The CPSC 213 Companion

(Second Edition - beta 2)

Mike Feeley

September 16, 2010

Chapter 1

Introduction to the Computer System

A computer is a machine, built from digital-electronic components — the hardware — and the programs those components execute — the software.

The *computer system* consists of the computing hardware and the *system software* that executes application programs. This software consists of an operating system, library software (called middleware) and compilers that translate programs written in *high-level* languages into the *machine instructions* that are executed by hardware.

CPSC 213 gives you an introduction to the computer system. Entering this course you know how to write programs in Java. You are starting to get pretty good at this. In 111 and 211 you developed a mental model of what a program is and what happens when it runs. The development of this model is the essential goal of these two courses. Once this model is deeply rooted in your brain, you will be able to implement complicated algorithms in Java and also in similar languages such as C, C++, C# with some review of how these languages compare to Java.

The goal of 213, and its companion course 313, is to help you extend your mental model of computation so that it is rooted in a physical manifestation of computation: the digital computer. Leaving 213 you will be able to explain the execution of a Java program in the context of the physical machine. You will see what happens in the machine as your program executes. You will know how the machine performs the computation that is your program.

The course is divided in two three main sections. In the first section we compare Java and C and see how these high-level languages are implemented by the CPU hardware — and the machine instruction set it implements — and the Main Memory hardware — which stores the program and all of the data it accesses.

The second section extends this basic model to include IO devices such as disks and network interfaces, virtual memory and the operating system. We examine how the devices communicates with the CPU. We examine the role of the operating system in controlling access to the devices.

The third section of the course adds concurrency, threads and networks to this model. We again investigate the role of the operating system in implementing network- protcol abstractions and we look at how to write programs that transfer data over the network.

1.1 Java and C

This course uses two high-level programming languages. One that you know and are in the process of become an expert in — Java — and a new language, the main language of system programming — C. Some of the programming you will do in this course is in Java — an auxiliary goal of this course is to reinforce your programming skills developed in 111 and 211. But, you will also learn to program in C.

Leaving 213 you will understand how to translate the key features of Java into C. You will be able to compare the two languages to describe the key differences between them and to explain their relative strengths and weaknesses. You will thus be able to write a simple C program.

The goal of 213 is not to make you an expert C programmer. However, the best way to become an expert C program **is** to become an expert Java programmer, know how to translate key Java features into C and to practice writing and debugging C programs. Together 111, 211, 213 and the rest of the curriculum tackle the first two of these goals. 221 gives you some experience with C++. 415 gives you considerable practice writing C programs.

Java and C# are *interpreted languages*. This means that their compilers translate programs into instructions of a *virtual machine* that is implemented in software. In the case of Java, the program that executes programs is called the *Java Virtual Machine*. For other languages such as C and C++, the compiler translates programs into instructions that are directly executed by machine hardware. The focus of 213 is to understand execution in hardware and with system software, other programs running on hardware. We thus think about Java execution by looking at the C equivalent of those programs and the execution of these programs in the machine.

1.2 The Compiler

The compiler is a program that translates a program written in a high-level language into a sequence of instructions a machine can execute. In the case of Java and C#, the machine is a virtual machine implemented by software. In the case of C, C++ and many other languages, the machine is implemented by hardware.

Throughout the first part of this course you will act as the compiler, translating very small snippets of C programs into machine instructions for a hypothetical CPU devised for this course.

A key issue we'll investigate in this course is to distinguish *static* information from *dynamic* information. We define anything that the compiler can know when it is compiling a program to be *static* and things that can only be know while the program is running to be *dynamic*. For example, some variables are *static* and others *dynamic*. The compiler knows the address of *static* variables and thus can hardcode the address of these variables in the instructions it generates. The compiler does not know the address of *dynamic* variables and so the code it generates for access to these variables first loads their from memory.

Chapter 2

Execution of a Program

2.1 The Plan

At the end of this section, you should be able to do the following.

1. You will be able to explain how a real machine executes a Java program. You will be able to identify the key features a machine required to execute a program. You will then be able to explain how each of the key components of the Java language are implemented by these machine capabilities.
2. You will be able to translate a simple Java program in to a C program, to the extent possible given the differences between the languages.
3. You will be able to translate a simple snippet of a C program into *machine instructions* for a simple computer.
4. You will be able to write and debug simple C programs.
5. You will be able to describe the differences between the two languages and to compare their relatives strengths and weaknesses.
6. You will be able to describe the key features of this simple computer and explain how each of its machine instructions uses these features to form a simple computation. You will see that each of these instructions is simple enough that it could be implemented by a combination circuit.

Our strategy will be to examine the execution of simple snippets of a program in a simple machine (called the *Simple Machine*). The course defines the basic characteristics of the machine and its instruction set, *SM213*. The course provides a simulator, written in Java, for running programs written in the *SM213* assembly language (or even in its machine language). The simulator has a powerful GUI designed to allow you to explore the execution of machine-language programs, debug them, understand them, add comments to them etc.

We examine the execution of a Java program, by taking a program bit by bit and looking at how it is implemented in the Simple Machine. Each bit of the program is called a *snippet*. Each snippet exemplifies a single key feature of the language, to focus on individual concerns for execution in the machine.

We first translate each Java snippet into C and talk about the similarities and differences between the way that Java and C treat this key idea exemplified by the snippet.

Then, we will work through a design of a set of instructions needed to execute this C program, when translated into their machine code. And finally, we'll manually compile the snippet into this instruction set. To simulate the program,

you will implement the instructions we design in the Simple Machine simulator. Then you'll execute the snippet and similar programs in the simulator. Some of this work will be done together in class, some as group exercises in the lab and some as individual lab work.

Here are the snippets we examine:

Variables

- **S1-global-static** Static variables for accessing static scalars and static arrays.
- **S2-global-dyn-array** Static variables for accessing dynamic arrays.
- **S3-C-pointer-math** Accessing dynamic objects in C: the & operator and the equivalence of pointer arithmetic with and array access with [].
- **S4-instance-var** Accessing instance variables (data members) of an object, both static and dynamic.

Control Flow

- **S5-loop** For loop that adds an array of numbers (snippet4b is loop unrolled) .
- **S6-if** If-then-else statement that computes $max(a, b)$.
- **S7-static-call** Invoking a static (class) method with no arguments.
- **S8-dynamic-call** Invoking an method on an object.
- **S9-switch** Switch statement.

Procedure Call Scope

- **SA-locals** Local variables of a method.
- **SB-args** Arguments of a method.

2.2 Introduction to a Simple CPU

A computer consists of several pieces of hardware that work together to execute programs: the central processing unit (CPU), the main memory, the IO device controllers (e.g., for disks, network), and the display-graphics processor (GPU). In this chapter we consider a simple computer that consists of only two of these: the CPU and main memory.

2.2.1 The CPU

A CPU consists of combinational circuitry built from transistors, a set of on-chip memories called *registers* and a clock.

The Combinational Circuit

Let's take a step back into your recent history and recall 121. In 121 you learned that finite computation can be modeled by propositional logic (also called Boolean algebra), which in turn can be implemented by a collection of logic gates built from transistors (though you don't know yet what a transistor is or how they implement gates — that comes in 313). This type of circuit is called a *combinational circuit*.

Another way to think of a combination circuit is that it can implement any function, but only functions. Recall from 121 that a function is a static mapping between two sets, an input set and an output set. Stated more simply, a function produces an output based *only* the value of it inputs. So, for any particular input values, the combinational circuit always gives the same output. For example, computation "a + b" is a function and so it can be implemented by a combination circuit. A computation that adds the values in an array of integers is only a function if the size of the input array is fixed. A combination circuit, however, can not add the values of an array if the size can vary (e.g., the size is an input parameter). To perform this type of computation requires a memory and a clock.

Registers

Combining a combinations circuit with memories and a clock yields a *sequential circuit*. The CPU is a sequential circuit where the memories that the combination circuit can access directly are called *registers*. A *register* is simply the name for a memory that resides on the processor chip and that is connected as either an input or an output to the combinational circuitry of the CPU.

Again think back to 121 and your implementation of an adder. The adder is a combination circuit with two multi-bit inputs, let's call them A and B, and computes and output S. In most computers, numeric values are 32-bits long. So, let's say that A, B and O are 32-bit numbers. Thus the input wires of the adder circuit are connected to two 32-bit *registers* and the output wires are connected two a third of these.

In CPU this notion of an adder is extended to something called the *arithmetic logic unit* (ALU). The ALU is a combination circuit with the same inputs and outputs as the adder, but with a third input F, a number that selects one of several different functions the circuit can compute. For example, $F = 3$ might cause the circuit to compute $O = A + B$, while $F = 4$ might cause it to compute $O = A * B$.

The Clock

The CPU is simply the combination circuitry, the registers and a clock. The clock regulates the memories as it ticks. Modern Intel processors typically tick 2 or 3 billion times a second.

To understand what happens on a clock tick, think of every register as having an input and output wire for every bit it stores. Normally the memory ignores its inputs and sets its outputs to the values it stores. But, on each clock tick the memory grabs the current value on each if its input wires, stores them and starts presenting those values on its output wires.

Each clock tick starts a new *cycle* of the processor. On each cycle the new value loaded into the registers provides new inputs to the combination circuit. This circuit computes a new output based on these values and presents this output to the input wires of the registers. Once this computation has completed, the clock can tick again. This keeps happening over and over, on each tick the CPU's state is transformed one step at a time. Each step is in fact the execution of a single *machine instruction*.

The Register File

While we could implement the ALU (and other parts of the CPU) using dedicated registers for its inputs and outputs, this approach is not ideal when you consider a computation that consists of a series of ALU operations; i.e., a program.

For example, consider a program that computes $r = (x + y) * x/y$ with dedicated registers. The execution of the machine would be something like this; each line is a separate cycle.

1. Load x into ALU_A, y into ALU_B, and 3 into ALU_F (assuming $ALU_F = 3$ means add).
2. ALU computes $ALU_A + ALU_B$ and puts result in ALU_O.

3. Copy ALU_O to ALU_A, x into ALU_B, and 4 into ALU_F (assuming 4 means multiply).
4. ALU computes $ALU_A * ALU_B$ and puts result in ALU_O.
5. Copy ALU_O to ALU_A, y into ALU_B, and 5 into ALU_F (assuming 5 means divide).
6. ALU computes ALU_A/ALU_B and puts result in ALU_O
7. Copy ALU_O to r.

Processors instead use a set of registers called the *register file* in way that any of these registers can be connected to any of the inputs or outputs of the ALU and all other parts of the combinational circuitry.

Now each bit of an ALU input or output is connected to a bit of every register in the register file using a component called a multiplexer (MUX), another thing you learned about in 121. A multiplexer is a combinational circuit that has n value inputs and an "selector" input. The selector is a number between 0 and $n-1$ that picks exactly one of the value inputs to be connected to the output of the multiplexer. For example, if the selector is 3, then the value of register 3 is connected to the output of the MUX and thus to the combinational circuit of the ALU. The ALU uses 32-bit input and output values, so the selector actually connects all 32-bits of the register named by the selector to the corresponding 32-bits of the output.

The ALU_A, ALU_B and ALU_O are connected to two different MUXes and a DECODER for ALU_O, each of these is connected to the register file. Each MUX/DECODER has a selector input that specifies a register for ALU_A, ALU_B and ALU_O. The input of this modified ALU is thus $ALUsel_A$, $ALUsel_B$, $ALUsel_O$ and F: three register numbers and a function number.

The execution of a machine with a register file to compute $r = (x + y) * x/y$ is like this; again each line is a separate cycle of the machine.

1. Load x into a register, say 0, and y into a register, say 1.
2. Pick a register to store the result, say 2. Load 0 into $ALUsel_A$, 1 into $ALUsel_B$, and 2 into $ALUsel_O$. Load 3 into ALU_F. ALU computes sum using values from register file and putting result in register file.
3. Pick a register to store the result, say 3. Load 2 into $ALUsel_A$, 0 into $ALUsel_B$, 3 into $ALUsel_O$, and 4 into ALU_F. ALU computes product using values from register file and putting result in register file.
4. Pick a register to store the result, say 3. Load 3 into $ALUsel_A$, 1 into $ALUsel_B$, 3 into $ALUsel_O$, and 5 into ALU_F. ALU computes quotient using values from register file and putting result in register file. Notice that register 3 is both an input and an output, which is fine because the new value of register 3 computed by the ALU waits at the register file until the clock ticks.
5. Copy value in register 3 into r.

The key benefit of this approach is that once the intial values are loaded into registers, then a sequence of ALU operations can be performed by repeated selecting different registers for inputs and output, repeating steps like 1–3. If the ALU had dedicated registers, then extra steps are required to move values from inputs to outputs or to and from main memory where the variables x, y and r are stored. In a real machine access to main memory is very slow compared to access to the register file and so the benefits of the register file are much greater than they appear in this example.

A typical machine has between 8 and 16 registers. Our simple CPU has 8, named by the numbers 0 through 7.

Special Purpose Registers

In addition to the general-purpose register file the CPU also has a set of special purpose registers. Our simple machine has these.

PC The memory address of the next instruction to execute (i.e., on the next cycle).

instruction The value of the current instruction being executed. Loaded from memory by the beginning of the cycle from memory at PC.

insOpCode The value of the operation code part of the instruction.

insOp0 The value of the instruction's first operand.

insOp1 The value of the instruction's second operand.

insOp2 The value of the instruction's third operand.

insOpImm For instructions that store a small immediate (i.e., constant) number, this is the value of that number.

insOpExt For instructions that store a large (32-bit) immediate, this is the value of that number.

