The Core i7's Sandy Bridge Pipeline

Figure 4-47 is a simplified version of the Sandy Bridge microarchitecture showing the pipeline. At the top is the front end, whose job is to fetch instructions from memory and prepare them for execution. The front end is fed new x86 instructions from the L1 instruction cache. It decodes them into micro-ops for storage in the micro-op cache, which holds approximately 1.5K micro-ops. A micro-op cache of this size performs comparably to a 6-KB conventional L0 cache. The micro-op cache holds groups of six micro-ops in a single trace line. For longer sequences of micro-ops, multiple trace lines can be linked together.

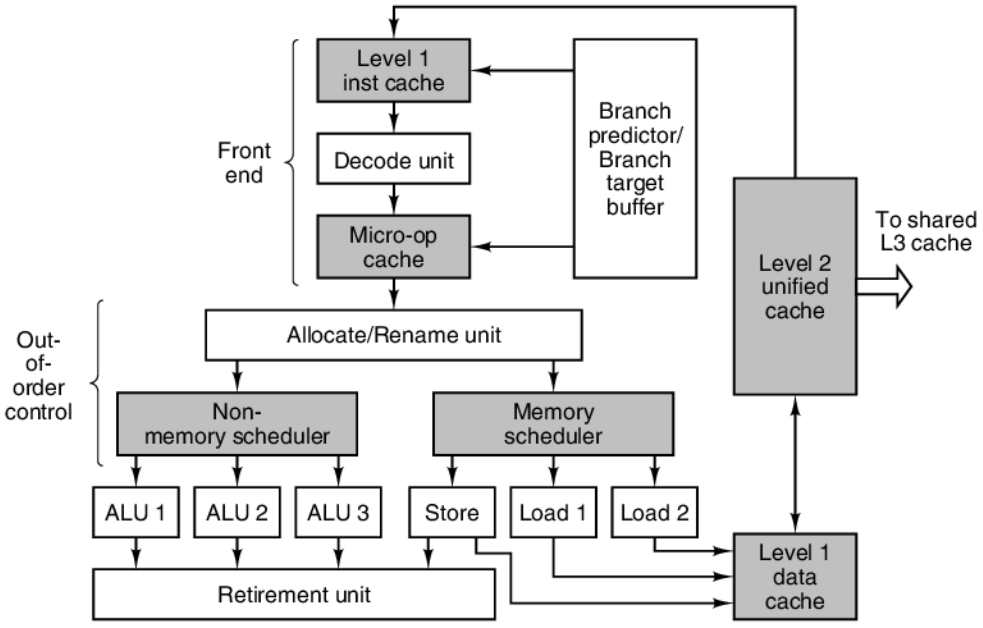

Figure 4-47. A simplified view of the Core i7 data path.

If the decode unit hits a conditional branch, it looks up its predicted direction in the **Branch Predictor**. The branch predictor contains the history of branches encountered in the past, and it uses this history to guess whether or not a conditional branch is going to be taken the next time it is encountered. This is where the top-secret algorithm is used.

If the branch instruction is not in the table, static prediction is used. A backward branch is assumed to be part of a loop and assumed to be taken. The accuracy of these static predictions is extremely high. A forward branch is assumed to be part of an if statement and is assumed not to be taken. The accuracy of these static predictions is much lower than that of the backward branches.

For a taken branch the **BTB** (**Branch Target Buffer**) is consulted to determine the target address. The BTB holds the target address of the branch the last time it was taken. Most of the time this address is correct (in fact, it is always correct for branches with a constant displacement). Indirect branches, such as those used by virtual function calls and C++ `switch` statements, go to many addresses, and they may be mispredicted by the BTB.

The second part of the pipeline, the out-of-order control logic, is fed from the micro-op cache. As each micro-op comes in from the front end, up to four per cycle, the **allocation/renaming unit** logs it in a 168-entry table called the **ROB** (**ReOrder Buffer**). This entry keeps track of the status of the micro-op until it is retired. The allocation/renaming unit then checks to see if the resources the micro-op needs are available. If so, the micro-op is enenqueued for execution in one of the **scheduler** queues. Separate queues are maintained for memory and nonmemory micro-ops. If a micro-op cannot be executed, it is delayed, but subsequent micro-ops are processed, leading to out-of-order execution of the micro-ops. This strategy was designed to keep all the functional units as busy as possible. As many as 154 instructions can be in flight at any instant, and up to 64 of these can be loads from memory and up to 36 can be stores into memory.

Sometimes a micro-op stalls because it needs to write into a register that is being read or written by a previous micro-op. These conflicts are called WAR and WAW dependences, respectively, as we saw earlier. By renaming the target of the new micro-op to allow it to write its result in one of the 160 scratch registers instead of in the intended, but still-busy, target, it may be possible to schedule the micro-op for execution immediately. If no scratch register is available, or the micro-op has a RAW dependence (which can never be papered over), the allocator notes the nature of the problem in the ROB entry. When all the required resources become available later, the micro-op is put into one of the scheduler queues.

The scheduler queues send micro-ops into the six functional units when they are ready to execute. The functional units are as follows:

1. ALU 1 and the floating-point multiply unit.
2. ALU 2 and the floating-point add/subtract unit.
3. ALU 3 and branch processing and floating-point comparisons unit.
4. Store instructions.
5. Load instructions 1.
6. Load instructions 2.

Since the schedulers and the ALUs can process one operation per cycle, a 3-GHz Core i7 has the scheduler performance to issue 18 billion operations per second; however, in reality the processor will never reach this level of throughput. Since the front end supplies at most four micro-ops per cycle, six micro-ops can only be

issued in short bursts since soon the scheduler queues will empty. Also, the memory units each take four cycles to process their operations, thus they could contribute to the peak execution throughput only in small bursts. Despite not being able to fully saturate the execution resources, the functional units do provide a significant execution capability, and that is why the out-of-order control goes to so much trouble to find work for them to do.

The three integer ALUs are not identical. ALU 1 can perform all arithmetic and logical operations and multiplies and divides. ALU 2 can perform only arithmetic and logical operations. ALU 3 can perform arithmetic and logical operations and resolve branches. Similarly, the two floating-point units are not identical either. The first one can perform floating-point arithmetic including multiplies, while the second one can perform only floating-point adds, subtracts, and moves.

The ALU and floating-point units are fed by a pair of 128-entry register files, one for integers and one for floating-point numbers. These provide all the operands for the instructions to be executed and provide a repository for results. Due to the register renaming, eight of them contain the registers visible at the ISA level (EAX, EBX, ECX, EDX, etc.), but which eight hold the "real" values varies over time as the mapping changes during execution.

The Sandy Bridge architecture introduced the Advanced Vector Extensions (AVX), which supports 128-bit data-parallel vector operations. The vector operations include both floating-point and integer vectors, and this new ISA extension represents a two-times increase in the size of vectors now supported compared to the previous SSE and SSE2 ISA extensions. How does the architecture implement 256-bit operations with only 128-bit data paths and functional units? It cleverly coordinates two 128-bit scheduler ports to produce a single 256-bit functional unit.

The L1 data cache is tightly coupled into the back end of the Sandy Bridge pipeline. It is a 32-KB cache and holds integers, floating-point numbers, and other kinds of data. Unlike the micro-op cache, it is not decoded in any way. It just holds a copy of the bytes in memory. The L1 data cache is an 8-way associative cache with 64 bytes per cache line. It is a write-back cache, meaning that when a cache line is modified, that line's dirty bit is set and the data are copied back to the L2 cache when evicted from the L1 data cache. The cache can handle two read and one write operation per clock cycle. These multiple accesses are implemented using **banking**, which splits the cache into separate subcaches (8 in the Sandy Bridge case). As long as as all three accesses are to separate banks, they can proceed in tandem; otherwise, all but one of the conflicting bank accesses will have to stall. When a needed word is not present in the L1 cache, a request is sent to the L2 cache, which either responds immediately or fetches the cache line from the shared L3 cache and then responds. Up to ten requests from the L1 cache to the L2 cache can be in progress at any instant.

Because micro-ops are executed out of order, stores into the L1 cache are not permitted until all instructions preceding a particular store have have been retired. The **retirement unit** has the job of retiring instructions, in order, and keeping track

of where it is. If an interrupt occurs, instructions not yet retired are aborted, so the Core i7 has "precise interrupts" so that upon an interrupt, all instructions up to a certain point have been completed and no instruction beyond that has any effect.

If a store instruction has been retired, but earlier instructions are still in progress, the L1 cache cannot be updated, so the results are put into a special pending-store buffer. This buffer has 36 entries, corresponding to the 36 stores that might be in execution at once. If a subsequent load tries to read the stored data, it can be passed from the pending-store buffer to the instruction, even though it is not yet in the L1 data cache. This process is called **store-to-load** forwarding. While this forwarding mechanism may seem straightforward, in practice it is quite complicated to implement because intervening stores may not have yet computed their addresses. In this case, the microarchitecture cannot definitely know which store in the store buffer will produce the needed value. The process of determining which store provides the value for a load is called **disambiguation**.

It should be clear by now that the Core i7 has a highly complex microarchitecture whose design was driven by the need to execute the old Pentium instruction set on a modern, highly pipelined RISC core. It accomplishes this goal by breaking Pentium instructions into micro-ops, caching them, and feeding them into the pipeline four at time for execution on a set of ALUs capable of executing up to six micro-ops per cycle under optimal conditions. Micro-ops are executed out of order but retired in order, and results are stored into the L1 and L2 caches in order.

4.6.2 The Microarchitecture of the OMAP4430 CPU

At the heart of the OMAP4430 system-on-a-chip are two ARM Cortex A9 processors. The Cortex A9 is a high-performance microarchitecture that implements the ARM instruction set (version 7). The processor was designed by ARM Ltd. and it is included with slight variations in a wide variety of embedded devices. ARM does not manufacture the processor, it only supplies the design to silicon manufacturers that want to incorporate it into their system-on-a-chip design (Texas Instruments, in this case).

The Cortex A9 processor is a 32-bit machine, with 32-bit registers and a 32-bit data path. Like the internal architecture, the memory bus is 32 bits wide. Unlike the Core i7, the Cortex A9 is a true RISC architecture, which means that it does not need a complex mechanism to convert old CISC instructions into micro-ops for execution. The core instructions are in fact already micro-op like ARM instructions. However, in recent years, more complex graphics and multimedia instructions have been added, requiring special hardware facilities for their execution.

Overview of the OMAP4430's Cortex A9 Microarchitecture

The block diagram of the Cortex A9 microarchitecture is given in Fig. 4-48. On the whole, it is much simpler than the Core i7's Sandy Bridge microarchitecture because it has a simpler ISA architecture to implement. Nevertheless, some of

the key components are similar to those used in the Core i7. The similarities are driven mostly by technology, power constraints, and economics. For example, both designs employ a multilevel cache hierarchy to meet the tight cost constraints of typical embedded applications; however, the last level of the Cortex A9's cache memory system (L2) is only 1 MB in size, significantly smaller than the Core i7 which supports last level caches (L3) of up to 20 MB. The differences, in contrast, are due mostly to the difference between having to bridge the gap between an old CISC instruction set and a modern RISC core and not having to do so.

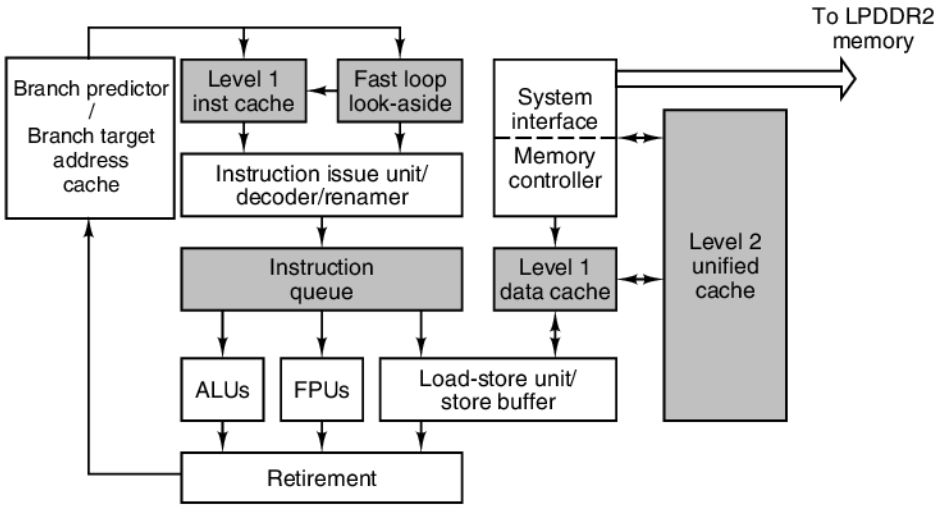

Figure 4-48. The block diagram of the OMAP4430's Cortex A9 microarchitecture.

At the top of Fig. 4-48 is the 32-KB 4-way associative instruction cache, which uses 32-byte cache lines. Since most ARM instructions are 4 bytes, there is room for about 8K instructions here in this cache, quite a bit larger than the Core i7's micro-op cache.

The **instruction issue unit** prepares up to four instructions for execution per clock cycle. If there is a miss on the L1 cache, fewer instructions will be issued. When a conditional branch is encountered, a branch predictor with 4K entries is consulted to predict whether or not the branch will be taken. If predicted taken, the 1K entry branch-target-address cache is consulted for the predicted target address. In addition, if the front end detects that the program is executing a tight loop (i.e., a non-nested small loop), it will load it into the fast-loop look-aside cache. This optimization speeds up instruction fetch and reduces power, since the caches and branch predictors can be in a low-power sleep mode while the tight loop is executing.

The output of the instruction issue unit flows into the decoders, which determine which resources and inputs are needed by the instructions. Like the Core i7,

the instructions are renamed after decode to eliminate WAR hazards that can slow down out-of-order execution. After renaming, the instructions are placed into the instruction dispatch queue, which will issue them when their inputs are ready for the functional units, potentially out of order.

The instruction dispatch queue sends instructions to the functional units, as shown in Fig. 4-48. The integer execution unit contains two ALUs as well as a short pipeline for branch instructions. The physical register file, which holds ISA registers and some scratch registers are also contained there. The Cortex A9 pipeline can optionally contain one more more compute engines as well, which act as additional functional units. ARM supports a compute engine for floating-point computation called **VFP** and integer SIMD vector computation called **NEON**.

The load/store unit handles various load and store instructions. It has paths to the data cache and the store buffer. The **data cache** is a traditional 32-KB 4-way associative L1 data cache using a 32-byte line size. The **store buffer** holds the stores that have not yet written their value to the data cache (at retirement). A load that executes will first try to fetch its value from the store buffer, using store-to-load forwarding like that of the Core i7. If the value is not available in the store buffer, it will fetch it from the data cache. One possible outcome of a load executing is an indication from the store buffer that it should wait, because an earlier store with an unknown address is blocking its execution. In the event that the L1 data cache access misses, the memory block will be fetched from the unified L2 cache. Under certain circumstances, the Cortex A9 also performs hardware prefetching out of the L2 cache into the L1 data cache, in order to improve the performance of loads and stores.

