory segment where memory is allocated dynamically at runtime. For instance, this is where memory is allocated upon a `calloc()` or `malloc()` call in the C language. The top of the heap goes up when the program keeps requesting dynamically more memory, and it goes down when memory is released, for instance, through `free()` calls in C. In fact, this allocation and release of memory, which in C is handled through pointers, is a major source of errors as programmers often times use "dangling pointers"; i.e., pointers pointing to memory that was released and cannot be referenced anymore, or by hogging memory when the program fails to release memory with a `free()` call once an allocated chunk of memory is not needed anymore.

The last chunk of logical memory is the stack. The stack is like a buffer of fixed size whose content increases and decreases during the course of the program execution. The peculiar thing about the stack is that that it grows from the opposite end of the memory space: it starts at the highest address in logic memory and it grows towards lower addresses. The top of the stack thus has the lowest logical address in the stack. The stack stores in a stack frame data associated with a particular function call: function parameters, the value of the return address; i.e., the address of the instruction that is executed upon return from the function call, as well as all the variables that are local to the function being called.



```
float bar(int a, float b)
{
    int initials[2];
    float t1, t2, t3;
    //..code here..
    //..no other variables
    return t1;
}
```

Figure 20. The anatomy of the stack, with emphasis placed on the content of a stack frame associated with a function `bar`. The function `bar` has two arguments and uses four variables: `initials`, `t1`, `t2`, and `t3`. Storage is set aside on the stack for these and at least one more variable: the return address, which specifies the address of the next instruction executed upon a return from `bar`. Note that the specific order in which the quantities end up placed on the stack might be different and to a large extent is dictated by the compiler. It is common though for arrays to be stored such that their values are ordered in memory as in the array; i.e., addresses grow with the array index, see the array `initials`.

Sidebar 6.
A trick used to hijack the execution in the past had to do with the fact that each stack frame has to store the value of the return address. Malicious software might try to overwrite values stored in a stack frame stored at a higher address in logical memory with the hope that it would overwrite its return address. Oversimplifying the discussion, in Figure 20, imagine that the function `bar` is a piece of malicious software and its body one overwrites the content of `initials[1000]` with a value that represents the address that points to the beginning of a sequence of rogue instructions. Note that in the current stack frame `initials` is dimensioned as an array of two integers. If by chance, this process overwrites the return address of a stack frame at a higher address, upon return from that function call the execution will jump right to the beginning of the set of rogue instructions. At that point, the execution is fully controlled by the malware. There are now techniques that prevent this type of attacks.

The stack keeps growing as a function calls another function, which can call another function, etc. A good example is a recursive function call, when basically in no time one can invoke a function so many times that the stack grows fast and at some point the top of the stack attempts to write at an address that is already outside the buffer pre-assigned to store the stack. A scenario like this leads to a stack overflow and program crash. The amount of free virtual memory is laid between the heap and the end of the area pre-assigned to hold the stack. It gets smaller and smaller as the user allocates through `malloc()` calls, for instance, more and more memory on the heap.

Example 1.11

The simple program below leads to stack overflow:

```
int foo() {
return foo();
}

int main(){
foo();
return 0;
}
```

In fact, the compilers can sometimes understand that a piece of code is slated to lead to a stack overflow scenario. Here's the warning issued by the Microsoft compiler used in relation to this code: "`work.cpp(Line 3): warning C4717: 'foo' : recursive on all control paths, function will cause runtime stack overflow.`"

Note that in general the compilers are not capable of figuring out ahead of time that a program will run into a stack overflow scenario. The execution of the program stops when a stack overflow occurs.

1.8.5. How Multitasking if Facilitated by Memory Virtualization

After a compiler processes a collection of source files and a linker handles all the external dependencies, a computer program ends up as a union of sequences of instructions that might get executed. Why is it a union? Consider the bits of code that provide below the skeleton for a program that solves a linear system. A first sequence of instructions is associated with the function `LUfactorization`. A second sequence of instructions is associated with/implements the algorithm that handles the forward elimination and backward substitution, steps that are coded in C in function `forwElim_backSubst`. Finally, a third sequence of instructions implements the operations defined by the C code in the `main` function. The compiler has a view of the memory that is identical to that of the processor: it operates in the logical memory. The sets of instructions that define `LUfactorization`, `forwElim_backSubst`, and `main` are placed at logical addresses in the TEXT segment (see Figure 19).

```
void LUfactorization(float* aMatrix, int N)
{
//.. code to carry out factorization; overwrite content of aMatrix ..
return;
}

void forwElim_backSubst(float* aMatrix, float* sol, float* rhs, int N)
{
//.. code here to find the solution of Ax=b ..
//.. assume the LU factorization is stored in aMatrix
return;
}

int main()
{
int dim;
float *matA, *solution, *rhsVec;
// ... initialize all variables defining a linear system
LUfactorization(matA, dim);
forwElim_backSubst(matA, solution, rhsVec, dim);
// ... print out solution or use it for a different purpose
return 0;
}
```

When launched for execution, the translation table is used to identify the physical address of the beginning of the program. For any instruction of a program a unique strategy is implemented: If a page of logical memory that contains the instruction slated for execution is stored as a frame in the main memory then the instruction is fetched for execution[3]. The strategy cannot be applied though when the instruction is not present in main memory. Then, through a process that requires millions of cycles, a block of secondary storage memory is fetched and stored as probably several frames in main memory, one of which containing the instruction of interest. At this point the strategy outlined can be applied.

Most often, at any given time, the main memory stores frames that hold instructions for various programs that are active on a system. The operating system is the layer in between the processor and the programs getting executed that feeds the processor sequences of instructions for program A, then a sequence of instructions from program B, goes back to executing program A, next moves and feeds the processor instructions associated with program C, etc. Note that the operating system itself is a program, so it should get its fair share of turns to be executed by the processor as well.

In this model, the pages that store instructions for a program are nicely laid out in logical memory in the TEXT segment; this holds for all programs. However, the frames that actually store the data are scattered all over the place in main memory. For instance, during the execution of the program that solves a linear system, the function **`forwElim_backSubst`** might end up at different times in different frames in main memory based on what other programs happen to be active at the time when program A is active.

Sidebar 7.
Embracing the idea of abstracting the physical memory by means of a virtual model also allows a program to be executed by different computers as long as the computers (i) run the same operating system; and (ii) have a processor that implements the same instruction set architecture, say x86. The fact that a laptop has 1 GB of RAM and an older AMD CPU while a newer workstation has 48 GB of RAM and a more recent generation Intel CPU should pose no barrier to using the same executable, say a 64 bit Firefox, to search the web. The idea is that the operating system has an abstract perception of the

[3] In reality a block of instructions is cached and as such the instruction is scheduled for execution from cache.

underlying hardware. The hardware allows both the laptop and the workstation to support the virtual universe in which the operating system initiates the necessary steps to start and subsequently manages the execution of the program.

1.8.6. Handling Page Faults

Any page fault; i.e., a reference to memory in a logical page that does not have a physical frame counterpart in main memory, requires an access to secondary memory. This scenario leads to latencies of the order of millions of cycles since it takes hundreds of thousands of times longer to access secondary vs. main memory. Upon a page fault, a sequence of steps are taken by the OS: the data requested is located in secondary memory; a request is made for a frame in main memory to bring over the chunk of memory that includes the requested data; a frame is found and made available (note that another frame might be evicted, which requires making a decision which frame to swap); the requested data is loaded in the frame made available; the page table is updated to reflect the new organization of the main memory; the OS returns control to the program that made the request, having it retry the instruction that led to the page fault.

Special attention has been paid to the issue of reducing as much as possible the occurrence of page faults. The solutions adopted are built around a familiar assumption: there is space and time locality to memory accesses. Strategies adopted in the past to avoid the steep penalty of secondary memory data accesses include [2]: (i) make memory pages large to have a lot of material available in a memory frame; (ii) adopt a write-back policy for virtual memory since a write-through would be prohibitively expensive when conversing with the secondary memory; (iii) build the page handling mechanism (placing, searching, evicting, writing, etc.) in software.; and (iv) embrace fully-associative maps for placing frames in main memory.

Strategy (i) has the undesired side effect that a smaller number of frames would be accommodated in main memory, which can lead to memory fragmentation and reduce the number of processes that can be active at the same time. In this context, an active process is a process that is executed although at any given time it might be parked for brief periods of time due to a decision of the operating system. Strategy (iii) is not as fast as having a design based on dedicated hardware but at a modest cost one can accommodate fancy policies that are (a) flexible to accommodate various hardware configurations, and (b) upgradable with each new release of an OS. Finally, strategy (iv) has overhead associated with locating a given frame in physical memory. To this end, one can carry out a global search, which is costly. Alternatively, one can use what has been introduced as the page table. The PT has at least two columns: the first stores the address of a virtual page; the second stores the address of a frame in main memory. The PT removes the overhead of address translation in strategy (iv) at the price of introducing an additional data structure that needs to be stored somewhere and has a nontrivial size, see Example 1.10.

Strictly speaking, the amount of memory needed to store the PT is slightly higher since it stores in each row several additional bits that further qualify the information stored in that PT row. One of these bits is the valid bit. It is used to indicate whether the physical address is to main or secondary storage. As a rule of thumb, a zero bit is interpreted as suggesting that the required information is not in main memory and a page fault is imminent. Upon the page fault occurring, the OS looks for bringing data into main memory from a region in secondary memory called the swap space. Each process has its own swap space and it holds the address that is stored in the PT upon a zero valid bit. The swap space of a program A is a chunk of secondary memory where memory frames of program A are evicted when the main memory needs to accommodate requests made by a different active program B. The decision to evict a frame associated with program A and place it from main memory to the swap space is made by the OS. A least recently used (LRU) policy is most often observed and implemented in software based on a so called use bit that is associated with each row in the PT and lives next to the valid bit. An in depth discussion data swapping in and out of main memory is available in [2].

Example 1.12

Assume that a `malloc()` call allocates a 6 GB array and that the code later on starts to reference array entries in a random fashion. As memory requests hit this array at random locations that are far apart from each other the main memory is depleted one 4 KB frame at a time. As long as frames can be assigned in the main memory to pages in virtual memory we are going to deal with a minor fault. It is minor since it does not require fetching of a page from secondary memory, unlike a typical page fault. If the system has a small amount of main memory, say 4 GB, sooner or later the main memory will be exhausted. At that point, the process traps the operating system, which needs to figure out a little used frame that will be evicted to the swap file that lives in secondary storage. The main memory frame is next populated by the information that the process needs, in this case another block of 4 KB. Note that under different circumstances this frame might have been populated by data that was in secondary storage. It becomes apparent that major page faults are costly; if they occur repeatedly one is faced with thrashing: pages are swapped in and out due to numerous major page faults. Note that the amount of swap space in secondary storage is typically twice the amount of in main memory and it can get exhausted too.

Sidebar 8. On the concept of active programs, executed process, multi-tasking, and time slicing.
A program is active if there is a PT associated with it in main memory. There can be many active programs, yet on a one core system there is only one program executed at any given moment in time. The active programs are waiting in the wings ready to get their turn at being executed by the processor. Switching the execution focus from one program to another program, a process managed with OS oversight, is a very common occurrence. The process has many stages to it, but a key one is that of saving all CPU registers associated with a process, turning the process into an active program, loading a different set of registers and proceeding the execution of the new process. Recall that one of the registers is the Program Counter (`PC`), which holds the address of a memory location storing the next instruction that needs to be executed. Multi-tasking is this process of swapping between active programs in response to an interrupt request. Most often, but not always, this interrupt comes as a consequence of the expiration of the slice of time associated with the process currently executed. Time slicing is the process in which chunks of time are assigned to the pool of active programs indicating how long they can be processed by the CPU when they get their turn. In addition to allocated time, processes have priorities: an active process with a large slice of time allocated might wait a while before it gets a turn of becoming the executed process.

1.8.7. The Translation Lookaside Buffer (TLB)

The page table is an artifice specific to a program. The information in the PT needs to be stored somewhere; it turns out that this "somewhere" is in the main memory starting at an address stored in a dedicated register, sometimes called the page table register (PTR). Once the OS has the PTR, it knows where to go to start a dialogue with the PT to gauge the status of a virtual page and understand where in physical memory a frame can be located.

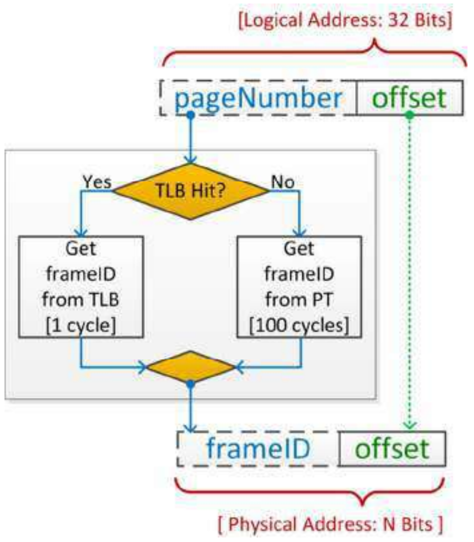

[Logical Address: 32 Bits]
pageNumber
offset
TLB Hit?
Yes
No
Get frameID from TLB [1 cycle]
Get frameID from PT [100 cycles]
frameID
offset
[Physical Address: N Bits]

Figure 21. Schematic of the page translation process that involves the TLB. The latency is two orders of magnitude higher upon a TLB miss.

This approach has the following drawback: it doubles the time required to fetch something from main memory. Recall that any address that the processor works with requires translation. With the PT placed in main memory, a first trip would be made to carry out the logical to physical memory translation. Once available, the physical address is used to execute the memory transaction, which might require a second trip.

Note that if this is the case, the whole purpose of having a cache, which was a mechanism put in place to avoid expensive trips to main memory, is defeated since the translation makes that trip anyway. A so called translation lookaside buffer (TLB) has been introduced in an attempt to alleviate this drawback. It plays the role of a cache that stores bits of information otherwise available in the PT. The TLB is on-chip, implemented in hardware, and physically provides a little bit of storage. Just like a cache hit, if the address translation can be performed based on information stored in the TLB, a memory transaction requires at most one trip to main memory. It can require no trip at all, if the memory transaction involves memory that turns out to be stored in cache.

The typical attributes of a TLB are as follows [2]: it has anywhere from 16 to 512 entries; since it's just like a cache, it has a line size, which is typically one to two PT rows; the mapping is most often fully associative; hit time is very short, typically 1 cycle; the miss penalty is 10 to 100 cycles, that is, what it takes to reach main memory; on average, the observed miss rates are 0.01 – 0.1 %.

1.8.8. Memory Access: The Big Picture

Figure 22 presents the workflow associated with a memory transaction. It is a simplified version of the actual workflow but it provides a big picture that explains how the hierarchical structure of a system memory operates to honor memory transactions elicited by the processor. The starting point is a memory transaction request that involves a logical address. The logical to physical translation can be fast, upon a TLB hit, or slow, upon a TLB miss. Once the translation is concluded, the flow branches based on the nature of the transaction. Upon a read, the cache is interrogated in relation to the address involved in the transaction. Upon a cache hit, the value is read and delivered to the processor. Upon a miss, a block is fetched from main memory, a line is populated in cache and the read is attempted again, this time successfully. A similar workflow is followed when handling a write.

As illustrated in Figure 13, the memory is organized as a hierarchy with a number N of levels. If one was to abstract the way information flows in this hierarchy, the focus would be on how the information is managed at the interface, moving back and forth between level i and level $i+1$, $i=1,\ldots,N-1$. If this process was optimal, it would be applied repeatedly between any two levels: between L1 cache and L2 cache, L2 cache and L3 cache, L3 cache and main memory, main memory and secondary memory, etc. The fact that the moving of data between two levels of the memory hierarchy is not entirely similar is due to the fact that various levels of this memory hierarchy have their own peculiarities. However, it is interesting that although they emerged independently and at different moments in time, the solutions for moving data across interfaces in the memory hierarchy have a lot in common. Specifically, looking at how data movement is managed at the interface between cache and main memory (interface "A"), and then at the interface between main memory and secondary memory (interface "B"), one notices a very similar abstraction expressed yet using a different terminology. For A, one has the concept of memory block; for B, one has a memory page. For A, one has a cache line; for B, the analog is a memory frame. For A, one can discuss about the line size, while for B of interest is the page size. A word in a line has the analog of an offset in a page. There is the concept of valid bit in both cases. For A we worry about cache misses; for B, we want to avoid page faults. To conclude, the message is that although developed at different moments in time and by different groups of people, the strategies that govern the data movement in the memory hierarchy are very similar and most likely explained by the abstraction that is inherent in any crossing of boundaries between levels in the hierarchy.

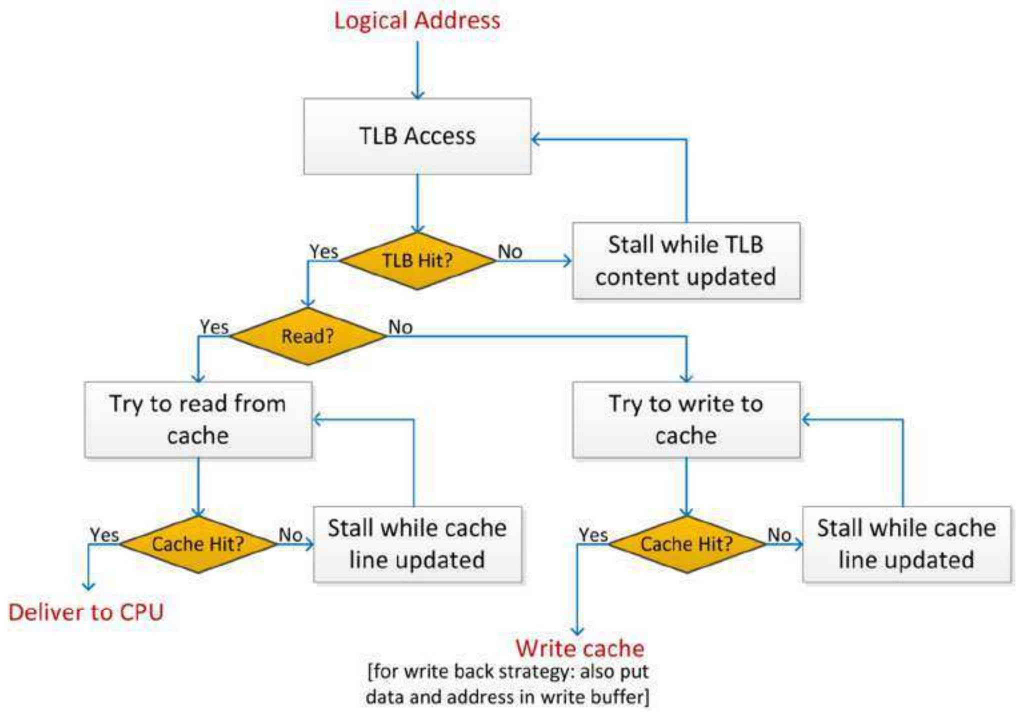

Logical Address
TLB Access
TLB Hit?
Yes
No
Stall while TLB content updated
Read?
Yes
No
Try to read from cache
Try to write to cache
Cache Hit?
Yes
No
Stall while cache line updated
Cache Hit?
Yes
No
Stall while cache line updated
Deliver to CPU
Write cache
[for write back strategy: also put data and address in write buffer]

Figure 22. Simplified schematic of a memory transaction. Given a logical address, translation takes place and the memory transaction involving the logical address is implemented in a memory hierarchy that involves cache.

The latency associated with a memory request following the workflow in Figure 22 can be very small if the request results in both TLB and cache hits. It can be extremely long though upon a TLB miss followed by a page fault. This means that a cache miss is basically guaranteed to follow as well. This aggregation of TLB miss, page fault and cache miss leads to a significant slowdown in the execution. A scenario like this substantiates the following crucial observation: *no matter how fast a processor a system has, the efficiency of the memory transactions dictates the overall performance of a program*.

In the light of this discussion it makes sense to alter the metric used in section 1.3 to define program performance:

$$\text{CPUtime} = \text{IC} \times \left(\text{CPI}_{execution} + \sum_i \frac{(\text{Memory accesses})_i}{\text{Instruction}} \times (\text{Miss Rate})_i \times (\text{Miss Penalty})_i \right) \times \text{CCT}$$

Above, IC stands for instruction count, CPI stands for clock cycles per instruction, CCT stands for Clock Cycle Time, and the index i runs over all possible types of memory transaction that can be associated with an instruction: access to L1 cache, L2 cache (if present), access to main memory, etc. Note that quantities such as $(\text{Miss Rate})_i$, for instance, are average values observed for a given program. . The value of the $(\text{Miss Rate})_i$ and $(\text{Miss Penalty})_i$ is provided in Table 8 for several types i of memory.

Attribute	L1 [per core]	L2 Cache [per core]	Main Memory	TLB
Size in Blocks	1,000	8,000	16,000,000	500
Size in KB	64	512	64,000,000	4
Block size in Bytes	64	64	4096	8
Hit Time [clocks]	1	10	100	1
Miss Penalty [clocks]	5	20	10,000,000	10
Miss Rate	3%	30% (out of L1 misses)	0.0001%	0.1%

Table 8. Attributes and typical values of several types of memory. The miss rate is specific to a program – the values reported are averages one could expect to encounter in typical programs.

1.9. From Code to an Executable Program

In Section 1.1 we briefly discussed how in a high-level language such as C a program, which is a set of C declarations and commands get translated into sequences of machine instruction executed by the processor. In Section 1.2, the discussion focused on the hardware support required to execute the machine instructions. The outcome of several logical operations such as AND, OR, NOT, etc., is easily reproducible using a reduced number of transistors. Increasingly complex logical operations, such as 32 bit addition of integers, can be supported in hardware by more sophisticated ways of combining transistors. In fact, we saw that there is a reasonable set of instructions that are supported in hardware. Examples include: add, move, load, jump to label, etc. This is a rather minimalistic group of operations that has been deemed critical to be supported in hardware. The union of these operations forms the Instruction Set Architecture of the chip and can be used to assemble, just like in a Lego game, complex constructs and algorithms, such as a program that finds the solution of a linear system.