2.2.2 The Memory

Main memory is implemented by a set of memory chips external to the CPU chip. Each memory chip is called DRAM (dynamic random- access memory), after the technology used to implement the memory. A DRAM bit is implemented by a single transistor and a capacitor. A set of memory chips that implement the entire memory is called a DIMM (dual-inline memory module).

Data in memory is named by its address. Most systems give a unique name to every byte of the memory (a byte is 8 bits). Most data access to memory is to 32-bit (4-byte) words (sometimes called longs). Real CPUs have instructions for accessing memory in 1, 2, 4 (and sometimes 8) byte chunks. Data in our simple machine can only be accessed in 4-byte chunks.

You should thus think of memory as if it were an array of bytes, but where you only access memory by reading or writing four-byte chunks at a time. In this analogy, array index is the memory address.

The CPU talks to memory by sending simple request messages that say one of two things: "please tell me the value of the four bytes starting at address A" or "please store V in the four bytes starting at address A.

The machine instructions that implement programs are also stored in memory. They are read by the CPU in a similar fashion, but using registers (PC and instruction). Once **every cycle** the memory reads delivers the instruction stored at the memory address in `PC` and delivers it to `instruction`. The CPU then further decodes `instruction` into the its various parts, described above.

In real systems, the interaction between the CPU and memory is a bit different than this, because the CPU chip stores some instructions and data values on chip in special memories called caches. Every memory read first looks in the cache to see if the desired value is there. If so, the CPU gets the value quickly. If not, the request is sent to memory and the execution of the instruction stall many cycles (perhaps 200 or more) until the memory is able to fetch the desired data. You will learn about caching in 313.

The key thing to keep in mind about the memory is that it is a very simple abstraction: a byte array. It stores data, but doesn't have any idea what that data means. It stores instructions and program variables, but it doesn't know that. All it knows is that it has a bunch of byte values and they all have a unique address.

There are, however, two remaining issues to consider.

Aligned Access

When reading or writing multi-byte data from or to memory, access to memory is fastest if the addresses used to access that data are *aligned*. For example, when accessing 32-bit words, aligned addresses are those addresses where the lower two bits are 0 (i.e., the address modulo 4 is zero). For example, 0, 4, 8, etc. are aligned addresses, but 1, 2, 3, 5 etc. are not.

The reason why aligned addresses are more efficient is that in this scheme every byte of the target word has the same *word address*. For example, the bytes 8, 9, 10 and 11 are all part of an aligned word with byte address 8. In binary these numbers are 1000, 1001, 1010, and 1011. Notice that if you look only at the bottom two bits you have the numbers 0-3, which pick the one of the bytes of the number. If you ignore these two bits, you have the number 10 for all of them. This is the value's word address.

If we allowed addresses to be unaligned, then an instruction could read a word starting at address 10 to access bytes 10, 11, 12, and 13. Again, in binary these are 1010, 1011, 1100, and 1101. Notice that the word address of the first two bytes is 10 and the word address of the second two is 11. It might not seem like much, but implementing hardware well is a difficult task, and even this simple complication is problematic.

As a result, some systems require that addresses be aligned and thus issue a hardware error if a program attempts to read a word at an address that is not word aligned. Other systems, such as Intel processors found in PCs, allow unaligned access, but implement the access more slowly than aligned access.

See also Bryant and O'Hallaron $\S$ *3.10.*

Endianness

The final issue for memory is to decide how the bytes of memory are assembled to make multi-byte integers. Again, consider the four-byte (32-bit) word. We know that each word in memory has a byte address, call it a, and thus the word is comprised of memory bytes with addresses a, $a+1$, $a+2$ and $a+3$. We also know that a 32-bit number is comprised of four bytes, one of which store the bits 0-7, another 8-15, another 16-23 and another 24-31. If n_i is the value of bit i of the number, then the value of the number is $\sum_{n=0}^{31} n_i \times 2^i$.

Now the question is, into which byte of the number do the memory bytes go. This decision is called the *endiannes* of the memory, named after Jonathan Swift's Gulliver's Travels in which two groups argue about which end of an egg is properly eaten first, the big end or the little end. Like those characters, computer systems are also divided into the Big Endians and the Little Endians.

Big Endian processors eat numbers from memory from the big part of the number first. Thus the byte with address a is interpreted to store the most-significant byte of the number, i.e., bits 24-31, and thus byte $a+3$ is the least-significant byte, i.e., bits 0-7. Systems like the PowerPC, Sun SPARC etc. are Big Endian (though the PowerPC can actually do both). Our simple machine is also Big Endian.

Little Endian processors, on the other hand, eat numbers starting with the little end of the number. Byte a is the least-signifant byte, bits 0-7, and $a+3$, the most, bits 24-31. Intel processors are Little Endian.

You need to pay attention to Endianness when transfering binary data between processors with different Endianness, either over the network or via secondary storage, such as DVD-ROM. For example, the number 1 stored on an Intel processor would be read as a 1048576 on a PowerPC or Sun computer (1048576 = 2^{24}).

Hexidecimal and Binary

One last thing. Sometimes we are more interested in the bit-pattern of a number than its in its decimal value. In these cases it is convenient to represent the number in base sixteen, called hexadecimal or hex for short. A hex digit corresponds to four bits. So the number 1048576, for example, is more easily understood to be 2^{24} if it is represented in hex: 0x100000. Of course we could see the same bit pattern in its binary representation, but this is tedious for large numbers: 0000 0001 0000 0000 0000 0000 0000 0000. In hex, you group the bits into groups of four and replace each group with a hex digit. The "0x" prefix is just a signal that the number is a hex number. So we know that 0x10 is not 10 base ten, but 16.

2.2.3 The Anatomy of a Cycle

Each cycle of the CPU executes a single instruction. What happens in the cycle is completely determined by the value stored in registers at the beginning of the cycle. All the computer does now is send these values through the combinational circuit to compute new values for the registers. The cycle ends when all of these new values are ready to be stored.

An important register is `instruction` (and its companion component registers such as `insOpCode`, `insOp0` etc.). These registers store a single instruction read from memory. It is this instruction that the CPU executes in the current cycle. Like other registers, the bits of these instruction registers feed into various parts of the combinational circuit selecting the particular parts of the circuit that implement this particular instruction. The instruction also names the registers from the register file that are the inputs and outputs of the computation (e.g., $ALUsel_A$ etc.).

One way to view the execution of a cycle is to divide it into two stages: fetch and execute. You will see in 313 that three additional stages (decode, memory and write-back) are used to more completely describe the execution of a processor and the way that instructions are *pipelined* through the processor. But, we ignore that stuff for now.

The Fetch Stage

The fetch stage has two very simple jobs. First, fetch the instruction stored in memory at the address in the register *PC* into the register *instruction*. Second, update *PC* to point to the next instruction. The fetch stage must thus know the size of the next instruction. Some systems address this issue by requiring that every instruction be the same size. Fetch can thus do its job without knowing anything about the instruction it just fetched. For example, if instructions are two bytes, the combination logic that implements the fetch stage would do something like this.

1. read two bytes from memory starting at address in PC
2. place result in instruction
3. add two to value in PC

If a CPU implements instructions of different sizes, however, the Fetch stage has a bit harder job. Intel processors, for example, have variable- length instructions.

Our Simple Machine uses 2- and 6-byte instructions. And so one way to implement the Fetch stage is to have the 2 bytes at PC from memory, check `insOpCode` to determine the instruction length and finally fetch the addition 4 bytes if required.

The Fetch stage takes the value in `instruction` and splits it into a set of *subfields*, one each for each part of the instruction. Every instruction consists of an `insOpCode` and one or more *operands*. The `insOpCode` is a number that identifies the function the instruction performs and the operands specify its inputs or outputs. There are three ways an operand can be specified: a constant value, a register number or a memory address. Instructions typically have one or two input operands and an output operand.

The Execute Stage

The execute stage performs the computation specified by `insOpCode` on the operands specified by the instruction. If you think of what the execute stage does in term of a Java program, its like a switch statement (i.e., `switch (insOpCode)`) where each case of the switch statement performs the computation of a particular instruction.

2.3 Instruction Set Architectures

An *Instruction Set Architecture* (ISA) is the interface to the processor. It describes the format of instructions, the meaning of opcodes, the way that operands are interpreted, and the endianness of numbers.

2.3.1 Simple Machine ISA Constraints

The ISA we develop for the Simple Machine is restricted to have the following properties

1. 2-byte instructions with optional 4-byte addendum
2. big endian numbers
3. byte-address memory
4. aligned-word data access to memory

2.3.2 Instruction Set Styles (RISC vs CISC)

A number of different instruction set architectures have been developed over the years and a number are still in production, implemented by various different companies such as Intel, AMD, IBM, Freescale Semicondutor, Sun Microsystems, and Sony. Intel, for example has products that implement a few different ISAs. Their most popular, of course, is the *IA32* architecture implemented by the Pentium chips used in most desktop and laptop PCs. Game consoles, TV set-top-boxes, iPods, and automobile control systems, for example, use different processors that implement different ISAs. There are two principle themes that distinguish ISAs from each other.

One design theme, followed by most companies other than Intel, and the one we use in the our SM213 ISA, is called the "Reduced Instruction Set Computer", more commonly called a RISC. The key idea of RISC Architectures is to design the ISA to make it as easy as possible to implement it efficiently in silicon (on the computer chip). RISC instructions are thus each as simple as possible. Only a few instructions are permitted to access memory; the rest are restricted to accessing data in registers. RISC instructions are typically fixed sized. The tradeoff made in keeping the ISA simple is that the compiler has to work a bit harder. A given high-level language statement when translated to machine instructions will typically require more instructions in a RISC ISA than a CISC. RISC machine code is thus a bit harder for humans to read. The rational for making this tradeoff is that we know how to make smart compilers and humans hardly ever have to read or write machine instructions directly (except in classes like this). Furthermore, the benefits of the simple instruction set on hardware implementation mean that even though a statement requires more instructions in a RISC than a CISC these instructions execute faster and thus RISC implementations typically offer superior performance.

On the other hand, ”Complex Instruction Set Computers”, CISC, are designed to make the compiler’s job easy and to give hardware designers fits. Prior to the 1980’s all computers were CISCs. The RISC idea developed in the 80’s in response to problems with implementing CISCs. At that time, all CISC computers were implemented by several chips connected on a circuit board. The first RISC architectures (MIPS and SPARC) were simple enough to be implemented on a single chip, as we do today for all processors, giving them a huge performance advantage.

The IA32 ISA implemented by Intel and AMD is a CISC ISA. Typical of CISC, IA32 allows virtually every instruction to access memory, has variable-length instructions, and includes fairly complex instructions and addressing modes. When comparing SM213 to IA32, you would see many cases where special-purpose instructions and addressing modes in IA32 allow high-level-language statements to be implemented by fewer instructions than in SM213.

2.3.3 Types of Opcodes

In a RISC ISA like SM213, there are three types of instructions: memory access, ALU and control flow.

Memory access instructions either *load* data from memory into a register or *store* the value in a register into memory. The instructions differ on how they specify the memory address they access.

ALU instructions perform a variety of math operations on values stored in registers. Some ALU instructions treat register values as integers to add, subtract, multiply, divide etc. them. Other ALU instructions treat register values as strings of bits to perform bitwise logical operations on them such as and, or, xor, shift etc. For example, if register r0 stores the value `0x0000000f` and register r1 stores the value `55555555`, an instruction that *ANDs* r0 and r1 and places the result in r1, puts the value `00000005` in r1. Similarly, an instruction that shifts r0 left 2 bits changes its value to `0x0000003c` and instruction that shifts r1 right 2 bits changes its value from `0x0000000f` to `0x00000003`. Shifting left n bits is like multiplying by 2^n shifting right is like dividing by 2^n.

There are two different ways to shift a number right: *sign extension* and *zero extension*. Instruction sets typically have two different shift-right instructions, one for each approach. While a compiler generates code to shift a number to the right, it selects the *zero extend* version unless the value being shifted is a signed integer. In Java, for instance, all numbers are *signed*, but Java provides a special unsigned (i.e., zero-extension) right shift operator, >>>; the normal right shift, >> performs sign-extension.

Recall from 121 that signed numbers are represented in computers in *two’s complement* form. The number -2 is thus stored as `0xfffffffe` (you can get the hex representation of any number by running `gdb`, the gnu debugger, and typing `p/x -2` or some other number). Shifting -2 to the right one bit should be like dividing it by two to get -1 (i.e., `0xffffffff`). The zero-extend-shift-right, however, shifts the number to the rigth and places a 0 in the vacated most-significant bit, yielding `0x7fffffff`. The sign-extend-shift-right, does the right thing by filling the vacated bit the the value just vacated. If the most-significant bit is a 1, then it stays a 1 when shifting right. If its a 0, it stays a zero. Sign-extended-shift right of 2 is 1 (`0x00000002` $\leftarrow$ `0x00000001`) and sign-extended-shift right of -2 is -1 (`0xfffffffe` $\leftarrow$ `0xffffffff`).

Finally, control-flow instructions comprise the third instruction type in a RISC ISA. These instructions specify the instruction to execute next, either conditionally or unconditionally. Normally, instructions are executed in sequence. To implement loops, if statements, switch statements and procedure calls, control-flow instructions, called *jumps* and *branches* are used to pick a different instruction to execute next.

Unconditional jumps and branches specify the memory address of the next instruction to execute, either by encoding its 32-bit value as a constant in the instruction, by specifying a register that stores this value, by specifying a register that stores the memory address where the brach address is located, or by specifying a signed constant to add to the current program counter to get the next-instruction address.

Conditional branches specify a register, a test condition and a target-instruction address. Test conditions typically are things like *equal to zero*, *less than zero*, etc. The instruction applies the specified test to the value in the named register and jumps target address only if the test succeeds. For example, the next instruction executed after a `branch-on-zero of r0 to 0x1000` is at address `0x1000` if r0 stores a 0 and the instruction following the branch sequentially, otherwise.