The OMAP 4430 chip also contains logic for controlling memory access. This logic is split into two parts: the system interface and the memory controller. The system interface interfaces with the memory over a 32-bit-wide LPDDR2 bus. All memory requests to the outside world pass through this interface. The LPDDR2 bus supports a 26-bit (word, not byte) address to 8 banks that return a 32-bit data word. In theory, the main memory can be up to 2 GB per LPDDR2 channel. The OMAP4430 has two of them, so it can address up to 4 GB of external RAM.

The memory controller maps 32-bit virtual addresses onto 32-bit physical addresses. The Cortex A9 supports virtual memory (discussed in Chap. 6), with a 4-KB page size. To speed up the mapping, special tables, called **TLBs** (**Translation Lookaside Buffers**), are provided to compare the current virtual address being referenced to those referenced in the recent past. Two such tables are provided for mapping instruction and data addresses.

The OMAP4430's Cortex A9 Pipeline

The Cortex A9 has an 11-stage pipeline, illustrated in simplified form in Fig. 4-49. The 11 stages are designated by short stage names shown on the left-hand side of the figure. Let us now briefly examine each stage. The *Fe*1 (Fetch

#1) stage is at the beginning of the pipeline. It is here that the address of the next instruction to be fetched is used to index the instruction cache and start a branch prediction. Normally, this address is the one following the previous instruction. However, this sequential order can be broken for a variety of reasons, such as when a previous instruction is a branch that has been predicted to be taken, or a trap or interrupt needs to be serviced. Because instruction fetch and branch prediction takes more than one cycle, the *Fe*2 (Fetch #2) stage provides extra time to carry out these operations. In the *Fe*3 (Fetch #3) stage the instructions fetched (up to four) are pushed into the instruction queue.

The *De*1 and *De*2 (Decode) stages decode the instructions. This step determines what inputs instructions will need (registers and memory) and what resources they will require to execute (functional units). Once decode is completed, the instructions enter the *Re* (Rename) stage where the registers accessed are renamed to eliminate WAR and WAW hazards during out-of-order execution. This stage contains the rename table which records which physical register currently holds all architectural registers. Using this table, any input register can be easily renamed. The output register must be given a new physical register, which is taken from a pool of unused physical registers. The assigned physical register will be in use by the instruction until it retires.

Next, instructions enter the *Iss* (Instruction Issue) stage, where they are dropped into the instruction issue queue. The issue queue watches for instructions whose inputs are all ready. When ready, their register inputs are acquired (from the physical register file or the bypass bus), and then the instruction is sent to the execution stages. Like the Core i7, the Cortex A9 potentially issues instructions out of program order. Up to four instructions can be issued each cycle. The choice of instructions is constrained by the functional units available.

The *Ex* (Execute) stages are where instructions are actually executed. Most arithmetic, Boolean, and shift instructions use the integer ALUs and complete in one cycle. Loads and stores take two cycles (if they hit in the L1 cache), and multiplies take three cycles. The *Ex* stages contain multiple functional units, which are:

1. Integer ALU 1.
2. Integer ALU 2.
3. Multiply unit.
4. Floating-point and SIMD vector ALU (optional with VFP and NEON support).
5. Load and store unit.

Conditional branch instructions are also processed in the first *Ex* stage and their direction (branch/no branch) is determined. In the event of a misprediction, a signal is sent back to the *Fe1* stage and the pipeline voided.

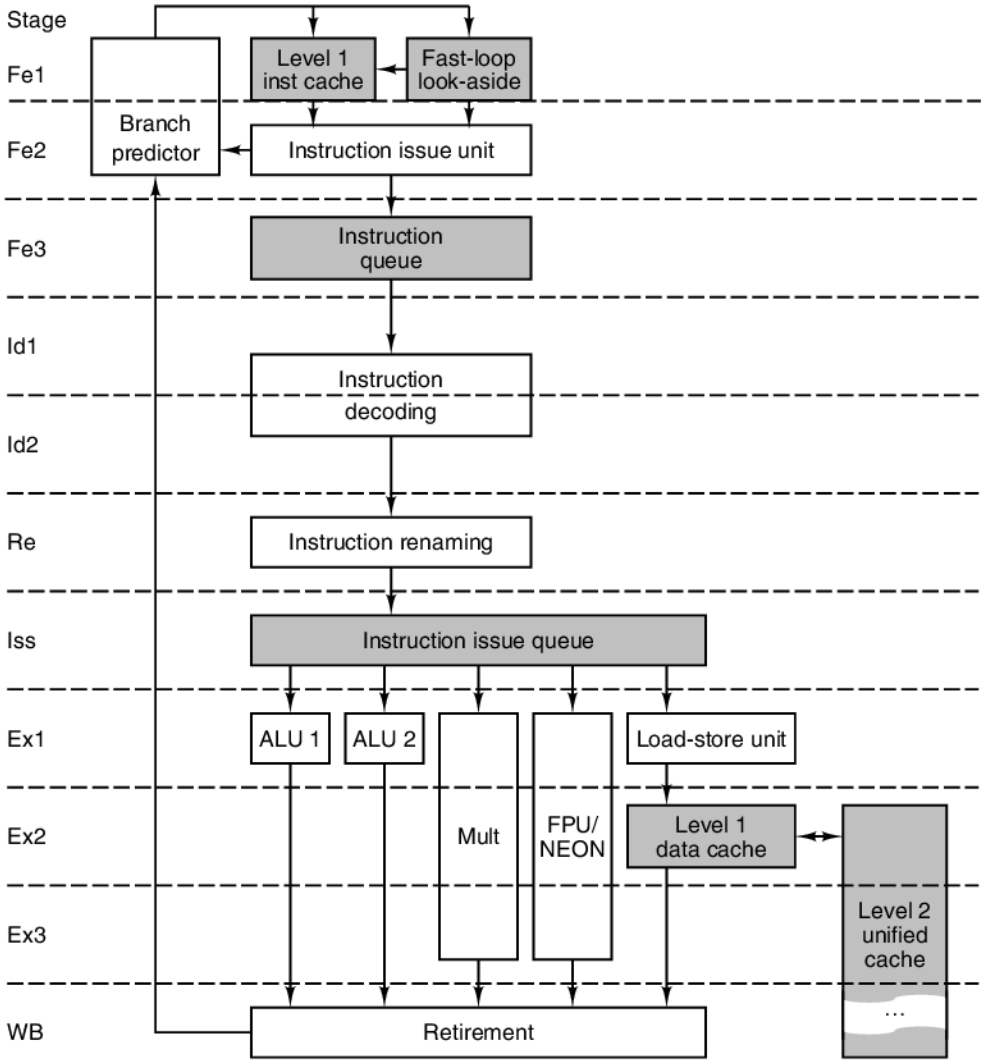

Figure 4-49. A simplified representation of the OMAP4430's Cortex A9 pipeline.

After completing execution, instructions enter the *WB* (WriteBack) stage where each instruction updates the physical register file immediately. Potentially later, when the instruction is the oldest one in flight, it will write its register result to the architectural register file. If a trap or interrupt occurs, it is these values, not those in the physical registers, that are made visible. The act of storing the register in the architectural file is equivalent to retirement in the Core i7. In addition, in the *WB* stage, any store instructions now complete writing their results to the L1 data cache.

This description of the Cortex A9 is far from complete but should give a reasonable idea of how it works and how it differs from the Core i7 microarchitecture.

4.6.3 The Microarchitecture of the ATmega168 Microcontroller

Our last example of a microarchitecture is the Atmel ATmega168, shown in Fig. 4-50. This one is considerably simpler than that of the Core i7 and OMAP4430. The reason is that the chip must be very small and cheap to serve the embedded design market. As such, the primary design goal of the ATmega168 was to make the chip cheap, not fast. Cheap and Simple are good friends. Cheap and Fast are not good friends.

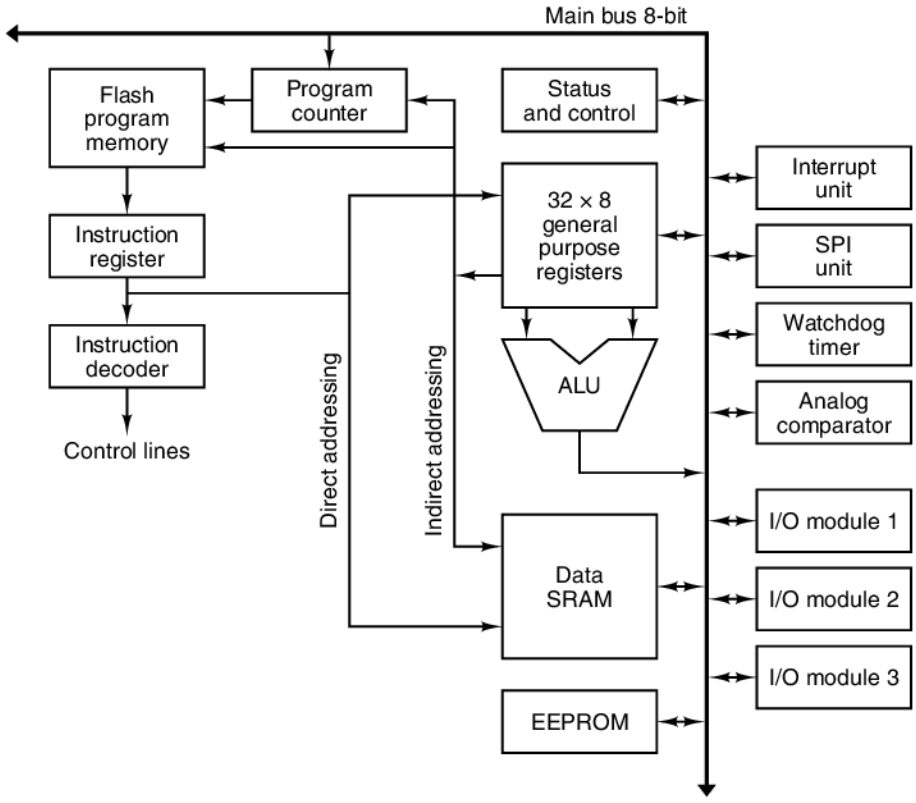

Figure 4-50. The microarchitecture of the ATmega168.

The heart of the ATmega168 is the 8-bit main bus. Attached to it are the registers and status bits, ALU, memory, and I/O devices. Let us briefly describe them now. The register file contains 32 8-bit registers, which are used to store temporary program values. The **status and control** register holds the condition codes of the last ALU operation (i.e., sign, overflow, negative, zero, and carry), plus a bit

that indicates if an interrupt is pending. The **program counter** holds the address of the instruction currently executing. To perform an ALU operation, the operands are first read from the register and sent to the ALU. The ALU output can be written to any of the writable registers via the main bus.

The ATmega168 has multiple memories for data and instructions. The data SRAM is 1 KB, too large to be fully addressed with an 8-bit address on the main bus. Thus, the AVR architecture allows addresses to be constructed with a sequential pair of 8-bit registers, thereby producing a 16-bit address that supports up to 64 KB of data memory. The EEPROM provides up to 1 KB of nonvolatile storage where programs can write variables that need to survive a power outage.

A similar mechanism exists to address program memory, but 64 KB of code is too small, even for low-cost embedded systems. To allow more instruction memory to be addressed the AVR architecture defines three RAM page registers (RAMPX, RAMPY, and RAMPZ), each 8 bits wide. The RAM page register is concatenated with a 16-bit register pair to produce a 24-bit program address, thereby allowing 16 MB of instruction address space.

Stop to think about that for a minute. 64 KB of code is too small for a microcontroller that might power a toy or small appliance. In 1964, IBM released the System 360 Model 30, which had 64 KB of total memory (with no tricks for upgrading it). It sold for $250,000, which is roughly $2 million in today's dollars. The ATmega168 costs about $1, less if you buy a lot of them at once. If you check out, say, Boeing's price list, you will discover that airplane prices have not dropped by a factor of 250,000 in the past 50 or so years. Nor have the prices of cars or televisions or anything except computers.

In addition, the ATmega168 has an on-chip interrupt controller, serial port interface (SPI), and timers, which are essential for real-time applications. There are also three 8-bit digital I/O ports, which allow the ATmega168 to control up to 24 external buttons, lights, sensors, actuators, and so on. It is the presence of the timers and I/O ports more than anything else that makes it possible to use the ATmega168 for embedded applications without any additional chips.

The ATmega168 is a synchronous processor, with most instructions taking one clock cycle, although some take more. The processor is pipelined, such that while one instruction is being fetched, the previous instruction is being executed. The pipeline is only two stages, however, fetch and execute. To execute instructions in one cycle, the clock cycle must accommodate reading the register from the register file, followed by executing the instruction in the ALU, followed by writing the register back to the register file. Because all of these operations occur in one clock cycle, there is no need for bypass logic or stall detection. Program instructions are executed in order, in one cycle, and without overlap with other instructions.

While we could go into more detail about the ATmega168, the description above and Fig. 4-50 give the basic idea. The ATmega168 has a single main bus (to reduce chip area), a heterogeneous set of registers, and a variety of memories and I/O devices hanging off the main bus. On each data path cycle, two operands are

read from the register file and run through the ALU and the results stored back into a register, just as on more modern computers.

4.7 COMPARISON OF THE I7, OMAP4430, AND ATMEGA168

Our three examples are very different, yet even they exhibit a certain amount of commonality. The Core i7 has an ancient CISC instruction set that Intel's engineers would dearly love to toss into San Francisco Bay, except that doing so would violate California's water pollution laws. The OMAP4430 is a pure RISC design, with a lean and mean instruction set. The ATmega168 is a simple 8-bit processor for embedded applications. Yet the heart of each of them is a set of registers and one or more ALUs that perform simple arithmetic and Boolean operations on register operands.

Despite their obvious external differences, the Core i7 and the OMAP4430 have fairly similar execution units. Both of the execution units accept micro-operations that contain an opcode, two source registers, and a destination register. Both of them can execute a micro-operation in one cycle. Both of them have deep pipelines, branch prediction, and split I- and D-caches.

This internal similarity is not an accident or even due to the endless job-hopping by Silicon Valley engineers. As we saw with our Mic-3 and Mic-4 examples, it is easy and natural to build a pipelined data path that takes two source registers, runs them through an ALU, and stores the results in a register. Figure 4-34 shows this pipeline graphically. With current technology, this is the most effective design.