In this section the discussion concentrates on the concept of file translation, which represents the process in which we start with one or several source files (in this discussion C files, sometimes called translation units) that get translated into a program, which subsequently can be loaded and executed on a system. The translation process is illustrated in Figure 23. Oval shapes represent stages where processing takes place (from where the pair of gears); the rectangles contain the input and output for each stage of the translation process. Specifically, a collection of source files (translation units) are processed by the compiler to produce assembly code (see Table 1). The assembly language closely mirrors the machine language; while the latter represents a sequence of 1 and 0 bits the former uses a mnemonic to represent their net effect. The assembler is the utility that in fact translates the assembly language instructions into machine instructions. Thus, the C code in a translation unit, say a file with the extension "c", **`fileA.c`**, yielded the assembly code stored in a file **`fileA.asm`** or **`fileA.s`** (depending on the operating system involved), which in turn yielded upon invocation of the assembler the object file **`fileA.obj`** or **`fileA.o`** (again, depending on the operating system). The linker can use this and several object files and/or static libraries to stitch information provided in these files into an executable, which is saved as a file saved with the extension .exe or .out (depending on the operating systems). Finally, this file, say myProgram.exe, can be launched for execution on the system where the translation process took place. At that point the loader is invoked to in order to have the executable stored in memory and the processor proceed to execute the first machine language instruction in the program **`myProgram.exe`**.

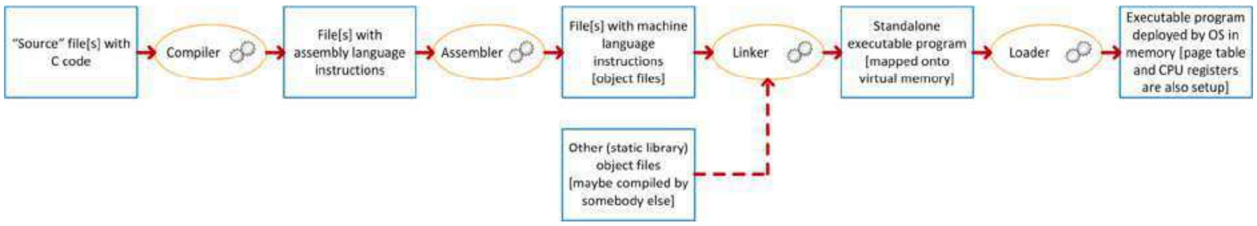

"Source" file[s] with C code
Compiler
File[s] with assembly language instructions
Assembler
File[s] with machine language instructions [object files]
Linker
Standalone executable program [mapped onto virtual memory]
Loader
Executable program deployed by OS in memory [page table and CPU registers are also setup]
Other (static library) object files [maybe compiled by somebody else]

Figure 23. The translation process for a C program. The case shown is that of an older approach that relied on the use of static libraries. Dynamically linked libraries require a slightly more convoluted linking and loading process but have been adopted for advantages related to the size of the executable and the opportunity to use continuously use the latest version of functions provided in a library.

The translation process in Figure 23 starts with a file that contains C code and ends up with an executable file that contains machine language instructions that the processor is slated to execute. The machine language instructions are the closest to the hardware that one can get; these instructions either belong to the ISA or are pseudoinstructions that represent constructs that are not directly implemented in hardware but they are the union of two or three instructions that are implemented in hardware (an example would be a human figure in Lego; it is used in so many setups that although it's made up of three basic Lego pieces is already comes assembled as a human figure). This being the case, a question whose answer is slightly counterintuitive is this regards two different system that each has say an AMD chip with the x86 ISA. Assume I start with **fileA.c** on the machine that uses Windows as the operating system and go through the translation process in Figure 23 to get **myProgram.exe**. You start with **fileA.c** on a machine that uses Linux as the operating system and go through the translation process in Figure 23 to get **yourProgram.out**. Since both systems rely on an AMD chip that implements the x86 ISA, the question is will **myProgram.exe** run on the Linux machine and will **yourProgram.out** run on the Windows machine? Without taking special steps, the answer is no, which is a bit counterintuitive. After all, the same C file was used in both cases, and the executable file contains instructions that belong to the same ISA and will eventually be executed by the same chip. The reason is that the way the assembler and linker work is quite different on the two operating systems. An explanation of the translation process in the next four sections will clarify the sources of these differences.

1.9.1. The Compiler and Related Issues

The compiler's job is that of taking a source file, say **fileA.c**, and express all the declarations and commands in this translation unit into a sequence of declaration and instructions in assembly language. In other words, in Table 1, the compiler takes the C code from the first column of the table and turns it into assembly code, shown in the second or third columns of the same table. As illustrated in the first column, the C code contains definitions such as double a=60, etc. Although the source file doesn't contain any there, functions are almost often defined in a C file. The way the variables and functions are declared/defined is very consequential when it comes to compiling code and later on how linking handles these declarations and definitions.

1.9.1.1. Definition v. Declaration

Each variable in a C program needs to be stored in memory at some address. The compiler attempts to produce this address and is in a position to do so in many but not all cases. It needs to decide first how much memory needs to be allocated to a variable. This is controlled by its type; for instance, typically an `int` requires four bytes and so does a `float`. A variable of type `double` requires eight bytes, and so on. There are user defined types, obtained using the `struct` construct, for which the size is dictated by the number and type of the variables making up the `struct`.

```c
double func1(int arg1, double arg2)
{
    double result;
    result = arg2 - (arg1-32.0)/1.8;
    return result;
}
```

Code Snippet 1. Content of fileA.c. The function does not actually do anything useful, some dummy code is provided that doesn't have a meaningful purpose beyond discussing the issues of definition v. declaration and variable scoping.

Consider for example Code Snippet 1, which shows the entire content of **fileA.c**. The C code in this file defines one function and one variable. The name `func1` is associated with a function that takes two arguments, one of time `int` and one of type `double`, which are subsequently used by the function to compute and return a variable of type `double`. This is all that it takes to define func1 as far as the compiler is concerned. Finally, one variable is defined by this translation unit: `result` is defined as a variable of type `double` and as such the compiler will have to decide on an address that identifies a block of eight bytes used to store `result` and will associate that address with the variable `result`.

At this point, the compiler is in a position to produce all the information required by a pushing on the stack of a call to the function `func1`. The compiler has an address for where the text in the body of the function is stored and where the execution should go upon a call to this function; it knows that there are three variables for which space on the stack needs to be reserved (`arg1`, `arg2`, and `result`); and it knows that when it pops this function from the top of the stack, `result` should be stored in a register subsequently to the caller.

Consider the C code in a different translation unit called **`fileB.c`**, see Code Snippet 2. This translation unit, or source file, defines one function, `func2`, and declares another one, `func1`. Note that `func1` has already been defined in **`fileA.c`** and there is no need to define it again. In fact, redefining `func1` will lead to an error during the linking phase since the same name; i.e., `func1`, would be associated with two definitions. More on this in the following subsection.

A declaration is just like a reminding to the compiler that the function `func1` has been or will be defined in a different translation unit and for all purposes, when dealing with this translation unit the compiler should treat any reference to `func1` as being to a function who prototype is provided at line 3 of **`fileB.c`** (see Code Snippet 2). Note that **`fileB.c`** also contains several variable definitions, such as result, dummy, `a1`, and `a2` being variables of type double. Variable `a3` is of type `int` and as such the compiler will allocate four bytes to store this variable at an address that it decides.

```
1.  extern const double valuePI;
2.
3.  double func1(int, double);
4.
5.  double func2(double a1, double a2, int a3)
6.  {
7.  double result=0;
8.  double dummy;
9.  dummy = func1(a3, valuePI);
10. result = a1 + a2 + dummy;
11. return result;
12. }
```

Code Snippet 2. Content of fileB.c. This translation unit defines one function, `func2`, and declares another function, `func1`. It defines several variables of type double, a1, a2, `finalValue`, dummy, and type `int`, a2.

To put things in perspective, for a function, a definition associates a name with the implementation of that function. At this point the compiler has all the information required for that function to choreograph a push to and pop from the stack upon a function call. For a variable, a definition instructs the compiler to allocate space in memory for that variable. This is different from a declaration, which only introduces/presents a function name of variable name to the compiler. The compiler is only made aware of their "fingerprint" and as such can handle any reference to these entities in the translation unit that is processing; i.e., can generate the assembly code that implements the push and pop from the stack.

1.9.1.2. Scoping in C Programming

There are two variables in Code Snippet 1 and Code Snippet 2 that are both called `result`. Will the compiler get confused? After all, a point was made in the previous section in that func1 in **`fileB.c`** is only declared since this name was already defined in **`fileA.c`** and this was done to make sure there is no ambiguity at compile and link time. The answer is no, the compiler will not get confused since the two variables called result have different scopes. The scope of a name represents the part or parts of one or possibly more translational units where the name has a type associated with it. There are some default rules that control scoping. For instance, a variable defined inside a function has a scope from the point where it is defined all the way to the end of that function; in other words, it's a local variable. By the same token, a function defined in a translation unit has a scope that extends from the point where it is defined all the way to the end of the translation unit.

```
1.  #include <stdio.h>
2.
3.  const double valuePI=3.1415;
4.  float crntTemperature;
5.  double func2(double, double, int);
6.
7.  int main() {
8.  double result;
9.
10. scanf("%f", &crntTemperature);
11. result = func2(crntTemperature, valuePI, 20);
12. printf("Result is %f\n", result);
13. return 0;
14. }
```

Code Snippet 3. Content of fileC.c. This translation unit defines the main function, which is the point where the processor begins to execute the program. Specifically, the processor starts by executing the first machine instruction associated with the `main()` function.

Consider the variable called `valuePI` at line 3 in Code Snippet 3. It is defined to be of type double. In addition to being defined, it's also initialized to the value 3.1415. Moreover, by using the qualifier const, it is indicated that it never changes its value, a piece of information that allows the compiler to slightly optimize the assembly code generated. `valuePI` is called a global variable, in that it is not specific to a function but instead its scope is from line 3 through 14. This variable can be made known to other translation units as well. Line 1 in Code Snippet 2 is a declaration preceded by the keyword `extern`. It indicates that a global variable `valuePI` is to be known to translation unit associated with fileB.c and it's supposed to be treated herein as a `const double`. The scope of `valuePI` has thus been extended beyond the one it had in **`fileC.c`**. to span the entire translation unit associated with **`fileB.c`**. The same argument applies to function `func1` in Code Snippet 1. The scope of `func1` is in the translation unit associated with **`fileA.c`**. However, by declaring it in **`fileB.c`**, its scope is augmented to include this file. As such it can be used inside this file, which is precisely what happens at line 9 in Code Snippet 2.

Note that there is another variable `result` that is this time defined by function `main()`, see line 8 in Code Snippet 3. Its scope is from line 9 to line 14. As such, this particular variable `result` does not collide with either the variable `result` defined at line 7 in Code Snippet 2 or variable `result` defined at line 3 in Code Snippet 1. This is because the intersection of the scopes of these three variables is empty. They are used in different places have local scope and they are completely different variables stored in memory at completely different locations. This cannot be said about `valuePI`: there is only one variable `valuePI` that is stored in one place in memory and available to be used both in **`fileB.c`** and **`fileC.c`**.

Next, consider the C code shown in Code Snippet 4. Line 2 shows a declaration of `crntTemperature`, which was defined in line 4 of fileC.c. Lines 3, 4, and 5 define three global variables whose scope extends from the line where they are defined till the end of the translation unit associated with **`fileD.c`**. Just like for the variable `crntTemperature`, there is only one memory location associated with `translUnitVarY_uninit`, `translUnitVarY_init`, and `translUnitVarY_initConst`. However, by using the keyword `static` these three variables are identified as being used only in this translation unit. This is unlike the variable `crntTemperature`, which was defined in **`fileC.c`** but now is used in **`fileD.c`**; i.e., in a different translation unit. This subtle distinction allows the compiler to have an extra degree of freedom when optimizing the process of translating the C code in **`fileD.c`** into assembly code. Note that of the three static variables one is initialized and one is initialized and declared constant (a constant variable is necessarily initialized at definition). Also our friends `func1` and `func2` are declared since they are used in `func3` at line 12 and in `func4` at line 18, respectively. Recall that the former is defined in **`fileA.c`** and the latter in **`fileB.c`**.

```c
1.  #define N 10
2.  extern float crntTemperature;
3.
4.  static double translUnitVarY_uninit;
5.  static double translUnitVarY_init = 2.0;
6.  static const double translUnitVarY_initConst = 3.14;
7.
8.  double func1(int, double);
9.  double func2(double, double, int);
10.
11. static double func3(double arg){
12. const int helperLocal = N;
13. double dummyLocal = func1(helperLocal, arg);
14. return arg*arg/helperLocal + translUnitVarY_init*dummyLocal;
15. }
16.
17. double func4(double arg1, double arg2) {
18. double helperLocal;
19. double dummyLocal = func2(arg1, arg2, -N);
20. helperLocal = arg2<1.0?1.0:arg2;
21. return func3(dummyLocal)/helperLocal+ crntTemperature + translUnitVarY_initConst;
22. }
```

Code Snippet 4. Content of fileD.c. The code further illustrates the idea of variable scoping by elaborating on the use of the `static` keyword.

Sidebar 9. On the concept of translation unit.
When a C file is passed to the compiler, all the `include` statements (such as in Line 1, Code Snippet 3), `const` variables (see `helperLocal`, Line 12, Code Snippet 4), `define` statements (such as in Line1, Code Snippet 4) are pre-processed. For instance, whenever possible, a `const` variable is replaced by its

value before generating assembly code. The define statement has a similar effect: the symbol N in lines 12 and 19 of Code Snippet 4 are replaced by the quantity 10. The `include` statement leads to more significant transformations of the code: all the definitions and declarations in the header file stdio.h are pulled into **`fileC.c`**. The **`stdio.h`** file contains basically the interface for the standard input/output C library, and as such it brings into the translation unit associated with **`fileC.c`** a lot of third party code/definitions/declarations. For instance, the **`gcc`** compiler can generate the translation unit associated with its file argument when invoked with the **`–E`** option. For instance, although **`fileC.c`** has 14 lines, the translation unit associated with it has 1154 lines. If the Microsoft compiler **`cl`** is invoked with the **`\E`** option as in

```
>> cl \E fileC.c
```

the translation unit has 5470 lines. For the translation unit associated with **`fileC.c`** all the extra inserted information ends up being superfluous; the only function used out of that `stdio` library is `scanf`, which reads from the input command line the value for the variable `crntTemperature`. This is reflected in a very small assembly file, **`fileC.s`** obtained as a result of the compilation stage. For illustration purposes, **`fileC.s`** and **`fileD.s`** provided in Table 9 are obtained using the gcc compiler:

```
>> gcc –S fileC.c fileD.c
```

Sidebar 10. The storage class issue.
In our discussion, as far as the scope of a variable is concerned, we have seen variables come in several flavors: local, static, extern. There is a story to be told in relation to where the compiler assigns memory to store these variables. The virtual memory space has been shown in Figure 19 to be split in several segments, of which the stack and the heap are changing during the execution of the program. For the sake of this discussion, the virtual memory space will be split in three segments: the stack, the heap, and everything else. In terms of what is called the storage class, local variables are variables that are stored on the stack while global variables are stored in the "everything else" segment. Like any respectable rules, this one has exceptions. For instance, if a variable is defined inside of a function then the variable's scope is from the place of definition till the end of the function; as such, the variable is allocated memory on the stack upon the function being called. However, if the variable is declared `static`, the storage class of the variable changes and memory space for it is set aside by the compiler in the "everything else" segment. Code Snippet 5 helps make this point: the variable called `interestingVariable` is defined and initialized to zero in both `func5` and `func6`. The latter also indicates in line 2 that this variable is of storage class "static". Although the scope of the two variables is only inside the functions where they are defined, memory is arranged to be made available only on the stack for `interestingVariable` when `func5` is compiled. As such, `interestingVariable` in `func5` will always assume the value 0 when the function is being executed. The value of the variable might change in line 6, but that is irrelevant since upon return, `interestingVariable` being a local variable its value is lost when the stack frame is popped. The situation is different for `func6`. When the compiler compiles `func6`, four bytes of memory are set aside in the "everything else" segment to store the value of `interestingVariable`; moreover, the bytes are set by the compiler such the value of the four byte integer is 0. Assume that the first time func6 is called anArg=99 and anotherArg=1. Then, due to the assignment in line 6, interestingVariable assumes the value 99. The function returns the value 100, and equally important, the value of interestingVariable has been modified and upon the next call of

func6 the interestingVariable will have the value 99 and not 0. The fact that the value 99 is "remembered" between function calls goes back to the class storage used for `interestingVariable` in func6; i.e., `static`, which sets aside memory in a persistent region (the "everything else" segment) as opposed to in a temporary region (the stack frame).

```
1   int func5(int anArg, int anotherArg){       int func6(int anArg, int anotherArg){
2       int interestingVariable = 0;                static int interestingVariable = 0;
3       if( interestingVariable>anArg )             if( interestingVariable>anArg )
4           anArg = interestingVariable;                anArg = interestingVariable;
5       else                                        else
6           interestingVariable = anArg;                interestingVariable = anArg;
7       return anArg + anotherArg;                  return anArg + anotherArg;
8   }                                           }
```

Code Snippet 5. Two functions that are identical bar the use of the keyword "`static`" in func6 at line 2.

Note that having a variable be assigned persistent memory space in the "everything else" segment does not necessarily require the use of the `static` keyword. Simply defining a variable in a translation unit outside the body of any function renders that variable non-local, and as such the compiler sets aside persistent memory for it. One though must use the keyword `static` when defining a variable inside a function, otherwise memory space for that variable will only be set aside somewhere in the stack frame associated with that function. This is the case of func5 in Code Snippet 5. To conclude, the keyword static has dual purpose: when used in a definition outside the body of a function it limits the scope of the variable; when used in a definition inside the body of a function it changes the storage class.

1.9.2. The Assembler and Related Issues

The role of the assembler is simple: it translates assembly code into machine code, see the discussion in Example 1.1. The outcome of the process is an object file, which has binary format. In most cases there is a one-to-one correspondence between an instruction in the assembly file and in the object file. After all, this was the purpose of using assembly language: operate at a level very close to the hardware yet do it using mnemonics to maintain a certain level of sanity to the process of writing code of this low of a level. On rare occasions, there are exceptions from the one-to-one mapping rule in the form of pseudoinstructions, in which one assembly language instruction translates into a couple of machine code instructions; i.e., one assembly instruction does not have an immediate correspondent in the ISA.

An object file has several sections whose need becomes apparent once we consider the implications of translating the source files in isolation. In Subsection 1.9.2 we considered a collection of files: **`fileA.c`**, **`fileB.c`**, and **`fileC.c`** that together combine to generate a program. The command

```
>> gcc fileA.c fileB.c fileC.c –o myProgram.exe
```

will combine the C code in three files to produce an executable stored in a binary file called **`myProgram.exe`**, a process illustrated in Figure 24.The object file **`fileA.o`** obtained from **`fileA.s`** is obtained without any knowledge of what files **`fileB.c`** and file **`fileB.s`** contain. This is a consequence of the fact that getting any object file from a translation unit is a process carried out in insolation. Therefore, the information stored in an object file should help other utility programs, such as the linker, get a clear image of the dependencies that the object file has on other object files or external libraries. Examples of such dependencies are numerous. Some of them are very intuitive, for instance the dependency of the object file **`fileC.o`** on the function `scanf`, which is provided by the standard input/output C library. Other dependencies are transparent to the programmer, such as the case when handling a jump in an if-then-else statement or a function call. A similar scenario is illustrated at line 57 in **`fileD.s`**, see Table 9, when a call to `_`**`func2`**, which is defined in a separate translation unit, is made. Therefore, a section needs to be present in the object file **`fileD.o`** to indicate that `_`**`func2`** is as an external dependency and it is expected to be provided by somebody else, which in this case is the translation unit associated with **`fileB.c`**. Note that the assembly file already contains at the very end, line 86 in Table 9, a brief description of the signature, or prototype, that is expected for **_func2**.

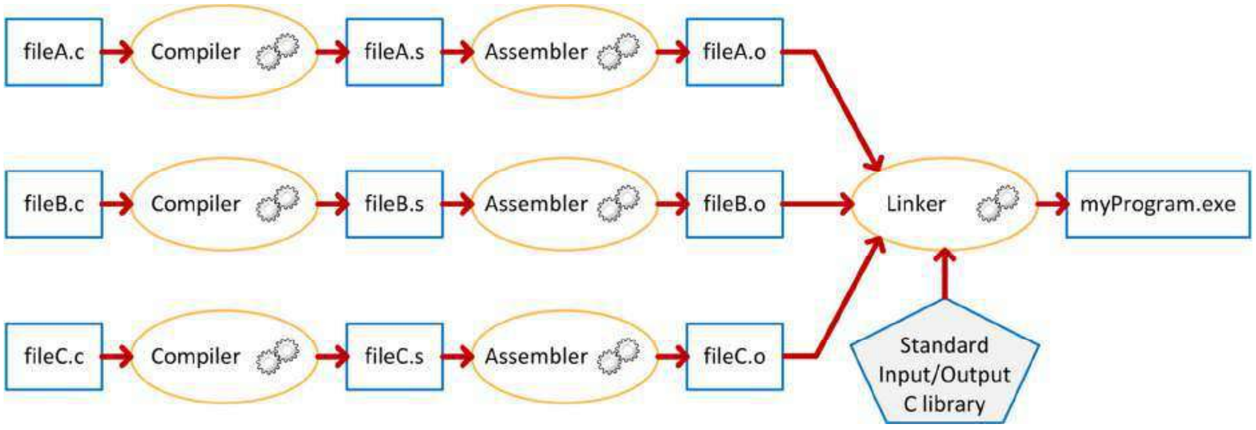


Figure 24. The steps triggered by the "**`gcc fileA.c fileB.c fileC.c –o myProgram.exe`**" command. Note that each of the three source files are processed in isolation and passed to the linker, which combines information from the object files and libraries (in this case only one library, the standard I/O C library) to produce an executable program.