2.3.4 Addressing Modes

Every instruction specifies an opcode and a set of operands. The ISA defines how these values are encoded into an instruction and how the operands are interpreted. There are typically one, two or three operands per instruction, depending on the instruction. Following the IA32 standard, the SM213 ISA puts source operands *before* destination operands. For example `mov r0, r1`, copies the value in register 0 into register 1.

The part of the ISA design that specifies how to interpret operand values is called the *addressing mode* specification. In the SM213 ISA each opcode itself specifies the addressing mode of its operands. In a real instruction set, however, the addressing mode of operands is usually specified by adding a few mode bits to the encoding of each operand. In SM213, for example, opcode 2 performs a *load indexed* operation and thus specifies that all three operands are registers and that the two source-specifier registers specify a memory address by taking the value of one, multiplying it by four, and adding it to the other. There are two other *load* instructions (opcodes 1 and 2) that differ only in how their operands are interpreted. In real instruction set, there would typically be a single *load* opcode and the instruction would encode an addressing-mode number and value (register or constant, depending on the mode) for each.

Figure 3.3 in Bryant and O'Hallaron (the text book), on page 137, lists the addressing modes provided by the IA32 instruction set architecture. The SM213 machine implements a subset of these addressing modes, using opcodes to specify which one an instruction uses.

Here are the SM213 ISA addressing modes:

Mode	Operands	Meaning
immediate	v	value ← v is a 32-bit constant
register	r	value ← register[r]
base + displacement	o b	value ← memory [o*4 + register[b]]
indexed	b i	value ← memory [register[b] + register[i] *4]

2.3.5 Assembly Language

Machine instructions are just patterns of bits (you could think of them as numbers). In the SM213 ISA we simplify reading and writing of machine instructions by using one or more hex digits to represent each key part of an instruction: opcode and each operand. The instruction `6123` thus performs *opcode 6* (ALU) using function code 1 (add) on registers 2 and 3. Still, machine code is very hard to read, because its just numbers.

Assembly Language is a symbolic (i.e., textual) representation of machine language. Its the way that humans normally read and write machine instructions. There is a direct, one-to-one, correspondence between assembly language instructions and machine instructions. Even though opcodes, operands and memory locations are named by symbol labels, somewhat like in a high-level language, assembly language isn't a high level language at all. It is not *compiled* to yield machine instructions, as Java or C is. Instead, an *assembler* performs a simple, instruction-by-instruction translation from assembly to machine instructions. In fact, this one-to-one property between assembly and machine code means that it is easy to translate in the other direction too. A *disassembler* can translate from machine instructions back in to assembly instructions.

The Gnu C compiler (i.e., gcc) can optionally generate an assembly-code file instead of compiling to machine code as it usually does. You select this option by adding the `-S` option when compiling. It simplify reading the generated code you should also ask the compiler to do some basic code optimization by also adding `-O1` or possibly `-O2` to the command line. Thus typing `gcc -O1 -S S1-global-static.c` produces the file `S1-global-static.s` that contains the assembly-code implementation of the C snippet in the native ISA of the machine on which you execute this command. To see IA32 assembly language, for example, you need to be on an Intel processor. To determine the ISA on which you are running, execute the command `uname -a` by tying this string in to a UNIX shell.

The SimpleMachine Simulator is also an *assembler*. You will normally write snippets and programs in its assembly language (based in the MIPS assembly language), either in an external editor or in the simulator itself. The simulator has a powerful gui editor for assembly language designed to facilitate debugging and program understanding. It allows you to write assembly language, add comments to assembly instructions, add symbolic labels to stand in for addresses and to save and restore this information to and from external files. The simulator translates assembly instructions in the fly into machine code (bits/numbers), shows you both in the display and allows you to execute the instructions to see what they do. The simulator also shows you the affect of instruction execution on values stored in CPU registers and in main memory.

In SM213 assembly language, instructions are described by an opcode name following by a set of operands separated by commas. Registers are named with an `r` following the a register number (e.g., `r0`). This is pretty much the same as IA32. The main difference is that IA32 registers have strange names like `%eax`.

Here are SM213 ISA opcode names:

OpCode	**Description**
ld	load from memory
st	store into memory
mov	move between registers
add	add integers
and	bitwise logical and of two 32-bit values
inc	increment an integer
inca	add four to an integer (increment word address)
dec	decrement an integer
deca	subtract four from integer (decrement word address)
not	bitwise compliment (i.e., not) of 32-bit value
gpc	get value of program counter
shr	shift right
shl	shift left
br	unconditional branch
beq	branch when equal to zero
bgt	branch when greater than zero
jmp	unconditional jump
halt	stop processor
nop	do nothing

Here is how the RISC assembler formats the SM213 ISA addressing modes.

Mode	**Format**	**Example**
immediate	`$#`	`ld $v, r0`
register	`r#`	`add r0, r1`
base + displacement	`#(r#)`	`ld -4(r0), r1`
indexed	`(r#,r#, 4)`	`ld (r0,r1,4), r2`

Recall that not all addressing modes can be applied to all opcodes and that in the SM213 ISA the addressing mode is specified by the opcode, not operand mode tags, as would be the case in most ISAs.

The IA32 assembly language and addressing modes are explained in Chapter 3 of Bryant and O'Hallaron (the textbook).

2.3.6 The SM213 ISA

A key strategy for helping you achieve the learning goals of this course is to design the ISA as we go, adding new instructions when necessary to implement new features found in the progression of snippets. We tackle this design problem in stages. At the beginning of a stage, we example a snippet of C code. We then work out how to compile the code into the SM213 instructions we have so far. We will see from time to time that we need a new instruction or addressing mode. If so, we'll consider the design tradeoffs and if it seems to make sense, we'll add the new instruction. So, it is helpful to not see the entire SM213 at once. For this reason, to maintain the suspense, the SM213 ISA is not revealed here. Instead, it is revealed in sections that cover each snippet. The full ISA is presented at the end and in the Appendix.

2.4 Accessing Variables

We classify variables by how the compiler determines their address, Java has three types of variables: static, instance and method locals (and arguments). The compiler knows the address of static variables, it knows that offset to instance variables from the beginning of the object that contains them and it knows the offset to local variables from the scope in which they are declared.

In a similar fashion, we can classify data values (e.g., objects) as being either static or dynamic based on when their values is allocated a store location in memory. In Java static scalar builtin-type variables (i.e., char, byte, short, int, long, double) have static storage. All other program value storage is allocated dynamically in Java, including static arrays.

Like Java, C has three types of variables: static, struct-members and procedure locals. Unlike Java, however, C allows arrays and objects (i.e., structs) to be allocated statically.

This section examines static and instance variables and static and dynamic objects. We save procedure-local and procedure-argument variables until we talk about procedure calls in Section 2.8.

2.4.1 SM213 Instructions for Accessing Variables

We now reveal the instructions the SM213 ISA includes for accessing variables. We'll also throw in the `halt` and `nop` instructions.

OpCode	Format	Semantics	Eg Machine	Eg Assembly
load immediate	`0d--`	$r[d] \leftarrow v$	`0100`	`ld $0x1000,r1`
	`iiiiiiii`		`00000100`	
load base	`1osd`	$r[d] \leftarrow m[o \times 4 + r[s]]$	`1123`	`ld 4(r2),r3`
load indexed	`2sid`	$r[d] \leftarrow m[r[s] + r[i] \times 4]$	`2123`	`ld (r1,r2,4),r3`
store base+dis	`3sod`	$m[o \times 4 + r[d]] \leftarrow r[s]$	`3123`	`st r1,8(r3)`
store indexed	`4sdi`	$m[r[d] + r[i] \times 4] \leftarrow r[s]$	`4123`	`st r1,(r2,r3,4)`
halt	`f000`		`f000`	`halt`
nop	`ff00`		`ff00`	`nop`

2.4.2 Static Variables

In Java static variables are declared by adding the keyword `static` before the type of the object. These variables are associated with the *class* that contains them and not the objects that are instances of that class. A class, for example, that contains the declaration `static Foo f;` stores one copy of the variable `f` shared by all instances of the class.

There are two ways to declare static variables in C. First, any variable declared in the global scope, outside of any procedure declaration, is static. Second, any variable whose type declaration is preceded by the keyword `static` is also static, even if it is declared within the scope of a procedure. There is exactly one copy of each static variable for the entire program. The C compiler allocates space for static variables at compile time, thus assigning each a static memory address. The compiler knows that address of all static variables. When it generates code for accessing a static variable, it inserts its static address as a constant in the accessing machine instructions. The compiler's job is thus to keep track of where all static variables are in memory. The compiler is in complete control of the layout of static information in memory. Once the program is compiled, no information about where static variables are need remain in the program, other that as constant numbers in the instructions that access them. Only the compiler need know the name and type of variables. In fact, however, for debugging purposes C compilers usually insert a table, called a *symbol table* in a part of the executable file they create. This information, however, **is not used to execute the program** it is simply there so that humans can examine variables by their name when debugging.

A static variable can store either a value or a *pointer* or *reference* to a value. In the former case, the value too is static, in the later case the value may be dynamic. A value is *dynamic* if the memory location that stores it is allocated when the program is running and not by the compiler.

In Java, only scalar values can be allocated statically; all arrays, even those accessed by static variables are allocated dynamically. In C, both scalar and arrays can be either static or dynamic. Let's look first at static scalars and arrays.

Static Scalars and Arrays

From `S1-global-static.java`, here is an example of the declaration and use of a static scalar in Java.

```
public class Foo {
  static int a;

  public void foo () {
    a = 0;
  }
}
```

Only C has static arrays. From the same snippet, here is a example of the declaration and use of a static scalar and array in C.

```
int a;
int b[10];

void foo () {
  a = 0;
  b[a] = a;
}
```

We now focus our attention on the machine code generated by a C compiler for `S1-global-static.c`. First, the compiler allocates four bytes to store the value of `a` and 40 bytes to allocate the values of the array `b[10]`. Let's assume that the compiler locates `a` at address `0x1000` and `b[10]` at address `0x2000`.

Here is an outline of machine code that executes the statement `a=0;`.

1. Copy the constant `0` into a register, say r0.
2. Copy the address constant `0x1000`, the address of `a`, into a register, say 0.
3. Store the value in register r0 into memory at the address in register r1.

From S1-global-static.machine, here is the SM213 machine code that implements this statement.

```
0000 00000000   # r0 = 0
0100 00001000   # r1 = address of a
3001            # a = 0
```

And finally here it is in assembly language (S1-global-static.s).

```
.address 0x100
                 ld   $0x0, r0           # r0 = 0
                 ld   $a, r1             # r1 = address of a
                 st   r0, 0x0(r1)        # a = 0
```

Notice that in addition to the three instructions execute by the CPU there is an additional line at the beginning of the assembly language file (i.e., `.address 0x100`). Operation names starting with a dot, like this, are *assembler directives* that exist to provide the assembler with information but that do not result in machine code. In this case, the `.address` directive tells the assembler where in memory to store the next instruction, in this case `0x100`.

If the compiler (or you) wishes provide initial values to the variables a and or b, assembly language provides a directive `.long` to do so. Keep in mind that this step is completely optional. It doesn't *allocate* the storage it simply assigns values to some parts of memory when the program loads — before it starts executing — that is, *statically*. Question: how are *static* variables *allocated*?

Here is the assembly code to statically initialize a and b to -1. Doing so is useful for debugging so that you can see if and when the 0 is stored in a by the snippet. Normally, compilers initialize variables to 0 (or don't initialize them) unless the high-level language program provides a static initializer (e.g., $a = -1$).

```
.address 0x1000
a:               .long 0xffffffff        # a
.address 0x2000
b:               .long 0xffffffff        # b[0]
                 .long 0xffffffff        # b[1]
                 .long 0xffffffff        # b[2]
                 .long 0xffffffff        # b[3]
                 .long 0xffffffff        # b[4]
                 .long 0xffffffff        # b[5]
                 .long 0xffffffff        # b[6]
                 .long 0xffffffff        # b[7]
                 .long 0xffffffff        # b[8]
                 .long 0xffffffff        # b[9]
```

Here is an outline for the array access in the second statement, `b[a]=a;`. We treat this statement entirely on its own, ignoring anything the compiler learned from generating the first statement. This is precisely what the initial *pass* of a compiler typically does: treat every statement as an island. Subsequent, *optimization passes* of the compiler, however, look to improve the generated code by using results computed in one instruction in later instructions etc., if possible. We will not optimize our code.

1. Copy the constant `0x2000`, the address of `b[10]`, into a register, say r0.
2. Copy the constant `0x1000`, the address of `a`, into a register, say r1.
3. Load the value in memory at the address stored in r1 into a register, say r2. This loads the value of `a` into r2.
4. Store the value in r2 into memory at the address that results from adding the value in register r0, the *base* of the array `b [10]` to *four times* the value in register r2, the *index* into the array.

This style of access, called *base plus index*, is very common in machine code. One register stores the base address of an object. Another register, or perhaps a constant, is an index into that array. The byte offset of the indexed value from the beginning of the object is determined by multiplying the index by the size of the things that it indexes; in this case this is an array of 4-byte `ints` and so we multiply by four. The memory address of the target value is the sum of the object's base and its offset.

For example, consider the case where `a=2`. The base address of the `b[10]` array is `0x2000` and so the address of `b[0]` is also `0x2000`. Thus the address of `b[1]` is `0x2004` and the address of `b[2]` is `0x2008` which is `0x2000` plus four times 2.