The main difference between the Core i7 and the OMAP4430 is how they get from their ISA instruction set to the execution unit. The Core i7 has to break up its CISC instructions to get them into the three-register format needed by the execution unit. That is what the front end in Fig. 4-47 is all about—hacking big instructions into nice, neat micro-operations. The OMAP4430 does not have to do anything because its native ARM instructions are already nice, neat micro-operations. This is why most new ISAs are of the RISC type—to provide a better match between the ISA instruction set and the internal execution engine.

It is instructive to compare our final design, the Mic-4, to these two real-world examples. The Mic-4 is most like the Core i7. Both of them have the job of interpreting a non-RISC ISA instruction set. Both of them do this by breaking the ISA instructions into micro-operations with an opcode, two source registers, and a destination register. In both cases, the micro-operations are deposited in a queue for execution later. The Mic-4 has a strict in-order issue, in-order execute, in-order retire design, whereas the Core i7 has an in-order issue, out-of-order execute, in-order retire policy.

The Mic-4 and the OMAP4430 are not really comparable at all because the OMAP4430 has RISC instructions (i.e., three-register micro-operations) as its ISA

instruction set. They do not have to be broken up. They can be executed as is, each in a single data path cycle.

In contrast to the Core i7 and the OMAP4430, the ATmega168 is a simple machine indeed. It is more RISC like than CISC like because most of its simple instructions can be executed in one clock cycle and do not need to be decomposed. It has no pipelining and no caching, and it has in-order issue, in-order execute, and in-order retirement. In its simplicity, it is much akin to the Mic-1.

4.8 SUMMARY

The heart of every computer is the data path. It contains some registers, one, two or three buses, and one or more functional units such as ALUs and shifters. The main execution loop consists of fetching some operands from the registers and sending them over the buses to the ALU and other functional unit for execution. The results are then stored back in the registers.

The data path can be controlled by a sequencer that fetches microinstructions from a control store. Each microinstruction contains bits that control the data path for one cycle. These bits specify which operands to select, which operation to perform, and what to do with the results. In addition, each microinstruction specifies its successor, typically explicitly by containing its address. Some microinstructions modify this base address by ORing bits into the address before it is used.

The IJVM machine is a stack machine with 1-byte opcodes that push words onto the stack, pop words from the stack, and combine (e.g., add) words on the stack. A microprogrammed implementation was given for the Mic-1 microarchitecture. By adding an instruction fetch unit to preload the bytes in the instruction stream, many references to the program counter could be eliminated and the machine greatly speeded up.

There are many ways to design the microarchitecture level. Many trade-offs exist, including two-bus versus three-bus designs, encoded versus decoded microinstruction fields, presence or absence of prefetching, shallow or deep pipelines, and much more. The Mic-1 is a simple, software-controlled machine with sequential execution and no parallelism. In contrast, the Mic-4 is a highly parallel microarchitecture with a seven-stage pipeline.

Performance can be improved in a variety of ways. Cache memory is a major one. Direct-mapped caches and set-associative caches are commonly used to speed up memory references. Branch prediction, both static and dynamic, is important, as are out-of-order execution, and speculative execution.

Our three example machines, the Core i7, OMAP4430, and ATmega168, all have microarchitectures not visible to the ISA assembly-language programmers. The Core i7 has a complex scheme for converting the ISA instructions into micro-operations, caching them, and feeding them into a superscalar RISC core for out-of-order execution, register renaming, and every other trick in the book to get

the last possible drop of speed out of the hardware. The OMAP4430 has a deep pipeline, but is further relatively simple, with in-order issue, in-order execution, and in-order retirement. The ATmega168 is very simple, with a straightforward single main bus to which a handful of registers and one ALU are attached.

PROBLEMS

1. What are the four steps CPUs use to execute instructions?
2. In Fig. 4-6, the B bus register is encoded in a 4-bit field, but the C bus is represented as a bit map. Why?
3. In Fig. 4-6 there is a box labeled "High bit." Give a circuit diagram for it.
4. When the JMPC field in a microinstruction is enabled, MBR is ORed with NEXT_ADDRESS to form the address of the next microinstruction. Are there any circumstances in which it makes sense to have NEXT_ADDRESS be 0x1FF and use JMPC?
5. Suppose that in the example of Fig. 4-14(a) the statement

   ```
   k = 5;
   ```

   is added after the if statement. What would the new assembly code be? Assume that the compiler is an optimizing compiler.
6. Give two different IJVM translations for the following Java statement:

   ```
   i = k + n + 5;
   ```

7. Give the Java statement that produced the following IJVM code:

   ```
   ILOAD j
   ILOAD n
   ISUB
   BIPUSH 7
   ISUB
   DUP
   IADD
   ISTORE i
   ```

8. In the text we mentioned that when translating the statement

   ```
   if (Z) goto L1; else goto L2
   ```

   to binary, *L2* has to be in the bottom 256 words of the control store. Would it not be equally possible to have *L1* at, say, 0x40 and *L2* at 0x140? Explain your answer.
9. In the microprogram for Mic-1, in if_icmpeq3, MDR is copied to H. A few lines later it is subtracted from TOS to check for equality. Surely it is better to have one statement:

```
if_cmpeq3       Z = TOS – MDR; rd
```

Why is this not done?

10. How long does a 2.5-GHz Mic-1 take to execute the following Java statement

```
i = j + k;
```

Give your answer in nanoseconds.

11. Repeat the previous question, only now for a 2.5-GHz Mic-2. Based on this calculation, how long would a program that runs for 100 sec on the Mic-1 take on the Mic-2?

12. Write microcode for the Mic-1 to implement the JVM POPTWO instruction. This instruction removes two words from the top of the stack.

13. On the full JVM machine, there are special 1-byte opcodes for loading locals 0 through 3 onto the stack instead of using the general ILOAD instruction. How should IJVM be modified to make the best use of these instructions?

14. The instruction ISHR (arithmetic shift right integer) exists in JVM but not in IJVM. It uses the top two values on the stack, replacing them with a single value, the result. The second-from-top word of the stack is the operand to be shifted. Its content is shifted right by a value between 0 and 31, inclusive, depending on the value of the 5 least significant bits of the top word on the stack (the other 27 bits of the top word are ignored). The sign bit is replicated to the right for as many bits as the shift count. The opcode for ISHR is 122 (0x7A).

a. What is the arithmetic operation equivalent to left shift with a count of 2?
b. Extend the microcode to include this instruction as a part of IJVM.

15. The instruction ISHL (shift left integer) exists in JVM but not in IJVM. It uses the top two values on the stack, replacing the two with a single value, the result. The second-from-top word of the stack is the operand to be shifted. Its content is shifted left by a value between 0 and 31, inclusive, depending on the value of the 5 least significant bits of the top word on the stack (the other 27 bits of the top word are ignored). Zeros are shifted in from the right for as many bits as the shift count. The opcode for ISHL is 120 (0x78).

a. What is the arithmetic operation equivalent to shifting left with a count of 2?
b. Extend the microcode to include this instruction as a part of IJVM.

16. The JVM INVOKEVIRTUAL instruction needs to know how many parameters it has. Why?

17. Implement the JVM DLOAD instruction for the Mic-2. It has a 1-byte index and pushes the local variable at this position onto the stack. Then it pushes the next higher word onto the stack as well.

18. Draw a finite-state machine for tennis scoring. The rules of tennis are as follows. To win, you need at least four points and you must have at least two points more than your opponent. Start with a state (0, 0) indicating that no one has scored yet. Then add a state (1, 0) meaning that *A* has scored. Label the arc from (0, 0) to (1, 0) with an *A*. Now add a state (0, 1) indicating that *B* has scored, and label the arc from (0, 0) with a *B*. Continue adding states and arcs until all the possible states have been included.

19. Reconsider the previous problem. Are there any states that could be collapsed without changing the result of any game? If so, which ones are equivalent?

20. Draw a finite-state machine for branch prediction that is more tenacious than Fig. 4-42. It should change only predictions after three consecutive mispredictions.

21. The shift register of Fig. 4-27 has a maximum capacity of 6 bytes. Could a cheaper version of the IFU be built with a 5-byte shift register? How about a 4-byte one?

22. Having examined cheaper IFUs in the previous question, now let us examine more expensive ones. Would there ever be any point to have a much larger shift register in the IU, say, 12 bytes? Why or why not?

23. In the microprogram for the Mic-2, the code for if_icmpeq6 goes to T when Z is set to 1. However, the code at T is the same as goto1. Would it have been possible to go to goto1 directly? Would doing so have made the machine faster?

24. In the Mic-4, the decoding unit maps the IJVM opcode onto the ROM index where the corresponding micro-operations are stored. It would seem to be simpler to just omit the decoding stage and feed the IJVM opcode into the queueing directly. It could use the IJVM opcode as an index into the ROM, the same way as the Mic-1 works. What is wrong with this plan?

25. Why are computers equipped with multiple layers of cache? Would it not be better to simply have one big one?

26. A computer has a two-level cache. Suppose that 60% of the memory references hit on the first level cache, 35% hit on the second level, and 5% miss. The access times are 5 nsec, 15 nsec, and 60 nsec, respectively, where the times for the level 2 cache and memory start counting at the moment it is known that they are needed (e.g., a level 2 cache access does not even start until the level 1 cache miss occurs). What is the average access time?

27. At the end of Sec. 4.5.1, we said that write allocation wins only if there are likely to be multiple writes to the same cache line in a row. What about the case of a write followed by multiple reads? Would that not also be a big win?

28. In the first draft of this book, Fig. 4-39 showed a three-way associative cache instead of a four-way associative cache. One of the reviewers threw a temper tantrum, claiming that students would be horribly confused by this because 3 is not a power of 2 and computers do everything in binary. Since the customer is always right, the figure was changed to a four-way associative cache. Was the reviewer right? Discuss your answer.

29. Many computer architects spend a lot of time making their pipelines deeper. Why?

30. A computer with a five-stage pipeline deals with conditional branches by stalling for the next three cycles after hitting one. How much does stalling hurt the performance if 20% of all instructions are conditional branches? Ignore all sources of stalling except conditional branches.

31. A computer prefetches up to 20 instructions in advance. However, on the average, four of these are conditional branches, each with a probability of 90% of being predicted correctly. What is the probability that the prefetching is on the right track?

32. Suppose that we were to change the design of the machine used in Fig. 4-43 to have 16 registers instead of 8. Then we change I6 to use R8 as its destination. What happens in the cycles starting at cycle 6?

33. Normally, dependences cause trouble with pipelined CPUs. Are there any optimizations that can be done with WAW dependences that might actually improve matters? What?

34. Rewrite the Mic-1 interpreter but having LV now point to the first local variable instead of to the link pointer.

35. Write a simulator for a 1-way direct mapped cache. Make the number of entries and the line size parameters of the simulation. Experiment with it and report on your findings.

This page intentionally left blank

5

THE INSTRUCTION SET ARCHITECTURE LEVEL

This chapter discusses the Instruction Set Architecture (ISA) level in detail. This level, as we saw in Fig. 1-2, is positioned between the microarchitecture level and the operating system level. Historically, this level was developed before any of the other levels, and, in fact, was originally the only level. To this day this level is sometimes referred to simply as "the architecture" of a machine or sometimes (incorrectly) as "assembly language."

The ISA level has a special importance for system architects: it is the interface between the software and the hardware. While it might be possible to have the hardware directly execute programs written in C, C++, Java, or some other high-level language, this would not be a good idea. The performance advantage of compiling over interpreting would then be lost. Furthermore, to be of much practical use, most computers have to be able to execute programs written in multiple languages, not just one.

The approach that essentially all system designers take is to have programs in various high-level languages be translated to a common intermediate form—the ISA level—and build hardware that can execute ISA-level programs directly. The ISA level defines the interface between the compilers and the hardware. It is the language that both of them have to understand. The relationship among the compilers, the ISA level, and the hardware is shown in Fig. 5-1.

Ideally, when designing a new machine, the architects will spend time talking to both the compiler writers and the hardware engineers to find out what features they want in the ISA level. If the compiler writers want some feature that the en-

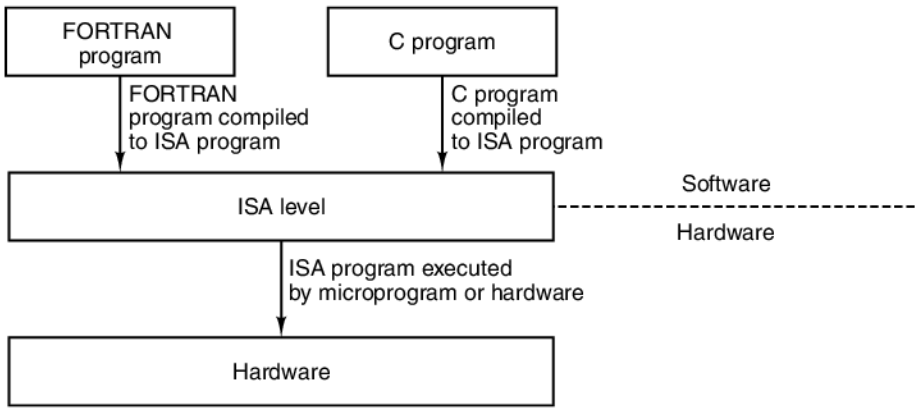

Figure 5-1. The ISA level is the interface between the compilers and the hardware.

gineers cannot implement in a cost-effective way (e.g., a branch-and-do-payroll instruction), it does not go in. Similarly, if the hardware folks have some nifty new feature they want to put in (e.g., a memory in which the words whose addresses are prime numbers are superfast), but the software folks cannot figure out how to generate code to use it, it will die on the drawing board. After much negotiation and simulation, an ISA perfectly optimized for the intended programming languages will emerge and be implemented.

That is the theory. Now the grim reality. When a new machine comes along, the first question all the potential customers ask is: "Is it compatible with its predecessor?" The second is: "Can I run my old operating system on it?" The third is: "Will it run all my existing application programs unmodified?" If any of the answers are "no," the designers will have a lot of explaining to do. Customers are rarely keen on throwing out all their old software and starting all over again.

This attitude puts a great deal of pressure on computer architects to keep the ISA the same between models, or at least make it **backward compatible**. By this we mean that the new machine must be able to run old programs without change. However, it is completely acceptable for the new machine to have new instructions and other features that can be exploited only by new software. In terms of Fig. 5-1, as long as the designers make the ISA backward compatible with the previous models, they are pretty much free to do whatever they want with the hardware, as hardly anyone cares about the real hardware (or even knows what it does). They can switch from a microprogrammed design to direct execution, or add pipelines or superscalar facilities or anything else they want, provided that they maintain backward compatibility with the previous ISA. The goal is to make sure that old programs run on the new machine. The challenge then becomes building better machines subject to the backward compatibility constraint.