Parsing the structure of **`fileD.s`**, one encounters several other sections: there is a preamble where information about data is provided, see lines 4 and 9. These are global variables, which end up logically stored in a particular region of the virtual memory, and to that end size information and initial values are provided if a variable was initialized as is the case with `translUnitVarY_init`. There is another section where the actual instructions are stored, which in Table 9 starts at line 13 for **`fileC.s`** and line 12 for **`fileD.s`**. Although we refer to these sections in relation to the assembly code and use in this discussion snippets out of assembly code in Table 9, the assembly code is translated into machine instructions or binary code that preserves these sections. Specifically, the end product yielded by the assembler; i.e., the object file, most often will include:

- The *object file header*, which contains information regarding the structure of the rest of the object file: where the other sections start, how many of them, how long they are, etc.,
- The *text section*, which contains the instructions to be executed by the processor should it have to execute a function defined in this translation unit,
- The *data section*, which stores global initialized data, see for instance the variables `valuePI` in **`fileC.c`** and `translUnitVarY_init` in **`fileD.c`**,

- The *BSS section*, which stores global uninitialized data, see for instance `crntTemperature` in **`fileC.c`** and `translUnitVarY_uninit` in **`fileD.c`**,
- An *external references section*, which indicates the functions that should be made available to functions defined in this translation unit. For **`fileD.s`**, the function func2 is expected to show up as an external reference since it's not defined in **`fileD.c`**,
- A *relocation information section*, which holds information about instructions and data that depend on absolute addresses and is relevant later in conjunction with the linker,
- A *dynamics linking information section*, which comes into play when the source file is compiled with the end goal of including some or all of the functions defined into a dynamically loaded library (more on this later),
- A debugging information section, which contains information that can be used to trace each machine instruction back to a line of code in the original C file. This type of information is needed when executing the program step-by-step in a debugger.

Comparing the nomenclature introduced above with the one used in conjunction with the virtual memory, see Figure 19, it is not but chance that they are very similar. The information stored in the object file's sections will end up mapped logically in the corresponding segments of virtual memory during the process of linking and/or loading of a program image for execution. For instance, the instructions stored in the text section of an object file will end up being assigned addresses in the text segment of the virtual memory; the BSS section in the object file provides information that is relevant in relation to the BSS segment of the virtual memory, etc.

```
.file   "fileC.c"
.globl _valuePI
.section .rdata,"dr"
.align 8
_valuePI:
.long   -1065151889
.long   1074340298
.def    ___main;        .scl    2;      .type   32;     .endef
LC0:
.ascii "%f\0"
LC1:
.ascii "Result is %f\12\0"
.text
.globl _main
.def    _main;  .scl    2;      .type   32;     .endef
_main:
leal    4(%esp), %ecx
andl    $-16, %esp
pushl   -4(%ecx)
pushl   %ebp
movl    %esp, %ebp
pushl   %ecx
subl    $52, %esp
call    ___main
movl    $_crntTemperature, 4(%esp)
movl    $LC0, (%esp)
call    _scanf
fldl    _valuePI
flds    _crntTemperature
fxch    %st(1)
movl    $20, 16(%esp)
fstpl   8(%esp)
fstpl   (%esp)
call    _func2
fstpl   -16(%ebp)
fldl    -16(%ebp)
fstpl   4(%esp)
movl    $LC1, (%esp)
call    _printf
movl    $0, %eax
addl    $52, %esp
popl    %ecx
popl    %ebp
leal    -4(%ecx), %esp
ret
.comm   _crntTemperature, 4, 2
.def    _scanf; .scl    2;      .type   32;     .endef
.def    _func2; .scl    2;      .type   32;     .endef
.def    _printf;        .scl    2;      .type   32;     .endef
```

```
.file   "fileD.c"
.data
.align 8
_translUnitVarY_init:
.long   0
.long   1073741824
.section .rdata,"dr"
.align 8
_translUnitVarY_initConst:
.long   1374389535
.long   1074339512
.text
.def    _func3; .scl    3;      .type   32;     .endef
_func3:
pushl   %ebp
movl    %esp, %ebp
subl    $40, %esp
movl    8(%ebp), %eax
movl    %eax, -24(%ebp)
movl    12(%ebp), %eax
movl    %eax, -20(%ebp)
movl    $10, -12(%ebp)
fldl    -24(%ebp)
fstpl   4(%esp)
movl    -12(%ebp), %eax
movl    %eax, (%esp)
call    _func1
fstpl   -8(%ebp)
fldl    -24(%ebp)
fmull   -24(%ebp)
fildl   -12(%ebp)
fdivrp  %st, %st(1)
fldl    _translUnitVarY_init
fmull   -8(%ebp)
faddp   %st, %st(1)
leave
ret
.globl _func4
.def    _func4; .scl    2;      .type   32;     .endef
_func4:
pushl   %ebp
movl    %esp, %ebp
subl    $72, %esp
movl    8(%ebp), %eax
movl    %eax, -24(%ebp)
movl    12(%ebp), %eax
movl    %eax, -20(%ebp)
movl    16(%ebp), %eax
movl    %eax, -32(%ebp)
movl    20(%ebp), %eax
movl    %eax, -28(%ebp)
movl    $-10, 16(%esp)
fldl    -32(%ebp)
fstpl   8(%esp)
fldl    -24(%ebp)
fstpl   (%esp)
call    _func2
fstpl   -8(%ebp)
fldl    -32(%ebp)
fld1
fucomip %st(1), %st
fstp    %st(0)
jbe     L9
L8:
fld1
fstpl   -40(%ebp)
jmp     L6
L9:
fldl    -32(%ebp)
fstpl   -40(%ebp)
L6:
fldl    -40(%ebp)
fstpl   -16(%ebp)
fldl    -8(%ebp)
fstpl   (%esp)
call    _func3
fdivl   -16(%ebp)
flds    _crntTemperature
faddp   %st, %st(1)
fldl    _translUnitVarY_initConst
faddp   %st, %st(1)
leave
ret
.lcomm _translUnitVarY_uninit,16
.def    _func1; .scl    2;      .type   32;     .endef
.def    _func2; .scl    2;      .type   32;     .endef
```

Table 9. Assembly code for fileC.s (left) and fileD.s (right). The compiler used was `gcc` with flag –S.

1.9.3. The Linker and Related Issues

Most often, the mission of the linker is to combine information provided in a collection of object files and libraries in order to produce an executable. This process is illustrated in a simple case in Figure 24: a set of three C files lead to three object files, which are used by the linker to produce the executable **`myProgram.exe`** after bringing into the picture information from the standard input/output C library. Another mission of the linker is to produce a library, just like the one used in Figure 24. The difference between an executable and a library is that in the former there is no dependency that is left dangling. Specifically, the external dependency section in fileC.s indicates in lines 47, 48, and 49 that there are dependencies on `_scanf`, `_func2`, and `_printf`, which are expected to be resolved by functions defined in other object files or libraries. In generating an executable file, the linker looks at all input files; i.e., fileA.o, fileB.o, fileC.o, and standard I/O C library. Indeed, `func2` is defined in **`fileB.c`** while the functions `scanf` and `printf` are made available by the standard I/O C library. In generating a library, unresolved symbols might be left, with the understanding that in order to generate an executable, other object files and/libraries will need to be linked with this library.

1.9.3.1. Generating an Executable

Recall that the linker takes a collection of object files and libraries and generates a binary file that has a structure very similar to that of an object file or a library. In other words, a binary file is produced whose structure resembles very closely that of an object file. Focusing on generating an executable, as shown in Figure 24, the data sections, the text sections, the BSS sections, etc. will be combined together into one bigger data section, one bigger text section, one bigger BSS section, etc. This is called *relocation*, and it is a process that eventually allows the linker to assign absolute addresses to labels that were put in place like stubs by the assembler: see lines 64, 64, 67, 68, 71 in Table 9. In other words, there will be no more labels but absolute addresses that will be used for data addresses (global variables like `crntTemperature`, for instance), jump instructions (where the code needs to jump to execute `func2`, for instance), and branch instructions (where the execution should proceed upon hitting the "`else`" part in an `if-then-else` instruction).

Example 1.13.

If one opens an object file in a text editor such as **`emacs`**, **`vim`**, or **`notepad`**, its content will look like a collection of illegible characters. There are several utility tools under Linux and Windows that allow one to peek inside an object file, or for that matter, an executable file. For instance, running in Linux the command

```
>> nm fileD.o
```

yields the following output (the numbers at the beginning of each line were not part of the output):

```
1.  00000000 b .bss
2.  00000000 d .data
3.  00000000 r .rdata
4.  00000000 t .text
5.           U _crntTemperature
6.           U _func1
7.           U _func2
8.  00000000 t _func3
9.  00000046 T _func4
10. 00000000 d _globalVarY_init
```

```
11.   00000000 r _globalVarY_initConst
12.   00000000 b _globalVarY_uninit
```

The file fileD.o, produced by the gcc compile driver, stores information in the Executable and Linkable Format (ELF), which is a common standard file format for object files, executables, and libraries. The **`nm`** utility reveals some of the information stored in fileD.o. For starters, the object file is indicated to contain information that will end up in the bss section, and data section. In terms of text, two functions are defined at line 8 and line 9. _func3 is of type "t", which indicates that it is local to this file. This is a consequence of the keyword `static` that qualifies the definition of this function in Code Snippet 4. A "T" indicates that the function is defined and can be invoked/used in a different translation unit. A letter "U" indicates global variables whose definition was not found, and it alerts the linker that it should look elsewhere for their definition. Indeed, `crntTemperature` can be picked up from **`fileC.o`**, `func1` from **`fileA.o`**, and `func2` from **`fileB.o`**. A symbol with a "d" or "D" type indicates an initialized global variable, which is local ("d") to this translation unit or visible elsewhere ("D"); each of these cases will prompt the linker to include the symbol in the right section of the executable file (BSS, data, etc.). An uninitialized global variable gets a type "b" if it's static, which is the case with `globalVarY_uninit`, see **`fileD.c`**. Finally, in this example the symbol `globalVarY_initConst` is associated with a different type "r", which prompts the linker to relocate this global and constant variable to a different section of the executable. Note that there are several other symbol types that the linker is aware of and not seen in this example. Each of them is a hint to the linker of what needs to be done for that symbol and where; i.e., at which address, it needs to be placed in the overall structure of the resulting executable file.

It was pointed out above that labels are replaced by absolute addresses. In what context are these "absolute address" defined? Multi-tasking enabled by modern day operating systems precludes the use of main memory as the space in which these absolute addresses point. This is because it is not known ahead of time where in main memory a program will store its data or where its instructions will be deployed after being fetched from secondary memory. The "absolute address" context is in fact that of the virtual memory, which provides an idealization that serves as a framework in which the program is anchored. In other words, absolute addresses are addresses in the virtual memory, which the program envisions as having exclusive and discretionary powers over its use.

1.9.3.2. Generating a Library

Assume we develop code to solve a linear system. The code calls for a matrix and a right-hand side to be loaded from a file. If an iterative approach is used to solve the linear system, a starting point is required, which we assume that is provided using the keyboard, interactively, by the user. At the end, the solution is displayed on the monitor and also saved, using a larger number of digits for higher precision, into a file. When linking the executable for this program, a large number of object files will be required to ensure that every function used in the source code is found in one of these object files. To make things even more interesting, imagine that you need to link against the object file that contains the definition of the function `scanf`, which is was used to read data in. Unbeknown to us, this function might call a different function of which we have not heard. As such, we should provide the linker yet another object file that contains this function that sneaked it because `scanf` was used. Locating this file might be extremely difficult because we basically don't know what we are exactly looking for. Even if we find this object file, maybe the sneaky function calls yet another function, further complicating the picture. Libraries have been introduced to alleviate this problem. All these object files that provide access to often-used utilities such as `printf`, `scanf`, etc., are packaged together in a *library*. Several libraries are often times needed at link time owing it to the fact that libraries are meant to encapsulate the support specific to a task or application. For instance, the task of writing, reading, printing to the standard output, parsing a line, etc.; i.e., input/output operations are supported by the standard C library; support for math functions such `sin`, `cos`, `exp`, etc. are provided by a math library; support for solving sparse linear systems using various methods might be provided by a user defined library; support for digital image processing can be provided by a different user defined library, etc.

Libraries typically come in two flavors: static and dynamic. *Static libraries* are increasingly a thing of the past and they are easily obtained by simply archiving together a collection of object files. The outcome of the archiving process is a file that ends in the suffix `.lib` on Windows systems or `.a` on UNIX platforms and which can subsequently be passed to the linker along with other object files that invoke functions defined in the library. There are at least three perceived disadvantages to the use of static libraries. First, the size of the executable file can be large. Whenever the linker finds a function definition in an object file that was archived in a library called, for instance, **`linAlgebra.a`** it includes in the executable the entire content of that object file. In other words, picking up a function definition from an object file brings along a lot of baggage. Second, linking an executable using static libraries represents the end of the journey for the executable. Any modifications made to the static library such as algorithm improvements, code optimization, bug fixes, etc. will be reflected into the executable only if the linker relinks the executable. Finally, if a powerful desktop multitasks 60 statically linked executables and 25 of them use a function called `foo`, the text segment of each of these 25 programs will store the machine code associated with foo. This redundancy might be costly due to the replication of the same machine instructions in several frames of the main memory.

There has been a manifest shift recently towards using shared libraries, sometimes called dynamic libraries. They address the three disadvantages listed above yet are conceptually more complex to produce, manage, and handle. Conceptually, one can think of a shared library as one big object file that is generated by the linker using a collection of input object files. Thus, the content of all text sections is relocated in one bigger text section, the information in all BSS sections is compounded into one BSS section of the library, etc. While generating a shared library is very much a process of mixing and stirring up the contents of several object files, generating a static library only requires an archiving of the input object files: the content of the object files is not mixed up. The consequence is that while linking against a static library the linker can pull in one object file on an as-needed basis, when handling shared libraries the entire library is pulled into the executable as soon as one function defined in the library is used in a program.

Example 1.14.

The purpose of this example is to illustrate how a static and a shared library is generated. The task that serves as the vehicle to this end is a simple program that converts a temperature provided in Fahrenheit to Celsius. There are three files that combine to provide this functionality: **driver.c**, **ioSupport.c**, and **conversion.c**. They are provided in Code Snippet 6 through Code Snippet 8. The functionality provided by the source files **ioSupport.c** and **conversion.c** will be provided through a library. On a Linux system, a static library is obtained by simply compiling the two source files with gcc and then archiving the two object files together:

```
>> gcc –c ioSupport.c conversion.c
>> ar rcs libUtility.a ioSupport.o conversion.o
>> gcc driver.c –L. –lUtility –o exmplStatLib.exe
>> ./exmplStatLib.exe
23
Input  Temp. [F]:  23.000000
Output Temp. [C]:  -5.000000
```

The –c flag passed to the gcc compile driver indicates that the driver should stop right after generating object files out of the two argument files (**ioSupport.c** and **conversion.c**). The ar command archives the object files to generate a library called **Utility.a**. The flags **r** forces the inclusion of each object file into the archive with replacement (if the archive already exists and has content coming from a previous object file with the same name); the **c** flag prompts the creation of the archive with the name specified; the **s** flag prompts **ar** to write an object-file index into the archive that is subsequently used by the linker to generate the executable. This index can also be created separately if need be. The next **gcc** command generates an executable called **exmplStatLib.exe** by compiling the source file **driver.c** and linking the object file against the library **Utility.a**. Note that the **–L** flag followed by a dot indicates that the compiler driver is to searched the current directory for a library called **libUtility.a**. This latter aspect is indicated by the **–l** flag, which is followed by the name of the library. Notice that in the library's name the prefix **lib** was dropped, as was the posfix **.a**. Finally, the executable is run to transform the value of 23 F in -5 C.

```c
#include <stdio.h>

float convertF2C(float);
void outputResult(float valF, float valC);

int main() {
   float tempC, tempF;
   // read in the temp. in F
   scanf("%f", &tempF);
   tempC = convertF2C(tempF);
   outputResult(tempF, tempC);
   return 0;
}
```

Code Snippet 6. Content of driver.c.

```c
float convertF2C(float valueF) {
   float tempC;
   tempC = 5*(valueF-32)/9;
   return tempC;
}
```

Code Snippet 7. Content of conversion.c.

```c
#include <stdio.h>

void outputResult(float valF, float valC) {
   printf("Input  Temp. [F]:  %f\n", valF);
   printf("Output Temp. [C]:  %f\n", valC);
}
```

Code Snippet 8. Content of ioSupport.c.

Generating a shared library requires similar steps.
gcc -fpic -c conversion.c ioSupport.c

```
>> gcc -fpic -c conversion.c ioSupport.c
>> gcc -shared -o libUtility.so *.o
>> gcc -o exmplSharedLib.exe driver.c -L. -lUtility
>> export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:.
>> ./exmplSharedLib.exe
32
Input  Temp. [F]:  32.000000
Output Temp. [C]:  0.000000
```

The files that make up the library are compiled with the flags “pic” to produce what is called position independent code, an aspect discussed in the next section. The second command generates the shared library, called **`Utility.so`**. Subsequently, the executable **`exmplSharedLib.exe`** is generated. Finally, the loader needs to be instructed where to search for the shared library. In this case, the current directory is appended at the end of the list of directories where the loader will look in order to find shared libraries. The executable is run to show that 32 F corresponds to 0 C.

1.9.4. The Loader and Related Issues

When a program is invoked for execution, a loader picks up the program for secondary storage and goes through a sequence of steps that eventually result in the processor being posed to execute the first machine instruction associated with the `main()` function. This sequence of steps is managed by the loader, which is employed by the operating to [2]:

- Parse the header of the executable and gauge the memory requirements associated with the text and data segments
- Resolve physical memory space issues so that enough frames are set aside for the data and text segments of the program
- Load the data and text segments into the physical memory while updating the TLB in the process
- If the program is passed a set of parameters copy the parameters onto the stack
- Update the argument registers and the machine registers
- Invoke a start-up routine of the operating system that eventually makes the Program Counter Register point to the first instruction of the main function associated with the program at hand.

The loading process is more complicated than described, see for instance [9] for a thorough description of both the linking and loading processes. Yet the points above provide enough intuition into the steps choreographed by the operating system upon requesting the execution of a program. They highlight how several actors come into play to enable the execution of a program. Specifically, the loading process requires moving information from secondary into primary memory, setting up a TLB, dealing the various segments of the virtual memory (DATA, BSS, text, etc.) and mapping them into physical memory, setting up the stack and heap, populating all registers of the processor with the appropriate information such that the execution of the main function can commence, etc.

The scenario just described is mainly associated with programs that use no libraries or if libraries used were static. Nowadays, due to a shift towards shared libraries, the loader can assume some of the linker's responsibilities. For instance, consider the shared library case in Example 1.14. Using a shared library calls for a separation of the data and code so that there is only one instance of the code present in the physical memory while having multiple data instances each associated a specific program that uses the shared library. To this end, like in Example 1.14, the object file should be produced/organized in a specific way by generating position-independent code (PIC) suitable for use in a shared library. This is critical since the program uses absolute addresses which will need to be updated through the use of a global offset table; i.e., one more level of indirection. The dynamic loader resolves the global offset table entries when the program starts, and it is this step that in normal circumstances would be handled by the linker. In fact, it is not necessary to completely and immediately establish this linkage between any library function potentially invoked by the program and the corresponding shared library implementation of the function. Rather than embracing a proactive stance, a lazy, reactive strategy is most often pursued by the loader. The actual indirection is resolved for a function call only if this function call is actually made. When it happens the first time, extra time is needed to resolve the indirection. However, upon any subsequent call of the function the price paid will be one additional jump that is associated with the indirection, which will add a negligible amount of overhead.

2. Parallel Computing: The Basic Architecture and Software Model

"Nothing is as powerful as an idea whose time has come."
- Victor Hugo

2.1. Introduction. Parallel Execution vs. Parallel Computing

Parallel computing represents the process of enlisting multiple hardware assets to simultaneously execute (*a*) multiple sequences of instructions belonging to the same program; or (*b*) the same sequence of instructions multiple times, in a fashion partially controlled by the application developer. At the other end of the spectrum, sequential computing can only handle one stream of instructions at a time. Up until early 1970s, the vast majority of computing was done on sequential systems whose layout was qualitatively similar to the one shown in Figure 25. The system was made up of memory and a basic processor whose execution focus was set on one instruction at a time. Instructions were stored and read from memory; they provided the work order for the processor, which was capable of reading and writing data from/to memory. This architecture gradually evolved, and although the processor kept executing code sequentially, it became more sophisticated by implementing pipelining. A schematic of a pipelined chip is provided in Figure 26.

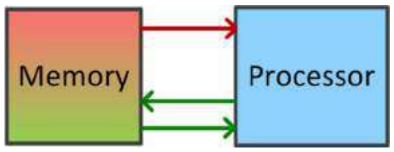


Figure 25. Schematic of early system architecture. Blue is used to indicate a hardware asset that processes information. Green is used for data flow. Red is used for commands/instructions that the processor is expected to honor. The memory shows a mixture of green and red as it stores both instructions and data.

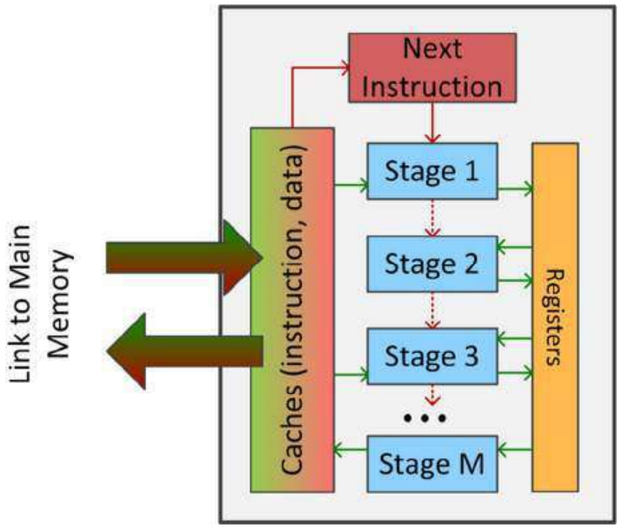


Figure 26. Typical chip layout in mid 1980s to mid-1990s. The chip was designed around the idea of pipelining. Registers represented in orange; otherwise observing the color conventions in Figure 25.