Again from `snippet1.gold`, here is the SM213 machine code that implements this statement.

```
0000 00002000  # r0 = address of b
0100 00001000  # r1 = address of a
1012           # r2 = a
4220           # b[a] = a
```

Notice that this code uses the *store indexed* SM213 instruction, `4220`, which multiplies the value in r2 (the second operand) by four and address that to the value in r0 (the third operand) to get the memory address here it stores the value of r2 (the first operand).

Finally, here is the assembly code.

```
ld   $b, r0              # r0 = address of b
ld   $a, r1              # r1 = address of a
ld   0x0(r1), r2         # r2 = a
st   r2, (r0, r2, 4)     # b[a] = a
halt                     # halt
```

2.4.3 Dynamic Arrays

In Java, all arrays are allocated dynamically. C has dynamic arrays too.

From `snippet1b.java` here is an example of declaring, allocating and accessing a static array in Java.

```
public class Foo {
    static int a;
    static int b[] = new int[10];

    void foo () {
      a=0;
      b[a]=a;
    }
  }
```

From `S2-global-dyn-array.c` here is the equivalent C code.

```
int  a;
int* b;

void foo () {
  a = 0;
  b = (int*) malloc (10*sizeof(int));
  b[a] = a;
}
```

In Java the array is allocated dynamically with the statement `b[] = new int[10];` executes. In C the array is allocated by the statement `b = (int*) malloc (10*sizeof(int));`.

Notice that C syntax for access static and dynamic arrays is exactly the same: `b[a]=a;` in this case. The code the compiler generates, however, is different for the two cases. They key difference is that when the array `b[10]` is allocated dynamically, the compiler does not know its address. Instead it knows that address of the variable `b`, which stores the address of the array once it is created. In C terminology, we say that `b` stores a *pointer* to the array.

Here is an outline of the machine code that implements `b[a]=a;` when `b[10]` is allocated dynamically and the static variable `b` stores a pointer to it. Compare this outline to the outline for static arrays given above. You should be able to explain what parts are different, what parts are the same, and why.

1. Copy the constant `0x2000`, the address of `b[10]`, into a register, say r0.
2. Load the value in memory at the address stored in r0 into a register, say r0.
3. Copy the constant `0x1000`, the address of `a`, into a register, say r1.
4. Load the value in memory at the address stored in r1 into a register, say r2. This loads the value of `a` into r2.
5. Store the value in r2 into memory at the address that results from adding the value in register r0, the *base* of the array `b [10]` to *four times* the value in register r2, the *index* into the array.

Here is the SM213 machine code for this statement. Again, compare this carefully to the code for the static case.

```
0000 00002000  # r0 = address of b, which stores pointer to array
1000           # r0 = value of b, the pointer to the array
0100 00001000  # r1 = address of a
1012           # r2 = a
4220           # b[a] = a
```

Finally, here is the SM213 assembly code.

```
.address 0x100
                 ld   $0x0, r0            # r0 = 0
                 ld   $a, r1              # r1 = address of a
                 st   r0, 0x0(r1)         # a = 0
                 ld   $b, r0              # r0 = address of b (which stores address
                 ld   0x0(r0), r0         # r0 = pointer to array
                 ld   $a, r1              # r1 = address of a
                 ld   0x0(r1), r2         # r2 = a
                 st   r2, (r0, r2, 4)     # b[a] = a
                 halt                     # halt
.address 0x1000
a:               .long 0xffffffff         # a
.address 0x2000
b:               .long 0x00003000         # address of b[0]; loaded dynamically
.address 0x3000
b_data:          .long 0xffffffff         # b[0]
                 .long 0xffffffff         # b[1]
                 .long 0xffffffff         # b[2]
                 .long 0xffffffff         # b[3]
                 .long 0xffffffff         # b[4]
                 .long 0xffffffff         # b[5]
                 .long 0xffffffff         # b[6]
                 .long 0xffffffff         # b[7]
                 .long 0xffffffff         # b[8]
                 .long 0xffffffff         # b[9]
```

2.4.4 Instance Variables

Java instance variables (also called data members) are allocated as part of an object (i.e., in the instance of the class in which they are defined). Before a program starts running, there are no objects and thus there are no instance variables. As the program runs, it allocates objects on the heap and the instance variables reside within them. If the program creates one thousand instances of a class `Foo` that declares an instance variable `int i;`, the program also creates one thousand instances of the variable `i`, one as part of each object.

In order for the machine to access any variable it needs to possess its memory address. The machine instructions that implement such an access are generated by the compiler. Let us again ask the question, what does the compiler know about the address of these variables?

In the case of the static variables, we have just examined, the compiler knows the address of the variables and can thus hard-code this address in the instruction it generates to implement access to them. The address of instance variables, however can not possibly be known by the compiler, because they are contained within an object that is allocated dynamically (i.e., at runtime). In fact the machine code for accessing the variable `i` in the example above must work for all one thousand copies of the variable and so it clearly can not hard code the address of `i`.

Instance variables are thus *dynamic variables*, because their address is not know until runtime.

There *is*, however, something about instance variables that *can* be known statically: their position within the object that contains them. The compiler is in complete control of how it formats objects when they are allocated. As long the compiler knows the size of every instance variable assigned to a class, it can know statically how far each will be from the beginning the object that contains them. It can thus hardcode these offsets in the instructions it generates for accessing the variable. Thus, while it does not know the address of `i`, it can know how far `i` is from the beginning of its containing object. If it can generate code to load the address of this object into a register, it can then access the variable using a static offset from this address, hardcoded into the memory-access instruction. In the SM213 ISA the instructions `1xxx` (load base+displacement) and `3xxx` (store base+displacement) exist precisely to implement this style of access.

Instance Variables in C

C does not have objects or classes. It does, however, allow a set of variables to be allocated, deallocated and accessed as group. In C this grouping is called a *struct*. Structs are very much like Java classes. There are two differences. First, unlike classes, structs have no associated methods; in C, the struct just defines the instance variables. Second, unlike classes, structs can be allocated either statically or dynamically. In Java, all objects are allocated dynamically.

For example, consider the following Java class with two public instance variables `e` and `f`.

```
public class D {
  public int e;
  public int f;
}
```

In C a similar struct type can be created like this:

```
typedef struct {
  int e;
  int f;
} D;
```

In Java there is only one way to declare variables of type `D`; these variables store a reference to objects of type D (or objects that implement the interface of class D).

```
D d = new D ();
```

In C, however, both static and dynamic declarations of instances of struct `D` are allowed. In this example `d0` is static and `d1` is dynamic.

```
D  d0;
D* d1;
```

Variables of type `D` statically allocate the object. Subsequent access to instance variables looks like the Java syntax: `D.e` and `D.f`. Accesses to these objects are like accesses to static scalars and arrays. The compiler knows the address of instance variables of these classes, because it statically allocates the object that contains them. There is no equivalent construct in Java.

To declare a variable to store a reference to instances of struct `D`, which can then be dynamically allocated, a C program declares their type to be `D*`. Subsequent access to instance variables using struct pointers uses a different syntax from their static variant and from Java: `D->e` and `D->f`. These references are the ones that are equivalent to the Java accesses. Be careful to notice that the C syntax that matches Java is for a type of access not possible in Java. The equivalent accesses in the two languages use different syntax.

Access from Outside Variable's Class

One way to access an instance variable in Java is to declare the variable `public` and access it from outside of the class that stores it. This style of access has two parts: a variable that stores on object reference and a name of a public instance variable in that object. For example, accessing variable `i` of the object referred to by variable `a` is expressed as `a.i`. At this point you are likely thinking it isn't good programming practice to expose instance variables in a class's public interface. You are right. It typically far better to access them through methods of that class, hiding the variable *behind* the class's interface. However, this outside-of-the-class access is a bit simpler to understand and it is how it is done in C, so let's start here.

Here is the Java code of such an access taken from `S4-instance-var.java`.

```
public class D {
  public int e;
  public int f;
}

public class Foo {
  static D d = new D ();
  public void foo () {
    d.e = d.f;
  }
}
```

And the corresponding code in C, taken from `S4-instance-var.c`. For completeness, this snippet includes both the static and dynamic versions of an instance of D.

```
typedef struct {
  int e;
  int f;
} D;

D  d0;
D* d1;

void foo () {
  d1 = (D*) malloc (sizeof(D));
  d0.e = d0.f;
  d1->e = d1->f;
}
```

We begin with the static case, which is very similar to the access to the element of a static array. The difference is that an array is a list of elements, all of which have the same type and that are named by a numeric index into the list. The struct (object), on the other hand, is a list of elements, each of which can be of a different type nd that are named by a variable name. In the case of the array, the element selector can be a dynamic value (e.g., `a[i]` where the value of `i` is determined at runtime). In the case of a struct, however, the element selector is always static: the name of a variable that the compiler turns into a constant offset from the beginning of the struct.

The compiler is responsible for laying out static data in memory. Here is how it might layout the static variables `d0` and `d1`, this is taken from `snippet3.gold`.

```
2000: 00000001      # d0.e
```

```
      00000002       # d0.f
3000: 00000000       # d1
```

At runtime, the execution of the procedure-call statement `d1 = (D*) malloc (sizeof(D));` will allocate a `D` object somewhere in the heap, say at address 0x4000, and place that address into memory at address 0x3000, the location of the variable `D1`. Also, the four instance variables in this example are assigned unique values (i.e, 1–4) to simplify debugging. In reality, static variables in C (i.e., `d0.e` and `d0.f` are initialized to zero and dynamic variables (i.e., `d1->e` and `d1->f)` are not initialized as so have the value of whatever was stored there previously. So, at the point in the execution just before the execution of the `d0` assignment, here is what the data part of memory looks like.

```
2000: 00000001       # d0.e
      00000002       # d0.f
3000: 00004000       # d1
4000: 00000003       # d1->e
      00000004       # d1->f
```

Let's first look at an outline of code the compiler would generate to implement the access to the static struct, the statement `d0.e = d0.f;`.

1. Copy the constant 0x2000, the address of d0, into a register, say r0.
2. Read the value in memory at the address in r0 plus 4, the address of `d0.f`, into a register, say r1.
3. Store the value in r1 into memory at the address in r0 plus 0, the address of `d0.e`.

From `S4-instance-var.gold`, here is the SM213 machine code that implements this statement.

```
0000 000200  # r0 = & d0
1101   # r1 = d0.f
3100   # d0.e = d0.f
```

Here is the SM213 assembly code, including initialization of data elements.

```
.address 0x100
                 ld   $d0, r0             # r0 = & d0
                 ld   0x4(r0), r1         # r1 = d0.f
                 st   r1, 0x0(r0)         # d0.e = d0.f
.address 0x2000
d0:              .long 0x00000001         # d0.e
                 .long 0x00000002         # d0.f
```

Now let's look at the access to the dynamically allocated instance of `D` using pointer `d1` in the statement `d1->e = d1->f;`.

Here's an outline of the code.

1. Copy constant 0x3000, the address of d1, into a register, say r0.
2. Read the value in memory at the address stored in r0, the value of d1, into a register, say r0.
3. Read the value in memory at the address in r0 plus 4, the address of `d1->f`, into a register, say r1.

4. Store the value in r1 into memory at the address in r0 plus 0, the address of `d1->e`.

Notice that the only difference between access to the static and dynamic instances is the addition of step 2 for the dynamic access. This is precisely what we say for static and dynamic arrays. In the dynamic case, there is one extra step required, read the value of the pointer variable from memory.

From `S4-instance-var.gold`, here is the SM213 machine code that implements this statement.

```
0000 00003000 # r0 = & d1
1000          # r0 = d1
1101          # r1 = d1->f
3100          # d1->e = d1->f
```

And here is assembly code. In this example we assume that the object pointed to by `d1`, which would be created dynamically, has already been created and resides at address `0x4000`. Don't be confused by this apparently-static allocation of a dynamic object. We do it this way *only* to allow you to see the code that accesses the dynamically-allocated object without having an implementation of the C dynamic allocator `malloc ()` at hand. Even though this assembly code clearly does statically assign the object to address `0x4000`, we must pretend that object is allocated dynamically and that the compiler could not know the address `0x4000` when it generates the code for accessing `d1->e` or `d1->f`, that is for *de-referencing* the pointer variable `d1`.

```
                 ld   $d1, r0            # r0 = & d1
                 ld   0x0(r0), r0        # r0 = d1
                 ld   0x4(r0), r1        # r1 = d1->f
                 st   r1, 0x0(r0)        # d1->e = d1->f
                 halt
.address 0x3000
d1:              .long 0x00004000        # d1
.address 0x4000
d1_data:         .long 0x00000003        # d1->e
                 .long 0x00000004        # d1->f
```

Access from Member Method of Variable's Class

In Java, but not C, instance variables can be accessed from a method of the class that contains the instance variable. The access in this case simply names the instance variable, leaving off the `a.`, used in the out-side-the-class case described above.

For example, here is this style of access in `S4-instance-var.java`.

```
public class D {
  int e;
  int f;
  public void foo () {
    e = f;
  }
}
```

The code for implementing the statement `e = f;` in this case is exactly the same as the dynamic access to `d1` above with one important difference. In the previous example, the address of the object that contains the instance variables `e` and f is stored in a static variable (i.e., d1). The compiler knows address of this variable and can thus generate code to access it by encoding this address as a constant in the first instruction it generate. In the current case, however, the address of the containing object is implicit. This is implemented by a hidden variable, in Java called this, that stores the address of the current object, when executing an instance method (i.e., a method that is not static). But, where is the this variable stored? How can the compiler access it?

One answer is that Java and other object-oriented languages such as C++ can reserve a general purpose register to store the address of the current object. This decision is arbitrary and its entirely up to the compiler to decide. For example, the compiler might decide to reserve r7 for this purpose. Really this hidden variable / reserved register is just a hidden argument of the method containing the access and is handled much the same as other arguments. More about arguments later.

