2.4 Operands of the computer hardware

Unlike programs in high-level languages, the operands of arithmetic instructions are restricted; they must be from a limited number of special locations built directly in hardware called registers. Registers are primitives used in hardware design that are also visible to the programmer when the computer is completed, so you can think of registers as the bricks of computer construction. The size of a register in the LEGv8 architecture is 64 bits; groups of 64 bits occur so frequently that they are given the name *doubleword* in the LEGv8 architecture. (Another popular size is a group of 32 bits, called a *word* in the LEGv8 architecture.)

Word: A natural unit of access in a computer, usually a group of 32 bits.

Doubleword: Another natural unit of access in a computer, usually a group of 64 bits; corresponds to the size of a register in the LEGv8 architecture.

One major difference between the variables of a programming language and registers is the limited number of registers, typically 32 on current computers, like LEGv8. Thus, continuing in our top-down, stepwise evolution of the symbolic representation of theLEGv8 language, in this section we have added the restriction that the three operands of LEGv8 arithmetic instructions must each be chosen from one of the 32 64-bit registers.

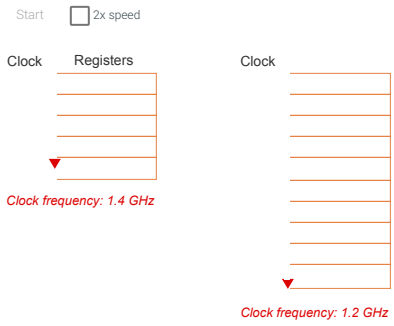
The reason for the limit of 32 registers may be found in the second of our three underlying design principles of hardware technology:

Design Principle 2: Smaller is faster

A very large number of registers may increase the clock cycle time simply because it takes electronic signals longer when they must travel farther.

Guidelines such as "smaller is faster" are not absolutes; 31 registers may not be faster than 32. Even so, the truth behind such observations causes computer designers to take them seriously. In this case, the designer must balance the craving of programs for more registers with the designer's desire to keep the clock cycle fast. Another reason for not using more than 32 is the number of bits it would take in the instruction format.

PARTICIPATION ACTIVITY 2.4.1: Smaller is faster: With fewer registers, the clock frequency can be made faster, because the clock signal requires less time to reach every register.



PARTICIPATION ACTIVITY 2.4.2: Registers.

1) In the instruction below, a, b, and c are operands. In such an arithmetic instruction, each operand comes from special hardware called a ____.

```
ADD a, b, c
```

Check Show answer

2) In the LEGv8 architecture, each register is ____ bits wide.

Check Show answer

3) More registers may benefit an assembly program, but may directly lead to a ____ clock frequency.

Type: slower, faster, broken.

Check Show answer

COD Chapter 4 (The Processor) shows the central role that registers play in hardware construction; as we shall see in that chapter, effective use of registers is critical to program performance.

Although we could simply write instructions using numbers for registers, from 0 to 31, the LEGv8 convention is X followed by the number of the register, except for a few register names that we will cover later.

Example 2.4.1: Compiling a C Assignment Using Registers.

It is the compiler's job to associate program variables with registers. Take, for instance, the assignment statement from our earlier example:

```
f = (g + h) - (i + j);
```

The variables `f`, `g`, `h`, `i`, and `j` are assigned to the registers `X19`, `X20`, `X21`, `X22`, and `X23`, respectively. What is the compiled LEGv8 code?

Answer

The compiled program is very similar to the prior example, except we replace the variables with the register names mentioned above plus two temporary registers, X9 and X10, which correspond to the temporary variables above:

```
ADD X9,  X20, X21   // register X9 contains g + h
ADD X10, X22, X23   // register X10 contains i + j
SUB X19, X9,  X10   // f gets X9 - X10, which is (g + h) - (i + j)
```

PARTICIPATION ACTIVITY 2.4.3: Associations of variables with registers by the compiler.

Consider the above example.

1) With what register did the compiler associate variable g?
   - ○ X20
   - ○ X19

2) Could the compiler have associated g with X26, assuming X26 wasn't used for something else?
   - ○ Yes
   - ○ No

CHALLENGE ACTIVITY 2.4.1: Arithmetic with registers.