According to the definition introduced at the beginning of the section, pipelining stops short of being parallel computing since the application developer did not have any control over the simultaneous processing of the instructions down the pipeline. This will be called *parallel execution* and a distinction will be made between it and parallel computing.

Note that in the parallel execution setup enabled by pipelining, M instructions were processed *at the same time* and the chip displayed what is called temporal parallelism: several consecutive instructions were processed simultaneously. To use an analogy from car manufacturing, the simple system layout in Figure 25 is how cars used to be assembled prior to Ford adopting an assembly product line. A team of jack-of-all-trades-and-master-of-none started to work on one car in the morning and by the end of the day they would put it together. Alternatively, in Ford's solution, an assembly line had M stages or stations. In Stage 1 one would receive two parts from the warehouse and assemble them towards completing a chassis; by the end of the assembly line, a car would be produced. This allowed for (*i*) a high degree of specialization of the individuals working on the assembly line as individual J mastered the tasks associated with Stage J; and (*ii*) increased output as each tick of the assembly line yielded a car. While individual J did not work on more cars at the same time, note that M cars were simultaneously being worked on.

Up until mid-1990s, a pipelined processor like the one in Figure 26 had one chip that was plugged in a motherboard using one socket. This chip had one core, which in turn had one control unit that was focused on sequential but pipelined execution of instructions on this one core. At this point parallelism was at work, in the form of parallel execution of instructions. Multiple programs could be executed by this core by having it switch its focus from program to program in a very short amount of time to enable multitasking. Yet, at any given time, there was only one program whose instructions were executed by the core; the rest of the programs were parked waiting for their turn to come in the spotlight. The trick worked since the chip operated so fast that every once in a while each program got a chance to be picked up for execution creating the illusion that all programs are moving forward at the same time.

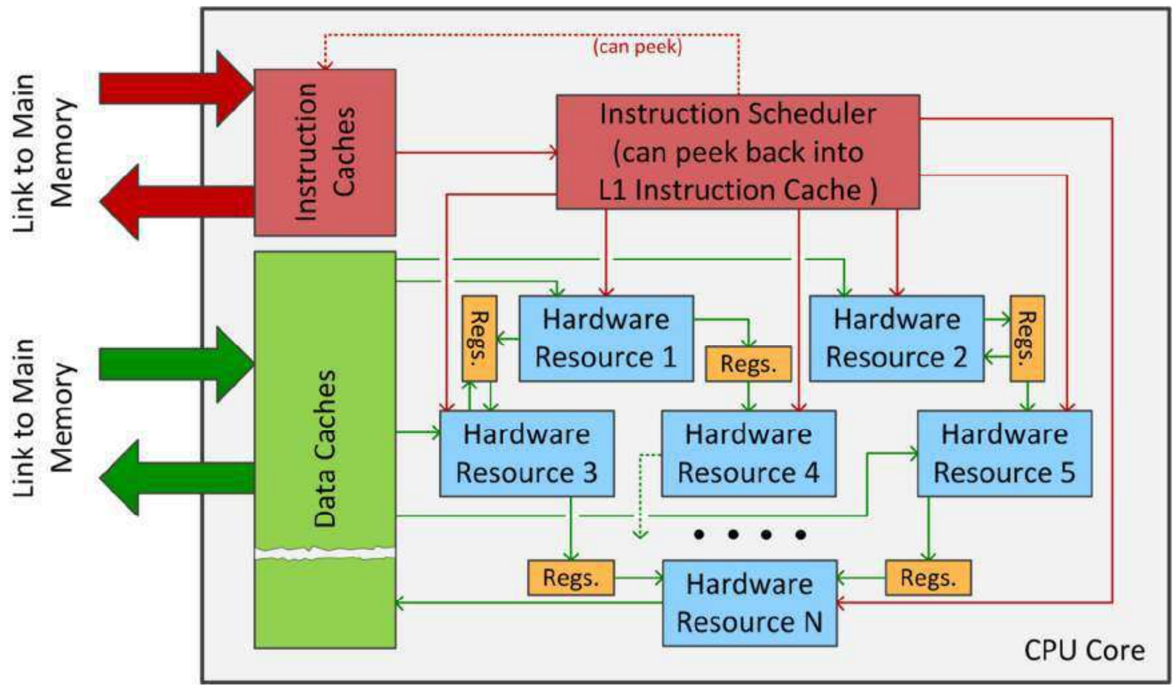


Figure 27. Schematic of a modern core architecture. Each chip contains a collection of several of these cores. The same color conventions are observed as in Figure 25. Several hardware resources shown in the picture are used, sometimes simultaneously, to execute micro-instructions. The micro-instructions are the most basic entities obtained after a machine instruction is broken down into more primitive elements.

In mid to late 1990s, as more transistors become available on the chip, engineers looked into new ways to boost the performance of the chip. Rather than exclusively focusing on effective instruction execution, the chip designs changed to (*i*) introduce specialized microarchitecture components and (*ii*) emphasize the effective use of all the hardware assets available on the chip. The new design layouts looked like the schematic shown in Figure 27: the layout was more complex and represented a departure from the pipelined view of the previous generation of design. In addition to pipelining, owing to its specialized microarchitecture components, the new chips supported out of order execution of instructions, branch prediction, speculative execution, hyper-threading, which are several techniques discussed in sections 2.2 and 2.3. Each of these techniques was meant to further enhance the parallel execution oomph of the chip. These techniques might have been all or some active at any given time, the idea being to utilize to the fullest extent possible all the hardware resources on the chip. Referring to Figure 27, this meant that "Hardware Resource 2" is utilized to process one instruction while "Hardware Resource 5" was used to process yet another instruction, etc. These hardware resources had the ability to communicate and coordinate through the use of caches or registers. Multiple instructions were issued for execution during each clock cycle and consequently many were retired at the same time. Tens to hundreds of instructions could be "in flight" at any given time: some instructions were at the point of taking off, some landing, some flying to destination, and finally some circling around waiting for resources to become available.

Example 2.1. IBM's PowerPC chip

An example of a sophisticated processor organization is the IBM PowerPC design. PowerPC chips were used by Apple's Macintosh between 1994 and 2006. The picture shows the microarchitecture organization of the PowerPC4 and PowerPC5 [10]. It had a superscalar organization (see section 2.3) and eight execution pipelines: two load/store units (LD1, LD2), two fixed point units (FX1, FX2), two double precision multiply-add execution units (FP1, FP2), and two execution units that handled the branch resolution (BR) and the management of the condition register (CR), where the latter acted as a bulletin board that displayed information related to the status of the instructions processed by the chip. I-cache stands for the instruction cache, which feeds into the instruction queue (Instr Q). Many resources are allocated for what is called instruction level parallelism (see section 2.2). Specifically, there were units for branch scanning and prediction (BR Scan, BR Predict) that provide the ability to peek into the code through a large instruction window for scheduling; a unit for decoding the instructions and forming groups of instructions scheduled for execution (Decode & Group Formation), the latter an attribute of Very Long Instruction Word (VLIW) chip architectures; a unit that handles speculative instruction fetches and manages the Instruction Fetch Address Register (IFAR); an execution unit that manages the global completion table (GCT) used in conjunction with instruction dispatch groups. Support for data caching came from two other execution units: D-Cache was used for reads and writes, the latter supported by the store queue (STQ) unit, for write-back caching strategies.

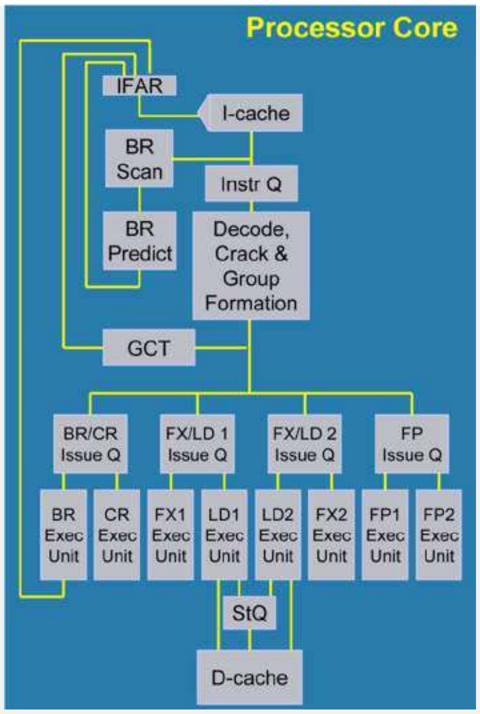


Figure 28. Layout of PowerPC architecture associated with the fourth and fifth generations of the chip [10].

This "different streams of instructions" can belong to the same program or to different programs. For the former, consider the case of an application that needs to perform three tasks that are completely independent; as such, they can be performed at the same time by three execution threads. For the latter, consider on operating system (OS) that manages the execution associated with a word processor and email client. The OS can manage the execution of these two tasks on the different compute cores

2.2. Increasing Execution Speed through Instruction Level Parallelism

The 1995-2005 decade represents a stage in the evolution of the chip design marked by increasingly complex layouts. Figure 27 and Figure 28 illustrate this point. The large number of transistors available in a small area provided the designers with an opportunity to cram new functionality inside the chip. This functionality translated into faster single stream execution speed through deeper instruction pipelining, superscalar execution, out-of-order execution, register renaming, branch prediction, and speculative execution. Pipelining was discussed in section 1.2.6; the only incremental gain associated with the new chip designs was the implementation of ever deeper pipelines and/or the presence of several pipelines on the same chip. Pipelining enables the parallel execution of instructions since effectively a number of instructions equal to the number of the pipeline stages can be worked on at the same time. This form of parallelism was called temporal parallelism, to distinguish it from spatial parallelism in which different resources on the chip; i.e., hardware components located in different areas of the chip and possibly executing completely different functions, execute different instructions simultaneously.

Out-of-order execution refers to the ability to enlist to support of existing chip hardware assets in selecting for execution and executing small sequence of instructions that are data independent and can be executed in parallel. A simple example is provided in Code Snippet 9. The matter is subtle and the user does not have any say in how the instructions get executed. Instead, either the compiler at compile time, or more often the instruction scheduler at run time, can figure out that the variable a occupies **a1** register in line 1 and it would be costly to vacate the register, wait maybe on a long memory fetch to bring **c1** and/or **d1** to perform the instructions associated with the line of code 2, and then bring **a1** back and execute the instructions associated with the line of code 3. Therefore, without user control, the instructions are executed as follows: first will be executed all instructions associated with the line of code 1; next, and here is where the out of order aspect crops up, are executed all the instructions associated with line 3; finally, all instructions associated with line 2 are executed. The key observation is that instructions are executed in an order dictated by the availability of input data, rather than the order induced by the C code wrote by a developer. Out of order execution gained popularity through the PowerPC line of chips introduced by IBM 1990. It required complex logic support and initially was available on higher end chips. It stands to benefit of a large out-of-order window, a concept that refers to the number of future instructions that the scheduler can peek at. Usually this window is of the order 100 to 200 instructions.

```
1. e1 = a1 + 2.*sin(b1);
2. f1 = tan(c1)/(1+d1)^2;
```

```
g1 = cos(a1);
```

Code Snippet 9. Three lines of C code that will lead to a sequence of instructions that might be the subject of an out of order execution.

Register renaming is a technique to increase the likelihood of identifying short sequences of 2-10 instructions that can be executed in parallel. Consider the following sequence of instructions in a pseudo-assembly language:

```
1.  MUL R2,R2,R3   ;     % implements R2 = R2 * R3
2.  ADD R4,R2,R2   ;     % implements R4 = R2 + R2
3.  ADD R2,R3,R3   ;     % implements R2 = R3 * R3
4.  ADD R2,R2,1    ;     % implements R2 = R2 + 1
5.  DIV R5,R4,R2   ;     % implements R5 = R4 / R2
```

Instructions 1 and 2 could be executed at the same time as instructions 3, 4, and 5 if it wasn't for the register R2 being used in instructions 3, 4, and 5. However, either the compiler or the processor can figure out that by using a different register, say R7, and renaming R2 with R7 in instructions 3, 4, and 5 can improve execution speed by issuing instructions from two streams per clock cycle. To support this kind of trick, the hardware should be resourceful since the renaming typically happens at run time and requires special logic. Moreover, it's not always possible to rename a register since there are not that many of them on a chip: X86 architectures rely on 8 registers; the 64 bit x86 ISA relies on 16; RISC architectures might have 32; and the Intel IA-64 ISA has 128 registers. A larger pool of registers certainly increases the opportunity for register renaming. Some high end chips go as far as having a set of registers that are not named and as such cannot be used by the compiler to generate assembly code. They are managed at run time by the processor to increase the opportunity of instruction-level parallelism.

Branch prediction is important insofar pipelining is concerned. When the execution encounters an `if-then-else` statement the pipeline might have to stall for the condition in the statement to be evaluated. Suppose that the condition reads `if(sin(a)<b/3.5)`. Evaluating this conditional branch requires the execution of a sizeable collection of instructions. But as long as these instructions are evaluated in theory the pipeline cannot be fed since it is not clear whether is the "if" or "else" branch that requires execution.

To prevent pipeline stalls, a good processor will attempt to execute the conditional ahead of time maybe through out of order execution or register renaming, or directly engaging a dedicated hardware resource, if available. Another approach is to not bother and simply execute both branches. If this is the case, branch prediction is basically preempted: the processor has enough resources to issue multiple streams of instructions with the understanding that some computations will be wasted. This is just one example of *speculative execution* is also often times applied for prefetching memory.

Sidebar 11. On speculative execution.
There are some caveats to **speculative execution**: intermediate results should not be sent out to caches or memory since this or other program running on the system at the same time might see these results and use them. Specifically, upon using speculative execution, imagine that a variable called `temperature` is computed on the "if" branch and the change in its value gets reflected in the main

memory. Suppose next that it turns out that the "else" branch provides the correct execution path, and the variable temperature is not referenced at all. The results associated with instructions on the "if" branch are discarded but this poses a small dilemma: what happens to `temperature`? If it got updated in the main memory during the speculative phase of the execution, some other program might have read and used a `temperature` value that is erroneous. The solution is to keep everything local and avoid creating data dependencies. This requires additional resources on the chip for temporary storage, ability to run more streams of instructions, etc., but this is precisely where Moore's law comes into play: every 18 months or so, the number of transistors doubles and they beg to be put to good use.

2.3. Superscalar Architectures. Hyper-Threading

A processor is said to have a superscalar architecture if it has the hardware units that allow it to finish more than one instruction at each clock tick; i.e., the processor has an instruction per cycle (IPC) higher than 1. The fact that the hardware units are sufficient to complete more than one instruction per cycle, there is no guarantee that any program executed on this architecture will complete with an average IPC higher than 1. Note also that pipelining alone cannot render an architecture superscalar since, even in an ideal scenario with no stalls, it can yield up to one instruction per cycle, IPC=1. For instance, the PowerPC5 design had a superscalar architecture since it could yield up to five instructions per cycle. A lot of dedicated hardware resources were made available on the chip to produce this yield, and IBM called the chip eight-way superscalar: it had two load/store units, two fixed point units, two floating point units, logical operations on the condition register, and a branch execution unit. This enabled the chip to manage more than 200 instructions during a clock cycle; i.e., it had more than 200 instructions "in flight", about five of them being retired during each cycle while eight instruction entered the execution sequence during each cycle. The difference between the number of instructions being picked up for execution (eight) and the number of instruction that completed execution during each cycle (five) represents the "unlucky" instructions that were executed speculatively (e.g. branch prediction, data prefetch) and eventually ended up discarded.

The two prerequisites for designing good superscalar architectures were (a) proficiency in relation to packing more hardware support units into the chip, which came as the length feature became ever so smaller; and (b) a good understanding of what hardware resources needed to be crammed onto the chip to avoid stalls due to lack of resources. Requirement (a) was eventually met owing to Moore's law. Requirement (b) is in equal parts science, art, and inspiration. Different applications have different resource requirements: a scientific computing program that uses finite differences might beat on a completely different set of instructions than a voice recognition application does. Thus, in some cases, the intended use of a chip will reflect in its architecture design. One example is the PowerPC chip, whose design is aligned well with its mission to work on scientific computing applications. Likewise, the design of modern GPU, discussed in section 2.5, aligns well with its mission to perform floating point operations fast.

Sidebar 12.
The idea of superscalar execution is not specific to the computer chip. In the auto industry, mass production led to a flavor a superscalar production. Rather than having only one production line, several were put in place. The layout of the line was also changed from something very linear, to something

more distributed. For instance, imagine that for each car there was a chemical treatment that was required on the newly applied paint, which it took 10 minutes. At the other end of the spectrum, there was an operation like installing the four wheels, that required one minute. It became apparent that there was a need for several chemical treatment stations and that only one station for wheel hook-up could serve more than one assembly line. Just like the PowerPC discussion above, the plant would then have the capacity to produce five cars per unit time (assembly line clock tick).

Hyper-threading refers to the process of using the same set of hardware resources to run more than one sequence of instructions through the processor. For all purposes, the OS sees this physical processor as two virtual processors. The rationale for hyper-threading was a desire to hide memory latency with useful computation and the observation that cramming more hardware resources on a chip can facilitate the issuing of two instructions from two different streams of execution to one processor. This point is important, since it captures the fundamental difference between hyper-threading and superscalar execution. The former handles simultaneously instructions belonging to totally different streams of instructions, most likely associated with different programs. The latter simultaneously handles several instructions that belong to the same sequence of instructions.

Hyper-threading increased somewhat the complexity of the chip since additional scheduling units were required to coordinate the execution of two different streams. Their execution needed to be choreographed so that they shared the set of resources on the chip and logic was in place to handle any possible contention. This was no easy task given that at any time a very large number of instructions were in flight. However, this increase in complexity was offset by a reduction of execution stalls due to data dependency and hiding memory latency with useful execution. For instance, for superscalar designs without hyper-threading, the scheduler would pick instruction 23 and 24 for execution, then attempt to get 25 and 26 executed, then 27 and 28, etc. Because of data locality, often times this strategy was bound to lead to stalls, since instruction 26 might need data produced by instruction 25. Or in a pipelined setup, 26 might need data produced in instruction 23, which is sent to a different pipeline and data forwarding might not be applicable between these two pipelines. With hyper-threading, a first stream would be associated with program **`a.exe`** while the second stream would be associated with **`b.exe`**. The chip would execute instruction 5 of **`a.exe`** and 239 of **`b.exe`**, then 6 of **`a.exe`** and 240 of **`b.exe`**, 7 and 241, etc. The chances of instruction 240 having a data dependency on instruction 5 of a different program are low. Moreover, this helped hide memory latency: if instruction 6 of **`a.exe`** called for a main memory access upon a cache miss, instructions 240, 241, 242, etc., of **`b.exe`** could be executed while **`a.exe`** was stalled waiting on the main memory transaction.

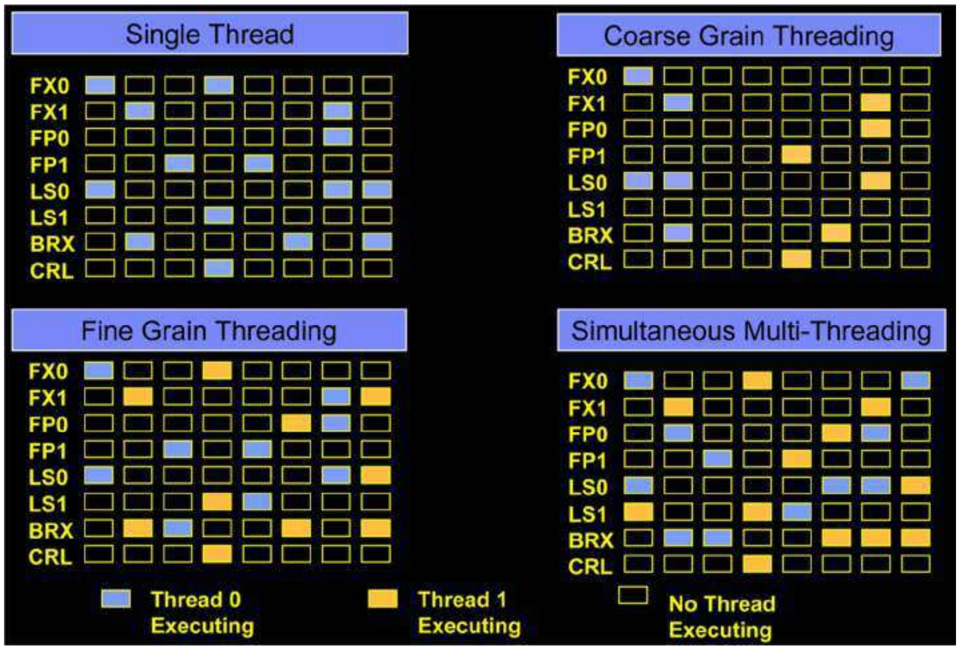


Figure 29. Chip designs become more versatile in time and today fully support hyper-threading; i.e., the ability of one processor to execute two streams of instructions without any interventions, or for that matter control, from the user. The image displays the evolution of the support for multi-threading [10].

In the beginning, the architectures were superscalar but not hyper-threaded. Note that hyper-threading cannot be employed unless the chip has the ingredients of a superscalar architecture. The PowerPC chips supported hyper-threading as of mid-2000s; the Intel chips followed suit shortly thereafter.

2.4. The Three Walls to Sequential Computing

For about three decades, between 1970 and early 2000s, sequential computing led to steady gains in computer power. Early on, as illustrated in Figure 30 [11], these gains could be traced back to a doubling of the number of transistors per unit area (Moore's law) each 18 to 24 months and a steady increase of the clock speed at which the chips operated. During 1990s, instruction level parallel execution support, as implemented by sophisticated chips similar to the one shown in Figure 28, provided an extra boost in the speed at which programs were executed.