To invoke a method on a object, the compiler generates code something like this.

1. Save the current value of r7 in memory, in a special region called the *runtime stack*.
2. Put the address of the object being invoked into r7.
3. Invoke the method.
4. Restore the old value of r7 by reading it back from memory.

Section 2.8 describes method invocation and the stack in detail.

Using r7 to store the value of this, the outline for implementing `e = f;` is a variant of the code for accessing d1 above, leaving out the first two step.

1. Read the value in memory at the address in r7 plus 4, the address of this.f, into a register, say r1.
2. Store the value in r1 into memory at the address in r7 plus 0, the address of this.e.

The SM213 machine code for this statement is:

```
1171          # r1 = this.f
3107          # this.e = this.f
```

And here is the disassembly comments generated by running the simulator

```
ld  0x4(r7), r1
st  r1, 0x0(r7)
```

2.4.5 Java References and C Pointers

A variable that stores the address of a program value is called a *reference* in Java and a *pointer* in C. In Java everything but the scalar base types are references. C, on the other had provides the ultimate flexibility in dealing with pointers and selecting between static and dynamic allocation.

C provides two different three types of syntax for dereferencing pointers: `*`, `[]`, and `->`.

The `&` operator can be applied to any variable in C to refer to the address of that variable. Consider the code below, for example.

```
        int i;
        int *ip = &i;
```

The variable `ip` is a pointer to an integer. Assigning it the value `&i` points it to the value stored in the variable `i`. A subsequent assignment `*ip=1`, therefore changes the value of `i` to one.

2.5 Dynamic Allocation and Deallocation

A key difference between C and Java is the way they handle dynamically created objects and the variables that store references to them.

2.5.1 Pointers and References

Read Bryant and O'Hallaron § *3.8.2 and 3.11.*

2.5.2 Allocating and Deallocating Dynamic Objects

Bryant and O'Hallaron § *10.9* describes allocation and deallocation in C. It talks about the *heap*, the location in memory from which dynamic objects are allocated and the implementation of `malloc ()` and `free ()` the procedures that handle dynamic allocation and deallocation in C. There is much more detail here about those implementations and the issues surrounding them than you are required to know. What you need to know is what the heap is, what a simple allocator does, how the problem of fragmentation can arise, and roughly- speaking how this problem can be tackled by the allocator.

Dynamic allocation in Java occurs when a program executes the *new* statement. Like C, Java allocates dynamic objects from a heap. Unlike C, Java does not have an operation that explicitly frees objects so that their memory space can be reused. Instead, a background process called a *garbage collector* runs periodically to determine which objects can not possibly be referenced again by the program. It automatically frees these objects and can also compact the rest of the objects to eliminate heap fragmentation.

2.5.3 Type Safety and Explicit Deallocation

Java is a *type safe* language, which means that the language *guarantees* that an object can never be used in a way that is not allowed by its type. Java allows programs to *type cast* an object to change its type by placing a type name in parentheses in front of an object reference, but it only allows casts that are permitted by the type system. The rule, called *co-variance*, states that you can cast objects only to types that those objects implement. Sometimes Java can enforce this type rule statically and can thus generate a compile-time error if a cast is not permitted. Other times, Java must insert a runtime type check that generates a runtime exception when violations occur.

For example, consider the following type declaration.

```
        public class Parent {
        void p();
        }
        public class Child {
        void c();
        }
```

The following are all permitted, because objects of type `Child` implement everything in the interface of class `Parent`. It is thus fine to have a pointer of type `Parent` refer to an object of type `Child`. Furthermore, if a pointer of type `Parent` actually stores an object of type `Child`, it is perfectly fine to cast that pointer back to a `Child` and perform `Child` operations on it; this form of type cast, called a *downcast*, however, requires a runtime check to ensure validity, because the actual type of objects referred to by a variable are not know until runtime.

```
Parent p = (Parent) new Child ();
((Child) p).c ();
```

Explicit deallocation is not easily compatible with type safety. The problem is that if an object is deallocated while the program still holds a reference to it in a variable and then that object's memory space is subsequently re-allocated to store another object, the original pointer can be used to access the new object. This bug is called the *dangling pointer problem* and it is covered in more detail in the next section. Now, if the old pointer and the new objects are if different type, this bug is also an un-caught type-safety violation, because the old pointer access the new object as if were the type of the object that used to reside at that location.

C programs explicitly deallocate objects and thus C suffers from the dangling pointer problem and the lack of type safety it entails. C is not type safe in other ways. A big way is that C performs virtually no type checks when programs cast an object from one type to another. The only restriction in C is that the new and object types be the same size. This check is extremely weak, however, because all pointers are the same size (4 bytes in a 32-bit processor). So, in C a pointer can be type cast into a pointer of any type. Even worse, is allowed to cast integers to pointers and vice versa. And so, a program can put an arbitrary number in an integer, cast it to a pointer and then write into memory at this location.

2.5.4 The Dangling Pointer Problem and Other Bugs With Using Pointers in C

Read Bryant and O'Hallaron § *10.11.1–10.11.9.*

2.5.5 Memory Leaks

Read Brant and O'Hallaron § *10.11.10* to see what a memory leak is and how you can create one in C.

Despite the fact that Java reclaims *garbage* objects automatically, it is still possible (easy, actually) to accidentally create memory leaks in Java by retaining a references to objects that are no long needed by a program. An example of a common error occurs when one uses a data structure such as a hash table to index a set of objects. Even when all other references to these objects have been released by the program, the objects remain allocated in the heap, because the hash table retains a reference to all of them.

It is thus as important in Java as it is in any language supports dynamic allocation to design for object lifetime and to take care that the program does retain references to objects past their planned lifetime. Good programming practices include (1) setting pointer (reference) variables to `null` when the object they point to is no longer needed, (2) remove objects from indexing data structures and other collections when they are no longer needed or (3) use Java's *WeakReference* type to store references in a hash table, for example, without inhibiting the garbage collector from reclaiming them when no normal (i.e., non-weak) references exist. You'll notice, for example, that collection objects have a `remove ()` method. This is really very much like and explicit delete. The key difference between this form of delete in Java and calls to the `free ()` procedure in C is that Java's approach is type safe. The remove object isn't reclaimed and thus the space it occupies is not reused until all references to the object are gone from the program. The call to `remove ()` simply removes one of the references to the object.

The bottom line is that Java programmers are responsible for explicitly removing references to objects, but not reclaiming the objects themselves. In C, programmers have the more difficult and dangerous responsibility explicitly reclaiming objects. You should be able to compare these two approaches and to point out at least one one advantage of each.

2.5.6 Garbage Collection

Read Bryant and O'Hallaron § 10.10.

2.6 ALU Instructions

The ALU performs basic math and bit-logical operations. Every instruction set includes a number of operations that are implemented by the ALU, these are commonly called the *ALU instructions*. In RISC ISAs, ALU instructions are not permitted to access memory. Instead their input and output operands are registers, or sometimes constants encoded in the instruction. In the SM213 ISA, operands of ALU instructions are all registers, with the exception of the shift operations, which use a constant in the instruction to specify the number of bits to shift.

2.6.1 SM213 Instructions for Doing Math

OpCode	Format	Semantics	Eg Machine	Eg Assembly
rr move	`60sd`	$r[d] \leftarrow r[s]$	`6012`	`mov r1, r2`
add	`61sd`	$r[d] \leftarrow r[d] + r[s]$	`6112`	`add r1, r2`
and	`62sd`	$r[d] \leftarrow r[d] \,\&\, r[s]$	`6212`	`and r1, r2`
inc	`63-d`	$r[d] \leftarrow r[d] + 1$	`6301`	`inc r1`
inc addr	`64-d`	$r[d] \leftarrow r[d] + 4$	`6401`	`add $4, r1`
dec	`65-d`	$r[d] \leftarrow r[d] - 1$	`6501`	`dec r1`
dec addr	`66-d`	$r[d] \leftarrow r[d] - 4$	`6601`	`add $-4, r1`
not	`67-d`	$r[d] \leftarrow !r[d]$	`6701`	`not r1`
shift	`7dss`	$r[d] \leftarrow r[d] << s$	`7102`	`shl $2, r1`
			`71fe`	`shr $2, r1`

2.7 Controlling Program Flow

All control flow changes in ISA are like GOTO <memory address>.

Some instructions only goto the address if a certain condition holds, for example, if a register value is 0, while other instructions jump unconditionally.

Some instructions specify the target instruction address using a 32-bit constant, while others specify an offset to add to the current program counter value. Absolute-address jumps are 6-byte instructions. The advantage of pc-relative addressing is that jumps instructions are only 2 bytes. We us the term *jump* to refer to instructions that use absolute addresses and *branch* to refer to instructions that use pc- relative addressing. Using pc-relative addressing, the address of the next instruction is typically the value of the current pc minus 2 plus two times the signed offset in the instruction. The minus two is because when an instruction address X is executing the value in pcReg has already been advanced by the fetch stage to $X + 2$. The times-two part is because instructions are at minimum two bytes. The constant in the instruction is thus an index, not an offset, into the instructions. If the calculation were instead pcReg = pcReg - 2 + offset, then odd offsets would never be used, and we would thus waste valuable space in the instruction.

Finally, some jump instructions specify the target jump address statically with a constant hard-coded in the instruction, while others specify the address dynamically, getting the target address from a register or from memory. Static control flow is used for loops, if-then-else and static procedure calls. Dynamic control flow is used for switch statements, return statements and dynamic method dispatch in languages like Java. This section describes both approaches, starting with the static flavour.

2.7.1 SM213 Instructions for Static Control Flow

We now reveal the instructions the SM213 ISA includes static-control-flow manipulation. Please do not read this until after the in-class exercise where your team designs its own. Please do not feel constrained to model your ISA after the SM213 ISA. Divergence from the SM213 ISA **is encouraged**. You ISA does not need to be prefect; it is a learning tool. And, there is not a *right answer* when it comes to ISA design. Some designs are better than others, of course, but for certain differences such as the choice between CISC and RISC, reasonable people can and do differ.

OpCode	Format	Semantics	Eg Machine	Eg Assembly
branch	`8-oo`	$\text{pc} \leftarrow \text{pc} - 2 + 2 \times o$	`1000: 8004`	`br 0x1008`
branch if equal	`9roo`	if $r[r] == 0,\ \text{pc} \leftarrow \text{pc} - 2 + 2 \times o$	`1000: 9104`	`beq r1, 0x1008`
branch if greater	`aroo`	if $r[r] > 0,\ \text{pc} \leftarrow \text{pc} - 2 + 2 \times o$	`1000: 9104`	`bgt r1, 0x1008`
jump	`b---` `aaaaaaaa`	$\text{pc} \leftarrow a$	`b000` `00001000`	`jmp 0x1000`

2.7.2 for loops

Let's look at a for loop that computes the sum of an array of numbers. We now want to focus just on the control flow, so we'll try to keep everything else as simple as possible. Thus, for example, all of the variables the code uses are static.

Summing numbers in Java from `S5-loop.java`.

```
public class Foo {
  static int s = 0;
  static int i;
  static int a[] = new int[10];

  static void foo () {
    for (i=0; i<10; i++)
      s += a[i];
  }
}
```

Summing numbers in C from `snippet4.c`.

```
int  s=0;
int  i;
int a[] = {2,3,4,6,8,10,14,16,18,20};

void foo () {
  for (i=0; i<10; i++)
    s += a[i];
}
```

Notice that in these examples the number of times the loop executes is actually known statically, by the compiler. In this case, the compiler might actually generate machine code that implements this loop by not using any control-flow instructions. It might instead *unroll* the loop like this:

```
s += a[0];
s += a[1];
s += a[2];
s += a[3];
s += a[4];
s += a[5];
s += a[6];
s += a[7];
s += a[8];
s += a[9];
```

If fact this strategy is just want the compiler would do if it was optimizing the machine code it generates to execute as quickly as possible, though at some sacrifice in code size.

However, we choose static values for the array size and thus the number of iterations of the loop just to keep the variable access simple. Our interest here is in generating code for loops where the number of loop executions is not know statically by the compiler and thus control-flow instructions are required.

Deconstructing the loop

The only control flow instructions provided by hardware are of the form `goto <address> if <condition>`. So, the first step in generating code for high-level-programmin-language control-flow construct such as the for loop is to deconstruct to use only gotos.

The loop show above can be deconstructed like this several pieces like this.

1. Set the initial value of the induction variable `i=0`;
2. Test the loop condition `i<10` and goto to the first statement after the end of the loop, step 6, if the condition is false.
3. Execute the body of the loop
4. Increment the induction variable `i++`.
5. Goto step 2.
6. Continue with code after the loop.

C actually has a goto statement and labels for goto targets. Its accepted wisdom that are considered to be harmful, thanks to Edgser Dijkstra's 1968 paper "Go To Statement Considered Harmful", so don't use C's gotos. Having said that, let's deconstruct the for loop above into C using gotos.

```
int i = 0;
loop: if (i>=10) goto done;
```

```
        s += a[i];
        i++;
        goto loop;
        done:
```

The translation of this code into machine instructions is now quite straightforward with the exception of the condition test `if (i<10) goto body;`. Typically of RISC ISAs, the SM213 ISA's condition branches test the value of a register and branch if the register is equal to zero, in one case, and if it is greater than zero, in another. Real RISC ISAs include other tests, like less than zero, less than or equal to etc., but they typically do to have tests that compare the value of two registers or that compare the value of a register to a constant other than zero. In this case, we can have `i` in a register and would like to compare it to 10, but we can't. We can only compare it to zero. So, how can we express a comparison with 10 as a comparison with 0? The answer is to subtract 10 from `i` and look at the result.