Memory operands

Programming languages have simple variables that contain single data elements, as in these examples, but they also have more complex data structures—arrays and structures. These composite data structures can contain many more data elements than there are registers in a computer. How can a computer represent and access such large structures?

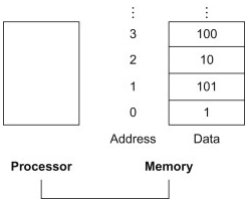
Recall the five components of a computer introduced in COD Chapter 1 (Computer Abstractions and Technology). The processor can keep only a small amount of data in registers, but computer memory contains billions of data elements. Hence, data structures (arrays and structures) are kept in memory.

As explained above, arithmetic operations occur only on registers in LEGv8 instructions; thus, LEGv8 must include instructions that transfer data between memory and registers. Such instructions are called *data transfer instructions*. To access a word or doubleword in memory, the instruction must supply the memory *address*. Memory is just a large, single-dimensional array, with the address acting as the index to that array, starting at 0. For example, in the figure below, the address of the third data element is 2, and the value of memory [2] is 10.

Data transfer instruction: A command that moves data between memory and registers.

Address : A value used to delineate the location of a specific data element within a memory array.

Figure 2.4.1: Memory addresses and contents of memory at those locations (COD Figure 2.2).



If these elements were doublewords, these addresses would be incorrect, since LEGv8 actually uses byte addressing, with each doubleword representing 8 bytes. The animation below (COD Figure 2.3) shows the correct memory addressing for sequential doubleword addresses.

PARTICIPATION ACTIVITY 2.4.4: Memory data and addresses.

Consider the above example.

1) What is the data in the memory word with address 3?
   - ○ 100
   - ○ 10

2) What is the data in the memory word with address 1?
   - ○ 0
   - ○ 101

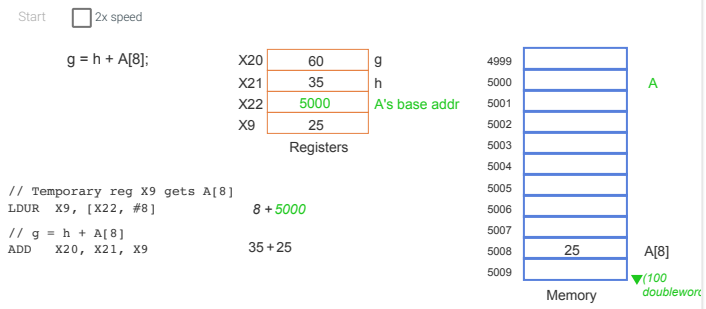
The data transfer instruction that copies data from memory to a register is traditionally called ***load***. The format of the load instruction is the name of the operation followed by the register to be loaded, then register and a constant used to access memory. The sum of the constant portion of the instruction and the contents of the second register forms the memory address. The real LEGv8 name for this instruction is `LDUR`, standing for *load register*.

Elaboration

The `U` in `LDUR` stands for unscaled immediate as opposed to scaled immediate, which we explain in COD Section 2.19 (Fallacies and Pitfalls).

The animation below illustrates the load instruction. Assume that A is an array of 100 doublewords. We'll be making a slight adjustment to the LDUR instruction, but we'll use the below simplified version for now. In an LDUR instruction, a ***base address*** is the starting address of an array in memory (5000 below), a ***base register*** is a register that holds an array's base address (X22 below), and an ***offset*** is a constant value added to a base address to locate a particular array element (8 below).

PARTICIPATION ACTIVITY 2.4.5: Example of compiling an assignment when an operand is in memory.



PARTICIPATION ACTIVITY 2.4.6: Load word instruction.