The foregoing is not intended to imply that ISA design does not matter. A good ISA has significant advantages over a poor one, particularly in raw computing power vs. cost. For otherwise equivalent designs, different ISAs might account for a difference of as much as 25% in performance. Our point is just that market forces make it hard (but not impossible) to throw out an ancient ISA and introduce a new one. Nevertheless, every once in a while a new general-purpose ISA emerges, and in specialized markets (e.g., embedded systems or multimedia processors) this occurs much more frequently. Consequently, understanding ISA design is important.

What makes a good ISA? There are two primary factors. First, a good ISA should define a set of instructions that can be implemented efficiently in current and future technologies, resulting in cost-effective designs over several generations. A poor design is more difficult to implement and may require many more gates to implement a processor and more memory for executing programs. It also may run slower because the ISA obscures opportunities to overlap operations, requiring much more sophisticated designs to achieve equivalent performance. A design that takes advantage of the peculiarities of a particular technology may be a flash in the pan, providing a single generation of cost-effective implementations, only to be surpassed by more forward-looking ISAs.

Second, a good ISA should provide a clean target for compiled code. Regularity and completeness of a range of choices are important traits that are not always present in an ISA. These are important properties for a compiler, which may have trouble making the best choice among limited alternatives, particularly when some seemingly obvious alternatives are not permitted by the ISA. In short, since the ISA is the interface between the hardware and the software, it should make the hardware designers happy (be easy to implement efficiently) and make the software designers happy (be easy to generate good code for).

5.1 OVERVIEW OF THE ISA LEVEL

Let us start our study of the ISA level by asking what it is. This may seem like a simple question, but it has more complications than one might at first imagine. In the following section we will raise some of these issues. Then we will look at memory models, registers, and instructions.

5.1.1 Properties of the ISA Level

In principle, the ISA level is defined by how the machine appears to a machine-language programmer. Since no (sane) person does much programming in machine language any more, let us redefine this to say that ISA-level code is what a compiler outputs (ignoring operating-system calls and ignoring symbolic assembly language for the moment). To produce ISA-level code, the compiler writer has

to know what the memory model is, what registers there are, what data types and instructions are available, and so on. The collection of all this information is what defines the ISA level.

According to this definition, issues such as whether the microarchitecture is microprogrammed or not, whether it is pipelined or not, whether it is superscalar or not, and so on are not part of the ISA level because they are not visible to the compiler writer. However, this remark is not entirely true because some of these properties do affect performance, and that is visible to the compiler writer. Consider, for example, a superscalar design that can issue back-to-back instructions in the same cycle, provided that one is an integer instruction and one is a floating-point instruction. If the compiler alternates integer and floating-point instructions, it will get observably better performance than if it does not. Thus the details of the superscalar operation *are* visible at the ISA level, so the separation between the layers is not quite as clean as it might appear at first.

For some architectures, the ISA level is specified by a formal defining document, often produced by an industry consortium. For others it is not. For example, the ARM v7 (version 7 ARM ISA) has an official definition published by ARM Ltd. The purpose of a defining document is to make it possible for different implementers to build the machines and have them all run exactly the same software and get exactly the same results.

In the case of the ARM ISA, the idea is to allow multiple chip vendors to manufacture ARM chips that are functionally identical, differing only in performance and price. To make this idea work, the chip vendors have to know what an ARM chip is supposed to do (at the ISA level). Therefore the defining document tells what the memory model is, what registers are present, what the instructions do, and so on, but not what the microarchitecture is like.

Such defining documents contain **normative** sections, which impose requirements, and **informative** sections, which are intended to help the reader but are not part of the formal definition. The normative sections constantly use words like *shall*, *may not*, and *should* to require, prohibit, and suggest aspects of the architecture, respectively. For example, a sentence like

Executing a reserved opcode shall cause a trap.

says that if a program executes an opcode that is not defined, it must cause a trap and not be just ignored. An alternative approach might be to leave this open, in which case the sentence might read

The effect of executing a reserved opcode is implementation defined.

This means that the compiler writer cannot count on any particular behavior, thus giving different implementers the freedom to make different choices. Most architectural specifications are accompanied by test suites that check to see if an implementation that claims to conform to the specification really does.

It is clear why the ARM v7 has a document that defines its ISA level: so that all ARM chips will run the same software. For many years, there was no formal defining document for the IA-32 ISA (sometimes called the **x86** ISA) because Intel did not want to make it easy for other vendors to make Intel-compatible chips. In fact, Intel went to court to try to stop other vendors from cloning its chips, although it lost the case. In the late 1990s, however, Intel finally released a complete specification of the IA-32 instruction set. Perhaps this was because they felt the error of their ways and wanted to help out fellow architects and programmers, or perhaps it was because the United States, Japan, and Europe were all investigating Intel for possibly violating antitrust laws. This well-written ISA reference is still updated today at Intel's developer website (http://developer.intel.com). The version released with Intel's Core i7 weighs in at 4161 pages, reminding us once again that the Core i7 is a *complex* instruction set computer.

Another important property of the ISA level is that on most machines there are at least two modes. **Kernel mode** is intended to run the operating system and allows all instructions to be executed. **User mode** is intended to run application programs and does not permit certain sensitive instructions (such as those that manipulate the cache directly) to be executed. In this chapter we will primarily focus on user-mode instructions and properties.

5.1.2 Memory Models

All computers divide memory up into cells that have consecutive addresses. The most common cell size at the moment is 8 bits, but cell sizes from 1 to 60 bits have been used in the past (see Fig. 2-10). An 8-bit cell is called a **byte** (or **octet**). The reason for using 8-bit bytes is that ASCII characters are 7 bits, so one ASCII character (plus a rarely used parity bit) fits into a byte. Other codes, such as Unicode and UTF-8, use multiples of 8 bits to represent characters.

Bytes are generally grouped into 4-byte (32-bit) or 8-byte (64-bit) words with instructions available for manipulating entire words. Many architectures require words to be aligned on their natural boundaries. For example, a 4-byte word may begin at address 0, 4, 8, etc., but not at address 1 or 2. Similarly, an 8-byte word may begin at address 0, 8, or 16, but not at address 4 or 6. Alignment of 8-byte words is illustrated in Fig. 5-2.

Alignment is often required because memories operate more efficiently that way. The Core i7, for example, fetches 8 bytes at a time from memory using a DDR3 interface which supports only aligned 64-bit accesses.. Thus the Core i7 could not even make a nonaligned memory reference if it wanted to because memory interface requires addresses that are multiples of 8.

However, this alignment requirement sometimes causes problems. On the Core i7, programs are allowed to reference words starting at any address, a property that goes back to the 8088, which had a 1-byte-wide data bus (and thus no requirement about aligning memory references on 8-byte boundaries). If a Core i7

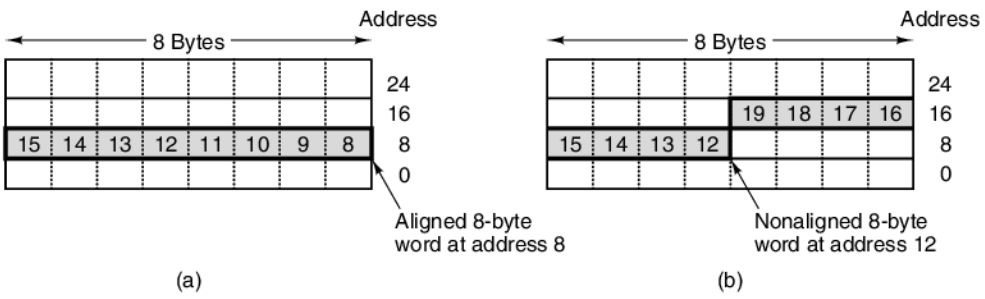

Figure 5-2. An 8-byte word in a little-endian memory. (a) Aligned. (b) Not aligned. Some machines require that words in memory be aligned.

program reads a 4-byte word at address 7, the hardware has to make one memory reference to get bytes 0 through 7 and a second to get bytes 8 through 15. Then the CPU has to extract the required 4 bytes from the 16 bytes read from memory and assemble them in the right order to form a 4-byte word. Doing this on a regular basis does not lead to blinding speed.

Having the ability to read words at arbitrary addresses requires extra logic on the chip, which makes it bigger and more expensive. The design engineers would love to get rid of it and simply require all programs to make word-aligned references to memory. The trouble is, whenever the engineers say: "Who cares about running musty old 8088 programs that reference memory wrong?" the folks in marketing have a succinct answer: "Our customers."

Most machines have a single linear address space at the ISA level, extending from address 0 up to some maximum, often $2^{32} - 1$ bytes or $2^{64} - 1$ bytes. However, a few machines have separate address spaces for instructions and data, so that an instruction fetch at address 8 goes to a different address space than a data fetch at address 8. This scheme is more complex than having a single address space, but it has two advantages. First, it becomes possible to have 2^{32} bytes of program and an additional 2^{32} bytes of data while using only 32-bit addresses. Second, because all writes automatically go to data space, it becomes impossible for a program to accidentally overwrite itself, thus eliminating one source of program bugs. Separating instruction and data spaces also makes attacks by malware much harder to pull off because the malware cannot change the program—it cannot even address it.

Note that having separate address spaces for instructions and data is not the same as having a split level 1 cache. In the former case the total amount of address space is doubled and reads to any given address yield different results, depending on whether an instruction or a data word is being read. With a split cache, there is still just one address space, only different caches store different parts of it.

Yet another aspect of the ISA level memory model is the memory semantics. It is perfectly natural to expect that a LOAD instruction that occurs after a STORE instruction and that references the same address will return the value just stored.

In many designs, however, as we saw in Chap. 4, microinstructions are reordered. Thus there is a real danger that the memory will not have the expected behavior. The problem gets even worse on a multiprocessor, with each of multiple CPUs sending a stream of (possibly reordered) read and write requests to shared memory.

System designers can take any one of several approaches to this problem. At one extreme, all memory requests can be serialized, so that each one is completed before the next is issued. This strategy hurts performance but gives the simplest memory semantics (all operations are executed in strict program order).

At the other extreme, no guarantees of any kind are given. To force an ordering on memory, the program must execute a SYNC instruction, which blocks the issuing of all new memory operations until all previous ones have completed. This design puts a great burden on the compilers because they have to understand how the underlying microarchitecture works in detail, but it gives the hardware designers the maximum freedom to optimize memory usage.

Intermediate memory models are also possible, in which the hardware automatically blocks the issuing of certain memory references (e.g., those involving a RAW or WAR dependence) but not others. While having all these peculiarities caused by the microarchitecture be exposed to the ISA level is annoying (at least to the compiler writers and assembly-language programmers), it is very much the trend. This trend is caused by the underlying implementations such as microinstruction reordering, deep pipelines, multiple cache levels, and so on. We will see more examples of such unnatural effects later in this chapter.

5.1.3 Registers

All computers have some registers visible at the ISA level. They are there to control execution of the program, hold temporary results, and serve other purposes. In general, the registers visible at the microarchitecture level, such as TOS and MAR in Fig. 4-1, are not visible at the ISA level. A few of them, however, such as the program counter and stack pointer, are visible at both levels. On the other hand, registers visible at the ISA level are always visible at the microarchitecture level since that is where they are implemented.

ISA-level registers can be roughly divided into two categories: special-purpose registers and general-purpose registers The special-purpose registers include things like the program counter and stack pointer, as well as other registers with a specific function. In contrast, the general-purpose registers are there to hold key local variables and intermediate results of calculations. Their main function is to provide rapid access to heavily used data (basically, avoiding memory accesses). RISC machines, with their fast CPUs and (relatively) slow memories, usually have at least 32 general-purpose registers, and the trend in new CPU designs is to have even more.

On some machines, the general-purpose registers are completely symmetric and interchangeable. Each one can do anything the others can do. If the registers

are all equivalent, a compiler can use R1 to hold a temporary result, but it can equally well use R25. The choice of register does not matter.

On other machines, however, some of the general-purpose registers may be somewhat special. For example, on the Core i7, there is a register called EDX that can be used as a general register, but which also receives half the product in a multiplication and holds half the dividend in a division.

Even when the general-purpose registers are completely interchangeable, it is common for the operating system or compilers to adopt conventions about how they are used. For example, some registers may hold parameters to procedures called and others may be used as scratch registers. If a compiler puts an important local variable in R1 and then calls a library procedure that thinks R1 is a scratch register available to it, when the library procedure returns, R1 may contain garbage. If there are system-wide conventions on how the registers are to be used, compilers and assembly-language programmers are advised to adhere to them to avoid trouble.

In addition to the ISA-level registers visible to user programs, there are always a substantial number of special-purpose registers available only in kernel mode. These registers control the various caches, memory, I/O devices, and other hardware features of the machine. They are used only by the operating system, so compilers and users do not have to know about them.

One control register that is something of a kernel/user hybrid is the **flags register** or PSW (**Program Status Word**). This register holds various miscellaneous bits that are needed by the CPU. The most important bits are the **condition codes**. These bits are set on every ALU cycle and reflect the status of the result of the most recent operation. Typical condition code bits include

N — Set when the result was Negative.

Z — Set when the result was Zero.

V — Set when the result caused an oVerflow.

C — Set when the result caused a Carry out of the leftmost bit.

A — Set when there was a carry out of bit 3 (Auxiliary carry—see below).

P — Set when the result had even Parity.

The condition codes are important because the comparison and conditional branch instructions (also called conditional jump instructions) use them. For example, the CMP instruction typically subtracts two operands and sets the condition codes based on the difference. If the operands are equal, then the difference will be zero and the *Z* condition code bit in the PSW register will be set. A subsequent BEQ (Branch EQual) instruction tests the *Z* bit and branches if it is set.

The PSW contains more than just the condition codes, but the full contents varies from machine to machine. Typical additional fields are the machine mode

(e.g., user or kernel), trace bit (used for debugging), CPU priority level, and interrupt enable status. Often the PSW is readable in user mode, but some of the fields can be written only in kernel mode (e.g., the user/kernel mode bit).

5.1.4 Instructions

The main feature of the ISA level is its set of machine instructions. These control what the machine can do. There are always LOAD and STORE instructions (in one form or another) for moving data between memory and registers and MOVE instructions for copying data among the registers. Arithmetic instructions are always present, as are Boolean instructions and instructions for comparing data items and branching on the results. We have seen some typical ISA instructions already (see Fig. 4-11) and will study many more in this chapter.

5.1.5 Overview of the Core i7 ISA Level

In this chapter we will discuss three widely different ISAs: Intel's IA-32, as embodied in the Core i7, the ARM v7 architecture, implemented in the OMAP4430 system-on-a-chip, and the AVR 8-bit architecture, used by the ATmega168 microcontroller. The intent is not to provide an exhaustive description of any of the ISAs, but rather to demonstrate important aspects of an ISA, and to show how these aspects can vary from one ISA to another. Let us start with the Core i7.

