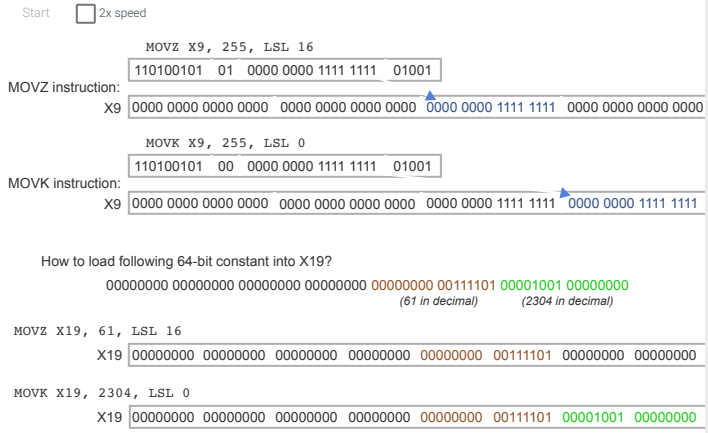
2.9 LEGv8 addressing for wide immediates and addresses

Although keeping all LEGv8 instructions 32 bits long simplifies the hardware, there are times where it would be convenient to have 32-bit or larger constants or addresses. This section starts with the general solution for large constants, and then shows the optimizations for instruction addresses used in branches.

Wide immediate operands

Although constants are frequently short and fit into the 12-bit fields, sometimes they are bigger. The LEGv8 instruction set includes the instruction *move wide with zeros* (`MOVZ`) and *move wide with keep* (`MOVK`) specifically to set any 16 bits of a constant in a register. The former instruction zeros the rest of the bits of the register and the latter leaves the remaining bits unchanged. The 16-bit field to be loaded is specified by adding `LSL` and then the number 0, 16, 32, or 48 depending on which quadrant of the 64-bit word is desired. These instructions allow, for example, a 32-bit constant to be created from two 32-bit instructions. The figure below shows the operation of `MOVZ` and `MOVK`.

PARTICIPATION ACTIVITY 2.9.1: The effect of the MOVX and MOVK instructions. (COD Figure 2.18).



Hardware/Software Interface

Either the compiler or the assembler must break large constants into pieces and then reassemble them into a register. As you might expect, the immediate field's size restriction may be a problem for memory addresses in loads and stores as well as for constants in immediate instructions.

Hence, the symbolic representation of the LEGv8 machine language is no longer limited by the hardware, but by whatever the creator of an assembler chooses to include. We stick close to the hardware to explain the architecture of the computer, noting when we use the enhanced language of the assembler that is not found in the processor.

PARTICIPATION ACTIVITY 2.9.2: 64-bit immediate operands.

1) Given: `MOVZ X19, 7, LSL 16`. How is the immediate operand 7 represented in the instruction?
   - ○ 00000000 00000111
   - ○ 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000111

2) What is X19 after:
   `MOVZ X19, 7, LSL 16`
   - ○ 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000111
   - ○ 00000000 00000000 00000000 00000000 00000000 00000000 00000000 01110000
   - ○ 00000000 00000000 00000000 00000000 00000000 00000000 00000111 00000000 00000000

3) What is X19 after:
   `MOVZ X19, 7, LSL 16`
   `MOVK X19, 8, LSL 0`
   - ○ 00000000 00000000 00000000

00001000 00000000 00000111 00000000 00000000

- 00000000 00000000 00000000 00000000 00000000 00000000 00001000
- 00000000 00000000 00000000 00000000 00000000 00000111 00000000 00001000

Addressing in branches

The LEGv8 branch instructions have the simplest addressing. They use the LEGv8 instruction format, called the *B-type*, which consists of 6 bits for the operation field and the rest of the bits for the address field. Thus,

```
B 10000   // go to location 10000ten
```

could be assembled into this format (it's actually a bit more complicated, as we will see):

where the value of the branch opcode is 5 and the branch address is 10000_{ten}.

Unlike the branch instruction, a conditional branch instruction can specify one operand in addition to the branch address. Thus,

```
CBNZ X19, Exit   // go to Exit if X19 ≠ 0
```

is assembled into this instruction, leaving only 19 bits for the branch address:

This format is called *CB-type*, for conditional branch. (The conditional branch instructions that rely on condition codes also use the CB-type format, but they use the final field to select among the many possible branch conditions.)