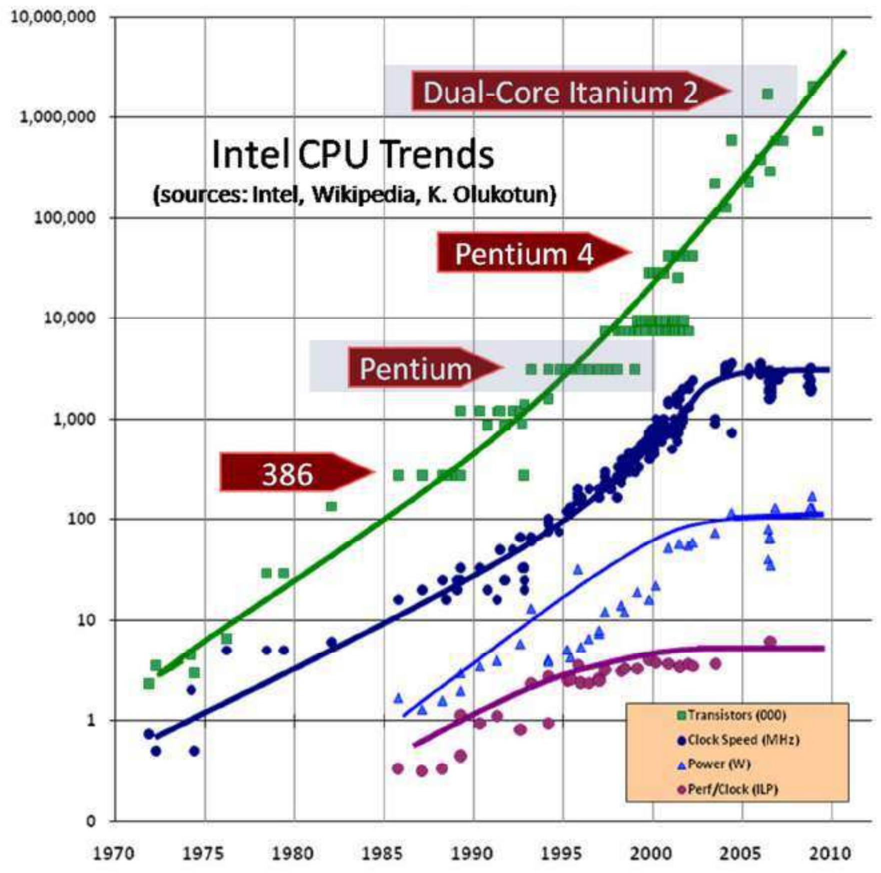


Figure 30. Evolution of several attributes of the Intel chip: the number of transistors, clock speed, power, and performance per clock, the latter envisioned as reflecting the instruction level parallelism prowess. The vertical axis reports data on a log scale.

Two of the three factors responsible for sequential execution performance growth started to abate in mid 2000s. First, the gains produced by instruction level parallelism plateaued, followed by a settling of the clock frequency at values between 1 and 4 GHz. This left the continuous increase in transistors per unit area as the only factor that had growth potential. What are the implications of these trends? In practice, a software vendor in the video game industry, or Computer Aided Design/Engineering, or database solutions cares about fast execution. In a first order approximation, proxies for fast sequential execution are high clock speed, high memory speeds, large memories, high instruction throughput. Note that there is no major effort required on the part of the software vendor to leverage improvements in any of these proxies. Once the sequential execution code is implemented, a doubling of the clock speed will reflect positively on the user experience when using the program. Not only that the software vendor was not required to change the code, but often times the source code didn't even require compiling and linking when running on an architecture with the same ISA. This was not true for certain instruction level parallelism aspects, which required a rebuilding of the code to capitalize on advances in compiler technology and chip designs.

Unlike improvements in clock speed, memory speeds, memory size/speed, and instruction level parallelism, improvements in the transistor density per unit area typically don't *immediately* reflect in faster program execution. A case could be made that an increase in the number of transistors per unit area leads to bigger fast memories, that is, bigger caches. This alone could not provide the momentum necessary to perpetuate the two fold increase in execution speed that software developers and consumers have come to expect every 18 to 24 months. However, what an increased number of transistors per unit area permits, is packing of more control units, arithmetic logic units, registers, etc. per die. Thus, a die ends up hosting not one processing unit, but rather several of them. They are called cores and combine to form one CPU. Over the last decade, the continuous increase in the number of cores per CPU has become virtually the only avenue that allows software vendors to continue to deliver programs with shorter execution time. Ideally, the compiler would be able to figure out how many cores on a chip, analyze the source code, and reorganize it to bring it to a form that is amenable to concurrent execution on multiple cores. Unfortunately, for the most widely used programming languages, such as C, Java, C++, etc., the compiler technology is not at the point where this is possible. Supporting concurrency in these languages requires source code modifications. At one end of the spectrum, the changes are small and target key parts of the source code to include compile directives that instruct the compiler that certain regions of the code can be executed concurrently. At the other end of the spectrum, leveraging concurrency in execution can require a full redesign and implementation of the code.

Does it make sense to act on this chip industry trend that is leading to more and more cores per chip? As illustrated in Figure 30, Moore's Law holds strong, and if anything, the plot indicates an increase in the pace at which transistors can be packed onto a die. Intel's roadmap predicts that the company is slated to produce dies with 14 nm feature size in 2014, 10 nm in 2016, 7 nm in 2018, and 5 nm 2020. In this respect, Moore's Law is projected to hold for at least one more decade. As such, for at least one more decade, a case can be made that concurrency will be the main avenue towards significant increases in execution speed since, as discussed next, the programs we develop nowadays are up against a memory wall, instruction level parallelism wall, and power wall.

2.4.1. The Memory Wall

The *memory wall* argument is built around the observation that memory access speed has not kept the pace with advances in the processing power of the chip. From 1986 to 2000, CPU speed improved at an annual rate of 55% while main memory access speed improved only at 10%. Here memory access speed is somewhat equivocally defined as a measure that combines the latency with the bandwidth of memory accesses. Figure 31 provides more quantitative details and indicates that the gap between memory speed and processor speed is increasing. This observation has practical implications. Since it is costly, programs should be designed so that they minimize data movement. While easier said than done, this piece of advice can be complemented by a practical observation: processing data is inexpensive, moving data back and forth is slow. This becomes apparent when inspecting the memory access speeds on a typical system. Figure 32 shows that a main memory access requires times of the orders of 100 ns. For a clock frequency of 2 GHz, this translates into roughly 200 clock cycles while in one clock cycle an AMD Interlagos chip can execute eight 32 bit fused multiply-add operations. This discrepancy between the speed at which, on one hand, data becomes available, and on the other hand, operations are carried out, led to the conclusion that there is no data crunching but only data movement, or equivalently, that number crunching is free.

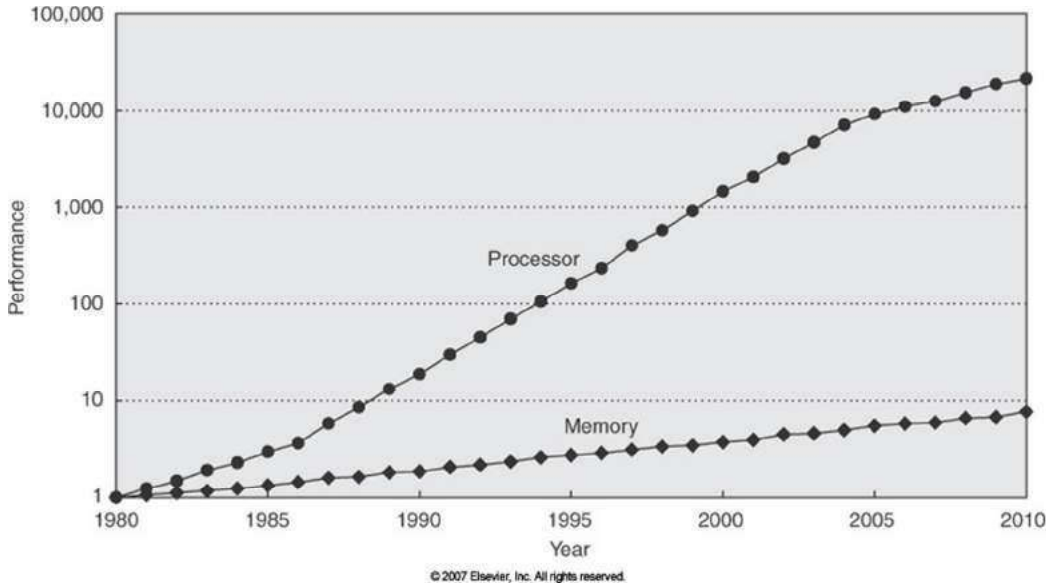


Figure 31. Processor speed versus memory speed; three decade evolution [8].

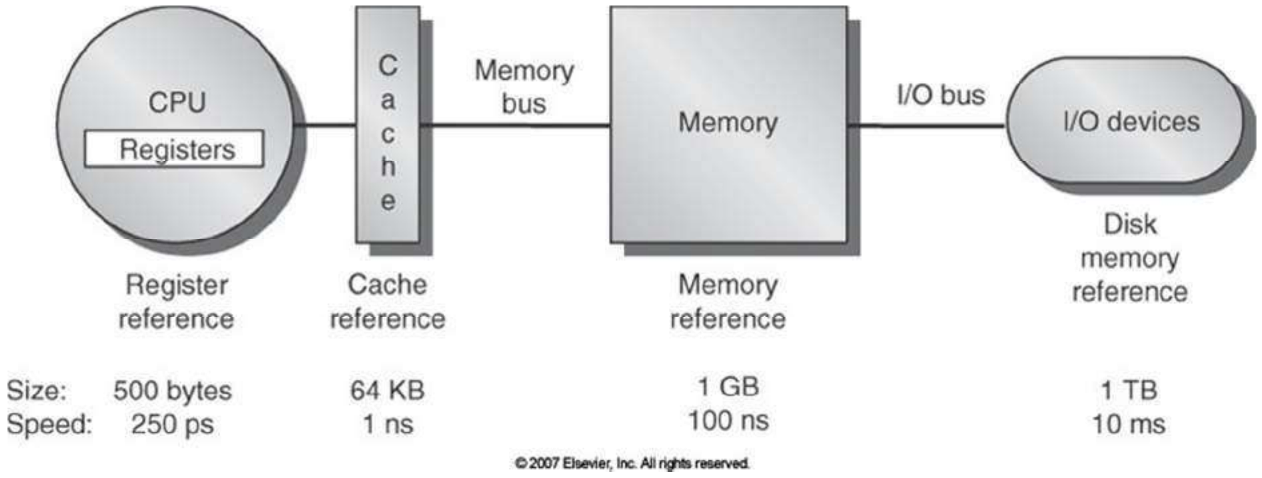


Figure 32. Memory bandwidths and latencies associated with various types of memories [8].

Promoting concurrent computing on multiple cores as the means to alleviate memory pressure is seemingly paradoxical. After all, if in the past the memory couldn't keep up with sequential execution on what today could be considered one core, how would it keep up with concurrent execution on multiple cores? To explain this, the argument starts with the observation that the memory speed is the interplay of two factors: latency and bandwidth. These concepts discussed in Section 1.4 can be summarized in the context of this discussion as follows: latency is essentially the amount of time it takes for a memory transaction the start being honored; once the transaction has started being honored, bandwidth is a measure of how much data per second flows down the conduit that connects the consumer of data and provider of data. Reducing latency is difficult; increasing bandwidth is relatively easy. This is confirmed by quantitative data plotted in Figure 33. Historically, the bandwidth was slightly easier to bump up by increasing the width of the bus over which data moves and/or increasing the frequency at which the data is moved. Recently, increasing bus widths has been gradually dropped in favor of having sometimes a serial link over which data is passed using a high clock frequency and possibly over several phases of the signal and/or wavelengths. Since the amount of time required to initiate a data transfer is relatively speaking large, to idea is to bring a lot of data if you bother to bring any. Nowadays, in a chip-main memory transaction it makes little difference if moving back and forth one integer or 64 integers and intuitively it makes common sense to squeeze 8 or 16 cores on a chip and have them process 64 integers stored in cache. It is interesting to reflect on how the bandwidth increased over the last fifteen years: in 1997 a single core Pentium II used to count on a 528 MB/s bandwidth, while today Opteron 6200 "Interlagos" has a bandwidth of about 52 GB/s and high end graphics card in 2013 delivers in excess of 300 GB/s when feeding thousands of lightweight scalar processors. What combining multi-core and high bandwidths accomplished was a hiding of the memory latency.

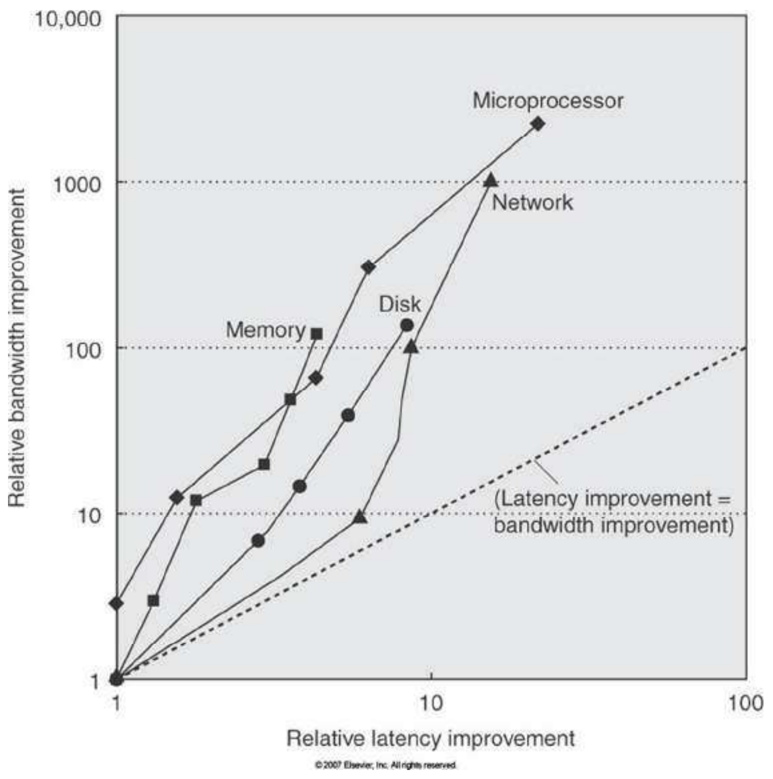


Figure 33. Relative bandwidth improvement vs. relative latency improvement over the last two decades. The dotted line in the figure marks the equal relative-improvement curve: for all types of data communication the bandwidth relative improvement was more prevalent than latency relative improvements [8].

2.4.2. The frequency/power wall

The discussion of the *frequency wall* or the *power wall*, starts with an observation made in 1974 by Robert Dennard and colleagues at IBM. They postulated that the transistors used in computers "continue to function as voltage-controlled switches – while all key figures of merit such as layout density, operating speed, and energy efficiency *improve*, provided geometric dimensions, voltages, and doping concentrations are consistently scaled to maintain the same electric field" [12]. This observation explains how technology advances and increased performance demand in the marketplace allowed for Moore's Law to hold for over four decades. Specifically, Dennard postulated that it makes sense to decrease both the typical feature size and voltage in a chip, since when these changes are corroborated with appropriate adjustments in doping concentrations the end product is superior. However, there are physical limits that challenge this quest for smaller feature lengths and operation voltages. The insulator that separates the gate from the rest of the MOS transistor becomes increasingly narrower as the feature length decreases. This translates into power leaks, which have been partially addressed recently by changing the material used for the dielectric: the silicon dioxide was replaced by high dielectric constant (high-k) materials. This provided a stop-gap solution: it temporarily alleviated the problem, but did not eliminate it as electrons are still leaking. In terms of voltage, the operating values are today of the order of 0.3 V. Aiming for lower values is challenging because controlling the transistors becomes very hard in a very electrically noisy environment. To tie this discussion back into the power wall, it should be pointed out that power dissipation on a chip changes as follows: it is increased by the leaks, it increases quadratically with the voltage and linearly with the clock frequency at which the transistors are operated. As long as the voltage could be reduced, it was a great time: a halving of the voltage in theory allowed for a quadrupling of the frequency. Unfortunately, nowadays the voltage can only be marginally decreased at a time when the feature length is decreased. If the frequency is not lowered, the amount of heat dissipated goes up. Because of the high density of the circuits in a die this can easily trigger a thermal runaway – a situation when an increase in temperature raises the resistance of the transistors and as such, even more power is dissipated, which further compounds the problem. The device either fails sooner or functions unreliably.

2.4.3. The Instruction Level Parallelism Wall

"Prediction is very difficult, especially if it's about the future." Niels Bohr, Mark Twain, Yogi Berra et al.

The *instruction level parallelism (ILP) wall* conjectures the inability of parallel execution at the chip level to yield further substantial efficiency gains in sequential programming. In other words, techniques that benefited sequential computing: instruction pipelining, superscalar execution, out-of-order execution, register renaming, branch prediction, and speculative execution – have fulfilled to a very large extent their potential. Moreover, future investments in these techniques will lead to marginal returns.

It's important to keep a historical perspective in mind when discussing the ILP wall. By early 2000s, the chips were capable of handling one stream of instructions, albeit some of these instructions could be executed in parallel in a superscalar fashion. Increasingly sophisticated designs introduced new tweaks that slightly increased the instruction horizon, or window, that fueled the speculative execution. Multiple pipelines were added to the chip just in case instructions were identified as being independent and they could be carried out simultaneously. Out of order execution logic was added to the chip, along with register renaming to further increase the chance of improving the IPC. Predicting the future is hard though, and the chance of getting predicting rightly decreases with the depth of the time horizon. The level of effort to get the right prediction is however not linearly linked to the depth of the time horizon. In fact, under a worst case scenario, the complexity grows exponentially because assessing all possible scenarios in the most general leads to a combinatorial problem. In other words, increasing the ability to predict for ILP called for substantially more complex designs and more power required to operate these units involved in deep horizon prediction. In the end, this process reached a point of diminishing returns in early 2000s when the return in investments was small enough to render ILP a mature technology likely to produce marginal gains as a result of tweaks of an otherwise dauntingly complex architecture.

There was a very compelling reason why the push for ILP was vigorous. Any increase in processor execution speed that came as a result of an ILP enhancement translated almost always in immediate benefit for the consumer. It came at no cost to the consumer; in many cases the consumer didn't even have to recompile a program, although doing so might have led to even more substantial gains when using smart compilers that were attuned to the new chips designs and knew how to capitalize on their ILP features.

2.4.4. Concluding Remarks

In early 2000s it became apparent that the momentum behind the sequential computing in the server and consumer market was waning. Memory access latencies could not be hidden behind one stream of instructions and the gap between data processing and data movement was getting wider; the frequency couldn't be increased any further without resorting to expensive solutions that the market was not willing to pay for; and the ILP had for most part exhausted its set of tricks. Moore's law remained one bright spot in the picture and provided a safe bet for continued performance improvement. Two of the leading chip manufacturers, Intel and IBM, had roadmaps in place that suggested that the feature length was bound to keep decreasing for close to two decades. The same message was coming out of pure wafer manufacturers, such as TSMC, which exclusively manufacture wafer and integrated circuits to assist chip designers such as AMD, Nvidia, Xilinx, with producing their chips.

Two observations explain the transition in mid-2000s to parallel computing as the new paradigm for program execution. First, the technology could deliver the hardware to sustain this paradigm shift. Second, by that time, the sequential computing model had reached its limit and the community was seeking solutions for faster computing. The outcome of this paradigm shift was twofold. From a hardware perspective, the chips designs embraced multicore layouts. From a software perspective, the software developers were forced to work with languages and libraries that supported these new hardware layouts. While the hardware shift has been fast paced, the software shift to using parallel computing languages and libraries to produce programs that leverage this hardware has been and continuous to be arduous.

2.5. Multi-Core Chips

Imagine that an architect/designer is provided a patch of land and a certain number of bricks, say 0.2 million of them. This architect wishes to build a small self-sufficient habitat and is given free rein over how the bricks are to be used. This is similar to a Lego game. The designer decides to build a factory, some storage area, a multiplex, etc. Two years later, the designer is presented another opportunity to build another habitat, yet this time around the designer can count on 0.4 billion bricks to be used on a similar patch of land. The designer learned what works and what doesn't based on the previous design. The new design will likely be more functional both because of the lesson learned but also because more bricks are available and additional support units can be built to fulfill functions only dreamed of in the previous habitat. This doubling of the number of bricks and wising up of the designer continues. Note that the increase in the number of bricks does not come at the cost of any increase in the size of the patch of land on which the habitat/habitats are supposed to be built on. It is just that the bricks are smaller and the design starts looking more packed. It embodies a lot of knowledge but also subjective choices that try to anticipate the requirements of the people that will use the habitat.

To continue this metaphor, at some point it does not make sense to have only one habitat on the typical patch of land. The complexity of the habitat would be too high, it cannot be supplied with the raw material required to operate well, there are power distribution issues as the voltage in the habitat is so high that it interferes with the ability of different units (the multiplex, or the factory) to operate properly etc. Moreover, complexity is not an end goal in itself; while in the beginning it enabled a harmonious and efficient operation of the habitat, at some point the return on designing more complex habitats is marginal in terms of benefits/outcomes. At some point, given the doubling of the number of bricks, it becomes clear that to put these bricks to good use one will have to go back to basics and build on the same patch of land two or more habitats that might share some resources but otherwise would work as independent entities. The habitats continue to improve incrementally, but given the number of bricks available the designers now focus on contriving entire cities. On the same patch of land there is a high level of functional and architectural integration and some coordination will be required for the habitats to function harmoniously as a city. For instance, to keep the city traffic going, they should coordinate who sends output to where and when.

Fast forward to 2013, and imagine a patch of land and 1.6 billion bricks. A situation like this is what the Intel designers were faced with in 2011. Specifically, they have a die of a certain area (about 0.5 inch by 0.5 inch) and 1.6 billion transistors. How should they organize these bricks? Having an entire city (chip) include several habitats (cores) and operate on a small patch of land (die) supported by all the necessary functional units represents what in the industry is called System on Chip (SoC) design. Note that a similar situation is encountered on an Nvidia card. On a slightly bigger die the designers could toy with seven billion transistors. The next two subsections discuss how the chip designers of Intel and Nvidia decided to build their chips, or cities.