The Core i7 processor has evolved over many generations, tracing its lineage back to some of the earliest microprocessors ever built, as we discussed in Chap. 1. While the basic ISA maintains full support for execution of programs written for the 8086 and 8088 processors (which had the same ISA), it even contains remnants of the 8080, an 8-bit processor popular in the 1970s. The 8080, in turn, was strongly influenced by compatibility constraints with the still-earlier 8008, which was based on the 4004, a 4-bit chip used back when dinosaurs roamed the earth.

From a pure software standpoint, the 8086 and 8088 were straightforward 16-bit machines (although the 8088 had an 8-bit data bus). Their successor, the 80286, was also a 16-bit machine. Its main advantage was a larger address space, although few programs ever used it because it consisted of 16,384 64-KB segments rather than a linear 2^{30}-byte memory.

The 80386 was the first 32-bit machine in the Intel family. All the subsequent machines (80486, Pentium, Pentium Pro, Pentium II, Pentium III, Pentium 4, Celeron, Xeon, Pentium M, Centrino, Core 2 duo, Core i7, etc.) have essentially the same 32-bit architecture as the 80386, called **IA-32**, so it is this architecture that we will focus on here. The only major architectural change after the 80386 was the introduction of the MMX, SSE, and SSE2 instructions in later versions of x86 series. These instructions are highly specialized and designed to improve performance on multimedia applications. Another important extension was 64-bit x86

(often called x86-64), which increased the integer computations and virtual address size to 64 bits. While most extensions were introduced by Intel and later implemented by competitors, this was one case where AMD introduced an extension that Intel had to adopt.

The Core i7 has three operating modes, two of which make it act like an 8088. In **real mode**, all the features that have been added since the 8088 are turned off and the Core i7 behaves like a simple 8088. If any program does something wrong, the whole machine just crashes. If Intel had designed human beings, it would have put in a bit that made them revert back to chimpanzee mode (most of the brain disabled, no speech, sleeps in trees, eats mostly bananas, etc.)

One step up is **virtual 8086 mode**, which makes it possible to run old 8088 programs in a protected way. In this mode, a real operating system is in control of the whole machine. To run an old 8088 program, the operating system creates a special isolated environment that acts like an 8088, except that if its program crashes, the operating system is notified instead of the machine crashing. When a Windows user starts an MS-DOS window, the program run there is started in virtual 8086 mode to protect Windows itself from misbehaving MS-DOS programs.

The final mode is protected mode, in which the Core i7 actually acts like a Core i7 instead of a very expensive 8088. Four privilege levels are available and controlled by bits in the PSW. Level 0 corresponds to kernel mode on other computers and has full access to the machine. It is used by the operating system. Level 3 is for user programs. It blocks access to certain critical instructions and control registers to prevent a rogue user program from bringing down the entire machine. Levels 1 and 2 are rarely used.

The Core i7 has a huge address space, with memory divided into 16,384 segments, each going from address 0 to address $2^{32} - 1$. However, most operating systems (including UNIX and all versions of Windows) support only one segment, so most application programs effectively see a linear address space of 2^{32} bytes, and sometimes part of this is occupied by the operating system. Every byte in the address space has its own address, with words being 32 bits long. Words are stored in little-endian format (the low-order byte has the lowest address).

The Core i7's registers are shown in Fig. 5-3. The first four registers, EAX, EBX, ECX, and EDX, are 32-bit, more-or-less general-purpose registers, although each has its own peculiarities. EAX is the main arithmetic register; EBX is good for holding pointers (memory addresses); ECX plays a role in looping; EDX is needed for multiplication and division, where, together with EAX, it holds 64-bit products and dividends. Each of these registers contains a 16-bit register in the low-order 16 bits and an 8-bit register in the low-order 8 bits. These registers make it easy to manipulate 16- and 8-bit quantities, respectively. The 8088 and 80286 had only the 8- and 16-bit registers. The 32-bit registers were added with the 80386, along with the E prefix, which stands for Extended.

The next four are also somewhat general purpose, but with more peculiarities. The ESI and EDI registers are intended to hold pointers into memory, especially for

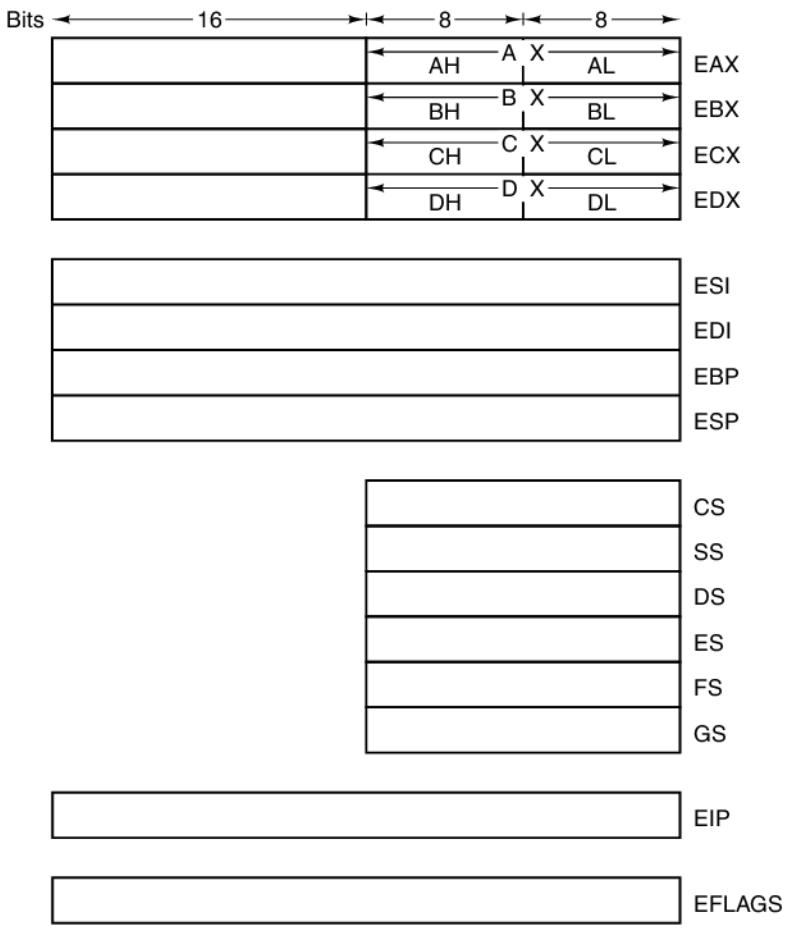

Figure 5-3. The Core i7's primary registers.

the hardware string-manipulation instructions, where ESI points to the source string and EDI points to the destination string. The EBP register is also a pointer register. It is typically used to point to the base of the current stack frame, the same as LV in IJVM. When a register (like EBP) is used to point to the base of the local stack frame, it is usually called the **frame pointer**. Finally, ESP is the stack pointer.

The next group of registers, CS through GS, are segment registers. To some extent, they are electronic trilobites, ancient fossils left over from the time the 8088 attempted to address 2^{20} bytes of memory using 16-bit addresses. Suffice it to say that when the Core i7 is set up to use a single linear 32-bit address space, they can be safely ignored. Next is EIP, which is the program counter (Extended Instruction Pointer). Finally, we come to EFLAGS, which is the PSW.

5.1.6 Overview of the OMAP4430 ARM ISA Level

The ARM Architecture was first introduced in 1985 by Acorn Computer. The architecture was inspired by the research done at Berkeley in the 1980s (Patterson, 1985, and Patterson and Séquin, 1982). The original ARM architecture (called the ARM2) was a 32-bit architecture that supported a 26-bit address space. The OMAP4430 utilizes the ARM Cortex A9 microarchitecture, which implements the version 7 of the ARM architecture, and that is the ISA we will describe in this chapter. For consistency with the rest of the book, we will refer to the OMAP4430 here, but at the ISA level, all designs based on the ARM Cortex A9 core implement the same ISA.

The OMAP4430 memory structure is clean and simple: addressable memory is a linear array of 2^{32} bytes. ARM processors are bi-endian, such that they can access memory with big- or little- endian order. The endian is specified in a system memory block that is read immediately after processor reset. To ensure that the system memory block is read correctly, it must be in little-endian format, even if the machine is to be configured for big-endian operation.

It is important that the ISA have a larger address-space limit than implementations need, because future implementations almost certainly will need to increase the size of memory the processor can access. The ARM ISA's 32-bit address space is giving many designers growing pains, since many ARM-based systems, such as smartphones, already have more than 2^{32} bytes of memory. To date, designers have worked around these problems by making the bulk of the memory flash-drive storage, which is accessed with a disk interface that supports a larger block-oriented address space. To address this potentially market-killing limitation, ARM (the company) recently published the definition of the ARM version 8 ISA, which support 64-bit address spaces.

A serious problem encountered with successful architectures has been that their ISA limited the amount of addressable memory. In computer science, the only error one cannot work around is not enough bits. One day your grandchildren will ask you how computers could do anything in the old days with only 32-bit addresses and only 4 GB of real memory when the average game needs 1 TB just to boot up.

The ARM ISA is clean, though the organization of the registers is somewhat quirky in an attempt to simplify some instruction encodings. The architecture maps the program counter into the integer register file (as register R15), as this allows branches to be created with ALU operations that have R15 as a destination register. Experience has shown that the register organization is more trouble than it is worth, but ye olde backwarde-compatibility rule made it well nigh impossible to get rid of.

The ARM ISA has two groups of registers. These are the 16 32-bit general-purpose registers and the 32 32-bit floating-point registers (if the VFP coprocessor is supported). The general-purpose registers are called R0 through R15,

although other names are used in certain contexts. The alternative names and functions of the registers are shown in Fig. 5-4.

Register	Alt. name	Function
R0–R3	A1–A4	Holds parameters to the procedure being called
R4–R11	V1–V8	Holds local variables for the current procedure
R12	IP	Intraprocedure call register (for 32-bit calls)
R13	SP	Stack pointer
R14	LR	Link register (return address for current function)
R15	PC	Program counter

Figure 5-4. The version 7 ARM's general registers.

All the general registers are 32 bits wide and can be read and written by a variety of load and store instructions. The uses given in Fig. 5-4 are based partly on convention, but also partly on how the hardware treats them. In general, it is unwise to deviate from the uses listed in the figure unless you have a Black Belt in ARM hacking and really, really know what you are doing. It is the responsibility of the compiler or programmer to be sure that the program accesses the registers correctly and performs the correct kind of arithmetic on them. For example, it is very easy to load floating-point numbers into the general registers and then perform integer addition on them, an operation that will produce utter nonsense, but which the CPU will cheerfully perform when so instructed.

The Vx registers are used to hold constants, variables, and pointers that are needed by procedures, and they should be stored and reloaded at procedure entries and exits if need be. The Ax registers are used for passing parameters to procedures to avoid memory references. We will explain how this works below.

Four dedicated registers are used for special purposes. The IP register works around the limitations of the ARM functional call instruction (BL) which cannot fully address all of its 2^{32} bytes of address space. If the target of a call is too far away for the instruction to express, the instruction will call a "veneer" code snippet that uses the address in the IP register as the destination of the function call. The SP register indicates the current top of the stack and fluctuates as words are pushed onto the stack or popped from it. The third special-purpose register is LR. It is used for procedure calls to hold the return address. The fourth special-purpose register, as mentioned earlier, is the program counter PC. Storing a value to this register redirects the fetching of instructions to that newly deposited PC address. Another important register in the ARM architecture is the program status register (PSR), which holds the status of previous ALU computations, including Zero, Negative, and Overflow among other bits.

The ARM ISA (when configured with the VFP coprocessor) also has 32 32-bit floating-point registers. These registers can be accessed either directly as 32 single-precision floating-point values or as 16 64-bit double-precision floating-point

values. The size of the floating-point register accessed is determined by the instruction; in general, all ARM floating-point instructions come in single- and double-precision variants.

The ARM architecture is a **load/store architecture**. That is, the only operations that access memory directly are load and store instructions to move data between the registers and the memory. All operands for arithmetic and logical instructions must come from registers or be supplied by the instruction (not memory), and all results must be saved in a register (not memory).

5.1.7 Overview of the ATmega168 AVR ISA Level

Our third example is the ATmega168. Unlike the Core i7 (which is used primarily in desktop machines and server farms), and the OMAP4430 (which is used primarily in phones, tablets, and other mobile devices), the ATmega168 is used in low-end embedded systems such as traffic lights and clock radios to control the device and manage the buttons, lights, and other parts of the user interface. In this section, we will give a brief technical introduction to the ATmega168 AVR ISA.

The ATmega168 has one mode and no protection hardware since it never runs multiple programs owned by potentially hostile users. The memory model is extremely simple. There is 16 KB of program memory and a second 1 KB of data memory. Each is its own distinct address space, so a particular address will reference different memory depending on whether the access is to the program or data memory. The program and data spaces are split to make it possible to implement the program space in flash and the data space in SRAM.

Several different implementations of memory are possible, depending on how much the designer wants to pay for the processor. In the simplest one, the ATmega48, there is a 4-KB flash for the program and a 512-byte SRAM for data. Both the flash and the RAM are on chip. For small applications, this amount of memory is often enough and having all the memory on the CPU chip is a big win. The ATmega88 has twice as much memory on chip: 8 KB of ROM and 1 KB of SRAM.

The ATmega168 uses a two-tiered memory organization to provide better program security. Program flash memory is divided into the *boot loader section* and *application section*, the size of each being determined by fuse bits that are one-time programmed when the microcontroller is first powered up. For security reasons, only code run from the boot loader section can update flash memory. With this feature, any code can be placed in the application area (including downloaded third-party applications) with confidence that it will never muck with other code in the system (because application code will be running from the application space which cannot write flash memory). To really tie down a system, a vendor can digitally sign code. With signed code, the boot loader loads code into the flash memory only if it is digitally signed by an approved software vendor. As such, the system

will run only code that has been "blessed" by a trusted software vendor. The approach is quite flexible in that even the boot loader can be replaced, if the new code has been properly digitally signed. This is similar to the way that Apple and TiVo ensure that the code running on their devices is safe from mischief.

The ATmega168 contains 32 8-bit general-purpose registers, which are accessed by instructions via a 5-bit field specifying which register to use. The registers are called R0 through R31. A peculiar property of the ATmega168 registers is that they are also present in the memory space. Byte 0 of the data space is equivalent to R0 of register set 0. When an instruction changes R0 and then later reads out memory byte 0, it finds the new value of R0 there. Similarly, byte 1 of memory is R1 and so on, up to byte 31. This arrangement is shown in Fig. 5-5.