If addresses of the program had to fit in this 19-bit field, it would mean that no program could be bigger than 2^{19}, which is far too small to be a realistic option today. An alternative would be to specify a register that would always be added to the branch offset, so that a branch instruction would calculate the following:

Program counter = Register + Branch offset

This sum allows the program to be as large as 2^{64} and still be able to use conditional branches, solving the branch address size problem. Then the question is, which register?

The answer comes from seeing how conditional branches are used. Conditional branches are found in loops and in *if* statements, so they tend to branch to a nearby instruction. For example, about half of all conditional branches in SPEC benchmarks go to locations less than 16 instructions away. Since the *program counter* (PC) contains the address of the current instruction, we can branch within $\pm 2^{18}$ words of the current instruction if we use the PC as the register to be added to the address. Almost all loops and *if* statements are much smaller than 2^{18} words, so the PC is the ideal choice. This form of branch addressing is called *PC-relative addressing*.

PC-relative addressing: An addressing regime in which the address is the sum of the *program counter* (PC) and a constant in the instruction.

Like most recent computers, LEGv8 uses PC-relative addressing for all conditional branches, because the destination of these instructions is likely to be close to the branch. On the other hand, branch-and-link instructions invoke procedures that have no reason to be near the call, so they normally use other forms of addressing. Hence, the LEGv8 architecture offers long addresses for procedure calls by using the B-type format for both branch and branch-and-link instructions.

Since all LEGv8 instructions are 4 bytes long, LEGv8 stretches the distance of the branch by having PC-relative addressing refer to the number of *words* to the next instruction instead of the number of bytes. Thus, the 19-bit field can branch four times as far by interpreting the field as a relative word address rather than as a relative byte address: ±1 MB from the current PC. Similarly, the 26-bit field in branch instructions is also a word address, meaning that it represents a 28-bit byte address.

The unconditional branch is also PC-relative, which means it can branch ±128 MB from the current PC.

Example 2.9.1: Showing branch offset in machine language.

The *while* loop was compiled into this LEGv8 assembler code:

```
Loop: LSL  X10, X22, #3   // Temp reg X10 = 8 * i
      ADD  X10, X10, X25  // X10 = address of save[i]
      LDUR X9, [X10,#0]   // Temp reg X9 = save[i]
      SUB  X11, X9, X24   // X11 = save[i] - k
      CBNZ X11, Exit      // go to Exit if save[i] ≠ k (X11 ≠ 0)
      ADDI X22, X22,#1    // i = i + 1
      B    Loop           // go to Loop
Exit:
```

If we assume we place the loop starting at location 80000 in memory, what is the LEGv8 machine code for this loop?

Answer

The assembled instructions and their addresses are:

80000	1691	0	3	22	10
80004	1112	25	0	10	10
80008	1986	0	0	10	9
80012	1624	24	0	9	11
80016	181	3			11
80020	580	1		22	22
80024	5	−6			
80028	...				

Remember that LEGv8 instructions have byte addresses, so addresses of sequential words differ by 4, the number of bytes in a word, and that branches multiply their address fields by 4, the size of

LEGv8 instructions in bytes. The `CBNZ` instruction on the fifth line adds 3 words or 12 bytes to the address of the instruction, specifying the branch destination relative to the branch instruction (12 + 80016) and not using the full destination address (80028). The branch instruction on the last line does a similar calculation for a backwards branch (-24 + 80024), corresponding to the label `Loop`.

Hardware/Software Interface

Most conditional branches are to a nearby location, but occasionally they branch far away, farther than can be represented in the 19 bits of the conditional branch instruction. The assembler comes to the rescue just as it did with large addresses or constants: it inserts an unconditional branch to the branch target, and inverts the condition so that the conditional branch decides whether to skip the unconditional branch.

Example 2.9.2: Branching far away.

Given a branch on register `X19` being equal to register zero,

```
CBZ X19, L1
```

replace it by a pair of instructions that offers a much greater branching distance.