We can rewrite the conditional branch above as follows.

```
        loop: j=i-10;
        if (j==0) goto done;
```

SM213 Code Example

Before giving the SM213 assembly code that implements this loop, let's decide that the compiler places the variables starting at address `0x1000` like this.

```
        .address 0x1000
        s:                .long 0x00000000         # s
        i:                .long 0x00000000         # i
        a:                .long 0x00000002         # a[0]
                          .long 0x00000004         # a[1]
                          .long 0x00000006         # a[2]
                          .long 0x00000008         # a[3]
                          .long 0x0000000a         # a[4]
                          .long 0x0000000c         # a[5]
                          .long 0x0000000e         # a[6]
                          .long 0x00000010         # a[7]
                          .long 0x00000012         # a[8]
                          .long 0x00000014         # a[9]
```

The SM213 assembly code for this loop, found in `snippet4.gold` is here.

Initialization

```
        .address 0x100
                          ld   $0x0, r0            # r0 = 0 (temp for i)
                          ld   $a, r1              # r1 = address of a[0]
                          ld   $0x0, r2            # r2 = 0 (temp for s)
                          ld   $0xfffffff6, r4     # r4 = -10
```

Loop Condition

```
        loop:             mov  r0, r5              # r5 = r0
                          add  r4, r5              # r5 = r0-10
                          beq  r5, end_loop        # if r0=10 goto +4
```

Loop Body
```
                      ld   (r1, r0, 4), r3      # r3 = a[r0]
                      add  r3, r2               # r2 += a[r0]
                      inc  r0                   # r0++
```

Goto the Top
```
        80fa              # goto -6
```

After the Loop
```
end_loop:             ld   $s, r0               # r0 = address of s
                      st   r2, 0x0(r0)          # s = r2 (temp for s)
                      halt
```

Notice that the branch instructions use a symbol to specify the branch-target address. Use the simulator to examine the machine code for these instructions. You will see that the machine code contains the PC offset (i.e., +4 or -7) that when added to the current PC yields the branch-target address. They symbols here are simply a convenience of the assembler (a bit one, because otherwise you'd need to compute the difference between address of the branch and the target to write the instruction. And, what would happen if you added or deleted (or changed the size of) an instruction between the branch and the branch target? Try it in the simulator and see what happens the the target offset.

2.7.3 if statements

If statements are a simple reapplication of the control-flow constructs used to implement for loops above.

Deconstructing the if statement

Once again, we can deconstruct if statements by using gotos, as follows.

1. If loop condition is true, goto "then" part (step 4).
2. Do the "else" part.
3. Goto to first statement following if statement (step 5).
4. Do the "then" part.
5. Continue with statement that follows if statement

Let's pick an example that computes the maximum of two integers. For simplicity, once again, variables are statics. Also, we'll skip the Java snippet this time, because the Java and C are virtually identical (other than the variable declarations and the lack of enclosing class declaration.

Here is `S6-if.c`.

```
int a=1;
int b=2;
int max;

void foo () {
  if (a>b)
    max = a;
  else
    max = b;
}
```

This if statement can be deconstructed to use C gotos as follows.

```
j = a-b;
if (j>0) goto thenp
max = b;
goto done;
thenp: max = a;
done:
```

SM213 Code Example

Let's assume that the compiler chooses to layout variables like this.

```
.address 0x1000
a:                .long 0x00000001         # a
.address 0x2000
b:                .long 0x00000002         # b
.address 0x3000
max:              .long 0x00000000         # max
```

Here is the SM213 code that impelements this if statement, taken from `snippet5.gold`.

Load a & b

```
0000 00001000   # r0 = address of a
1000            # r0 = value of a
0100 00002000   # r1 = address of b
1011            # r1 = value of b
```

Test a>b

```
.long.address 0x100
                  ld   $a, r0              # r0 = address of a
                  ld   0x0(r0), r0         # r0 = value of a
                  ld   $b, r1              # r1 = address of b
                  ld   0x0(r1), r1         # r1 = value of b
                  mov  r1, r2              # r2 = value of b
                  not  r2                  # complement r2
                  inc  r2                  # r2 = -(value of b)
                  add  r0, r2              # r0 = a-b
                  bgt  r2, then            # if (a>b) goto +2
```

Else Part

```
else:             mov  r1, r3              # r3 = b
                  br   end_if              # goto +1
```

Then Part

```
then:             mov  r0, r3              # r3 = a
```

After If
```
end_if:          ld   $max, r0            # r0 = address of max
                 st   r3, 0x0(r0)         # max = r3
                 halt
```

2.7.4 SM213 Instructions for Dynamic Control Flow

Some high-level language features can not be implemented with static control flow instructions. As we will see, method return, dynamic method invocation and switch statements all require or benefit from a type of jump instruction in which the target jump address is not determined until runtime. In these cases, the compiler can not know the target address when it compiles the program.

Here now are the SM213 ISA includes dynamic-control-flow manipulation. Once again, please do not read this until after the in-class exercise where your team designs its own. Please do not feel constrained to model your ISA after the SM213 ISA. Divergence from the SM213 ISA **is encouraged**. You ISA does not need to be prefect; it is a learning tool. And, there is not a *right answer* when it comes to ISA design. Some designs are better than others, of course, but for certain differences such as the choice between CISC and RISC, reasonable people can and do differ.

OpCode	Format	Semantics	Eg Machine	Eg Assembly
get program counter	`6f-d`	$r[d] \leftarrow$ pc	`6f01`	`gpc r1`
jump indirect	`croo`	pc $\leftarrow r[r] + o \times 2$	`c102`	`jmp 4(r1)`
jump double indirect, base+displacement	`droo`	pc $\leftarrow m[o \times 4 + r[r]]$	`d102`	`jmp *8(r1)`
jump double indirect, indexed	`eri-`	pc $\leftarrow m[4r[r] + r[i] \times 4]$	`e120`	`jmp *(r1,r2,4)`

2.7.5 Static Method Invocation

Method invocation involves both control flow — jumping to the invoked method and back when the method returns — and data flow — the passing of arguments to the invoked method and of a return value back to the invoking code. In this and the next two sections, we consider *only the control flow* aspect of method invocation. Argument passing is discussed separately in Section 2.8.

Static method methods and procedures can be invoked by static control flow constructs. In Java, a method declared with the `static` keyword preceding its return type is a method of the *class* in which it is declared and not of the objects that are instances of that class. Static methods thus do not have access to instance variables of objects, but only the the static variables defined in the class. A program invokes a static method by putting the class name before the procedure name, separating the two by a period. The compiler uses static control flow instructions to implement these calls. In C, all procedures are static and all direct procedure calls are implemented with static control flow instructions (procedure variables are discussed in Section 2.7.7 below).

Here is an example of a static method call in Java, taken from `S7-static-call.java`.

```
public class A {
  static void ping () {}
}

public class Foo {
  static void foo () {
    A.foo ();
  }
}
```

And in C, taken from `S7-static-call.c`.

```
void ping () {}

void foo () {
  ping ();
}
```

In both cases, the procedure call itself is simply a jump to the address of the first instruction of `ping()`, an address that the compiler knows and that it can thus hardcode in an instruction. For example, if the code for `ping()` starts at address 0x500, the SM213 assembly instruction for direct jump would be.

```
j 0x500       # goto 500
```

Or, using a symbol for the address.

```
j ping
```

Equivalently, the compiler could load the address of `ping()` into a register and then us an indirect jump through that register. To simplify the implementation of the `return` statement, as will be clear in a moment, we will use this alternative.

```
foo:              ld   $ping, r0            # r0 = address of ping ()
                  j    0x0(r0)              # goto ping ()
```

To complete the implementation of this method invocation, we need to consider how the compiler will generate code for the `return` statement in ping (note that in this example, ping's return is implicit).

2.7.6 Method Return

All methods and procedures return to their caller, sometimes the return occurs implicitly after the last statement of the procedure executes and sometimes it is explicit when a `return` statement is executed. In either case, the return is a jump to a code address, called the *return address*. For any call, the return address is address of the instruction immediately following the procedure call. For example, if the code that makes a call to `ping()` is at address `0x100`, and the 8-byte version of the procedure call is used, then the return address is `0x108`.

```
100:    0000 00000500     # r0 = 500
        c000              # goto address in r0
108:
```

The value of a method's return address is determined at runtime when the method is invoked. The compiler can not possibly know the value of this address, because a method can be invoked from multiple places in a program. The code the compiler generates for the return statement must work for all of these calls and thus clearly, the return address can not be hardcoded in the return instructions. Instead, this address must be placed in memory or a register by the code that implements the procedure call, so that the return instructions can use this value in an indirect branch.

Typically, the calling code places the return address in a register, sometimes called the return-address register or `ra`. In our case, we will arbitrarily reserve the register `r6` to hold the return address. It is entirely up to the compiler to decide where to save this address. All that is required is for it to use the same convention for both method calls and returns.

It remains to be seen how the method invocation code determines the value of the return address it stores in `r6`. To this end, we have added to the SM213 ISA instruction `6f0r` or `gpc`, which copies the current value of the program-counter register, `PC`, into a general-purpose register. The invocation code includes this instruction, which loads the program counter into `r6`. The return address is a fixed number of bytes after this instruction. The compiler thus adds this constant to `r6` to compute the return address. To understand this calculation, recall that when executing one instruction, the program counter stores the address of the next instruction. In the code below, there are two instructions, totaling four bytes, between the instruction that reads the program counter and the halt instruction to which the call returns in this case. The two instructions are `6406` which adds four to `r6` and `c000` which jumps to `ping()`.

The following code thus completes the static procedure call example we started above.

```
0000 00000500   # r0 = address of ping ()
6f06            # r6 = pc of next instruction
6406            # r6 = pc + 4
c000            # goto ping ()
f000
```

Or in assembly.

```
foo:              ld   $ping, r0           # r0 = address of ping ()
                  gpc  r6                  # r6 = pc of next instruction
                  inca r6                  # r6 = pc + 4
                  j    0x0(r0)             # goto ping ()
```

We now see the benefit of using the indirect-jump variant of the static method call: doing so has us add four to r6 and we have a special instruction to do this. If we had instead used the direct-jump version, which is 6 bytes instead of two, we would have to add fourteen, instead of four, because the add would require 6 bytes to load the immediate value 14 into a register and two bytes to add. Here is the code, which does the same thing as above, but is larger and slower.

```
foo: gpc # r6 = address of next instruction
ld    0x14, r0 # r0 = 14 (size of ld (6), add (2) and j (6))
add   r0, r6 # r6 = r6 + 14, the return address
j     ping # goto ping
```

Returning from a method is simply an indirect jump using the return-address stored in r6.

```
.address 0x500
ping:             j    0x0(r6)             # return
```

2.7.7 Dynamic Method Invocation

The cornerstone of object-oriented languages is the ability to implement new classes by extending existing classes or by implementing existing interfaces. The goal in either case is to allow code written for a base class or interface to operate on objects of a new class without changing this code, as long as the new class extends that base class or implements that interface.

With this in mind we see that while all variables in Java has a static type (i.e., class or interface), the type of the objects to which they refer is determined dynamically. Now consider what the compiler must to to implement an instance-method invocation (i.e., invocation on a non-static method). What it knows, statically, is the type of the variable and the name of the method to invoke. What it does not know, however, is the type of the object to which the variable refers. This object is allocated and assigned to the variable at runtime and thus, in general, its type is determined dynamically (i.e., at runtime). But, the compiler must know the actual type of the object in order to correctly dispatch control flow to the correct method, because different implementations of the static type can have different implementations of the same method (e.g., if a subclass overrides a superclass method). This type of method invocation is called *polymorphic dispatch*, because the dispatch decision must be made at runtime based on the dynamic type of the object in question.

Dispatching to `static` or to `final` methods can be implemented using static method invocation, because the compiler either knows that the method to invoke is determined by the variables static class or that the method can not be extended by any subclasses. All other methods, however, require dynamic method invocation that implements poylmorphic dispatch.

Example of Poylmorphic Dispatch in Java

For example, consider this Java code taken from `S8-dynamic-call.java`.

```
public class A {
  void ping () {}
}

public class B extends A {
  void ping () {}
  void ping () {}
}

public class Foo {
  static A a;
  static void foo () {
    a = new B ();
    a.ping ();
  }
}
```

The variable `a` is statically declared to store a reference to an object that implements the interface defined by the class A. It can thus store instances of class A, but also instances of class B as well, because class B extends A. Class B, however, defines a new implementation of the `ping` method. An thus, when `a` stores an instance of `A`, the statement `a.ping()` must invoke the method `A.ping`, but when `a` stores an instances of `B`, it must invoke `B.ping`. In general, the compiler can not know the actual type of the object to which `a` refers and thus it **can not know statically the memory address of the procedure to invoke.**

Like other procedure calls, the machine code that implements the statement `a.ping()` will contain a `jump` to the first instruction of the appropriate version of the `ping` method. In this case, however, the jump instruction must get this address from some data structure referred to by the object itself.

We will described a simplified, but essentially correct, solution that includes something called a *jump table* in ever object. In fact, Java does something a bit trickier so that it can store jump tables with classes and not objects, to minimize the space overhead of storing jump tables. In both cases, however, the key element is the same: there is a jump table in memory and the code that implements a method invocation access the jump table using the object's memory address.

In our simplified view, every object stores a *jump table* when an entry for every new instance method it implements. The object also stores the instance variables declared by its classes. The compiler organizes these two things together by class, so it can determined *statically* the offset to methods and variables of an object without knowing the objects actual class, but only knowing the static type of a variable that stores a reference to it.