Assume X22 has 5000, and doublewords addressed 5000..5002 have the data shown:
5000: 99
5001: 77
5002: 323

1) What address will be computed by:
   `LDUR X9, [X22, #2]`

   Check Show answer

2) What value will be put in X9 by:
   `LDUR X9, [X22, #0]`

   Check Show answer

3) What value will be put in X10 by:
   `LDUR X10, [X22, #2]`

   Check Show answer

4) Assume X21 has 5001. What value will

be put in X11 by:
`LDUR X11, [X21, #1]`

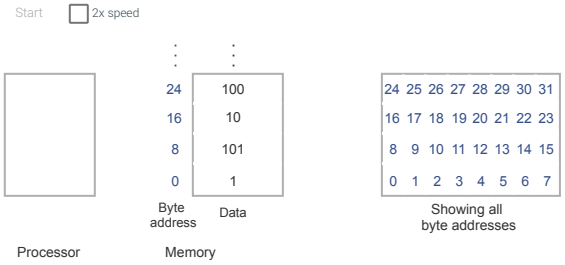
Check Show answer

Hardware/Software Interface

In addition to associating variables with registers, the compiler allocates data structures like arrays and structures to locations in memory. The compiler can then place the proper starting address into the data transfer instructions.

Since 8-bit *bytes* are useful in many programs, virtually all architectures today address individual bytes. Therefore, the address of a doubleword matches the address of one of the 8 bytes within the doubleword, and addresses of sequential doublewords differ by 8. For example, the animation below shows the actual LEGv8 addresses for the doublewords in COD Figure 2.2 (Memory addresses and contents of memory at those locations); the byte address of the third doubleword is 16.

PARTICIPATION ACTIVITY 2.4.7: Actual LEGv8 memory addresses and contents of memory for those doublewords (COD Figure 2.3).



Computers divide into those that use the address of the leftmost or "big end" byte as the doubleword address versus those that use the rightmost or "little end" byte. LEGv8 can work either as *big-endian* or *little-endian*. Since the order matters only if you access the identical data both as a doubleword and as eight bytes, few need to be aware of the "endianess." (COD Appendix A (Assemblers, Linkers, and the SPIM Simulator) shows the two options to number bytes in a word.)

Byte addressing also affects the array index. To get the proper byte address in the code above, *the offset to be added to the base register* `X22` *must be 8 × 8, or 64,* so that the load address will select `A[8]` and not `A[8/8]`.

PARTICIPATION ACTIVITY 2.4.8: Alignment restriction.

1) Each doubleword consists of ____ bytes.
   - ○ 1
   - ○ 4
   - ○ 8

2) Does every byte in memory have a unique address?
   - ○ Yes
   - ○ No

3) An array A has a base address of 2000. A[0] is thus at address 2000. What is the address of A[1]?
   - ○ 2000
   - ○ 2001
   - ○ 2008

4) An array A has a base address of 2000. What is the address of A[9]?
   - ○ 2009
   - ○ 2072
   - ○ 2080

5) Assuming X22 has 5000, what address will be stored in X9 by:
   `LDUR X9, [X22, #7]`?
   - ○ 5008
   - ○ 5007

6) Consider the 64-bit binary number 11100000 00000000 00000000 00000000 00000000 00000000 00000000 00000001, stored in the doubleword with address 5000. For a big-endian architecture, what value is stored in byte 5007?

- ○ 11100000
- ○ 00000000
- ○ 00000001

The instruction complementary to load is traditionally called store; it copies data from a register to memory. The format of a store is similar to that of a load: the name of the operation, followed by the register to be stored, then the base register, and finally the offset to select the array element. Once again, the LEGv8 address is specified in part by a constant and in part by the contents of a register. The actual LEGv8 name is `STUR`, standing for *store register*.

Elaboration

In many architectures, words must start at addresses that are multiples of 4 and doublewords must start at addresses that are multiples of 8. This requirement is called an ***alignment restriction****. (COD Chapter 4 (The Processor) suggests why alignment leads to faster data transfers.) ARMv8 and Intel x86 do not have alignment restrictions, but ARMv7 and MIPS do.*

Alignment restriction: A requirement that data be aligned in memory on natural boundaries.