Answer

These instructions replace the short-address conditional branch:

```
      CBNZ X19, L2
      B   L1
L2:
```

LEGv8 addressing mode summary

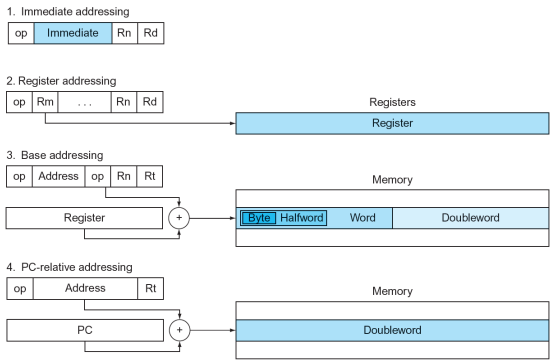
Multiple forms of addressing are generically called *addressing modes*. The figure below shows how operands are identified for each addressing mode. The addressing modes of the LEGv8 instructions are the following:

1. ***Immediate addressing***: The operand is a constant within the instruction itself
2. ***Register addressing***: The operand is a register
3. ***Base addressing / displacement addressing***: The operand is at the memory location whose address is the sum of a register and a constant in the instruction
4. ***PC-relative addressing***: The branch address is the sum of the PC and a constant in the instruction

Addressing mode: One of several addressing regimes delimited by their varied use of operands and/or addresses.

Figure 2.9.1: Illustration of the four LEGv8 addressing modes (COD Figure 2.19).

The operands are shaded in color. The operand of mode 3 is in memory, whereas the operand for mode 2 is a register. Note that versions of load and store access bytes, halfwords, words, or doublewords. For mode 1, the operand is part of the instruction itself. Mode 4 addresses instructions in memory, with mode 4 adding a long address shifted left 2 bits to the PC. Note that a single operation can use more than one addressing mode. Add, for example, uses both immediate (`ADDI`) and register (`ADD`) addressing.



PARTICIPATION ACTIVITY 2.9.3: Addressing modes.

PC-relative addressing | Register addressing | Base or displacement addressing | Immediate addressing

The operand is a constant within the instruction itself.

The operand is a register.

The operand is at the memory location whose address is the sum of a register and a constant in the instruction.

The branch address is the sum of the PC and a constant in the instruction.

Reset

Decoding machine language

Sometimes you are forced to reverse-engineer machine language to create the original assembly language. One example is when looking at "core dump". The figure below shows the LEGv8 encoding of the opcodes for the LEGv8 machine language. This figure helps when translating by hand between assembly language and machine language.

Figure 2.9.2: LEGv8 instruction encoding (COD Figure 2.20).

The varying size opcode values can be mapped into the space they occupy in the widest opcodes. By looking at the first 11 bits of the instruction and looking up the value, you can see which instruction it refers to.

Instruction	Opcode	Opcode Size	11-bit opcode range		Instruction Format
			Start	End	
B	000101	6	160	191	B - format
STURB	00111000000	11	448		D - format
LDURB	00111000010	11	450		D - format
B.cond	01010100	8	672	679	CB - format
ORRI	1011001000	10	712	713	I - format
EORI	1101001000	10	840	841	I - format
STURH	01111000000	11	960		D - format
LDURH	01111000010	11	962		D - format
AND	10001010000	11	1104		R - format
ADD	10001011000	11	1112		R - format
ADDI	1001000100	10	1160	1161	I - format
ANDI	1001001000	10	1168	1169	I - format
BL	100101	6	1184	1215	B - format
ORR	10101010000	11	1360		R - format
ADDS	10101011000	11	1368		R - format
ADDIS	1011000100	10	1416	1417	I - format
CBZ	10110100	8	1440	1447	CB - format
CBNZ	10110101	8	1448	1455	CB - format
STURW	10111000000	11	1472		D - format
LDURSW	10111000100	11	1476		D - format
STXR	11001000000	11	1600		D - format
LDXR	11001000010	11	1602		D - format
EOR	11001010000	11	1616		R - format
SUB	11001011000	11	1624		R - format
SUBI	1101000100	10	1672	1673	I - format
MOVZ	110100101	9	1684	1687	IM - format
LSR	11010011010	11	1690		R - format
LSL	11010011011	11	1691		R - format
BR	11010110000	11	1712		R - format
ANDS	11101010000	11	1872		R - format
SUBS	11101011000	11	1880		R - format
SUBIS	1111000100	10	1928	1929	I - format
ANDIS	1111001000	10	1936	1937	I - format
MOVK	111100101	9	1940	1943	IM - format
STUR	11111000000	11	1984		D - format
LDUR	11111000010	11	1986		D - format

Example 2.9.3: Decoding machine code.