To do this, the compiler lays out the jump table and instances variables for a class starting with its base class. It places these values at the beginning of the object. It now knows a static offset to every method or object declared in the object's based class — and this offset is the same for objects of the base-class type and all classes that extend it. It repeats this process, step-by-step for all of the object's parent classes, starting with the base. For each subclass it adds only the new methods and variables declared in that subclass. Methods declarations that override parent methods do not get new entries in the jump table, they use the entry created by the parent-class declaration, but modified to store the address of the overridden procedure. The compiler performs this procedure for every class once to determine the size objects that instantiate the classes and to produce a template for the jump table, filled in with the appropriate code addresses for every method implemented by the class. It uses this information to implement statements that created `new` instances of the class, to determine the size of the object and to initialize the object's jump table.

Let's return to the `S8-dynamic-call` example and assume that method addresses are `0x500` for `A.ping`, `0x600` for `B.ping` and `0x700` for `.pong`. The jump tables for objects of type `A` and `B` would be initialized as follows.

Jump table for instances of class `A`:

0x500	*(address of ping)*

Jump table for instances of class `B`:

0x600	*(address of ping)*
0x700	*(address of pong)*

The code that implements the method invocation `a.ping ()` with then jump to the address stored in memory at offset 0 (constant known to the compiler and hard-coded in the instruction) from the address stored in variable `a`. If `a` were a static variable stored at address `0x100`, then the SM213 instructions that implement this are as follows.

```
.address 0x100
                 ld   $a, r0               # r0 = address of a
                 ld   0x0(r0), r0          # r0 = pointer to object
                 gpc  r6                   # r6 = pc
                 inca r6                   # r6 = r6 + 4
                 j    *0x0(r0)             # goto address stored in a
```

Notice that these calls use the double-indirect, base-plus-displacement jump instruction, `dr--`, which was designed specifically for this purpose.

Jump tables in C

In C, all procedure calls are static. The idea of polymorphic dispatch is so powerful, however, that many large C programs — and all modern operating systems written in C — implement jump tables and a form of polymorphic dispatch using themselves using structs and explicitly coded procedure variables.

Procedure variables are a feature of C not found in Java, but found in C#. They were, in fact, the source of a major lawsuit between Sun Microsystems and Microsoft Corporation that Sun won and that lead to Microsoft developing C#. Sun controls the Java language specification. Microsoft implemented a variant of Java that included procedure variables and a few other things. Sun complained and got the court to order Microsoft to stop. Microsoft did and eventually stopped supporting Java and instead developed their own language, C#, which is remarkably similar to Java, but with some variations.

A procedure variable is a variable that stores a pointer to a procedure and that can be used to invoke procedures dynamically. In C, a procedure variable is declared using a syntax very similar to the declaration of the procedure for which it will store pointers, but with a * in front of the procedure name and parentheses around the name of the procedure. Thus, a procedure variable named `foo` that can store pointers to procedures that have no arguments and no return value would be declared like this.

```
void (*foo)();
```

This variable is assigned a value by a statement whose right-hand-side is the name of a real procedure, with without listing its arguments. For example, the following code assigns foo the address of the procedure `bar`.

```
void bar () { /* do something */ }
void zot () {
    foo = bar;
}
```

Subsequently a statement of the form `foo ();` will call `bar ()`. Notice, however, that this is a dynamic procedure call. The compiler will not know the address of the procedure to call when it generates the code for this statement. Just as in the Java case described above, it will use a double-indirect jump to go to the instruction stored in memory in the variable `foo`.

Finally, it is now possible to implement the sort of dynamic dispatch found in `snippet7.java` in the following way, taken from `S8-dynamic-call.c`.

```
typedef struct {
  void (*ping)();
} A;

void A_ping () {}

A* new_A () {
  A* a = (A*) malloc (sizeof(A));
  a->ping = A_ping;
  return a;
}

typedef struct {
  void (*ping)();
  void (*pong)();
} B;

void B_ping () {}
void B_pong () {}

B* new_B () {
  B* b = (B*) malloc (sizeof(B));
  b->ping = B_ping;
  b->pong = B_pong;
  return b;
}
```

```
A a;

void foo () {
  A* a = (A*) new_B ();
  a->ping ();
}
```

The SM213 implementation is in `S8-dynamic-call.s` as follows.

```
.address 0x100
                 ld   $a, r0              # r0 = address of a
                 ld   0x0(r0), r0         # r0 = pointer to object
                 gpc  r6                  # r6 = pc
                 inca r6                  # r6 = r6 + 4
                 j    *0x0(r0)            # goto address stored in a
                 halt
.address 0x200
a:               .long 0x00001000         # a - allocated dynamically by new_A
.address 0x500
A_ping:          j    0x0(r6)             # return
.address 0x600
B_ping:          j    0x0(r6)             # return
.address 0x700
B_pong:          j    0x0(r6)             # return
.address 0x1000
object:          .long 0x00000600         # allocated dynamically by new_B
                 .long 0x00000700
```

As in previous examples with dynamic objects, we have allocated the object statically here (i.e., the memory labelled `object`). As before, we do this only to simplify the example. In reality, this memory would have been allocated and initialized dynamically when the `new` or `malloc` statement/procedure executed.

STOP READING HERE — A NEW VERSION WILL MODIFY WHAT YOU FIND BELOW.

2.7.8 Switch Statements

A switch statement can be implemented by converting it to a set of if-then-else statements. Similarly, programmers can always choose to use if statements instead of switch statements, if they like. The reverse, however, is not true. If statements are more flexible than switch statements and thus only restricted types of if statements can be converted to a switch statement. Most languages place two key restrictions on switch statements that separate them from if statements: (1) the switch condition must evaluate to an integer and (2) case labels must be constant integer values. These restrictions allow many switch statements to be implemented more efficiently than the equivalent set of if statements, by jumping directly to the matching case. If if statements are used, on the other hand, each case requires the evaluation of the conditional expression in the if and a conditional branch.

For example, consider this C switch statement taken from `snippet11.c` (Java switch statements have exactly the same syntax).

```
switch (i) {
  case 0:  j=10; break;
  case 1:   j=11; break;
  case 2:   j=12; break;
  case 3:   j=13; break;
  default: j=14; break;
}
```

This statement is logically equivalent to the following if-then-else statement:

```
if (i==0)
  j=10;
else if (i==1)
  j=11;
else if (i==2)
  j=12;
else if (i==3)
  j=13;
else
  j=14;
```

Notice that in the if-statement case, the expression $(i ==?)$ is evaluated over and over again for each case label. If i is 3, for example, this expression will be evaluated four times at runtime and each time a conditional branch will be computed as well.

This switch statement can alternatively be implemented using an *double indirect jump* and a branch table, similar to that used for dynamic method invocation. The key difference here is that the index into the table is a dynamically determined value, the value of the switch statement expression (i in this case). Switch statements are thus best implemented using the double-indirect-indexed instruction `eri-`.

The key idea is that the compiler allocates a jump table for the switch statement and places the code address of each case *arm* (the statement block associated with a particular case) in the table, which is then index by the value of the switch expression at runtime.

The size of this jump table is determined by the difference between the minimum and maximum case-label values; there must be an entry in the table for every integer between the minimum and maximum, inclusively. The example thus requires a four-entry jump table. Four entries would be required even if cases 1 and 2 were missing; in this case the entries for 1 and 2 would store the address of the switch statement's *default* arm. Notice that if the minimum case value is 100, for example, no jump-table entries for cases 0-99 are required, because the switch value can be *normalized* at runtime by subtracting 100 from it (i.e., using jump-table entry 0 for case 100).

Here is an outline of the code that implements the switch statement itself.

1. Evaluate the switch expression. In the example, this just requires reading i from memory into a register, say r0.
2. Subtract the value of the minimum case label. Notice this is a constant value hard coded in an instruction. In the example, the minimum case label is 0, so this step is left out.
3. Load the address of the branch table into a register, say r1. Again, this is a constant know to the compiler and thus hard coded in an instruction.
4. Jump double-indirectly using r0 and the index value and r1 as the base value.

Here is the SM213 machine code for this switch statement, where the variables i and j are declared as global (i.e., static) ints.

```
100:  0000 00001000
      1000
      0100 fffffffd
      6101
      a117            # if (i>3) goto default
      0100 00000800
      e100
120:  0100 0000000a   # case 0:
      8010            # goto done
      0100 0000000b   # cose 1:
      800c            # goto done
      0100 0000000c   # case 2:
      8008            # goto done
      0100 0000000d   # case 3:
      8004            # goto done
      0100 0000000e   # default:
      8000            # goto done

      0000 00001004   # done:
      3100
      f000

800:  00000120        # & (case 0)
      00000128        # & (case 1)
      00000130        # & (case 2)
      00000138        # & (case 3)

1000: 00000002    # i
1004: 00000000    # j
```

And, here is the disassembly comments generated by running the simulator.

```
(sm) x/i 0x100:5
0x00000100: 0000 00001000          ld   $0x1000, r0
0x00000106: 1000                   ld   0x0(r0), r0
0x00000108: 0100 fffffffd          ld   $0xfffffffd, r1
0x0000010e: 6101                   add  r0, r1
0x00000110: a117                   bgt  r1, 0x140
(sm) x/i 0x120:13
0x00000120: 0100 0000000a          ld   $0xa, r1
0x00000126: 8010                   br   0x148
0x00000128: 0100 0000000b          ld   $0xb, r1
0x0000012e: 800c                   br   0x148
0x00000130: 0100 0000000c          ld   $0xc, r1
0x00000136: 8008                   br   0x148
0x00000138: 0100 0000000d          ld   $0xd, r1
0x0000013e: 8004                   br   0x148
0x00000140: 0100 0000000e          ld   $0xe, r1
0x00000146: 8000                   br   0x148
0x00000148: 0000 00001004          ld   $0x1004, r0
0x0000014e: 3100                   st   r1, 0x0(r0)
0x00000150: f000                   halt
```

2.8 Method Scope: Local Variables, Method Arguments and the Stack

Finally, we turn the question of how to implement the local variables and arguments of methods. To do so. These variables are part of the *scope* of the a method and exist in memory only while the method is executing and only for a particular instance of that execution. If a method is invoked 100 times without returning, for example, memory stores 100 distinct copies of that methods local variables, one for each execution instance.

While local variables are arguments together make on a method's local scope, they are slightly different from each other in how they are used and implements. We'll first discuss local variables and then arguments.

Accessing Local Variables of a Method

Conceptually the local variables of a method are like a class defined for that method. An instance of this class is implicitly allocated when the method is invoked and is explicitly deallocated when the method returns. As with instance variables, the compiler determines the layout of local variables in this scope object and thus knows their offset from the beginning of the scope. Statements that access local variables are implemented in precisely the same manner as access to instance variables, as a static offset from an address stored in a register. In the case of instance variables, this register is `r7` for methods of the class in which the variables are defined (e.g., the implicit access to `this`. In the case of local variables it will be a special register that, for reasons will be clear in a moment, is the called the *stack pointer*. The SM213 "compiler" convention will be to use `r5` for this purpose.

Here, for example, is a bit of Java that contains an assignment to an instance variable and a local variable.

```
public class Foo {
    int i;
    int k;
    void foo () {
        int l;
        int m;

        k=0;
        m=0;
    }
}
```

The two assignment statements, to `k` and `m` are implemented by the following SM213 instructions.

```
0000 00000000    # r0 = 0
3017             # k = 0
3015             # m = 0
```

In this case, by coincidence, the offset to `k` from the beginning of the object and to `m` from the beginning of the local scope are both `4`. For other variables these values might be different, but in every case the compiler will know the offset and will thus be able to encode it as a constant in an instruction, as was done here (the `1` in the two store instructions — recall that the gold machine multiplies this value by four to get the offset it uses).

What remains is to show how these local stacks are created and freed and thus how we ensure at runtime that the register `r5` always points to the right spot: the beginning of the currently executing method's local scope.

2.8.1 Local Scopes and the Runtime Stack

As a concept, local scopes are much like objects, as we have seen. They could, in fact, be allocated from the `heap` just as objects are. The only reasons not to do this are (1) that there is alternative that has better runtime performance and (2) that it is very important that method invocation be as fast as possible because it is frequent, particularly in object-oriented-style programs where each method is small by design.

The key thing to notice about local scopes is that, unlike other objects, they are deallocated in exactly the reverse order of their allocation. If `a` calls `b()` and `b()` calls `c()`, then the local scopes are allocated in the order $a \rightarrow b \rightarrow c$ and deallocated $c \rightarrow b \rightarrow a$. This pattern suggested a very simple design for data structure used to store local scopes: a *stack*.

Every thread of a program (our programs have only a single thread) is allocated a region of memory called its *runtime stack*. The stack stores a set of *activation frames*, one to hold the local scope of every method that has been invoked but has not yet returned. Activation frames are ordered sequentially on the stack in order of their creation. By convention the stack *grows* toward lower addresses. If we stick with our view of memory as an array starting with address zero on the top, then stacks grow *up*. Sometimes stacks are drawn the other way, with high addresses on the top, to emphasize that stacks grow to lower addresses (in this case down) — this is what the textbook does, don't be confused by this.

The first few instructions of every method allocate that method's activation frame by subtracting the frame's size (a constant) from the current stack pointer (in `r5` in our case). The last few instructions executed by every procedure release the frame by adding the same number to the stack pointer. If a procedure has no local variables, then these two steps are left out.