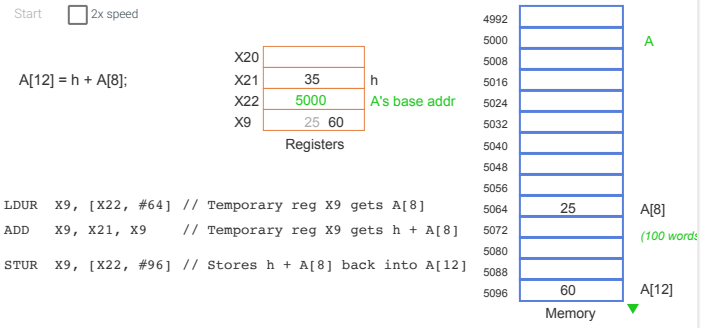
Elaboration

It is not quite true that ARMv8 has no alignment restrictions. While it does support unaligned access to normal memory for most data transfer instructions, stack accesses and instruction fetches do have alignment restrictions.

Hardware/Software Interface

As the addresses in loads and stores are binary numbers, we can see why the DRAM for main memory comes in binary sizes rather than in decimal sizes. That is, in gibibytes (2^{30}) or tebibytes (2^{40}), not in gigabytes (10^9) or terabytes (10^{12}).

PARTICIPATION ACTIVITY 2.4.9: Example of compiling using load and store.



Load register and store register are the instructions that copy doublewords between memory and registers in the ARMv8 architecture. Some brands of computers use other instructions along with load and store to transfer data. An architecture with such alternatives is the Intel x86, described in COD Section 2.18 (Real Stuff: x86 Instructions).

PARTICIPATION ACTIVITY 2.4.10: Store instruction.

1) If X22 has 1000, what address does this instruction compute?

```
STUR X9, [X22, #40]
```

Check Show answer

2) If X22 has 1000, X9 has 77, and memory locations 1000, 1008, and 1016 have 10, 15, 20 respectively, what do those locations have after the following instruction?

```
STUR X9, [X22, #8]
```

Type answer as: 10, 15, 20

Check Show answer

Hardware/Software Interface

The process of putting less frequently used variables (or those needed later) into memory is called ***spilling registers***.

The hardware principle relating size and speed suggests that memory must be slower than registers, since there are fewer registers. This suggestion is indeed the case; data accesses are faster if data are in registers instead of memory.

Moreover, data are more useful when in a register. A LEGv8 arithmetic instruction can read two registers, operate on them, and write the result. A A LEGv8 data transfer instruction only reads one operand or writes one operand, without operating on it.

Thus, registers take less time to access *and* have higher throughput than memory, making data in registers both considerably faster to access and simpler to use. Accessing registers also uses much less energy than accessing memory. To achieve the highest performance and conserve energy, an instruction set architecture must have enough registers, and compilers must use registers efficiently.

Elaboration

Let's put the energy and performance of registers versus memory into perspective. Assuming 64-bit data, registers are roughly 200 times faster (0.25 vs. 50 nanoseconds) and are 10,000 times more energy efficient (0.1 vs 1000 picoJoules) than DRAM in 2015. These large differences led to caches, which reduce the performance and energy penalties of going to memory (see COD Chapter 5 (Large and Fast: Exploiting Memory Hierarchy)).

Constant or immediate operands

Many times a program will use a constant in an operation—for example, incrementing an index to point to the next element of an array. In fact, more than half of the LEGv8 arithmetic instructions have a constant as an operand when running the SPEC CPU2006 benchmarks.

Using only the instructions we have seen so far, we would have to load a constant from memory to use one. (The constants would have been placed in memory when the program was loaded.) For example, to add the constant 4 to register `X22`, we could use the code

```
LDUR   X9, [X20, AddrConstant4]   // X9 = constant 4
ADD    X22, X22, X9               // X22 = X22 + X9 (X9 == 4)
```

assuming that `X20 + AddrConstant4` is the memory address of the constant 4.

An alternative that avoids the load instruction is to offer versions of the arithmetic instructions in which one operand is a constant. This quick add instruction with one constant operand is called *add immediate* or `ADDI`. To add 4 to register `X22`, we just write

```
ADDI X22, X22, #4               // X22 = X22 + 4
```

Constant operands occur frequently, and by including constants inside arithmetic instructions, operations are much faster and use less energy than if constants were loaded from memory.

