2.3 Operations of the computer hardware

> “ There must certainly be instructions for performing the fundamental arithmetic operations.
> *Burks, Goldstine, and von Neumann, 1947*

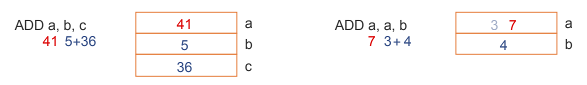
Every computer must be able to perform arithmetic. The LEGv8 assembly language notation

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

instructs a computer to add the two variables `b` and `c` and to put their sum in `a`.

PARTICIPATION ACTIVITY 2.3.1: A simple add instruction.

Start 2x speed



PARTICIPATION ACTIVITY 2.3.2: A simple add instruction.

1) Type a LEGv8 add instruction that computes:
z = x + y

Check Show answer

2) Type a LEGv8 add instruction that computes:
a = a + b

Check Show answer

This notation is rigid in that each LEGv8 arithmetic instruction performs only one operation and must always have exactly three variables. For example, suppose we want to place the sum of four variables `b`, `c`, `d`, and `e` into variable `a`. (In this section we are being deliberately vague about what a "variable" is; in COD Section 2.3 (Operands of the computer hardware) we'll explain in detail.)

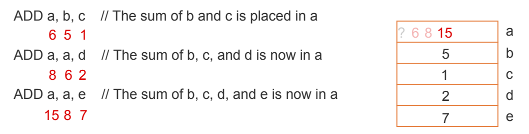
The following sequence of instructions adds the four variables:

```
ADD a, b, c   // The sum of b and c is placed in a
ADD a, a, d   // The sum of b, c, and d is now in a
ADD a, a, e   // The sum of b, c, d, and e is now in a
```

Thus, it takes three instructions to sum the four variables.

PARTICIPATION ACTIVITY 2.3.3: Three add instructions executing one at a time.

Start 2x speed



ADD a, b, c // The sum of b and c is placed in a
6 5 1
ADD a, a, d // The sum of b, c, and d is now in a
8 6 2
ADD a, a, e // The sum of b, c, d, and e is now in a
15 8 7
? 6 8 15
5
1
2
7
a
b
c
d
e

Above, the words to the right of the double slashes (//) on each line are ***comments*** for the human reader, so the computer ignores them. Note that unlike other programming languages, each line of this language can contain at most one instruction. Another difference from C is that comments always terminate at the end of a line.

Similarly to the add instruction, `SUB a, b, c` computes b - c and puts the result in a. A table at the end of this section lists more LEGv8 instructions.

PARTICIPATION ACTIVITY 2.3.4: Basic instruction execution.

Given: b = 2, c = 5, d = 1.

1)
```
ADD a, b, c
```

Final value of a is ____.

Check Show answer

2)
```
ADD t, d, c
ADD a, t, c
```

Final value of a is ____.

Check Show answer

3)
```
SUB t, c, b
ADD a, t, d
```

Final value of a is ____.

Check Show answer

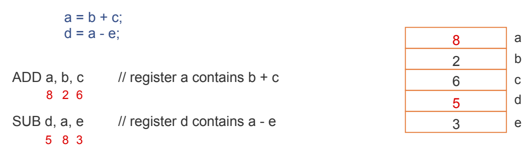
The natural number of operands for an operation like addition is three: the two numbers being added together and a place to put the sum. Requiring every instruction to have exactly three operands, no more and no less, conforms to the philosophy of keeping the hardware simple: hardware for a variable number of operands is more complicated than hardware for a fixed number. This situation illustrates the first of three underlying principles of hardware design:

Design Principle 1: Simplicity favors regularity.

We can now show, in the two examples that follow, the relationship of programs written in higher-level programming languages to programs in this more primitive notation.

PARTICIPATION ACTIVITY 2.3.5: Example of compiling two C assignment statements into LEGv8.

Start ☐ 2x speed



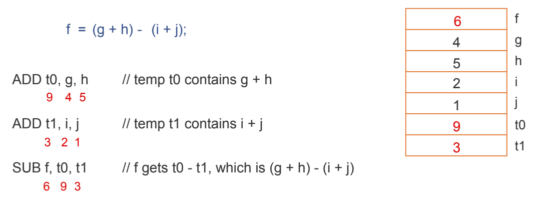
```
a = b + c;
d = a - e;

ADD a, b, c      // register a contains b + c
    8  2  6
SUB d, a, e      // register d contains a - e
    5  8  3
```

8	a
2	b
6	c
5	d
3	e

PARTICIPATION ACTIVITY 2.3.6: Example of compiling a complex C assignment into LEGv8.