2.5.1. The Intel Haswell Architecture

Chip Architecture Overview

The Intel Haswell microarchitecture was released in June 2013. It is a 22 nm design that draws on approximately 1.4 billion 3D tri-gate transistors (see **Sidebar 1**). It typically has four cores each with private L1 and L2 cache; the L3 cache is shared and is the last level cache (LLC). It has an execution pipeline that is 14 to 19 stages long, depending on whether some instruction decoding stages can be eliminated upon a micro-instruction cache hit. It is a complex design aimed at increasing the average IPC of sequential programs through ILP. Each of its cores is five-wide superscalar, out of order, supports speculative execution and simultaneous multithreading.

Haswell continues Intel's SoC push to cram as much functionality on the chip as possible. The most manifest illustration of the trend is the presence of an integrated graphics card on the chip, see Figure 34. Moreover, many of the services provided by a motherboard's northbridge are now hosted on the chip for improved speed and power management. For instance, the voltage regulator has been migrated onto the chip to improve power management by enabling faster switching between various operational states such as idle/peak as discussed shortly. The display engine is managed on the chip, which seems natural given the proximity of the graphics processor. Likewise, there is an integrated memory controller (IMC) for managing transactions with the main memory. As further proof that the chip includes most of the functionality of the northbridge, a direct media interface (DMI), which typically connects the northbridge and the southbridge on a motherboard, is now hosted on the chip. So is the PCI-Express controller, which essentially is similar in implementation and purpose to the DMI in that it controls multiple lanes to form a point-to-point link. The voltage regulator, display engine, DMI, PCI controller, and IMC combine to form the "System Agent". The System Agent is connected to a ring bus and thus has a low latency high bandwidth connection to the rest of the chip components: cores, LLC, graphics processor, and eDRAM. The latter is a chunk of 128 MB of off-die memory connected to the chip through a 512 bit bus. Compared to the DDR3 system memory, it provides lower latency and higher bandwidth for faster rendering in the graphics processor.

The die has several clocks running on it, the most important ones being associated with (a) the cores ticking at 2.7 to 3.0 GHz but adjustable up to 3.7-3.9 GHz; (b) the graphics processor ticking at 400 MHz but adjustable up to 1.3 GHz; and (c) the ring bus and the shared L3 cache with a frequency that is close to but not necessarily identical to that of the cores. These frequencies are controlled by the voltage regulator. For instance, when the graphics processor runs at peak, if the cores idle their frequency is reduced while the bus ring can still run at high frequency to support the graphics processor. In fact, a core can be completely turned off (power gated off) if idle.

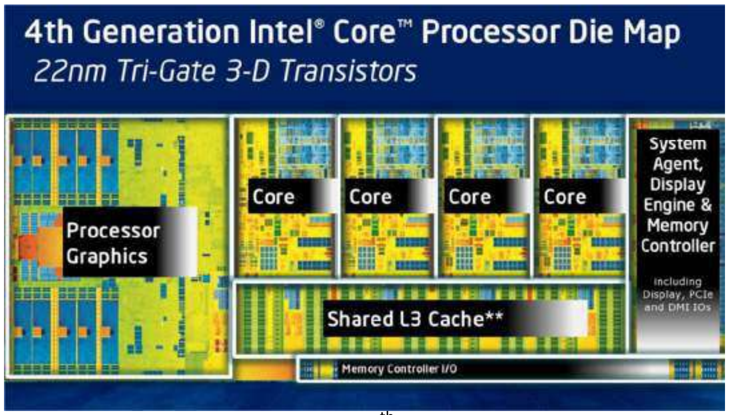
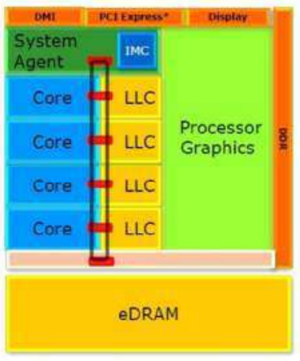

4th Generation Intel® Core™ Processor Die Map
22nm Tri-Gate 3-D Transistors
Processor Graphics
Core
Core
Core
Core
System Agent, Display Engine & Memory Controller
including Display, PCIe and DMI IOs
Shared L3 Cache**
Memory Controller I/O


Figure 34. Left: Actual die layout for 4th generation Intel Core processor using 22 nm tri-gate 3D transistors. Right: Schematic of the layout of the major units on the Intel Haswell die [13].

Caches

There are three levels of caches in Haswell. There is a 32KB L1 instruction cache and a 32 KB L1 data cache. Both L1 caches are 8-way associative and the fastest load-to-use operation takes 4 clock cycles. The load bandwidth is 64 Bytes/cycle; in other words, after the initial latency of four cycles, two integers or one double precision value can be fetched from the L1 data cache each cycle. The store bandwidth is 32 Bytes/cycle. Finally, the L2 cache, at sizes of 512 KB or 1024 KB, is also core specific and has higher latency; i.e., 11 cycles, with a bandwidth between L1 and L2 of 64 Bytes/cycle. The typical L3 cache size is 8 MB and it is shared by the cores, the graphics processor, and the System Agent (see Figure 34). There is an interconnect; i.e., a wide ring bus, that links all these components with the L3 cache. Its bandwidth is 32 bytes/cycle for every stop.

The Front End

The front end of a chip is the set of functional units in charge of priming the execution units for performing the operations encoded in the machine instructions. Its output becomes the input for the execution engine, which sometimes is called the back end of the chip. The front end plays an important role in supporting the instruction level parallelism associated with the chip and its ultimate goal is to never keep the back end idle. The Haswell front end fetches up to four instructions at a time. It decodes them, performs branch prediction, seeks opportunities for speculative execution and initiates memory prefetching in anticipation of future memory transaction requests.

Recall that Intel is a CISC (complex instruction set architecture). As such, the instruction size is not fixed. Consequently, when instructions are fetched their decoding is not straightforward, at least not when compared with a RISC ISA. On Haswell there is a small amount of memory sometimes called LØ cache that stores the decoded instructions in the form of sets of fixed-length micro-operations (uops). The net effect of one <u>set</u> of these uops is one CISC instruction. Having this LØ uops cache saves both time and power since decoding variable length complex instructions, some of them with many operands, is nontrivial. For instance, consider a program that has a loop in it, see Code Snippet 10. In this example, the "`for`" loop gets executed 1000 times. Consequently, there is a collection of instructions associated with the loop that get executed time and again. The idea is to store uops associated with these instructions in fast memory and have them ready to go when the execution loops over them.

```
const int NMAX=1000;
const double M_PI=3.1415926535898;
const double incrm = M_PI/NMAX;
double usefulData[NMAX];
for (int i=0; i<NMAX; i++) {
   double dummy = i*incrm;
   double temp = sin(dummy);
   temp *= temp;
   //..several lines of code
   //..further modify
   //..the value of "temp"
   usefulData[i] = temp;
}
```

Code Snippet 10. An example where caching uops can save time by not necessitating a repeated use of the instruction decoding.

The front end includes logic units that understand when the loop is being executed. Upon looping, the branch prediction as well as the fetch/decode units are shut down since they are superfluous. The execution proceeds by using uops stored in the LØ cache, which (i) improves performance, due to the proximity of the LØ cache and the skipping of the decoding; and (ii) saves power, since the branch prediction and decoding units are shut down. The size of the uop cache is 1,500 uops, each four byte long, which can be regarded as a 6KB LØ instruction cache that adds to the typical L1 instruction cache. The Intel's x86 ISA being a CISC, not having to decode instructions can be a significant factor. This saving is less impactful for RISC architecture whose instructions are equal length and more straightforward to decode. Indeed, the x86 ISA has instructions that range in size from 1 to 15 bytes; the position of the operands is not fixed; the instruction prefix is also of variable length. Given a sequence of bytes, understanding where an instruction starts and ends, what the operands are, what the instruction calls for is thus no simple thing. To make things even more challenging, there are several instructions in the current x86 ISA that require microcode for execution; i.e., they lead to the invocation of a sequence of steps that are technically implemented in software to take advantage of the specific circuitry of the chip. In other words, microcode is used whenever an instruction leads to more than four uops by essentially calling a subroutine or making a function call that is very low level and is not directly accessible to the programmer.

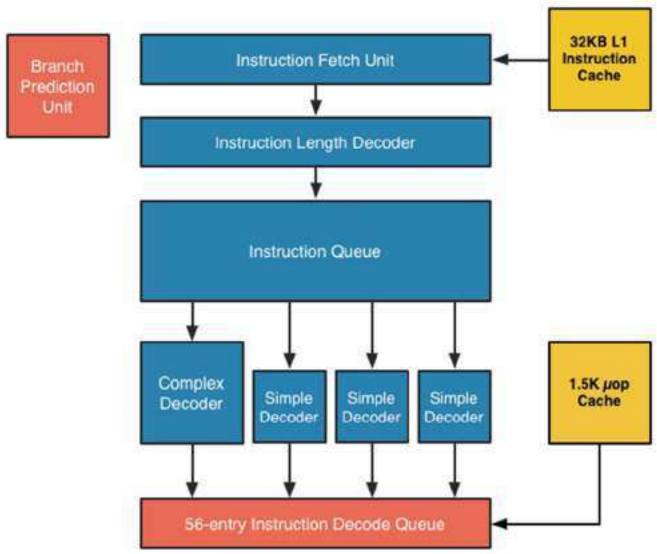


Figure 35. Schematic of Haswell's front end [13].

Once its bounds and operands are figured out, each instruction is sent to an "Instruction Queue", see Figure 35. When its turn comes, each instruction is decoded into uops. Most of the machine instructions decode to one uop. However, some lead to multiple uops. For the latter, there is a special "Complex Decoder", which produces up to four uops per clock. In general, the decoding happens in parallel, with three simple decoders and a complex one to produce no more than four uops per clock: if the complex decoder issues four uops, the three simple decoders idle.

This entire decoding process can be skipped when an instruction hits the LØ cache thus reducing the pipeline of Haswell from 16 to 14 stages. Whenever an instruction is identified as being cached all the uops that would otherwise associate with this instruction are directly picked up from the uop cache. Each uop cache line contains up to six uops, and each cache block contains eight cache lines making the cache eight-way associative. There are 32 blocks for a total of 32X8X6=1536 uops.

As uops are churned out, they are stored in an "Instruction Decode Queue", which acts as a buffer that can hold up to 56 entries. If the core processes only one instruction stream, all 56 uops are associated with the execution of this stream. Since Haswell supports hyper-threading, two instruction streams might be processed by the core, in which case the 56 entries are shared in a 28/28 fashion by the two streams.

The Back End

As illustrated in Figure 36, the Instruction Decode Queue feeds a large Reorder Buffer (ROB), which sometimes is called the "out-of-order window". It can hold on to 192 uops representing the maximum number of uops that can be in flight at any given time. The purpose of ROB is to re-order the uops for improved use of the execution assets. This reordering draws on Tomasulo's algorithm [14], which relies on register renaming for facilitating out of order execution. The ROB can be regarded as a large structure that keeps track of all the uops that are at various stages of the execution: some have not started yet, some are buffered waiting for their turn, some uops are about to be retired, etc.

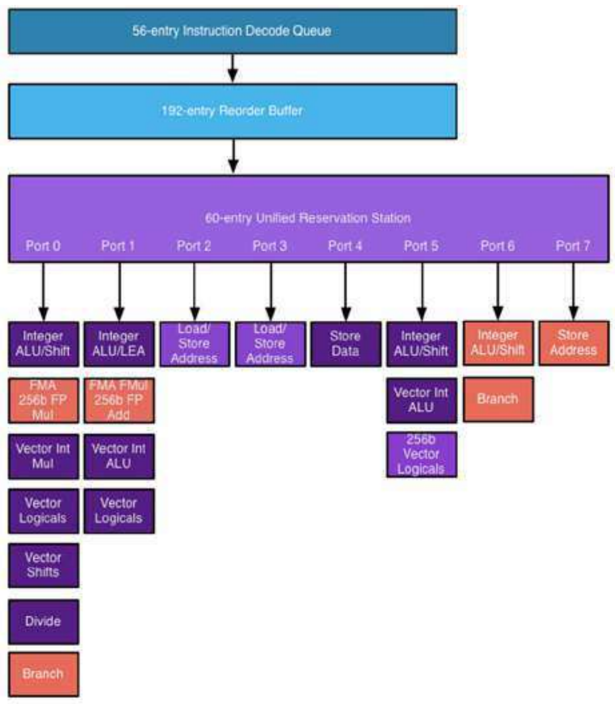


Figure 36. Schematic of Haswell's back end (the execution engine) [13].

ROB is keeping tabs on uops since, for instance, upon a failed branch prediction the execution should be unrolled to a valid state from which the execution can commence.

Before a uop can be issued for execution, it needs to find its way into a 60-entry "Unified Reservation Station". The Unified Reservation Station acts as a scheduler that matches uops to execution units: any uop that has its operands ready is dispatched to an execution unit through one of eight ports (Port 0 through Port 7) depending on the job order coded in the uop. Note that there are more execution units than ports. As each execution unit is associated with a stack, the ports should be regarded as gates to these stacks. Ports 2 and 3, for instance, have it easy: they each serve only one execution unit. At the other end of the spectrum, Port 0 services several execution units that take care of integer and floating point (FP) arithmetic. Roughly speaking, of the eight ports, four are dedicated to servicing memory operations: Ports 2, 3, 4, 7. The rest are on math duty, be it for integer or FP operations.

Table 10 summarizes several key parameters that shape the ILP prowess of a core for several Intel architecture designs. The allocation queue for Haswell is 56 uops long. For the previous two designs, Sandy Bridge and before that Nahelm, the queue was organized to hold 28 entries per thread. Recall that these cores support the simultaneous execution of two threads owing to their hyper-threaded design. The Haswell has the ability to use all of the 56 entries to service one thread, which is convenient if only one thread gets executed; for Sandy Bridge and Nahelm 28 entries would stay unused. The size of the reorder buffer is 192. These many uops are in flight at each given time. Some uops are associated with loads and as such will find their way in a buffer of size 72. Other uops are associated with stores and will end up in a buffer of size 42. Producing an output requires more often more than one input, and this is reflected in the number of in-flight loads and in-flight stores that can be handled by the design. A number of registers, for integers and floating point values, are used for register renaming that sustains out-of-order execution. Finally, the size of the Unified Reservation Station, which is a staging area before a uop is picked by execution, is 60. The word "unified" suggests that both memory requests and arithmetic operations are stored here before issue. The Unified Reservation Station plays the role of a scheduler: a set of the oldest non-conflicting uops, up to eight of them, are sent through the dispatch ports to the appropriate execution units. Note that it is not always the case that during a cycle all of these eight execution units will see action but rather this represents a best case scenario. Finally, the size of the register file is 168 for both integer and floating point values. These registers hold the actual input and output operands for the uops.

Intel Design Generation	Conroe (Core 2 Duo)	Nehalem	Sandy Bridge	Haswell
Allocation Queue	7	28/thread	28/thread	56
Out-of-order Window (Reorder Buffer)	96	128	168	192
In-flight Loads	32	48	64	72
In-flight Stores	20	32	36	42
Integer Register File	N/A	N/A	160	168
FP Register File	N/A	N/A	144	168
Scheduler Entries (Unified Reservation Station)	32	36	54	60

Table 10. Key attributes that determine ILP prowess for several Intel design generations.

Putting It All Together

Figure 37 contains a simplified schematic of the Haswell microarchitecture. It provides an opportunity to contemplate the entire layout from a different perspective. Thus, what was above called the "front end" can be regarded as the sum of two parts: the instruction pre-fetch and the instruction decode. Likewise, what was called the "back-end" is the sum of two components that serve different purposes: the instruction level parallelism stage, and the execution engine. Different colors are used to mark the four functionally different components of the microarchitecture.

Figure 37 also contains information about bandwidths. For instance, 32 bytes of instructions can be fetched each cycle upon a L1 instruction cache hit. How many instructions these 32 bytes represent cannot be said since the x86 ISA is a CISC: it handles different length instructions. The decoder can issue six instructions per cycle, which can actually belong to up to two execution threads given the hyper-threaded nature of the Haswell core. The machine instructions are decoded at a rate of up to four uops/cycle, which represents also the peak rate at which the ROB can be supplied with uops. Four uops can be subsequently issued by the scheduler. Given that one uop can be fused and actually contain information about a load/store and some other arithmetic operation, the four fused uops can be dispatched in a cycle to eight ports as eight basic uops. Three other relevant bandwidths are as follows: L1 stores have a bandwidth of 32 bytes/cycle, L1 load requests can honored at two 32 bytes each cycle, while L1 to/from L2 transactions have 64 bytes per cycle bandwidth.

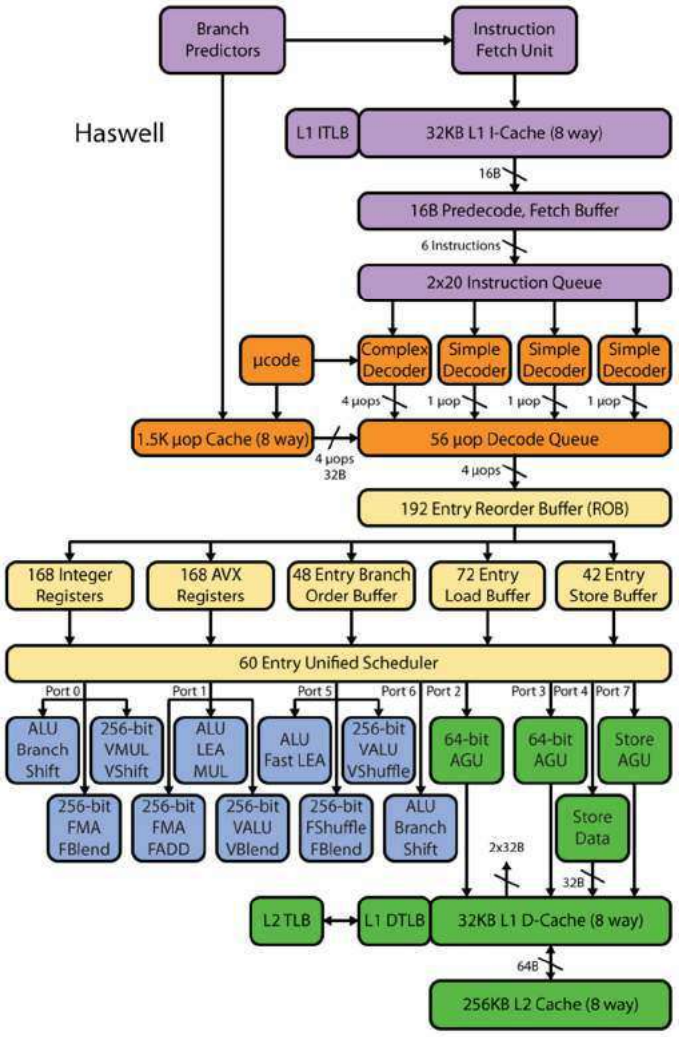


Figure 37. Overview of the Haswell microarchitecture [15]. Note the dedicated Address Generation Unit (AGU), which is needed in X86 to generate addresses for memory access operations. This is because instruction operands can specify memory locations.

The Haswell core is out of order and dual-threaded. It supports a set of uops that handle vectors in arithmetic operations: rather than adding the first entry in array **a** to the first entry in array **b**, followed by adding the second entry in array **a** to the second entry in an array **b**, etc., it can leverage an AVX (Advanced Vector Extension) mode in which each core can deliver 16 double precision (DP) operations in one cycle for a theoretical peak rate of 160 GFlop/second. This number is obtained by multiplying 16 DP operations times the number of cores (4) times a base clock frequency of 2.5 GHz (this can be slightly lower or higher, depending on the processor model). In the AVX mode four pairs of integers of double precision values are processed simultaneously.

Sidebar 13.
Since 2007, the new Intel chip designs are released on a tick-tock schedule with the tick happening in even years and tock releases happening in odd years. A release in which the design sports a smaller design feature; i.e., a shrinking of the design process technology, is a tick. For "tock" releases the focus is on providing updates to the microarchitecture using the existing feature technology. Released in 2013, Haswell is thus a tock design that changed the microarchitecture in at least three major ways: (i) the AVX technology was extended beyond floating point (FP) to include integer operations; (ii) FP operations got two times faster for cases where an addition was done in relation to a multiplications by supporting fused multiply-add (FMA) operations; (iii) Intel introduce support for transactional memory, which comes into play in parallel computing when multiple threads are modifying the cache and special measures need to be in place to avoid read/write hazards (one thread reading a value right when a different thread is modifying that value). The issue of cache coherency is discussed later.

2.5.2. The Nvidia Fermi Architecture

Fermi is a co-processor that assists a CPU with performing compute intense segments of an application. A schematic of its architecture is shown in Figure 38. It represents one of the best illustrations of Moore's law at work: a lot of transistors could be etched per unit area, and this large number, 3 billion for Fermi, has been organized in a hardware architecture that has 512 Scalar Processors, grouped into a set of 16 Streaming Multiprocessors (SM). The former are shown in green; the latter can be noticed as repeating patterns, eight above and eight under, around the L2 cache block. One SM is shown schematically in Figure 44.

Sidebar 14.
Nvidia is currently the most successful manufacturer of video cards that are also used for scientific applications. There are several families of graphics cards offered by Nvidia, but for the purpose of this discussion, one can regard these cards as being single or dual use. The cards of the Tesla line of products are pure co-processors; in fact they don't even have an output to a monitor, screen, etc. These cards are exclusively dedicated to scientific computing. The dual use cards are the core product of Nvidia and are aimed at the graphics enthusiasts but at the same time can perform, albeit not at the level of a Tesla card, scientific computing by relying on a software infrastructure called CUDA (Compute Unified Device Architecture). CUDA is also used by Tesla cards, and it was first released in 2007. It provides the software ecosystem and defines the programming model that enables users to tap into the resources available on Nvidia cards.