For example, let's consider the following Java class taken from `snippet8.java`.

```
public class A {
  public static void b () {
    int l0 = 0;
    int l1 = 1;
  }
}

public class Foo {
  static void foo () {
    A.b ();
  }
}
```

And the corresponding C snippet `snippet8.c`.

```
void b () {
  int l0 = 0;
  int l1 = 1;
}

void foo () {
  b ();
}
```

The SM213 instructions for the procedure `b()` are the following (taken from `snippet8.gold`).

```
100:  0000 fffffff8   # r0 = -8 = -(size of activation frame)
```

```
6105            # create activation frame on stack

0000 00000000   # r0 = 0
3005            # l0 = 0
0000 00000001   # r0 = 1
3015            # l1 = 1

0000 00000008   # r0 = 8 = size of activation frame
6105            # teardown activation frame

c600            # return
```

And the disassembler comments generated by the simulator.

```
00000100:   0000 fffffff8   ld  $0x-8, r0
00000106:   6105            add r0, r5
00000108:   0000 00000000   ld  $0x0, r0
0000010e:   3005            st  r0, 0x0(r5)
00000110:   0000 00000001   ld  $0x1, r0
00000116:   3015            st  r0, 0x4(r5)
00000118:   0000 00000008   ld  $0x8, r0
0000011e:   6105            add r0, r5
00000120:   c600            jmp 0x0(r6)
```

2.8.2 Saving the Return Address

In Section 2.7.6 we encounter the question of what to with the current value of the return-address register (i.e., `r6`) when making a new procedure call, which needs to place its own return address in `r6`. The solution is to save this as part of the current method's scope on the stack.

Thus the complete machine code for calling a procedure follows this outline.

1. Save `r6` on stack.
2. Compute new return address and place this in `r6`
3. Jump to target method.
4. Restore `r6` to the value on the stack.

The SM213 code for these steps show in this sequence that implements the call to `b()` from `foo()`, again taken from `snippet8.gold`.

```
6605            # sp-=4
3605            # save r6 to stack
0000 00000100   # address of c ()
6f06            # r6 = pc
6406            # r6 = r6 + 4
c000            # goto b ()
1056            # restore r6 from stack
6405            # sp+=4
```

And with the disassembler comments generated by the similator.

```
00000200:   6605            add $-4, r5
00000202:   3605            st  r6, 0x0(r5)
00000204:   0000 00000100   ld  $0x100, r0
0000020a:   6f06            gpc r6
0000020c:   6406            add $4, r6
0000020e:   c000            jmp 0x0(r0)
00000210:   1056            ld  0x0(r5), r6
00000212:   6405            add $4, r5
```

2.8.3 Arguments and the Return Value

Invoking a method involves a transfer of control and data between two locations in a program, the *caller* and the *callee*. In the preceding sections we have examined the control transfer only. We now turn to the transfer of data arguments to the callee and the transfer of a return value to the caller.

We'll use the following Java program and its C equivalent (`snippet9.java` and `snippet9.c`).

```
public class A {
  static int add (int a, int b) {
    return a+b;
  }
}

public class foo {
  static int s;
  static void foo () {
    s = add (1,2);
  }
}


int add (int a, int b) {
  return a+b;
}

int s;

void foo () {
  s = add (1,2);
}
```

Return Value

In Java and C methods (and procedures) can return one value to their caller. In C this value is typically returned in a register. Typically register 0 is used for this purpose; in IA32, register 0 is called `%eax`. In the SM213 "compiler" we will use `r0`. It is entirely up to the compiler to choose a convention; the only requirement is the code machine code generated for caller and callee both follow the same convention.

Assuming the values of arguments `a` and `b` are in registers `r0` and `r1`, respectively, the SM213 implementation of `add ()` returns their sum to its callers as follows.

```
6110            # return (r0) = a (r0) + b (r1)
c600            # return
```

Its disassembly.

```
00000100:   6110            add r1, r0
00000102:   c600            jmp 0x0(r6)
```

The code code in `foo()` that calls `add()` and then stores the value it returns in `s` is as follows, provided that `s` is a global variable with static address `0x280`.

```
0300 00000100   # address of add ()
6f06            # r6 = pc
6406            # r6 = r6 + 4
c300            # goto add ()
0100 00000280   # r1 = address of s
3001            # s = add (1,2)
```

And its disassembly.

```
00000210:   0300 00000100   ld  $0x100, r3
00000216:   6f06            gpc r6
00000218:   6406            add $4, r6
0000021a:   c300            jmp 0x0(r3)
00000220:   0100 00000280   ld  $0x280, r1
00000226:   3001            st  r0, 0x0(r1)
```

Arguments

A method declaration includes the declaration of its *formal arguments*. These arguments are variables that are contained in by the method's scope while it is running. Method invocations provide an *actual argument* value for each the method's formal arguments. One of the tasks for the machine instructions that implement the method invocation is to assign these formal argument values to the formal argument variables. In this way arguments are a bit different from the local variables of the called method, even though they both reside in the same scope.

There are two alternatives available for passing arguments from caller to callee: through registers or on the stack. It is up to the compiler to decide which convention to use, or perhaps to use a combination of the two, when it compiles the code for a method. Subsequent compilation of invocations of that method must then follow the same convention as the method they call.

Typically small methods with few arguments are best implemented by passing arguments through registers. The runtime performance of this method is much better than the through-stack alternative, because it avoids the memory-access cost of copying values into stack memory on the caller side and back out of memory on the callee side. On the other hand, registers can be a scarce resource (particularly in IA32, which has only six generally available). Through the stack argument passing is thus favoured for methods with more arguments than available registers or methods that are large enough that they will need most of the registers to store other values and that run long enough to *amortize* the cost argument passing over a longer overall method execution.

Here is the SM213 implementation of `add` and the call to it from `foo` that passes values for `add`'s two arguments from `foo` to `add` in the registers `r0` and `r1`. These are taken from `snippet9a.gold`.

The implementation of `add` is the same as shown in the return-value example above.

The implementation of the call to `add` adds these two states just before those listed above in the return-value example.

```
0000 00000001   # arg0 (r0) = 1
0100 00000002   # arg1 (r1) = 2
```

Their disassembly.

```
0000020a:   0100 00000002   ld  $0x2, r1
00000210:   0300 00000100   ld  $0x100, r3
```

To pass arguments through the stack, however, requires more work in both `add` and in `foo`; taken from `snippet9b.gold`.

The following statements are added to `add` to copy the values of the formal arguments into registers.

```
1050            # r0 = arg0
1151            # r1 = arg1
```

And their disassembly.

```
00000100:   1050            ld  0x0(r5), r0
00000102:   1151            ld  0x4(r5), r1
```

And in `foo`, instead of the two instructions that copy the values 1 and 2 into registers `r0` and `r1`, respectively, we have these instructions that precede the call to `add`.

```
0000 00000002   # r0 = 2
6605            # sp-=4
3005            # save arg1 on stack
0000 00000001   # r0 = 1
6605            # sp-=4
3005            # save arg0 on stack
```

And their disassembly.

```
00000204:   0000 00000002   ld  $0x2, r0
0000020a:   6605            add $-4, r5
0000020c:   3005            st  r0, 0x0(r5)
0000020e:   0000 00000001   ld  $0x1, r0
00000214:   6605            add $-4, r5
00000216:   3005            st  r0, 0x0(r5)
```

Notice that the two arguments are pushed on the stack in reverse order, with the value of the right-most argument pushed first.. The reason for this is that when they are subsequently accessed in `add`, the will be access as positive offsets from the stack pointer. And if we want the left-most argument to have the smallest offset, similar to local variables and instance variables, then the value for this argument must be pushed last.

In addition, these instruction must following the call to `add` to discard the two arguments from the stack.

```
6405            # discard arg0 from stack
6405            # discard arg1 from stack
```

And their disassembly.

```
00000224:   6405            add $4, r5
00000226:   6405            add $4, r5
```

Comparing the code between the register- and stack-passing approaches, we see why register-passing typically leads to faster code. A total of two additional instructions were needed in this example to pass through registers, while ten instructions were required to pass through the stack. Of even greater significance is that passing through registers require zero memory accesses, while passing through the stack requires four, two for each argument. These differences can be very significant for small procedures that are called frequently.

2.8.4 Arguments and Local Variables Together

In general, the local scope of a method includes its local variables, its formal arguments and temporarily saved registers, such as `r6`, the saved return address. The caller of a method is responsible for creating the part of the new frame that contains the formal arguments, by pushing on to the stack the actual values of those arguments, and subsequently discarding those arguments from the stack when the called method returns. The called method is responsible for allocating space on the stack for local variables and for freeing this stack space just before the method returns. This method is also responsible for saving registers such as `r6` just before it calls another procedure.

Consider the procedure below that has both locals and arguments and assume that arguments for this procedure are passed on the stack.

```
void foo(int i, int j) {
    int k;
    int l;
}
```

If this is the current procedure then its frame is on the top of the stack and thus the register `r5` stores its memory address. The frame, along with an instruction that reads the variable at each location, looks something like this. Notice that the offset to the first argument depends on the number of local variables (and their size) declared by the method; both of these values are specified statically and so the compiler always knows the offset to every local and every argument.

local variable 0: `k`	`ld 0x0(r5), r0`
local variable 1: `l`	`ld 0x4(r5), r0`
formal argument 0: `i`	`ld 0x8(r5), r0`
formal argument 1: `j`	`ld 0xc(r5), r0`

If this is not the current procedure, the the frame includes its saved value of the return address `r6` and `r5`, the stack pointer, does not store its address. The frame would thus look like this.

saved `r6` (return address)
local variable 0: `k`
local variable 1: `l`
formal argument 0: `i`
formal argument 1: `j`

2.8.5 The First Stack Frame

At the bottom of every stack frame is the procedure that calls the first application method for that thread. In a single-threaded Java program, the first application method is always the `static void main (String args[])` method of some class. Below this stack frame is a special procedure called `_start` (or possibly `crt0` or `crt1`) that initializes the stack pointer, calls the first method and stops the current thread (in our case by halting the processor) when that method (e.g., `main`) returns.

Here is a SM213 ISA implementation of a `_start` method that places the stack at the *bottom* of 4-MB of memory (i.e., at address `0x40000` - 4 = `0x3ffffc`). It is taken from `snippet9a.gold` (or `9b`).

```
# _start
0:    0000 003ffffc   # base of stack
      6005            # initialize stack pointer
      0000 00000200   # r0 = address of main ()
      6f06            # r6 = pc
      6406            # r6 = r6 + 4
      c000            # goto main ()
      f000
```

And its disassembly.

```
00000000:   0000 003ffffc   ld  $0x3ffffc, r0
00000006:   6005            mov r0, r5
00000008:   0000 00000200   ld  $0x200, r0
0000000e:   6f06            gpc r6
00000010:   6406            add $4, r6
00000012:   c000            jmp 0x0(r0)
00000014:   f000            halt
```

And here, for completeness, is `snippet9b.gold` in its entirety.

```
# _start
0:    0000 003ffffc   # base of stack
      6005            # initialize stack pointer
      0000 00000200   # r0 = address of foo ()
      6f06            # r6 = pc
      6406            # r6 = r6 + 4
      c000            # goto foo ()
      f000

# int add (int a, int b)
100:  1050            # r0 = arg0
      1151            # r1 = arg1
      6110            # return (r0) = a (r0) + b (r1)
      c600            # return

# void foo ()
200:  6605            # sp-=4
      3605            # save r6 to stack
```

```
        0000 00000002   # r0 = 2
        6605            # sp-=4
        3005            # save arg1 on stack
        0000 00000001   # r0 = 1
        6605            # sp-=4
        3005            # save arg0 on stack

        0300 00000100   # address of add ()
        6f06            # r6 = pc
        6406            # r6 = r6 + 4
        c300            # goto add ()

        6405            # discard arg0 from stack
        6405            # discard arg1 from stack
        1056            # restore r6 from stack
        6405            # sp+=4

        0100 00000280   # r1 = address of s
        3001            # s = add (1,2)

        c600            # return

280:    00000000        # s
```

Exercises

2.1 For each of the following, indicate whether the value is know statically or dynamically by placing the work **static** or **dynamic** next to each.

(a) The address of a global variable in C:

(b) The address of an element of a static array in Java:

(c) The address of an instance variable in Java.

(d) The offset of an instance variable from the beginning of the object that contains it in Java:

(e) The address of the code that implements a return statement.

(f) The address of the instruction that executes immediately after a return statement.

(g) The value of `&a.i` where `a` is a global variable in the C program with the following declaration.

```
typedef struct { int i; } A;
A a;
```

(h) The value of `&a->i` where `a` is a global variable in the C program with the following declaration.

```
typedef struct { int i; } A;
A* a;
```

(i) The address of a local variable.

(j) The position of a local variable in its activation frame.

(k) The code address of a static method.

(l) The code address of an instance method (i.e., a method that is not declared `static`).

2.2 What is the value of i after the following C code executes?

```
int a[10] = {0,1,2,3,4,5,6,7,8,9};
int i = *( &a[3] + 2 + *(a+1) );
```

2.3 Consider the following C snippet. All three variables are global variables.

```
int i;
int a[i];
int * b;
void foo () {
    a[i] = b[3];
}
```

Give the SM213 instructions that implement the statement `a[i] = b[3];`. Give all variables allocated by the compiler some arbitrary address and clearly comment your code.

2.4 Java and C deallocate dynamically created objects (i.e., heap-allocated objects) differently. Briefly describe the approach each uses. Give one significant benefit of each approach.

2.5 What is a *memory leak*? Give an example of a type of program where it could be a big problem. State whether Java solves this problem entirely, somewhat or not at all. Justify your answer.

2.6 Would it be possible to implement a compiler that used the heap, instead of the stack, to store activation frames? Give one important drawback of this approach and explain.

2.7 What is the difference between pc-relative and absolute jumps? Give one advantage of each. Explain.

2.8 Java and C do not allow the size of local variables or instance variables to be determined at runtime. How does this restriction simplify the machine instructions the compiler generates for accessing these variables?