What is the assembly language statement corresponding to this machine instruction?

$8b0f00130_{hex}$

Answer

The first step in converting hexadecimal to binary:

```
1000 1011 0000 1111 0000 0000 0001 0011
```

To know how to interpret the bits, we need to determine the instruction format, and to do that we first need to find the opcode field. The problem is that the opcode varies from 6 bits to 11 bits depending on the format. Since opcodes must be unique, one way to identify them is to see how many 11-bit opcodes do the shorter opcodes correspond.

For example, the branch instruction `B` use a 6-bit opcode with the value:

```
00 0101
```

Measured in 11-bit opcodes, it occupies all the opcode values from

```
00 0101 00000
```

to

```
00 0101 11111
```

That is, if any 11-bit opcode had a value in that range, such as

```
00 0101 00100
```

it would conflict with the 6-bit opcode of branch.

The figure above lists the instructions in LEGv8 in numerical order by opcode, showing the range of the 11-bit opcode space they occupy. For example, branch goes from 160 to 191, which is the decimal version of the bit patterns above. To determine the opcode, you just take the first 11 bits of the instruction, convert it to decimal representation, and then look up the table to find the instruction and its format.

In this example, the tentative opcode field is then 10001011000_{two}, which is 1112_{ten}. When we search the figure above, we see that opcode corresponds to the `ADD` instruction, which uses the R-format. Thus, we can parse the binary format into fields listed in the figure below:

```
op          Rm      shamt     Rn        Rd
10001011000 00101   000000    01111     10000
```

We decode the rest of the instruction by looking at the field values. The decimal values are 5 for the Rm field, 15 for Rn, and 16 for Rd (shamt is unused). These numbers represent registers `X5`, `X15`, and `X16`. Now we can reveal the assembly instruction:

```
ADD X16, X15, X5
```

Figure 2.9.3: LEGv8 instruction formats (COD Figure 2.21).

Name	Fields							Comments
Field size		6 to 11 bits	5 to 10 bits	5 or 4 bits	2 bits	5 bits	5 bits	All LEGv8 instructions are 32 bits long
R-format	R	opcode	Rm	shamt		Rn	Rd	Arithmetic instruction format
I-format	I	opcode	immediate			Rn	Rd	Immediate format
D-format	D	opcode	address		op2	Rn	Rt	Data transfer format
B-format	B	opcode	address					Unconditional Branch format
CB-format	CB	opcode	address				Rt	Conditional Branch format
IM-format	IM	opcode	immediate				Rd	Wide Immediate format

The figure above shows all the LEGv8 instruction formats. COD Figure 2.1 (LEGv8 assembly language revealed in this chapter) shows the LEGv8 assembly language revealed in this chapter. The next chapter covers LEGv8 instructions for multiply, divide, and arithmetic for real numbers.

PARTICIPATION ACTIVITY 2.9.4: Check yourself: 64-bit immediates and addresses.

1) What is the range of addresses for conditional branches in LEGv8 (K = 1024)?
   - ○ Addresses between 0 and 512K - 1
   - ○ Addresses up to about 256K before the branch to about 256K after
   - ○ Addresses up to about 1024K before the branch to about 1024K after

2) What is the range of addresses for branch and branch and link in LEGv8 (M = 1024K)?
   - ○ Addresses between 0 and 64M - 1
   - ○ Addresses up to about 128M before the branch to about 128M after

Elaboration

*An easy-to-understand way for hardware to figure out the format of the instruction in parallel is to think of the hardware as having a read-only memory whose address size matches the largest opcode and whose content tells the hardware what to do for the specific instruction. Thus, instructions like add (**ADD**) that have an 11-bit opcode have a single entry in the memory, but instructions like branch (**B**) with a 6-bit opcode have many redundant copies. In fact, B has $2^{11} / 2^{6} = 2^{5}$ or 32 entries. A more efficient hardware structure than a read-only memory that will accomplish the same task is a programmable-logic array (PLA), which essentially modifies the address decoder so that there is a single entry for every opcode, no matter its size (see COD Appendix B (Graphics and Computing GPUs)).*

Provide feedback on this section