When discussing the architecture of Fermi, one should keep in mind the type of computing this device is meant to carry out. A GPGPU is designed first and foremost to support the Single Instruction Multiple Data (SIMD) computing paradigm in which a collection of threads execute a single instruction over a set of data. A trivial example is that of multiplying the entries in two arrays that each has three million entries. A single instruction, in this case "multiply", is issued and applied to a set of three million pairs of numbers: `c[i] = a[i]*b[i]`, for $0 \leq i < 3{,}000{,}000$.

A step up in terms of complexity is the task of multiplying two large matrices, say of dimension 2000 by 1000 and 1000 by 3000. The result will be a matrix of dimension 2000 by 3000 in which entry in row `i` and column `j` is obtained as the dot product of row `i` from the first matrix and column `j` from the second matrix. One will then launch a set of six million threads to compute the six million entries in the result matrix. This matrix multiplication exercise can also be viewed as a SIMD process since the same sequence of instructions are executed for different pairs (`i,j`) thus accessing different rows `i` and columns `j`. At the end of the sequence of instructions, each of the six million parallel threads will have computed one entry in the product matrix.

Architecture Overview and Execution Model

From a high vantage point, the Fermi architecture is simple. It is the sum of

- A host interface that facilitates the CPU-GPGPU data exchange.
- A block of L2 cache and six banks of main memory, both shared by all 16 SMs
- A high level scheduler unit, and
- 16 Streaming Multiprocessors, each with its L1 cache, register file, thread scheduler, and set of execution units

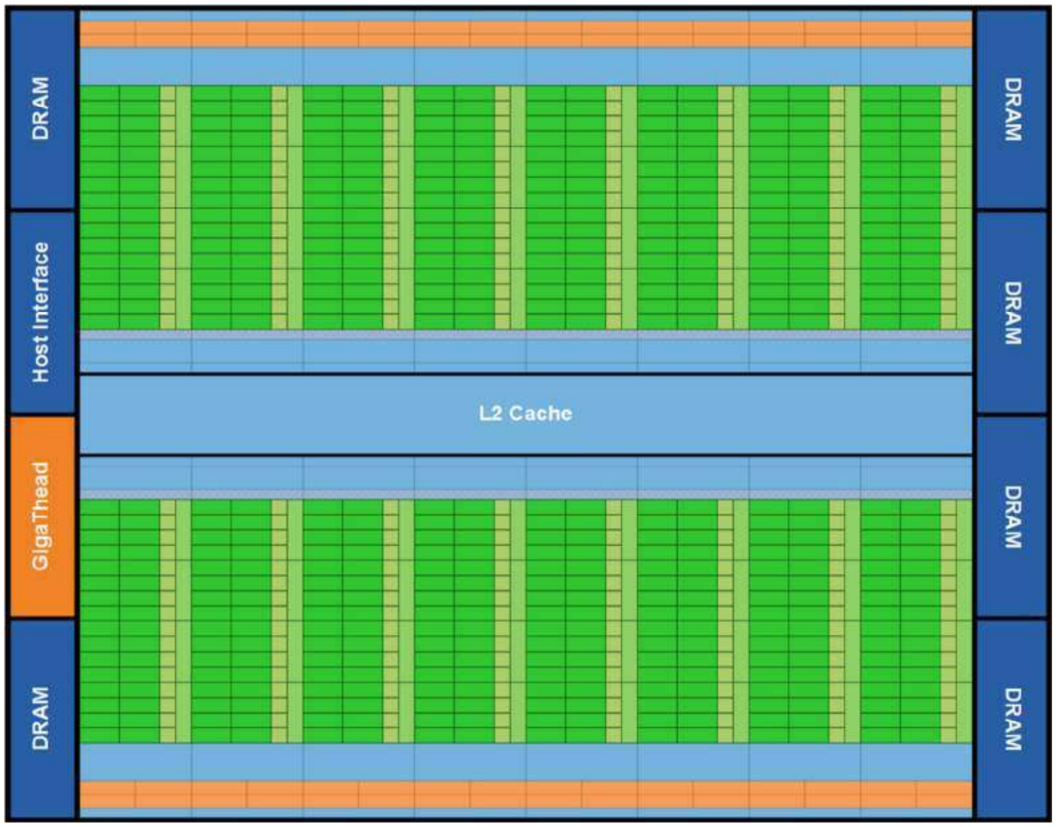


Figure 38. A schematic of the Fermi architecture. Color code: orange - scheduling and dispatch; green - execution; light blue – registers and caches. The co-processor is made up of a collection of 16 Stream Multiprocessors (SMs). Details of an SM are provided in Figure 44. Credit: Nvidia.

To discuss the execution model, recall the two array multiplication example introduced above. Assume that the two input arrays are on the CPU, which is called the host to differentiate it from the GPGPU, which is called the device. The Host Interface will manage the connection between the host and device. Physically, the host-device back-and-forth communication takes place over a PCI Express v2 bus with a theoretical peak rate of 8 GB/s simultaneously in each direction. In our example, it facilitates the copying of the two arrays of values from the host to the device. Once the data is on the device in global memory, a function can be invoked to sum up the three million pairs of numbers. In CUDA parlance, a function like this, which is called/invoked on the host but executed on the device, is called a kernel. For this example, the kernel is simple: probably in less than five lines of code it reads two values from global memory, multiplies them and writes the result back to a third array `c`. This kernel is executed three million times by three million threads. Each of the three million threads has a "tunnel vision" in that it only operates on one pair of entries based on its thread index. Oversimplifying things, thread 0 executes the kernel to multiply the first pair of values, thread 1 executes the kernel and based on its index understands that it needs to multiply the second pair of values, etc., all the way to the last thread, which when executing the kernel, based on its index, ends up loading the last pair of values and multiplies them.

It becomes apparent that in this model three million threads cannot be simultaneously executed by 16 SMs that collectively have 512 functional units that can perform multiplications. Instead, the device will place these threads in a queue and will grab as many threads as it can possibly do, say N, and execute the kernel N number of times through the N threads; the rest of the threads will have to wait. Each Fermi SM can handle up to 1,536 threads; since there are 16 SMs, the total number of threads that can be active on the device is 24,576. The device is said to have finished the execution of the kernel once all three million threads get their turn to finish executing the kernel.

Given the limits of the device, the scheduler must organize the threads in blocks of threads. It is such a block that gets scheduled for execution on an SM. How many blocks of threads an SM can handle at any given time is something that the user cannot control. It is anywhere between one and eight blocks, depending on how demanding these blocks are in terms of resources required for execution. What the user can control though, is the number of threads that each block will contain. For instance, if the multiplication kernel needs to be executed by 3,000,000 threads to multiply the 3,000,000 pairs of values in arrays `a` and `b`, then the user might decide to launch blocks of 256 threads, in which case a number of 11,719 blocks would be needed. If instead each block is decided to have 1024 threads, a number of 2,930 blocks would be needed to get the multiplication job done. Why not launch then one block of 3,000,000 threads and get over with it? CUDA on Fermi doesn't allow one to have more than 1024 threads per block. For reasons that will be explained later, most often the number of threads is chosen to be a power of 2. If one chooses to have a large number of threads in a block, there is an increased chance that the scheduler will have to send a smaller number of blocks to an SM. The scheduler decides on the number of blocks allocated to an SM based on the amount of resources required for the execution of the block. For instance, if each thread requires a large number of registers and the block contains a large number of threads, the scheduler might only be able to send one or two blocks to an SM. In the extreme case, the scheduler will be in a situation where the amount of resources required for execution by one block are so excessive that a kernel cannot be launched for on the device since the SM cannot accommodate the kernel's execution.

Code Snippet 11 lists the code that makes up the kernel that multiplies a pair of values stored on the device in two arrays `a` and `b` and subsequently writes the results to an array `c`. Assume that 512 threads have been selected to make up a block, which means that 5,860 blocks will be launched. Line 3 of the code snippet shows how each of the 3,000,320 that eventually get to execute the kernel computes the entry in the c array that it is supposed to compute on. The `if` statement on Line 4 ensures that we are not going to read/write beyond the bounds of the arrays.

```
1.  __global__ void multiply_ab(int* a, int* b, int* c, int size)
2.  {
3.     int whichEntry = threadIdx.x + blockIdx.x*blockDim.x;
4.     if( whichEntry<size )
5.        c[whichEntry] = a[whichEntry]*b[whichEntry];
6.  }
```

Code Snippet 11. Kernel to compute the pairwise product of two arrays a and b, with the result stored in c. Arrays are of dimension `size`.

The interplay between the host and device; i.e., between CPU and GPU execution, can be summarized in four steps.

Step 1: Allocate memory space on the device to hold data
Step 2: Copy data from host memory to device memory
Step 3: Set up an execution configuration
Step 4: Launch kernel execution on the device to process data
Step 5: Copy back data from the device to the host
Step 6: Free memory allocated on the device

The main function follows this six-step process, see Code Snippet 12. A preamble takes care of housekeeping associated with this example. It sets aside memory on the device and host and then initializes the values on the host to random integers between 0 and 8. Step 1 actually happens in the function call `setupDevice`, see Code Snippet 13. Note that pointers that start with an "h" prefix point to data on the host; a prefix "d" is used in relation to data stored on the device. The second step of the above six-step process is implemented at Lines 10 and 11, which take care of copying the content of arrays a and b, respectively, from host to the device. Lines 13 and 14 take care of Step 3: a number of 512 threads will be used per block, which means that 5,860 blocks need to be launched. Line 15 takes care of Step 4, while for Step 5 the results are copied back in Line 16. Additional housekeeping frees memory, Step 6, once it checks that the results are correct. The code for this example available in **`$CODEROOT/CUDArelated/multiplyAB.cu`**.

```
1.  int main(int argc, char* argv[])
2.  {
3.      const int arraySize = 3000000;
4.      int *hA, *hB, *hC;
5.      setupHost(&hA, &hB, &hC, arraySize);
6.
7.      int *dA, *dB, *dC;
8.      setupDevice(&dA, &dB, &dC, arraySize);
9.
10.     cudaMemcpy(dA, hA, sizeof(int) * arraySize, cudaMemcpyHostToDevice);
11.     cudaMemcpy(dB, hB, sizeof(int) * arraySize, cudaMemcpyHostToDevice);
12.
13.     const int threadsPerBlock = 512;
14.     const int blockSizeMultiplication = arraySize/threadsPerBlock + 1;
15.     multiply_ab<<<blockSizeMultiplication,threadsPerBlock>>>(dA,dB,dC,arraySize);
```

```
    cudaMemcpy(hC, dC, sizeof(int) * arraySize, cudaMemcpyDeviceToHost);

    if( resultsAreOK(hA, hB, hC, arraySize) )
    printf("Results are OK.\n");
    else
        printf("Results are not OK.\n");

    cleanupHost(hA, hB, hC);
    cleanupDevice(dA, dB, dC);
    return 0;
}
```

Code Snippet 12. The main function for the program that multiplies pairwise two arrays of integers.

```
void setupDevice(int** pdA, int** pdB, int** pdC, int arraySize)
{
    cudaMalloc((void**) pdA, sizeof(int) * arraySize);
    cudaMalloc((void**) pdB, sizeof(int) * arraySize);
    cudaMalloc((void**) pdC, sizeof(int) * arraySize);
}

void cleanupDevice(int *dA, int *dB, int *dC)
{
    cudaFree(dA);
    cudaFree(dB);
    cudaFree(dC);
}
```

Code Snippet 13. Two support functions implement Step 1 and Step 6: `setupDevice` and `cleanupDevice`, respectively.

Note that no synchronization can take place on Fermi among all the threads launched in conjunction with a kernel. In this context, a synchronization request represents a barrier in the execution flow where the threads must pause the execution; i.e., take a break, until all threads executing the kernel reach this mark; i.e., all threads catch up. At that moment in time the threads can resume their work. Going back to the array multiplication example, on Fermi one cannot have all three million threads synchronize at a certain point in their execution of the kernel (not that there would be any good reason for requesting a synchronization in this simple example). This goes back to the observation that the resources on an SM are limited and only 1536 threads can be handled at most at any given time. Consequently, some threads finish executing the kernel long before the last batch of threads actually starts their execution.

If a global synchronization is needed in CUDA, this is enforced in a brute force fashion by breaking a computation in two parts. A first kernel implements the first part of the process while the second part is implemented by a different kernel. Exiting a kernel necessarily means that all threads in the execution configuration got their turn to execute the kernel. Note that synchronization of certain small subsets of threads is nonetheless possible.

Sidebar 15.
Imagine that in the two array multiplication example, one would have to sum up all the entries in the three million entry array `c`. In fact, this implements the dot product of the arrays `a` and `b`. A global synchronization point is needed, in that the final summation cannot be carried out before all threads got their chance to take care of multiplying the corresponding pair of `a` and `b` entries. Once the array `c` is available, one can proceed to sum up all the entries in `c`. This process is called a reduction operation

carried out in conjunction with `c`. Performing this reduction has a very good chance of yielding a wrong result should the reduction start before all threads got their chance to complete the multiplication part. This situation is avoided by launching two kernels. Each thread executing the first kernel populates an entry `c[i]` of the `c` array. A follow up kernel is launched to perform, in parallel, the reduction step.

The Memory Ecosystem

Fermi has a deep memory ecosystem with a big register file (32,768 four byte words per SM), a small amount of shared memory (scratch pad memory) and a little bit of L1 cache that add up to 48 KB per SM, L2 cache (768 KB) shared by all SMs, and global memory (up to 6 GB). There is also texture memory and a small amount of what Nvidia calls constant memory. The former is a living proof of the graphics origin of the Fermi architecture and will not be discussed. The latter can only be used for read purposes: values are copied by the host in this memory and subsequently any thread executing on the device can read but not write values from there.

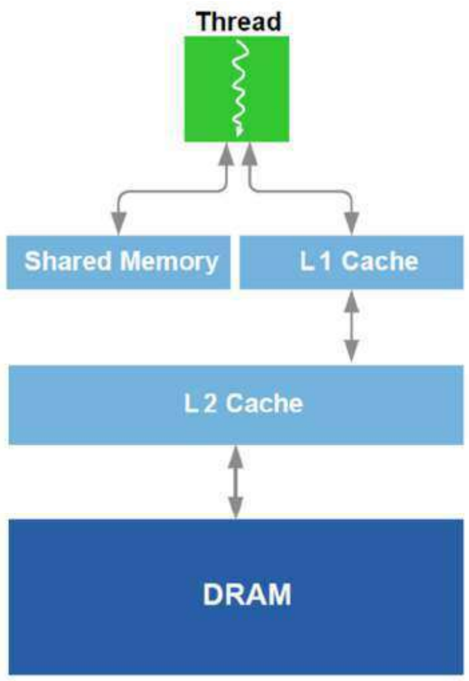


Figure 39. Memory hierarchy on Fermi, as perceived by a thread running on an SM. Credit: Nvidia.

As illustrated in Figure 39, a thread has direct access to the shared memory and it has assigned to it a certain number of registers. These are not exactly the hardware registers that come up when discussing register forwarding or register renaming, etc., in the context of CPU computing. One should think of them as a small amount of very fast memory that a thread accesses privately; i.e., no other thread can access data stored in the registers associated with a different thread. A transaction with the global memory first checks if the data is in L1 cache. If not, it checks against the L2 cache. Upon a L2 cache miss, the data is brought over from global memory. Although the memory hierarchy has several layers, Fermi has a unique overarching address space that spans the thread-local memory (register), block-specific memory (shared and L1), and grid-specific memory (global).

The latency of a memory access is correlated with the level targeted by the transaction. A register access has the lowest latency; accesses to global memory upon a cache miss incur the largest latency. The Fermi family encompasses a spectrum of cards, but orders of magnitude are as follows: a register access is of the order of one cycle ($O(0)$), shared/L1 less than 10 cycles ($O(1)$), while a global memory access has latencies of the order ($O(2)$): 400-800 cycles. In terms of bandwidths, register bandwidth is approximately 8000 GB/s, shared memory and L1 about 1000 GB/s, and global memory bandwidth, depending on the model, is in the vicinity of 150 GB/s. To arrive at the last value, one can use the memory clock rate, 1.674 GHz, and the memory bus width, 320 lanes: 1.674 GHZ times 2 (dual rate technology) times 320 divided by eight (number of bits per byte) yields a bandwidth of 133.92 GB/s, which is the peak bandwidth of the GeForce GTX480 member of the Fermi family. For comparison purposes, the PCI Express version 2 bus that connects a Fermi card to the host has a theoretical bandwidth of 8 GB/s, with the caveat that Fermi is capable of simultaneously controlling the data transfer to/from the host.

As illustrated in Figure 40, there are six memory controllers that talk to the global GDDR5 (Graphics Double Data Rate version 5) memory, which is a fast and more pricy SDRAM graphics card memory.

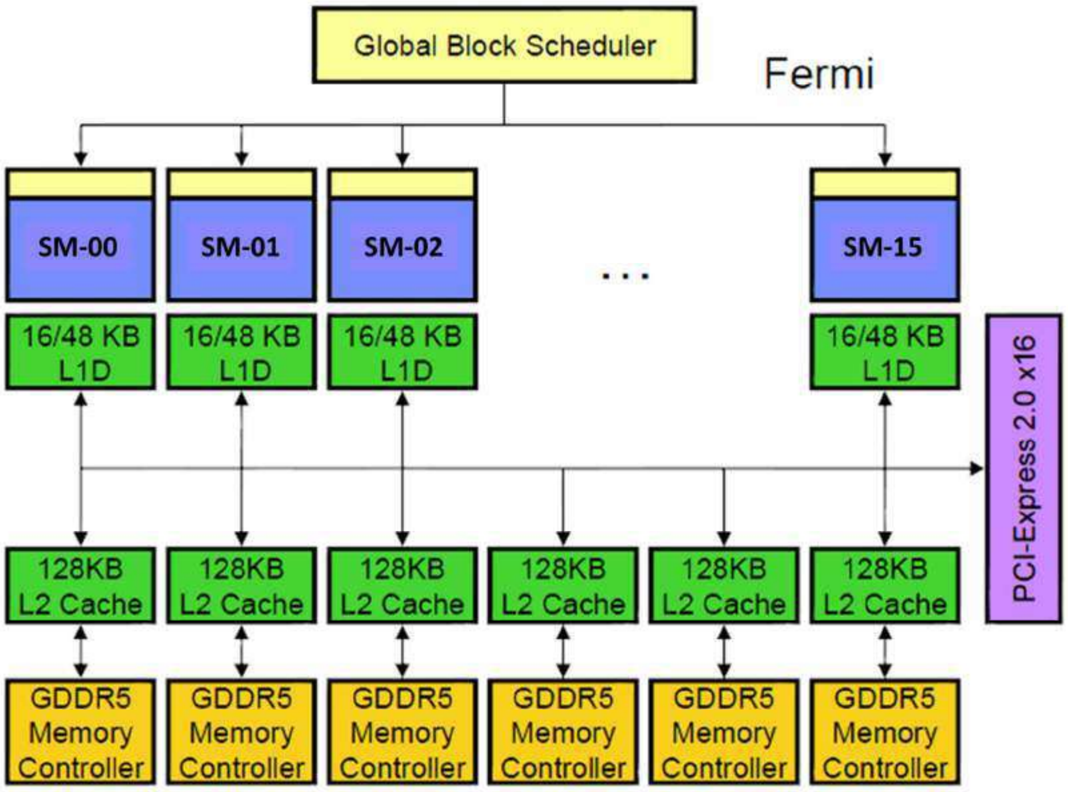


Figure 40. Memory layout in the Fermi architecture [16].

Fermi uses one unified memory model with 40 bit addressing, which means that it can address 1 TB of memory. All memory spaces: local thread-private, block-shared, and kernel global, are ranges within this address space and can be accessed by load/store instructions.

The Front End (the Scheduler)

There are two schedulers at work during the execution of a kernel: a device-level scheduler and an SM-level scheduler. The former operates at a higher level by dispatching blocks to SMs that advertise excess capacity and thus can accommodate the execution of a new block of threads. The latter controls what happens once a block gets assigned to the SM.

The device-level scheduler is capable to maintain state for up to 16 different kernels. To understand the motivation behind this feature, recall that each SM can handle up to 1,536 threads. A GTX480 Nvidia card has 15 SMs and as such it can handle simultaneously up to 23,040 threads. In case the work associated with a task, which is implemented as a kernel, is limited and requires, for instance, a total of 5,000 threads, the excess hardware capacity can be used to launch a second kernel with a different execution configuration. The device scheduler maintains state information for both kernels and schedules to keep the SMs as busy as possible. If two or more kernels are executed by a device with multiple SMs, an SM can only execute one kernel at any given time; for instance, it cannot execute four blocks of which three belong to a kernel and one block that belongs to a different one.

Moving on to the SM-scheduler, its details are sketchy as the manufacturer has not provided a clear account of the pertinent design decisions adopted for Fermi. The most plausible scenario that would support the CUDA programming model requires the presence of a number of 48 independent instruction queues in an instruction buffer, see Figure 41. The reason for having 48 queues brings into the discussion the concept of warp of threads. A warp represents a collection of 32 threads, all belonging to the same block, that are scheduled for execution on the SM. The size of a warp cannot be controlled by the user and might be changed by Nvidia in the future. The fact that an SM can handle at any given time in the execution of a kernel up to 1,536 threads is equivalent to saying that the SM can manage up to 48 warps, from where the number of queues. Since there are two schedulers, effectively there are two buffers, each with 48 queues in them.

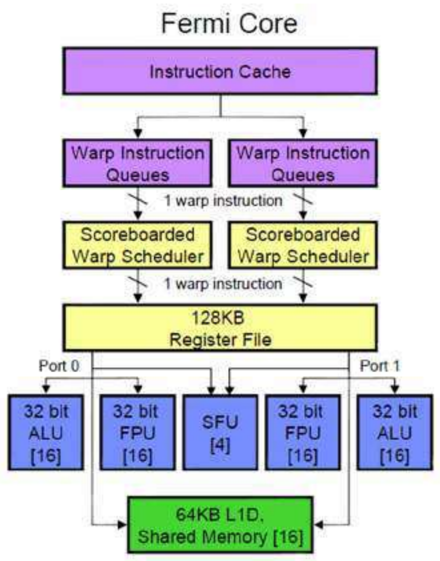


Figure 41. The Fermi SM scheduler [16].