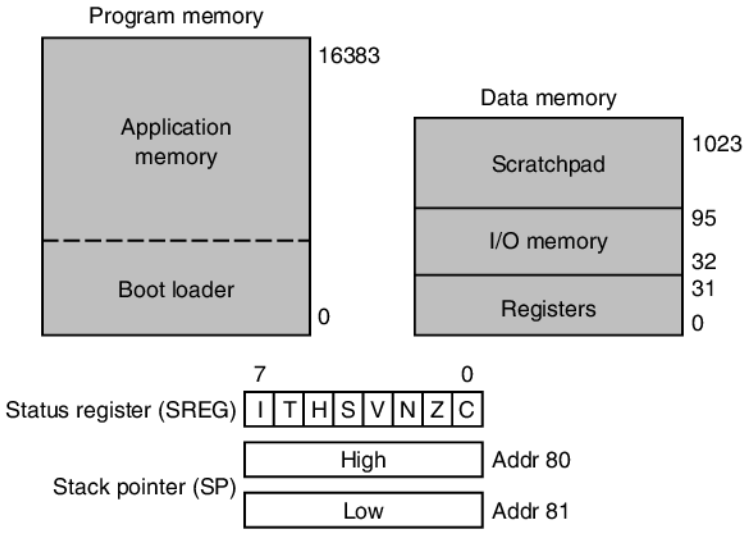

Figure 5-5. On-chip register and memory organization for the ATmega168.

Directly above the 32 general-purpose registers, at memory addresses 32 through 95, are 64 bytes of memory reserved for accessing I/O device registers, including the internal system-on-a-chip devices.

In addition to the four sets of eight registers, the ATmega168 has a small number of special-purpose registers, the most important of which are illustrated in Fig. 5-5. The *status register* contains, from left to right, the interrupt enable bit, the half-carry bit, the sign bit, the overflow bit, the negative flag, the zero flag, and the carry-out bit. All of these status bits, except the interrupt enable bit, are set as a result of arithmetic operations.

The status register I bit allows interrupts to be enabled or disabled globally. If the I bit is 0, all interrupts are disabled. Clearing this bit makes it possible to disable any further interrupts in a single instruction. Setting the bit allows any interrupts currently pending to occur as well as future ones. Each device has associated

with it an interrupt enable bit. If the device enable is set and the global interrupt enable I bit is set, the device can interrupt the processor.

The stack pointer SP holds the current address in data memory where PUSH and POP instructions will access their data, similar to the similarly named instruction in the Java JVM of Chap. 4. The stack pointer is located in I/O memory at address 80. A single 8-bit memory byte is too small to address 1024 bytes of data memory, so the stack pointer is composed of two consecutive locations in memory, forming a 16-bit address.

5.2 DATA TYPES

All computers need data. In fact, for many computer systems, the whole purpose is to process financial, commercial, scientific, engineering, or other data. The data have to be represented in some specific form inside the computer. At the ISA level, a variety of different data types are used. These will be explained below.

A key issue is whether there is hardware support for a particular data type. Hardware support means that one or more instructions expect data in a particular format, and the user is not free to pick a different format. For example, accountants have the peculiar habit of writing negative numbers with the minus sign to the right of the number rather than to the left, where computer scientists put it. Suppose that in an effort to impress his boss, the head of the computer center at an accounting firm changed all the numbers in all the computers to use the rightmost bit (instead of the leftmost bit) as the sign bit. This would no doubt make a big impression on the boss—because all the software would instantly cease to function correctly. The hardware expects a certain format for integers and does not work properly when given anything else.

Now consider another accounting firm, this one just having gotten a contract to verify the federal debt (how much the U.S. government owes everyone). Using 32-bit arithmetic would not work here because the numbers involved are larger than 2^{32} (about 4 billion). One solution is to use two 32-bit integers to represent each number, giving 64 bits in all. If the machine does not support **double-precision** numbers, all arithmetic on them will have to be done in software, but the two parts can be in either order since the hardware does not care. This is an example of a data type without hardware support and thus without a required hardware representation. In the following sections we will look at data types that are supported by the hardware, and thus for which specific formats are required.

5.2.1 Numeric Data Types

Data types can be divided into two categories: numeric and nonnumeric. Chief among the numeric data types are the integers. They come in many lengths, typically 8, 16, 32, and 64 bits. Integers count things (e.g., the number of screwdrivers

a hardware store has in stock), identify things (e.g., bank account numbers), and much more. Most modern computers store integers in two's complement binary notation, although other systems have also been used in the past. Binary numbers are discussed in Appendix A.

Some computers support unsigned as well as signed integers. For an unsigned integer, there is no sign bit and all the bits contain data. This data type has the advantage of an extra bit, so for example, a 32-bit word can hold a single unsigned integer in the range from 0 to $2^{32} - 1$, inclusive. In contrast, a two's complement signed 32-bit integer can handle only numbers up to $2^{31} - 1$, but, of course, it can also handle negative numbers.

For numbers that cannot be expressed as an integer, such as 3.5, floating-point numbers are used. These are discussed in Appendix B. They have lengths of 32, 64, or 128 bits. Most computers have instructions for doing floating-point arithmetic. Many computers have separate registers for holding integer operands and for holding floating-point operands.

Some programming languages, notably COBOL, allow decimal numbers as a data type. Machines that wish to be COBOL-friendly often support decimal numbers in hardware, typically by encoding a decimal digit in 4 bits and then packing two decimal digits per byte (binary code decimal format). However, binary arithmetic does not work correctly on packed decimal numbers, so special decimal-arithmetic-correction instructions are needed. These instructions need to know the carry out of bit 3. This is why the condition code often holds an auxiliary carry bit. As an aside, the infamous Y2K (Year 2000) problem was caused by COBOL programmers who decided that they could represent the year in two decimal digits (8 bits) rather than four decimal digits (or an 8-bit binary number), which can hold even more values (256) than two decimal digits (100). Some optimization!

5.2.2 Nonnumeric Data Types

Although most early computers earned their living crunching numbers, modern computers are often used for nonnumerical applications, such as email, surfing the Web, digital photography, and multimedia creation and playback. For these applications, other data types are needed and are frequently supported by ISA-level instructions. Characters are clearly important here, although not every computer provides hardware support for them. The most common character codes are ASCII and Unicode. These support 7-bit characters and 16-bit characters, respectively. Both were discussed in Chap. 2.

It is not uncommon for the ISA level to have special instructions intended for handling character strings, that is, consecutive runs of characters. These strings are sometimes delimited by a special character at the end. Alternatively a string-length field can be used to keep track of the end. The instructions can perform copy, search, edit, and other functions on the strings.

Boolean values are also important. A Boolean value can take on one of two values: true or false. In theory, a single bit can represent a Boolean, with 0 as false and 1 as true (or vice versa). In practice, a byte or word is used per Boolean value because individual bits in a byte do not have their own addresses and thus are hard to access. A common system uses the convention that 0 means false and everything else means true.

The one situation in which a Boolean value is normally represented by 1 bit is when there is an entire array of them, so a 32-bit word can hold 32 Boolean values. Such a data structure is called a **bit map** and occurs in many contexts. For example, a bit map can be used to keep track of free blocks on a disk. If the disk has n blocks, then the bit map has n bits.

Our last data type is the pointer, which is just a machine address. We have already seen pointers repeatedly. In the Mic-x machines, SP, PC, LV, and CPP are all examples of pointers. Accessing a variable at a fixed distance from a pointer, which is the way ILOAD works, is extremely common on all machines. While pointers are useful, they are also the source of a vast number of programming errors, often with very serious consequences. They must be used with great care.

5.2.3 Data Types on the Core i7

The Core i7 supports signed two's complement integers, unsigned integers, binary coded decimal numbers, and IEEE 754 floating-point numbers, as listed in Fig. 5-6. Due to its origins as a humble 8-bit/16-bit machine, it handles integers of these lengths as well as 32-bits, with numerous instructions for doing arithmetic, Boolean operations, and comparisons on them. The processor can optionally be run in 64-bit mode which also supports 64-bit registers and operations. Operands do not have to be aligned in memory, but better performance is achieved if word addresses are multiples of 4 bytes.

Type	8 Bits	16 Bits	32 Bits	64 Bits
Signed integer	×	×	×	× (64-bit)
Unsigned integer	×	×	×	× (64-bit)
Binary coded decimal integer	×			
Floating point			×	×

Figure 5-6. The Core i7 numeric data types. Supported types are marked with ×. Types marked with "64-bit" are only supported in 64-bit mode.

The Core i7 is also good at manipulating 8-bit ASCII characters: there are special instructions for copying and searching character strings. These instructions can be used both with strings whose length is known in advance and with strings whose end is marked. They are often used in string manipulation libraries.

5.2.4 Data Types on the OMAP4430 ARM CPU

The OMAP4430 ARM CPU supports a wide range of data formats, as shown in Fig. 5-7. For integers alone, it can support 8-, 16-, and 32-bit operands, both signed and unsigned. The handling of small data types in the OMAP4430 is slightly more clever than in the Core i7. Internally, the OMAP4430 is 32-bit machine with 32-bit datapaths and instructions. For loads and stores, the program can specify the size and sign of the value to be loaded (e.g., load signed byte: LDRSB). The value is then converted by load instructions into a comparable 32-bit value. Similarly, stores also specify the size and sign of the value to write to memory, and they access only the specified portion of the input register.

Signed integers use two's complement. Floating-point operands of 32 and 64 bits are included and conform to the IEEE 754 standard. Binary coded decimal numbers are not supported. All operands must be aligned in memory. Character and string data types are not supported by special hardware instructions. They are manipulated entirely in software.

Type	8 Bits	16 Bits	32 Bits	64 Bits
Signed integer	×	×	×	
Unsigned integer	×	×	×	
Binary coded decimal integer				
Floating point			×	×

Figure 5-7. The OMAP4430 ARM CPU numeric data types. Supported types are marked with ×.

5.2.5 Data Types on the ATmega168 AVR CPU

The ATmega168 has a very limited number of data types. With one exception, all the registers are 8 bits wide, so integers are also 8 bits wide. Characters are also 8 bits wide. In essence the only data type that is really supported by the hardware for arithmetic operations is the 8-bit byte, as shown in Fig. 5-8.

Type	8 Bits	16 Bits	32 Bits	64 Bits
Signed integer	×			
Unsigned integer	×	×		
Binary coded decimal integer				
Floating point				

Figure 5-8. The ATmega168 numeric data types. Supported types are marked with ×.

To facilitate memory accesses, the ATmega168 also includes limited support for 16-bit unsigned pointers. The 16-bit pointers X, Y, and Z, can be formed from the concatenation of 8-bit register pairs R26/R27, R28/R29, and R30/R31, respectively. When a load uses X, Y, or Z as an address operand, the processor will also optionally increment or decrement the value as needed.

5.3 INSTRUCTION FORMATS

An instruction consists of an opcode, usually along with some additional information such as where operands come from and where results go to. The general subject of specifying where the operands are (i.e., their addresses) is called **addressing** and will be discussed in detail later in this section.

Figure 5-9 shows several possible formats for level 2 instructions. An instruction always has an opcode to tell what the instruction does. There can be zero, one, two, or three addresses present.

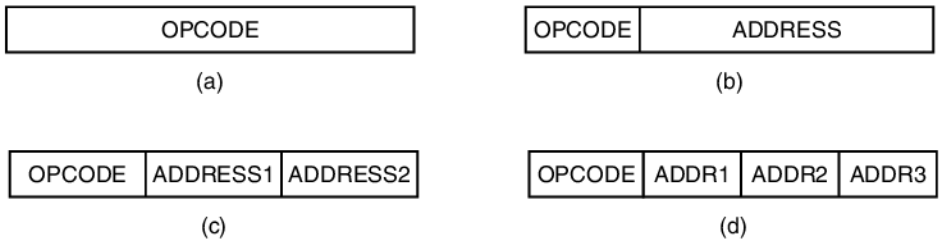

Figure 5-9. Four common instruction formats: (a) Zero-address instruction. (b) One-address instruction (c) Two-address instruction. (d) Three-address instruction.

On some machines, all instructions have the same length; on others there may be many different lengths. Instructions may be shorter than, the same length as, or longer than the word length. Having all the instructions be the same length is simpler and makes decoding easier but often wastes space, since all instructions then have to be as long as the longest one. Other trade-offs are also possible. Figure 5-10 shows some possible relationships between instruction length and word length.

5.3.1 Design Criteria for Instruction Formats

When a computer design team has to choose instruction formats for its machine, they must consider a number of factors. The difficulty of this decision should not be underestimated. The decision about the instruction format must be made early in the design of a new computer. If the computer is commercially successful, the instruction set may survive for 40 years or more. The ability to add

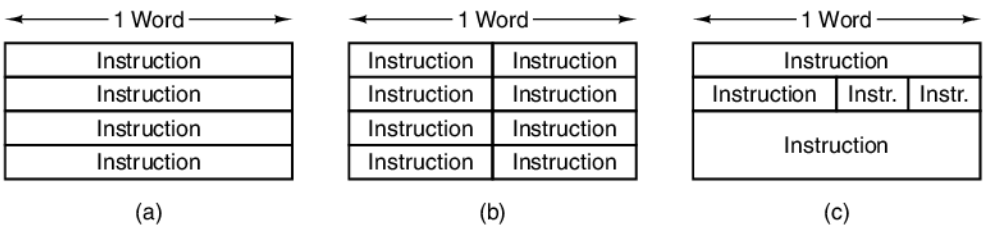

Figure 5-10. Some possible relationships between instruction and word length.

new instructions and exploit other opportunities that arise over an extended period is of great importance, but only if the architecture—and the company building it—survive long enough for the architecture to be a success.

The efficiency of a particular ISA is highly dependent on the technology with which the computer is to be implemented. Over a long period of time, this technology will undergo vast changes, and some of the ISA choices will be seen (with 20/20 hindsight) as unfortunate. For example, if memory accesses are fast, a stack-based design (like IJVM) is a good one, but if they are slow, then having many registers (like the OMAP4430 ARM CPU) is the way to go. Readers who think this choice is easy are invited to find a slip of paper and write down their predictions for (1) a typical CPU clock speed, and (2) a typical RAM access time for computers 20 years in the future. Fold this slip neatly and keep it for 20 years. Then unfold and read it. The humility-challenged can forget the slip of paper and just post their predictions to the Internet now.

Of course, even far-sighted designers may not be able to make all the right choices. And even if they could, they have to deal with the short term, too. If this elegant ISA is a little more expensive than its current ugly competitors, the company may not survive long enough for the world to appreciate the elegance of the ISA.

All things being equal, short instructions are better than long ones. A program consisting of n 16-bit instructions takes up only half as much memory space as n 32-bit instructions. With ever-declining memory prices, this factor might be less important in the future, were it not for the fact that software is metastasizing even faster than memory prices are dropping.