Start ☐ 2x speed



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

ADD t0, g, h     // temp t0 contains g + h
    9   4  5
ADD t1, i, j     // temp t1 contains i + j
    3   2  1
SUB f, t0, t1    // f gets t0 - t1, which is (g + h) - (i + j)
    6  9   3
```

6	f
4	g
5	h
2	i
1	j
9	t0
3	t1

PARTICIPATION ACTIVITY 2.3.7: Compiling an expression.

Order the assembly instructions to calculate the expression: a = b + c + d - e

```
ADD t0, b, c    SUB a, t1, e    ADD t1, t0, d
```

1

2

3

Reset

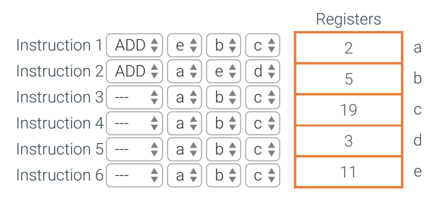
PARTICIPATION ACTIVITY 2.3.8: Compiling an expression.

Which instruction is *incorrect* for calculating a = b + c + d - e?

1)
```
ADD  t0, c, d
ADD  t0, t0, b
SUB  a, e, t0
```

PARTICIPATION ACTIVITY 2.3.9: Instruction set simulator: ADD and SUB.

Start



Instruction 1	ADD	e	b	c
Instruction 2	ADD	a	e	d
Instruction 3	---	a	b	c
Instruction 4	---	a	b	c
Instruction 5	---	a	b	c
Instruction 6	---	a	b	c

Registers	
2	a
5	b
19	c
3	d
11	e

Execute first instruction

PARTICIPATION ACTIVITY 2.3.10: Check yourself: High-level language and lines of code.

For a given function, which programming language likely takes the most lines of source code?

Java **C** **LEGv8 assembly language**

1 (Requires most lines)

2

3 (Requires fewest lines)

Reset

Elaboration

To increase portability, Java was originally envisioned as relying on a software interpreter. The instruction set of this interpreter is called Java bytecodes (see COD Section 2.15 (Advanced Material: Compiling C and Interpreting Java)), which is quite different from the LEGv8 instruction set. To get performance close to the equivalent C program, Java systems today typically compile Java bytecodes into the native instruction sets like LEGv8. Because this compilation is normally done much later than for C programs, such Java compilers are often called Just In Time (JIT) compilers. COD Section 2.12 (Translating and Starting a Program) shows how JITs are used later than C compilers in the start-up process, and COD Section 2.13 (A C Sort Example to Put It All Together) shows the performance consequences of compiling versus interpreting Java programs.

Table 2.3.1: LEGv8 operands revealed in this chapter (COD Figure 2.1).

Name	Example	Comments
32 registers	`X0...X30, XZR`	Fast locations for data. In LEGv8, data must be in registers to perform arithmetic, and register `XZR` always equals 0.
2^{62} memory doublewords	Memory[0], Memory [4], ..., Memory[4,611,686,018,427,387,904]	Accessed only by data transfer instructions. LEGv8 uses byte addresses, so sequential doubleword addresses differ by 8. Memory holds data structures, arrays, and spilled registers.

PARTICIPATION ACTIVITY 2.3.11: LEGv8 registers.

Indicate whether each name refers to a LEGv8 register.

1) X0

○ Yes

○ No

2) X32
- Yes
- No

3) X
- Yes
- No

4) XZR
- Yes
- No

5) XONE
- Yes
- No

6) Memory[0]
- Yes
- No

Table 2.3.2: LEGv8 assembly language revealed in this chapter (COD Figure 2.1).

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	`ADD X1, X2, X3`	`X1 = X2 + X3`	Three register operands
	subtract	`SUB X1, X2, X3`	`X1 = X2 – X3`	Three register operands
	add immediate	`ADDI X1, X2, 20`	`X1 = X2 + 20`	Used to add constants