The most likely setup is that in which four to eight instructions are brought into the queue for each warp and then decoded. After decoding, each warp is scoreboarded, that is, each warp logs its data dependencies for the instruction it is about to execute. A warp of threads is subsequently dispatched for execution only when there are no outstanding dependencies that prevent the warp instruction from being executed. Effectively, a warp of threads executes an instruction only when there are no conflicts with previously dispatched but incomplete instructions. For instance, imagine that the instruction requires a result that is computed by the previous instruction. Due to pipelining, it takes several cycles for this result produced by the previous instruction to become available. Scoreboarding is the mechanism that brings sanity to this way of operation in which multiple warps are handled at the same time and are executing instructions with vastly different execution latencies: for instance a fused multiply-add takes one cycle while the evaluation of transcendental function by the threads of a warp might take four times longer. It doesn't come as a surprise that often times there are situations when no warp is ready to be dispatched, which leads to underutilization of the back end (execution units).

Let's discuss next the concept of thread divergence and understand how instructions are counted. In an ideal scenario, if SIMD were to be strictly enforced, all 32 threads in a warp would execute instructions in lockstep fashion; i.e., the same instruction is executed by all 32 threads. This is an ideal scenario that happens often times, but not always. The term "thread divergence" or "warp divergence" is used to indicate such a scenario in which threads *of the same* warp execute different instructions. In lockstep fashion, the scheduler issues for execution one instruction at each cycle. As such, the throughput is 32 instructions per clock cycle. This allows one to quickly figure out what the peak flop rate is: for Nvidia GTX 480, the core clock ticks at 607 MHz. The two schedulers can schedule in an ideal scenario 32 instructions each for a total of 64 instructions. If the instruction scheduled is a fused multiply-add (FMA), it counts as two floating point operation. This yields 1.165 Tflops in single precision: 15 SMs times 607 MHz (core clock) times 64 instructions times two floating point operations. That is, a GTX 480 can perform more than one thousand billion operations per second.

There is one subtle difference between the SIMD computational paradigm, in which threads operate in lockstep fashion by executing the same instruction, and the model supported by CUDA, which is called Single Instruction Multiple Threads (SIMT). SIMT is more general than SIMD in that threads can execute at the same time different instructions. The important thing to keep in mind is that this SIMD vs. SIMT discussion is relevant only in the context of a warp. On the same SM, warps are in fact expected to be about to execute different instructions at a certain moment; the important question is whether *within* one warp all threads execute the same instruction or not. Note that the device can reach peak performance only when it operates in SIMD fashion, otherwise the throughput gets reduced. An extreme case is discussed in Example 2.2.

Example 2.2. Thread Divergence

An extreme thread divergence case is shown in Code Snippet 14. Each thread in each warp ends up executing a different instruction. It is impossible for a kernel that includes this type of code to reach more than 1/32 of the peak performance of an Nvidia device.

The setup is as follows: the pointer `arrA` points to an array of floats. The array is 1.28 million entries long and as such a number of 5000 blocks will be launched, each with 256 threads. On a GTX 480, given that there are 15 SMs, each SM will deal with about 333 or 334 blocks. The exact number cannot be specified since it is the prerogative of the device scheduler to manage the scheduling of block launching; the user has not control over scheduling.

Recall that what get scheduled for execution are warps of threads. The 32 threads in a warp always have consecutive indices, and information regarding this index is picked up at Line 4 from the CUDA native variable `threadIdx`. Owing to the presence of the modulo operation, each thread of a warp executing the kernel `cookedUpExampleThreadDivergence` will end up at Line 4 with a different `indx` value. As such, based on the switch statement at line 7 each thread in the warp is scheduled to execute a different instruction. The instruction throughput will be impacted by a 1/32 factor since the 32 threads in the warp will be scheduled to execute 32 different instructions. Compare this with a previous example in which two arrays of integers were multiplied pairwise. There all threads picked up to integers, multiplied them, and stored them in a results array; all threads executed the same set of instructions but on different data. In the current example the first thread in the warp does nothing, the second one doubles the corresponding value of `arrA`, the third thread in the warp does yet something else, etc.

```
1.  __global__ void cookedUpExampleThreadDivergence(float* arrA)
2.  {
3.     const unsigned int threadsPerWarp=32;
4.     int indx = threadIdx.x % threadsPerWarp;
```

```
    int offset = threadIdx.x + blockDim.x*blockIdx.x;

    switch( indx ) {
    case 1:
       arrA[offset] += arrA[offset];
       break;
    case 2:
       arrA[offset] += 2*arrA[offset];
       break;
    case 3:
       arrA[offset] += 3*arrA[offset];
       break;
       // .. and so on...
    case 31:
       arrA[offset] += 31*arrA[offset];
       break;
    default:
       break;
    }
}
```

Code Snippet 14. A thread divergence example: each of the 32 threads in a warp executes different instructions. Only a small portion of the kernel code is shown here.

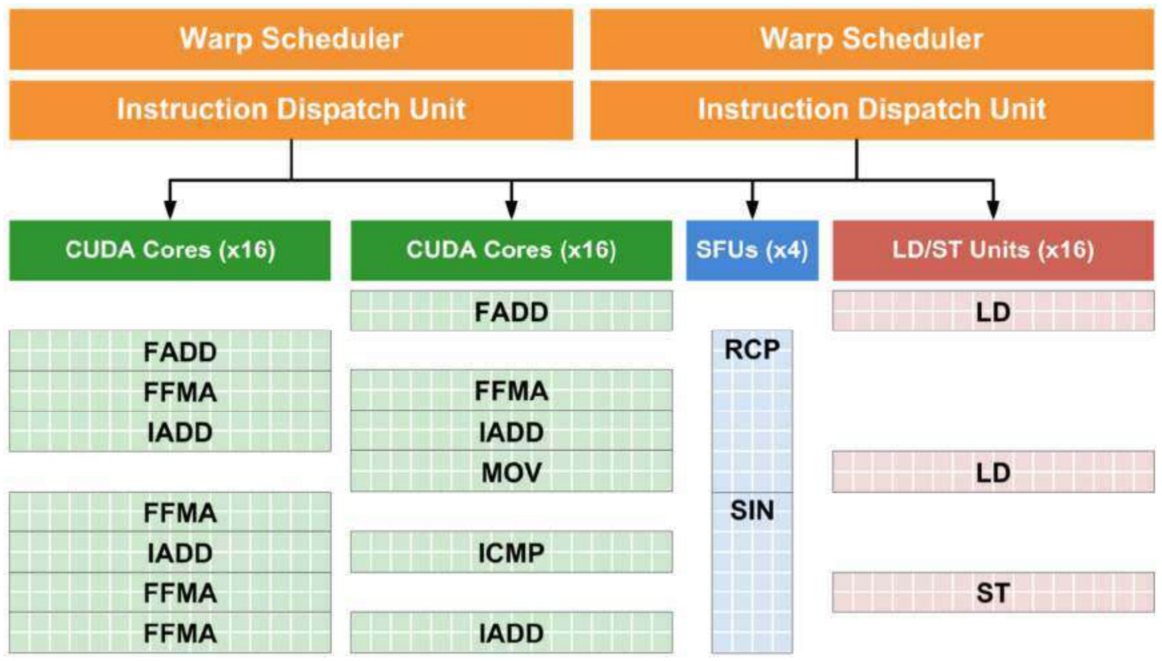


Figure 42. Instruction dispatching for Fermi [17]. Image shows the four functional units in three different colors. At no point in time are there more than two instructions being dispatched to the four functional units. However, at times there can be more than two functional units busy, something that is highly desirable – the mark of a program that fully utilizes the hardware. Credit: Nvidia.

Figure 42 and Figure 43 provide two different perspectives to the topic of scheduling: the former illustrates how instructions are dispatched to the four functional units, while the latter illustrates how different warps are executed by the two pipelines. Figure 42 makes it clear (a) at most two batches of 32 instructions can be issued at each cycle; and (b) not any combination of instructions is feasible. For instance, the two schedulers cannot issue two sets of 32 instructions in which each set draws on the SFU. This is because there is only one set of four SFUs. Examples of pairs of instructions that can be issued by the two schedulers are “memory access + integer arithmetic”, “integer arithmetic + single precision floating point arithmetic”, “integer arithmetic + integer arithmetic”, “single precision floating point arithmetic + single precision floating point arithmetic”, etc.

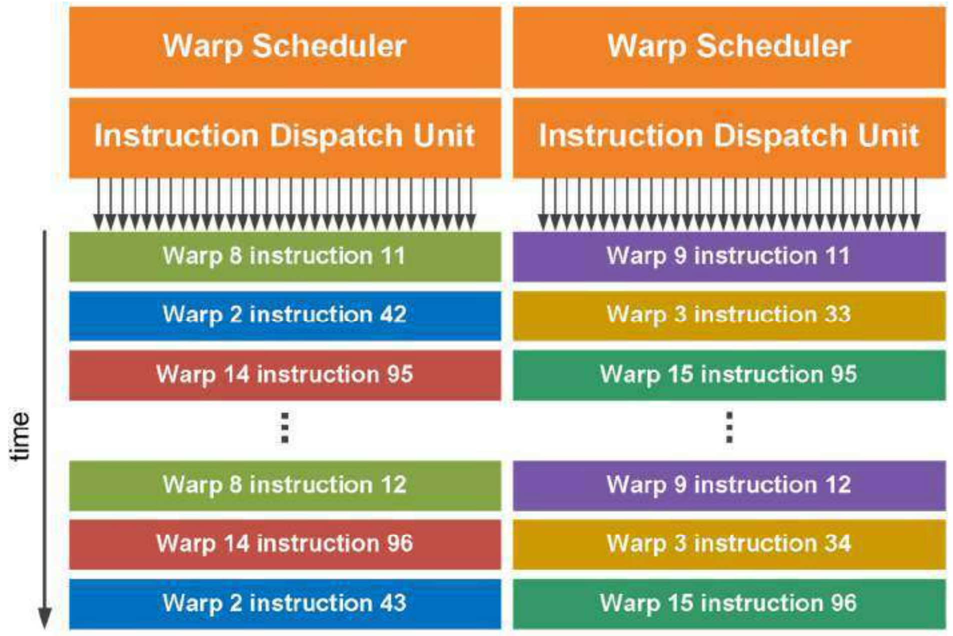


Figure 43. Another perspective on scheduling: time slices are allocated to execute different warps on one SM [18]. These warps are made up of threads that can belong to any of the blocks handled by the SM. Ideally, a warp would be ready for dispatch at each clock cycle. The more threads are deployed for execution on an SM, the bigger the pool of warps waiting to be picked up for execution. There is a limitation on Fermi of 1,536 threads (48 warps) being assigned to an SM. This number went up to 2,048 (64 warps) on the Kepler microarchitecture.

Figure 43 illustrates a possible time-slicing scenario where in the time windowed provided several warps are picked up for execution: Warp 2, Warp 3, Warp 8, Warp 9, Warp 14, and Warp 15. The case presented represents an ideal situation, at no time slice do we see any of the pipeline idle. This is rarely the case since data dependencies introduce bubbles in the pipeline and only a fraction of the cycles end up performing useful execution.

The Back End (the Stream Multiprocessor)

An SM is an entity similar to a processor core in that it has its own scheduler (in fact two of them for Fermi), register file, L1 cache, and functional units. While the Intel Haswell CPU core handles two different threads owing to its hyper-threading attribute, an SM is organized to handle a very large number of threads at the same time. For a CPU core and a Fermi SM though, the word “handle” has different meanings. The Haswell core takes up a second execution thread with the intention to execute its instructions whenever the other thread is stalled most likely due to memory operations. On average, this strategy provides a 20% boost in performance; i.e., a 1.2X (1.2 times) speedup of the execution of two programs instead of a theoretical 2X improvement that could be expected from the use of a second thread. By comparison, a Fermi SM is overcommitted to handle up to 48 warps; i.e., 1,536 threads, in an attempt to have enough opportunities to hide stalls: if a warp is busy waiting on resources to become available, chances are that some other warp will be ready for execution. Effectively, although 1,536 threads are "handled", the Fermi SM can dispatch for execution up to two warps; i.e., 64 threads, at each clock cycle.

The back end in a Fermi SM is organized as follows (see Figure 44): there are two pipelines fed by two schedulers. These two pipelines are serviced by four functional units: two blocks of 16 scalar processors (SPs, or shaders), a group of four special function units (SFUs), and a set of 16 load/store units that allow for source and destination addresses to be calculated for sixteen threads per clock cycle. As indicated in the medallion on the left side of Figure 44, each SP in the two execution block functional units carries out single precision or integer arithmetic and as such is a combination of a floating point unit (FPU) and an integer arithmetic logic unit (ALU), which cannot operate simultaneously.

The functional units tick at different clock frequencies: the schedulers operate at one frequency; the FPU/ALU and SFU operate at twice the frequency of the schedulers; the 16 load/store units operate at what is called the memory clock rate. For GTX 480 the SM scheduler rate is 607 MHz, the ALU and SFU operate at 1.215 GHz (called the shader frequency), and the memory clock ticks at 1.674 GHz. Since the shader's frequency is twice that of the scheduler, whenever the scheduler issues a single precision or integer arithmetic instruction the shader gets two runs at it. This explains why a warp of 32 threads is executed in one (scheduler) cycle although there are only 16 shaders or SPs. A higher latency; i.e., four cycles, is experienced when an instruction requires the evaluation of a transcendental function such as `log`, `exp`, `sin`, reciprocal, etc. Only four SFUs are available to service such a request. Since they operate at twice the frequency of the scheduler it takes four cycles to fulfill the request. Since the four SFUs are shared between two schedulers, the SFUs can become a subject of contention that might lead to the idling of a pipeline. A similar situation can occur for memory transactions since there is only one set of 16 load/store units shared by two pipelines. Finally, there is no direct hardware support on Fermi for 64-bit (double precision) operations. A 64-bit arithmetic operation requires the participation of both pipelines, which incurs scheduling overhead and reduces the instruction throughput since one of the scheduler has to idle while its pipeline services the double precision arithmetic operation. This has changed recently and Kepler, the generation immediately following Fermi, has a microarchitecture with direct hardware support for 64-bit arithmetic.

SM microarchitectures rely on a large number of transistors but in their inner workings these microarchitectures are less complicated when compared to the microarchitecture of a CPU core. The SM designs dedicated more transistors to FPUs rather than in support of complex logic required by instruction level parallelism. This should come as no surprise, as the original purpose of the SM was fast floating point arithmetic for image generation. The shaders on the SM used to be exclusively dedicated to processing information associated with image pixels, which was a straightforward, fixed function, job in which throughput was of the essence. An Intel processor is designed with a different philosophy in mind. Since the CPU should be a jack of all trades, a large contingent of transistors is dedicated to identifying opportunities for parallelism and increasing the IPC through hyper-threading and superscalar execution. This explains why Haswell is superscalar while neither Fermi nor Kepler is. Moreover, the SM has no support for out-of-order or speculative execution. An argument could be made that since each SM dispatches two instructions per cycle it might be regarded as superscalar. In reality, there are two distinct pipelines and neither one is superscalar.

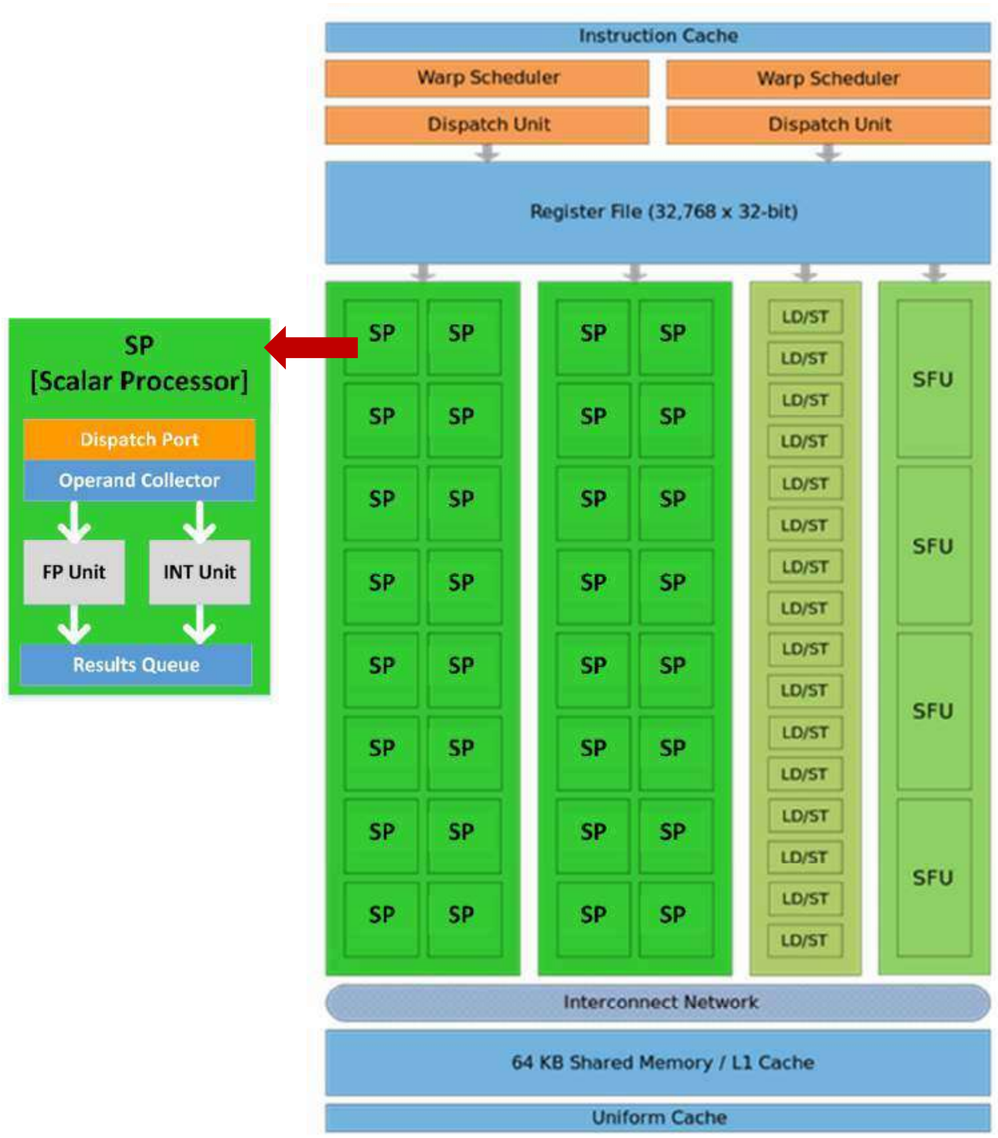


Figure 44. Architecture of a Stream Multiprocessor (SM). Fermi has four functional units: two blocks of 16 SPs each, a group of four SFUs, and a set of 16 load/store units. Color code: orange - scheduling and dispatch; green - execution; light blue -registers and caches. Credit: Nvidia.

References

[1] MIPS-Convertor. (2013). *The MIPS Online Helper*. Available: http://mipshelper.com/mips-converter.php

[2] D. A. Patterson and J. L. Hennessy, *Computer organization and design: the hardware/software interface*, Revised Fourth ed.: Morgan Kaufmann, 2011.

[3] T. Steinke. (2012). *MIPS Helper*. Available: http://mipshelper.com/mips-converter.php

[4] CUDA. (2013). *CUDA Programming Guide 5.0*. Available: https://developer.nvidia.com/cuda-downloads

[5] SPEC. (2012). *Standard Performance Evaluation Corporation*. Available: http://www.spec.org/

[6] H. Meuer, J. Dongarra, E. Strohmaier, and H. Simon. (2013). *TOP500 List*. Available: http://top500.org/

[7] (2004). *Oxford Dictionaries (11th ed.)*. Available: http://oxforddictionaries.com/words/what-is-the-frequency-of-the-letters-of-the-alphabet-in-english

[8] J. L. Hennessy and D. A. Patterson, *Computer architecture: a quantitative approach*: Elsevier, 2012.

[9] J. R. Levine, *Linkers & Loaders*: Morgan-Kaufman, 2001.

[10] C. Grassl. *POWER5 Processor and System Evolution, ScicomP11 May 31–June 3, 2005*. Available: http://www.spscicomp.org/ScicomP13/Presentations/IBM/Tutorial2Feyereisen.pdf

[11] H. Sutter, "The free lunch is over: A fundamental turn toward concurrency in software," *Dr. Dobb's Journal,* vol. 30, pp. 202-210, 2005.

[12] R. H. Dennard, F. H. Gaensslen, V. L. Rideout, E. Bassous, and A. R. LeBlanc, "Design of ion-implanted MOSFET's with very small physical dimensions," *Solid-State Circuits, IEEE Journal of,* vol. 9, pp. 256-268, 1974.

[13] A. L. Shimpi. (2012). *Intel's Haswell Architecture Analyzed: Building a New PC and a New Intel*. Available: http://www.anandtech.com/show/6355/intels-haswell-architecture/8

[14] R. M. Tomasulo, "An efficient algorithm for exploiting multiple arithmetic units," *IBM Journal of research and Development,* vol. 11, pp. 25-33, 1967.

[15] D. Kanter. (2012). *Intel's Haswell CPU Microarchitecture*. Available: http://www.realworldtech.com/haswell-cpu/

[16] D. Kanter. (2009). *Inside Fermi: Nvidia's HPC Push*. Available: http://www.realworldtech.com/fermi/

[17] P. N. Glaskowsky, "Nvidia's Fermi: the first complete GPU computing architecture," *NVIDIA Corporation, September,* 2009.

[18] NVIDIA. (2009). *NVIDIA's Next Generation CUDA Compute Architecture: Fermi*. Available: http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIAFermiComputeArchitectureWhitepaper.pdf