Furthermore, minimizing the size of the instructions may make them harder to decode or harder to overlap. Therefore, achieving the minimum instruction size must be weighed against the time required to decode and execute the instructions.

Another reason for minimizing instruction length is already important and becoming more so with faster processors: memory bandwidth (the number of bits/sec the memory is capable of supplying). The impressive growth in processor speeds over the last few decades has not been matched by equal increases in memory bandwidth. An increasingly common constraint on processors stems from the inability of the memory system to supply instructions and operands as rapidly as

the processor can consume them. Each memory has a bandwidth that is determined by its technology and engineering design. The bandwidth bottleneck applies not only to the main memory but also to all the caches.

If the bandwidth of an instruction cache is t bps and the average instruction length is r bits, the cache can deliver at most t/r instructions per second. Notice that this is an *upper limit* on the rate at which the processor can execute instructions, though there are current research efforts to breach even this seemingly insurmountable barrier. Clearly, the rate at which instructions can be executed (i.e., the processor speed) may be limited by the instruction length. Shorter instructions means a faster processor. Since modern processors can execute multiple instructions every clock cycle, fetching multiple instructions per clock cycle is imperative. This aspect of the instruction cache makes the size of instructions an important design criterion that has major implications for performance.

A second design criterion is sufficient room in the instruction format to express all the operations desired. A machine with 2^n operations with all instructions smaller than n bits is impossible. There simply will not be enough room in the opcode to indicate which instruction is needed. And history has shown over and over the folly of not leaving a substantial number of opcodes free for future additions to the instruction set.

A third criterion concerns the number of bits in an address field. Consider the design of a machine with an 8-bit character and a main memory that must hold 2^{32} characters. The designers could choose to assign consecutive addresses to units of 8, 16, 24, or 32 bits, as well as other possibilities.

Imagine what would happen if the design team degenerated into two warring factions, one advocating making the 8-bit byte the basic unit of memory, and the other advocating the 32-bit word. The former group would propose a memory of 2^{32} bytes, numbered 0, 1, 2, 3, ..., 4,294,967,295. The latter group would propose a memory of 2^{30} words numbered 0, 1, 2, 3, ..., 1,073,741,823.

The first group would point out that in order to compare two characters in the 32-bit word organization, the program would not only have to fetch the words containing the characters but would also have to extract each character from its word in order to compare them. Doing so costs extra instructions and therefore wastes space. The 8-bit organization, on the other hand, provides an address for every character, thus making the comparison much easier.

The 32-bit word supporters would retaliate by pointing out that their proposal requires only 2^{30} separate addresses, giving an address length of only 30 bits, whereas the 8-bit byte proposal requires 32 bits to address the same memory. A shorter address means a shorter instruction, which not only takes up less space but also requires less time to fetch. Alternatively, they could retain the 32-bit address to reference a 16-GB memory instead of a puny 4-GB memory.

This example demonstrates that in order to gain a finer memory resolution, one must pay the price of longer addresses and thus longer instructions. The ultimate in resolution is a memory organization in which every bit is directly addressable

(e.g., the Burroughs B1700). At the other extreme is a memory consisting of very long words (e.g., the CDC Cyber series had 60-bit words).

Modern computer systems have arrived at a compromise that, in some sense, captures the worst of both. They require all the bits necessary to address individual bytes, but memory accesses read one, two, or sometimes four words at a time. Reading 1 byte from memory on the Core i7, for example, brings in a minimum of 8 bytes and probably an entire 64-byte cache line.

5.3.2 Expanding Opcodes

In the preceding section we saw how short addresses and good memory resolution could be traded off against each other. In this section we will examine new trade-offs, involving both opcodes and addresses. Consider an $(n + k)$ bit instruction with a k-bit opcode and a single n-bit address. This instruction allows 2^k different operations and 2^n addressable memory cells. Alternatively, the same $n + k$ bits could be broken up into a $(k - 1)$ bit opcode, and an $(n + 1)$ bit address, meaning only half as many instructions but either twice as much memory addressable, or the same amount of memory but with twice the resolution. A $(k + 1)$ bit opcode and an $(n - 1)$ bit address gives more operations, but the price is either a smaller number of cells addressable, or poorer resolution and the same amount of memory addressable. Quite sophisticated trade-offs are possible between opcode bits and address bits as well as the simpler ones just described. The scheme discussed in the following paragraphs is called an **expanding opcode**.

The concept of an expanding opcode can be most clearly seen by a simple example. Consider a machine in which instructions are 16 bits long and addresses are 4 bits long, as shown in Fig. 5-11. This situation might be reasonable for a machine that has 16 registers (hence a 4-bit register address) on which all arithmetic operations take place. One design would be a 4-bit opcode and three addresses in each instruction, giving 16 three-address instructions.

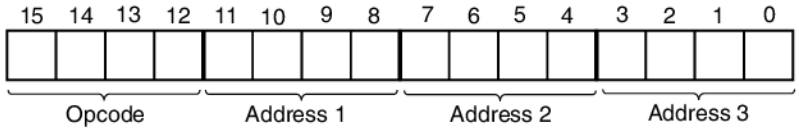

Figure 5-11. An instruction with a 4-bit opcode and three 4-bit address fields.

However, if the designers need 15 three-address instructions, 14 two-address instructions, 31 one-address instructions, and 16 instructions with no address at all, they can use opcodes 0 to 14 as three-address instructions but interpret opcode 15 differently (see Fig. 5-12).

Opcode 15 means that the opcode is contained in bits 8 to 15 instead of in bits 12 to 15. Bits 0 to 3 and 4 to 7 form two addresses, as usual. The 14 two-address

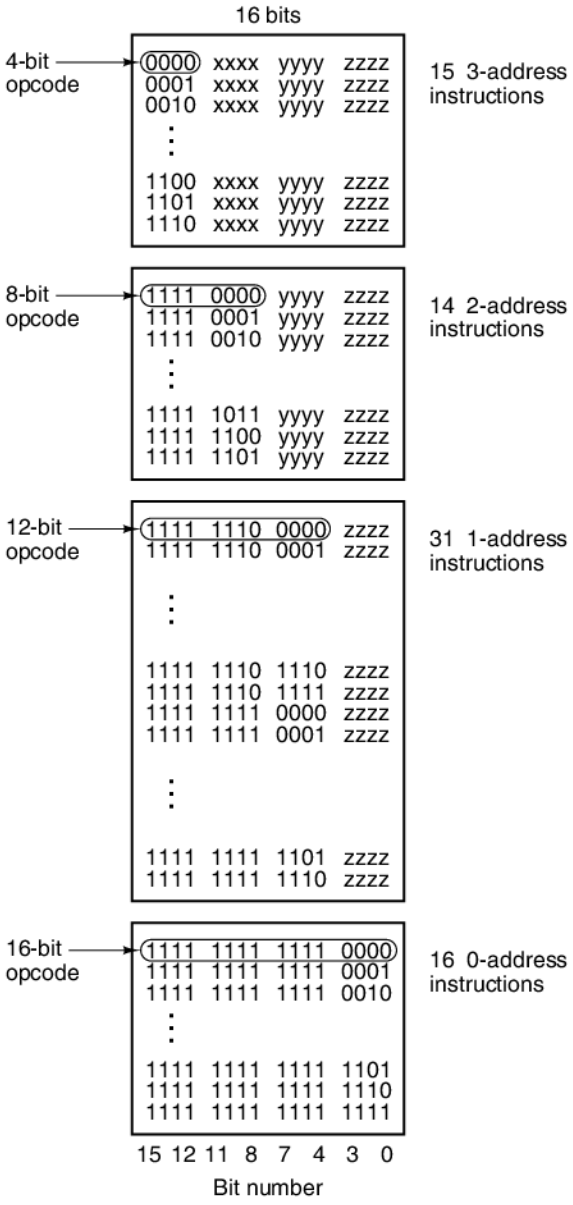

Figure 5-12. An expanding opcode allowing 15 three-address instructions, 14 two-address instructions, 31 one-address instructions, and 16 zero-address instructions. The fields marked *xxxx*, *yyyy*, and *zzzz* are 4-bit address fields.

instructions all have 1111 in the leftmost 4 bits, and numbers from 0000 to 1101 in bits 8 to 11. Instructions with 1111 in the leftmost 4 bits and either 1110 or 1111 in bits 8 to 11 will be treated specially. They will be treated as though their opcodes were in bits 4 to 15. The result is 32 new opcodes. Because only 31 are needed, opcode 111111111111 is interpreted to mean that the real opcode is in bits 0 to 15, giving 16 instructions with no address.

As we proceeded through this discussion, the opcode got longer and longer: the three-address instructions have a 4-bit opcode, the two-address instructions have an 8-bit opcode, the one-address instructions have a 12-bit opcode, and the zero-address instructions have a 16-bit opcode.

The idea of expanding opcodes demonstrates a trade-off between the space for opcodes and space for other information. In practice, expanding opcodes are not quite as clean and regular as in our example. In fact, the ability to use variable sizes of opcodes can be exploited in either of two ways. First, the instructions can all be kept the same length, by assigning the shortest opcodes to the instructions that need the most bits to specify other things. Second, the size of the *average* instruction can be minimized by choosing opcodes that are shortest for common instructions and longest for rare instructions.

Carrying the idea of variable-length opcodes to an extreme, it is possible to minimize the average instruction length by encoding every instruction to minimize the number of bits needed. Unfortunately, this would result in instructions of various sizes not even aligned on byte boundaries. While there have been ISAs that had this property (for example, the ill-fated Intel 432), the importance of alignment is so great for the rapid decoding of instructions that this degree of optimization is almost certainly counterproductive.

5.3.3 The Core i7 Instruction Formats

The Core i7 instruction formats are highly complex and irregular, having up to six variable-length fields, of which five are optional. The general pattern is shown in Fig. 5-13. This state of affairs occurred because the architecture evolved over many generations and included some poor choices early on. In the name of backward compatibility, these early decisions could not be reversed later. In general, for two-operand instructions, if one operand is in memory, the other may not be in memory. Thus instructions exist to add two registers, add a register to memory, and add memory to a register, but not to add a memory word to another memory word.

On earlier Intel architectures, all opcodes were 1 byte, though the concept of a prefix byte was used extensively for modifying some instructions. A **prefix byte** is an extra opcode stuck onto the front of an instruction to change its action. The WIDE instruction in IJVM is an example of a prefix byte. Unfortunately, at some point during the evolution, Intel ran out of opcodes, so one opcode, 0xFF, was designated as an **escape code** to permit a second instruction byte.

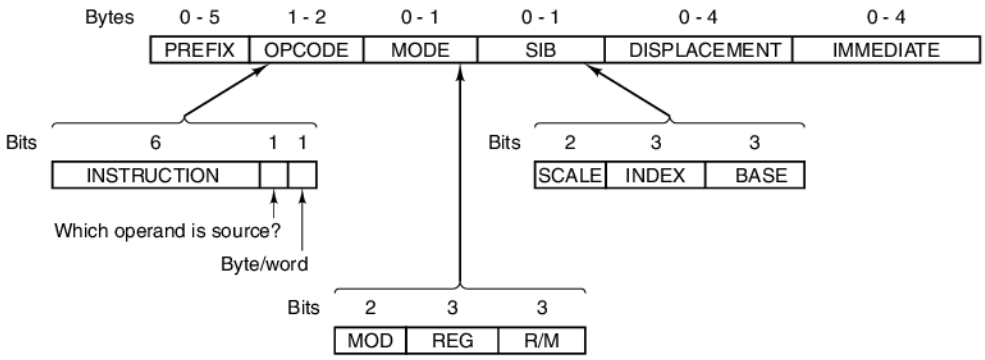

Figure 5-13. The Core i7 instruction formats.

The individual bits in the Core i7 opcodes do not give much information about the instruction. The only structure in the opcode field is the use of the low-order bit in some instructions to indicate byte/word, and the use of the adjoining bit to indicate whether the memory address (if it is present) is the source or the destination. Thus in general, the opcode must be fully decoded to determine what class of operation is to be performed—and thus how long the instruction is. This makes high-performance implementations difficult, since extensive decoding is necessary before it can even be determined where the next instruction starts.

Following the opcode byte in most instructions that reference an operand in memory is a second byte that tells all about the operand. These 8 bits are split up into a 2-bit MOD field and two 3-bit register fields, REG and R/M. Sometimes the first 3 bits of this byte are used as an extension for the opcode, giving a total 11 bits for the opcode. However, the 2-bit mode field means that there are only four ways to address operands and one of the operands must always be a register. Logically, any of EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP should be specifiable as either register, but the encoding rules prohibit some combinations and use them for special cases. Some modes require an additional byte, called **SIB** (**Scale, Index, Base**), giving a further specification. This scheme is not ideal, but a compromise given the competing demands of backward compatibility and the desire to add new features not originally envisioned.

In addition to all this, some instructions have 1, 2, or 4 more bytes specifying a memory address (displacement) and possibly another 1, 2, or 4 bytes containing a constant (immediate operand).

5.3.4 The OMAP4430 ARM CPU Instruction Formats

The OMAP4430 ARM ISA consists of both 16- and 32-bit instructions, aligned in memory. Instructions are generally simple, specifying only a single action. A typical arithmetic instruction specifies two registers to supply the source

operands and a single destination register. The 16-bit instructions are pared-down versions of the 32-bit instruction. They perform the same operations, but allow only two register operands (i.e., the destination register must be the same as one of the inputs) and only the first eight registers can be specified as inputs. The ARM architects called this smaller version of the ARM ISA the Thumb ISA.

Additional variants allows instructions to supply a 3, 8, 12, 16, or 24-bit unsigned constant instead of one of the registers. For a load instruction, two registers (or one register and an 8-bit signed constant) are added together to specify the memory address to read. The data are written into the other register specified.

The format of the 32-bit ARM instructions is illustrated in Fig. 5-14. The careful reader will notice that some of the formats have the same fields (e.g., LONG MULTIPLY and SWAP). In the case of the SWAP instruction, the decoder knows that the instruction is a SWAP only when it sees that the combination of field values for the MUL is illegal. Additional formats have been added for instruction extensions and the Thumb ISA. At the time of this writing, the number of instruction formats was 21 and rising. (Can it be long before we see some company advertising the "World's most complex RISC machine"?) The majority of instructions, however, still use the formats shown in the figure.