The constant zero has another role, which is to simplify the instruction set by offering useful variations. For example, the move operation is just an add instruction where one operand is zero. Hence, LEGv8 dedicates a register `XZR` to be hard-wired to the value zero. (It corresponds to register number 31.) Using frequency to justify the inclusions of constants is another example of the great idea of making the **common case** fast.

PARTICIPATION ACTIVITY 2.4.11: Immediate operands.

Click on the error. Assume X19 has 5, and X20 has 20.

1) ADD X9, X19, X20
Result: X9 gets 20

2) ADD X9 X19 #44

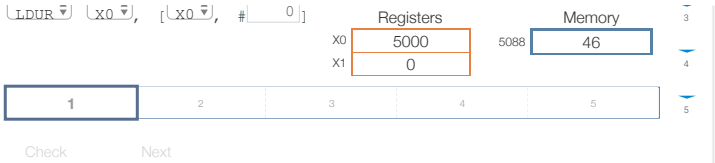
3) ADD X9, X19 #23
Result: X9 gets 28

4) ADDI X9, X20 #99
Result: X9 gets 104

CHALLENGE ACTIVITY 2.4.2: Loading and storing from memory.

Start

Compute: X1 = Memory[5088]



PARTICIPATION ACTIVITY 2.4.12: Check yourself: Registers trend.

1) Given the importance of registers, what is the rate of increase in the number of registers in a chip over time?
   - ○ Very fast, like Moore's Law
   - ○ Very slow

Elaboration

Although the LEGv8 registers in this book are 64 bits wide, the full ARMv8 instruction set has two execution states: AArch32, in which registers are 32 bits wide (see COD Section 2.19(Real Stuff: The Rest of the ARMv8 Instruction Set)), and AArch64, which has a 64-bit wide register. The former supports the A32 and T32 instruction sets and the later supports A64. In this chapter, we use a subset of A64 for LEGv8.

Elaboration

The LEGv8 offset plus base register addressing is an excellent match to structures as well as arrays, since the register can point to the beginning of the structure and the offset can select the desired element. We'll see such an example in COD Section 2.13 (A C Sort Example to Put It All Together).

Elaboration

The register in the data transfer instructions was originally invented to hold an index of an array with the offset used for the starting address of an array. Thus, the base register is also called the index register. Today's memories are much larger and the software model of data allocation is more sophisticated, so the base address of the array is normally passed in a register since it won't fit in the offset, as we shall see.

Table 2.4.1: Popular data type options.

Operating System	pointers	int	long int	long long int
Microsoft Windows	64 bits	32 bits	32 bits	64 bits
Linux, Most Unix	64 bits	32 bits	64 bits	64 bits

Elaboration

The migration from 32-bit address computers to 64-bit address computers left compiler writers a choice of the size of data types in C. Clearly, pointers should be 64 bits, but what about integers? Moreover, C has the data types `int`, `long int`, *and* `long long int`. *The problems come from converting from one data type to another and having an unexpected overflow in C code that is not fully standard compliant, which unfortunately is not rare code. The table above shows the two popular options.*

While each compiler could have different choices, generally the compilers associated with each operating system make the same decision. To keep the examples simple, in this book we'll assume pointers are all 64 bits and declare C integers as `long long int` *to keep them the same size. We also follow C99 standard and declare variables used as indexes to arrays to be* `size_t`, *which guarantees they are the right size no matter how big the array. They typically declared the same as* `long int.`

Elaboration

In the full ARMv8 instruction set, register 31 is `XZR` *in most instructions but the stack point (*`SP`*) in others. We think it is confusing, so register 31 is always* `XZR` *in LEGv8 and* `SP` *is always register 28. Besides confusing the reader, it would also complicate datapath design in COD Chapter 4 (The Processor) if register 31 meant 0 for some instructions and* `SP` *for others.*

Elaboration

The full ARMv8 instruction set does not use the mnemonic `ADDI` *when one of the operands is an immediate; it just uses* `ADD`, *and lets the assembler pick the proper opcode. We worry that it*

might be confusing to use the same mnemonic for both opcodes, so for teaching purposes LEGv8 distinguishes the two cases with different mnemonics.

Provide feedback on this section