	subtract immediate	`SUBI X1, X2, 20`	`X1 = X2 – 20`	Used to subtract constants
	add and set flags	`ADDS X1, X2, X3`	`X1 = X2 + X3`	Add, set condition codes
	subtract and set flags	`SUBS X1, X2, X3`	`X1 = X2 – X3`	Subtract, set condition codes
	add immediate and set flags	`ADDIS X1, X2, 20`	`X1 = X2 + 20`	Add constant, set condition codes
	subtract immediate and set flags	`SUBIS X1, X2, 20`	`X1 = X2 – 20`	Subtract constant, set condition codes
Data transfer	load register	`LDUR X1, [X2, 40]`	`X1` = Memory[`X2` + 40]	Doubleword from memory to register
	store register	`STUR X1, [X2, 40]`	Memory[`X2` + 40] = `X1`	Doubleword from register to memory
	load signed word	`LDURSW X1, [X2, 40]`	`X1` = Memory[`X2` + 40]	Word from memory to register
	store word	`STURW X1, [X2, 40]`	Memory[`X2` + 40] = `X1`	Word from register to memory
	load half	`LDURH X1, [X2, 40]`	`X1` = Memory[`X2` + 40]	Halfword memory to register
	store half	`STURH X1, [X2, 40]`	Memory[`X2` + 40] = `X1`	Halfword register to memory
	load byte	`LDURB X1, [X2, 40]`	`X1` = Memory[`X2`	Byte from

			+ 40]	memory to register
	store byte	`STURB X1, [X2, 40]`	Memory[`X2` + 40] = `X1`	Byte from register to memory
	load exclusive register	`LDXR X1, [X2, 0]`	`X1` = Memory[`X2`]	Load; 1st half of atomic swap
	store exclusive register	`STXR X1, X3, [X2]`	Memory[`X2`] = `X1`; `X3` = 0 or 1	Store; 2nd half of atomic swap
	move wide with zero	`MOVZ X1, 20`	`X1` = 20 or 20*2^{16} or 20*2^{32} or 20*2^{48}	Loads 16-bit constant, rest zeros
	move wide with keep	`MOVK X1, 20`	`X1` = 20 or 20*2^{16} or 20*2^{32} or 20*2^{48}	Loads 16-bit constant, rest unchanged
Logical	and	`AND X1, X2, X3`	`X1 = X2 & X3`	Three reg. operands; bit-by-bit AND
	inclusive or	`ORR X1, X2, X3`	`X1 = X2 | X3`	Three reg. operands; bit-by-bit OR
	exclusive or	`EOR X1, X2, X3`	`X1 = X2 ^ X3`	Three reg. operands; bit-by-bit XOR
	and immediate	`ANDI X1, X2, 20`	`X1 = X2 & 20`	Bit-by-bit AND reg with constant
	inclusive or immediate	`ORRI X1, X2, 20`	`X1 = X2 | 20`	Bit-by-bit OR reg with constant
	exclusive or immediate	`EORI X1, X2, 20`	`X1 = X2 ^ 20`	Bit-by-bit XOR reg with constant
	logical shift left	`LSL X1, X2, 10`	`X1 = X2 << 10`	Shift left by constant
	logical shift right	`LSR X1, X2, 10`	`X1 = X2 >> 10`	Shift right by constant
Conditional branch	compare and branch on equal 0	`CBZ X1, 25`	if (X1 == 0) go to PC + 4 + 100	Equal 0 test; PC-relative branch
	compare and branch on not equal 0	`CBNZ X1, 25`	if (X1 != 0) go to PC + 4 + 100	Not equal 0 test; PC-relative
	branch conditionally	`B.cond 25`	if (condition true) go to PC + 4 + 100	Test condition codes; if true, branch
Unconditional jump	branch	`B 2500`	go to PC + 4 + 10000	Branch to target address; PC-relative
	branch to register	`BR X30`	go to `X30`	For switch, procedure return
	branch with link	`BL 2500`	`X30` = PC + 4; PC + 4 + 10000	For procedure call PC-relative

PARTICIPATION ACTIVITY 2.3.12: LEGv8 instructions.

Indicate whether each is a valid LEGv8 instruction.

1) `ADD X1, X2, X3`

- Valid
- Not valid

2) `ADDI X1, X2, 50`

- Valid
- Not valid

3) `LDUR X1, [X35, 20]`

- Valid
- Not valid

4) `BRANCH 2500`

- Valid
- Not valid

Provide feedback on this section