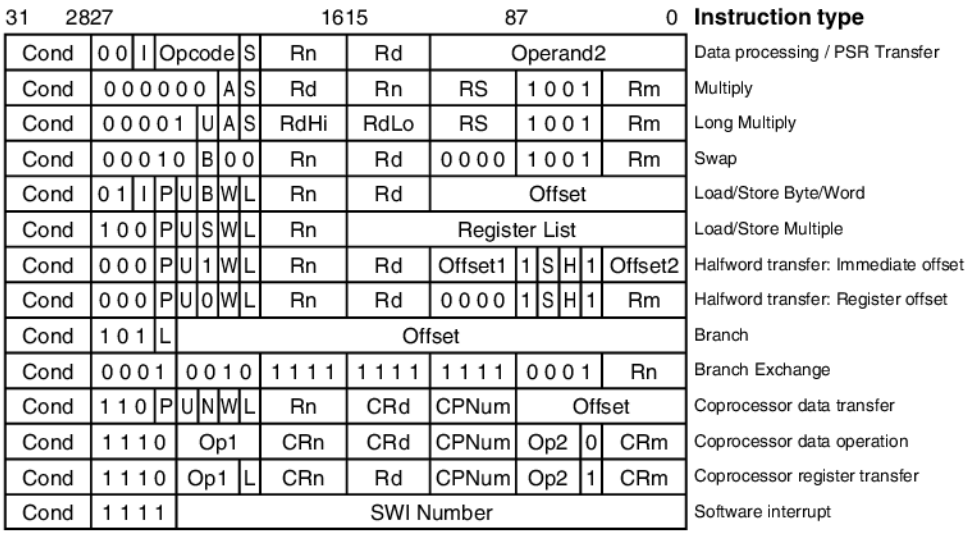

Figure 5-14. The 32-bit ARM instruction formats.

Bits 26 and 27 of every instruction are the first stop in determining the instruction format and tell hardware where to find the rest of the opcode, if there is more. For example, if bits 26 and 27 are both zero, and bit 25 is zero (operand is not an immediate), and the input operand shift is not illegal (which indicates the instruction is a multiply or branch exchange), then both sources are registers. If bit 25 is one, then one source is a register and the other is a constant in the range 0 to 4095.

In both cases, the destination is always a register. Sufficient encoding space is provided for up to 16 instructions, all of which are currently used.

With 32-bit instructions, it is not possible to include a 32-bit constant in the instruction. The MOVT instruction sets the 16 upper bits of a 32-bit register, leaving room for another instruction to set the remaining lower 16 bits. It is the only instruction to use this format.

Every 32-bit instruction has the same 4-bit field in the most significant bits (bits 28 to 31). This is the condition field, which makes any instruction a **predicated instruction**. A predicated instruction executes as normal in the processor, but before writing its result to a register (or memory), it first checks the condition of the instruction. For ARM instructions, the condition is based on the state of the processor status register (PSR). This register holds the arithmetic properties of the last arithmetic operation, (e.g., zero, negative, overflowed, etc). If the condition is not met, the result of the conditional instruction is dropped.

The branch instruction format encodes the largest immediate value, used to compute a target address for branches and function procedure calls. This instruction is special, because it is the only one where 24 bits of data are needed to specify an address. For this instruction, there is a single, 3-bit opcode. The address is the target address divided by four, making the range approximately $\pm 2^{25}$ bytes relative to the current instruction.

Clearly, the ARM ISA designers wanted to fully utilize every bit combination, including otherwise illegal operand combinations, for specifying instructions. The approach makes for extremely complicated decoding logic, but at the same time, it allows the maximum number of operations to be encoded into a fixed-length 16- or 32-bit instruction.

5.3.5 The ATmega168 AVR Instruction Formats

The ATmega168 has six simple instruction formats, as illustrated in Fig. 5-15. Instructions are 2 or 4 bytes in length. Format one consists of an opcode and two register operands, both of which are inputs and one is also the output of the instruction. The ADD instruction for registers uses this format, for example.

Format 2 is also 16 bits, consisting of an additional 16 opcodes and a 5-bit register number. This format increases the number of operations encoded in the ISA at the cost of reducing the number of instruction operands to one. Instructions that use this format perform a unary operation, taking a single register input and writing the output of the operation to the same register. Examples of this type of instruction include "negate" and "increment."

Format 3 has an 8-bit unsigned immediate operand. To accommodate such a large immediate value in a 16-bit instruction, instructions which use this encoding can have only one register operand (used as an input and output) and the register can only be R16–R31 (which limits the operand encoding to 4 bits). Also, the

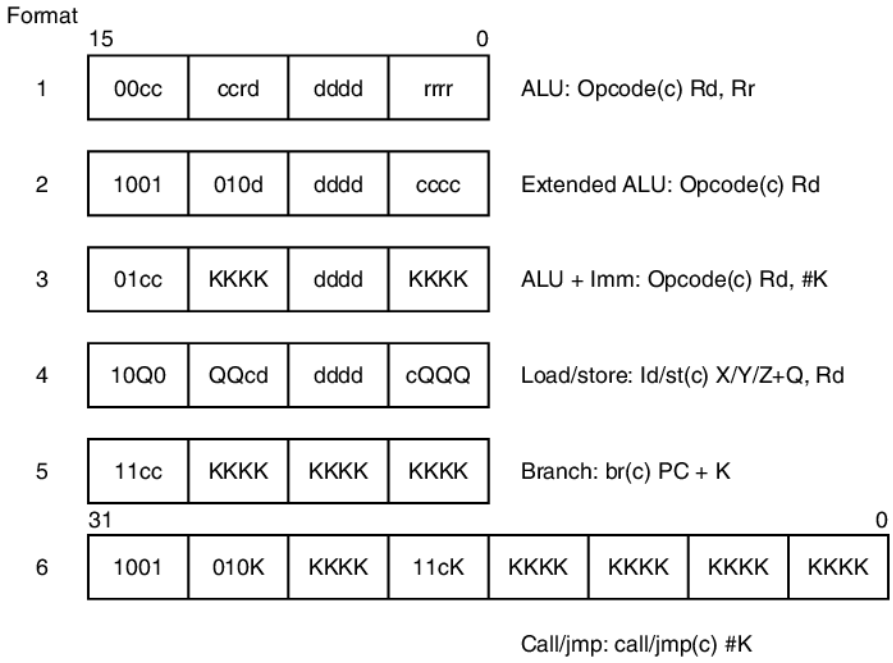

Figure 5-15. The ATmega168 AVR instruction formats.

number of opcode bits is cut in half, allowing only four instructions to use this format (SUBCI, SUBI, ORI, and ANDI).

Format 4 encodes load and store instruction, which includes a 6-bit unsigned immediate operand. The base register is a fixed register not specified in the instruction encoding because it is implied by the load/store opcode.

Formats 5 and 6 are used for jumps and procedure calls. The first format includes a 12-bit signed immediate value that is added to the instruction's PC value to compute the target of the instruction. The last format expands the offset to 22 bits, by making the AVR instruction 32 bits in size.

5.4 ADDRESSING

Most instructions have operands, so some way is needed to specify where they are. This subject, which we will now discuss, is called **addressing**.

5.4.1 Addressing Modes

Up until now, we have paid little attention to how the bits of an address field are interpreted to find the operand. It is now time to investigate this subject, called **address modes**. As we shall see, there are many ways to do it.

5.4.2 Immediate Addressing

The simplest way for an instruction to specify an operand is for the address part of the instruction actually to contain the operand itself rather than an address or other information describing where the operand is. Such an operand is called an **immediate operand** because it is automatically fetched from memory at the same time the instruction itself is fetched; hence it is immediately available for use. A possible immediate instruction for loading register R1 with the constant 4 is shown in Fig. 5-16.

Figure 5-16. An immediate instruction for loading 4 into register 1.

Immediate addressing has the virtue of not requiring an extra memory reference to fetch the operand. It has the disadvantage that only a constant can be supplied this way. Also, the number of values is limited by the size of the field. Still, many architectures use this technique for specifying small integer constants.

5.4.3 Direct Addressing

A method for specifying an operand in memory is just to give its full address. This mode is called **direct addressing**. Like immediate addressing, direct addressing is restricted in its use: the instruction will always access exactly the same memory location. So while the value can change, the location cannot. Thus direct addressing can only be used to access global variables whose address is known at compile time. Nevertheless, many programs have global variables, so this mode is widely used. The details of how the computer knows which addresses are immediate and which are direct will be discussed later.

5.4.4 Register Addressing

Register addressing is conceptually the same as direct addressing but specifies a register instead of a memory location. Because registers are so important (due to fast access and short addresses) this addressing mode is the most common one on most computers. Many compilers go to great lengths to determine which variables will be accessed most often (for example, a loop index) and put these variables in registers.

This addressing mode is known simply as **register mode**. In load/store architectures such as the OMAP4420 ARM architecture, nearly all instructions use this addressing mode exclusively. The only time this addressing mode is not used is when an operand is moved from memory into a register (LDR instruction) or from a register to memory (STR instruction). Even for those instructions, one of the operands is a register—where the memory word is to come from or go to.

5.4.5 Register Indirect Addressing

In this mode, the operand being specified comes from memory or goes to memory, but its address is not hardwired into the instruction, as in direct addressing. Instead, the address is contained in a register. An address used in this manner is called a **pointer**. A big advantage of register indirect addressing is that it can reference memory without paying the price of having a full memory address in the instruction. It can also use different memory words on different executions of the instruction.

To see why using a different word on each execution might be useful, imagine a loop that steps through the elements of a 1024-element one-dimensional integer array to compute the sum of the elements in register R1. Outside the loop, some other register, say, R2, can be set to point to the first element of the array, and another register, say, R3, can be set to point to the first address beyond the array. With 1024 integers of 4 bytes each, if the array begins at *A*, the first address beyond the array will be *A* + 4096. Typical assembly code for doing this calculation is shown in Fig. 5-17 for a two-address machine.

```
        MOV R1,#0          ; accumulate the sum in R1, initially 0
        MOV R2,#A          ; R2 = address of the array A
        MOV R3,#A+4096     ; R3 = address of the first word beyond A
LOOP:   ADD R1,(R2)        ; register indirect through R2 to get operand
        ADD R2,#4          ; increment R2 by one word (4 bytes)
        CMP R2,R3          ; are we done yet?
        BLT LOOP           ; if R2 < R3, we are not done, so continue
```

Figure 5-17. A generic assembly program for computing the sum of the elements of an array.

In this little program, we use several addressing modes. The first three instructions use register mode for the first operand (the destination) and immediate mode for the second operand (a constant indicated by the # sign). The second instruction puts the *address* of *A* in R2, not the contents. That is what the # sign tells the assembler. Similarly, the third instruction puts the address of the first word beyond the array in R3.

What is interesting to note is that the body of the loop itself does not contain any memory addresses. It uses register and register indirect mode in the fourth instruction. It uses register and immediate mode in the fifth instruction and register mode twice in the sixth instruction. The BLT might use a memory address, but more likely it specifies the address to branch to with an 8-bit offset relative to the BLT instruction itself. By avoiding the use of memory addresses completely, we have produced a short, fast loop. As an aside, this program is really for the Core i7, except that we have renamed the instructions and registers and changed the

notation to make it easy to read because the Core i7's standard assembly-language syntax (MASM) verges on the bizarre, a remnant of the machine's former life as an 8088.

It is worth noting that, in theory, there is another way to do this computation without using register indirect addressing. The loop could have contained an instruction to add *A* to R1, such as

```
ADD R1,A
```

Then on each iteration of the loop, the instruction itself could be incremented by 4, so that after one iteration it read

```
ADD R1,A+4
```

and so on until it was done.

A program that modifies itself like this is called a **self-modifying** program. The idea was thought of by none other than John von Neumann and made sense on early computers, which did not have register indirect addressing. Nowadays, self-modifying programs are considered horrible style and hard to understand. They also cannot be shared among multiple processes at the same time. Furthermore, they will not even work correctly on machines with a split level 1 cache if the I-cache has no circuitry for doing writebacks (because the designers assumed that programs do not modify themselves). Lastly, self-modifying programs will also fail on machines with separate instruction and data spaces. All in all, this is an idea that has come and (fortunately) gone.

5.4.6 Indexed Addressing

It is frequently useful to be able to reference memory words at a known offset from a register. We saw some examples with IJVM where local variables are referenced by giving their offset from LV. Addressing memory by giving a register (explicit or implicit) plus a constant offset is called **indexed addressing**.

Local variable access in IJVM uses a pointer into memory (LV) in a register plus a small offset in the instruction itself, as shown in Fig. 4-19(a). However, it is also possible to do it the other way: the memory pointer in the instruction and the small offset in the register. To see how that works, consider the following calculation. We have two one-dimensional arrays of 1024 words each, *A* and *B*, and we wish to compute A_i AND B_i for all the pairs and then OR these 1024 Boolean products together to see if there is at least one nonzero pair in the set. One approach would be to put the address of *A* in one register, the address of *B* in a second register, and then step through them together in lockstep, analogous to what we did in Fig. 5-17. This way of doing it would certainly work, but it can be done in a better, more general way, as illustrated in Fig. 5-18.

```
        MOV R1,#0          ; accumulate the OR in R1, initially 0
        MOV R2,#0          ; R2 = index, i, of current product: A[i] AND B[i]
        MOV R3,#4096       ; R3 = first index value not to use
LOOP:   MOV R4,A(R2)       ; R4 = A[i]
        AND R4,B(R2)       ; R4 = A[i] AND B[i]
        OR R1,R4           ; OR all the Boolean products into R1
        ADD R2,#4          ; i = i + 4 (step in units of 1 word = 4 bytes)
        CMP R2,R3          ; are we done yet?
        BLT LOOP           ; if R2 < R3, we are not done, so continue
```

Figure 5-18. A generic assembly program for computing the OR of A_i AND B_i for two 1024-element arrays.

Operation of this program is straightforward. We need four registers here:

1. R1 — Holds the accumulated OR of the Boolean product terms.
2. R2 — The index, *i*, that is used to step through the arrays.
3. R3 — The constant 4096, which is the lowest value of *i* not to use.
4. R4 — A scratch register for holding each product as it is formed.

After initializing the registers, we enter the six-instruction loop. The instruction at *LOOP* fetches A_i into R4. The calculation of the source here uses indexed mode. A register, R2, and a constant, the address of *A*, are added together and used to reference memory. The sum of these two quantities goes to the memory but is not stored in any user-visible register. The notation

```
MOV R4,A(R2)
```

means that the destination uses register mode with R4 as the register and the source uses indexed mode, with *A* as the offset and R2 as the register. If *A* has the value, say, 124300, the actual machine instruction for this is likely to look something like the one shown in Fig. 5-19.

MOV	R4	R2	124300

Figure 5-19. A possible representation of MOV R4,A(R2).

The first time through the loop, R2 is 0 (due to it being initialized that way), so the memory word addressed is A_0, at address 124300. This word is loaded into R4. The next time though the loop, R2 is 4, so the memory word addressed is A_1, at 124304, and so on.

As we promised earlier, here the offset in the instruction itself is the memory pointer and the value in the register is a small integer that is incremented during the