2.19 Concluding remarks

> “ Less is more.
> *Robert Browning, Andrea del Sarto, 1855*

The two principles of the *stored-program* computer are the use of instructions that are indistinguishable from numbers and the use of alterable memory for programs. These principles allow a single machine to aid cancer researchers, financial advisers, and novelists in their specialties. The selection of a set of instructions that the machine can understand demands a delicate balance among the number of instructions needed to execute a program, the number of clock cycles needed by an instruction, and the speed of the clock. As illustrated in this chapter, three design principles guide the authors of instruction sets in making that tricky tradeoff:

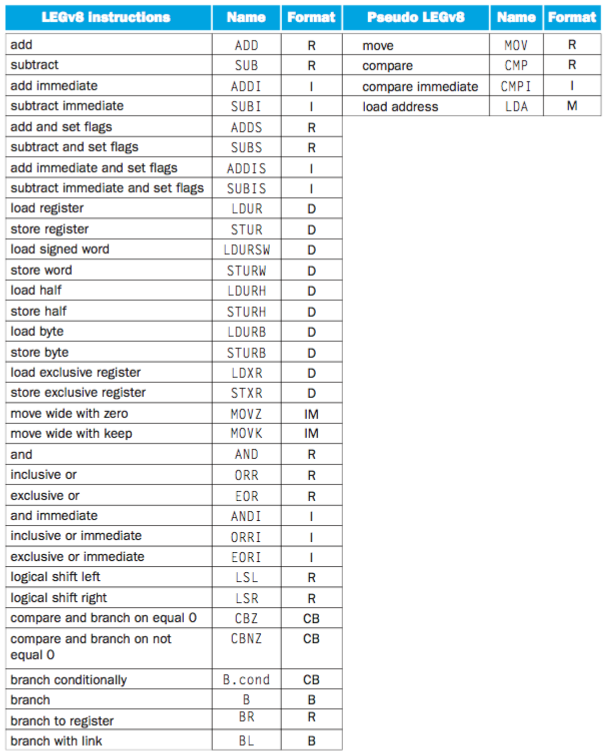
1. *Simplicity favors regularity.* Regularity motivates many features of the ARMv8 instruction set: keeping all instructions a single size, always requiring three register operands in arithmetic instructions, and keeping the register fields in the same place in most instruction formats.
2. *Smaller is faster.* The desire for speed is the reason that ARMv8 has 32 registers rather than many more.
3. *Good design demands good compromises.* One ARMv8 example was the compromise between providing for larger addresses and constants in instructions and keeping all instructions the same length.

We also saw the great idea from COD Chapter 1 (Computer Abstractions and Technology) of making the **common case fast** applied to instruction sets as well as computer architecture. Examples of making the common ARMv8 case fast include PC-relative addressing for conditional branches and immediate addressing for larger constant operands.

Above this machine level is assembly language, a language that humans can read. The assembler translates it into the binary numbers that machines can understand, and it even "extends" the instruction set by creating symbolic instructions that aren't in the hardware. For instance, constants or addresses that are too big are broken into properly sized pieces, common variations of instructions are given their own name, and so on. The figure below lists the LEGv8 instructions we have covered so far, both real and pseudoinstructions. Hiding details from the higher level is another example of the great idea of **abstraction**.

Figure 2.19.1: The LEGv8 instruction set covered so far, with the real LEGv8 instructions on the left and the pseudoinstructions on the right (COD Figure 2.45).

COD Section 2.19 (Real stuff: The rest of the ARMv8 instruction set) describes the full ARMv8 architecture. COD Figure 2.1 (LEGv8 assembly language revealed in this chapter) shows more details of the LEGv8 architecture revealed in this chapter.



LEGv8 instructions	Name	Format
add	ADD	R
subtract	SUB	R
add immediate	ADDI	I
subtract immediate	SUBI	I
add and set flags	ADDS	R
subtract and set flags	SUBS	R
add immediate and set flags	ADDIS	I
subtract immediate and set flags	SUBIS	I
load register	LDUR	D
store register	STUR	D
load signed word	LDURSW	D
store word	STURW	D
load half	LDURH	D
store half	STURH	D
load byte	LDURB	D
store byte	STURB	D
load exclusive register	LDXR	D
store exclusive register	STXR	D
move wide with zero	MOVZ	IM
move wide with keep	MOVK	IM
and	AND	R
inclusive or	ORR	R
exclusive or	EOR	R
and immediate	ANDI	I
inclusive or immediate	ORRI	I
exclusive or immediate	EORI	I
logical shift left	LSL	R
logical shift right	LSR	R
compare and branch on equal 0	CBZ	CB
compare and branch on not equal 0	CBNZ	CB
branch conditionally	B.cond	CB
branch	B	B
branch to register	BR	R
branch with link	BL	B

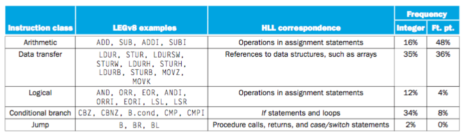
Pseudo LEGv8	Name	Format
move	MOV	R
compare	CMP	R
compare immediate	CMPI	I
load address	LDA	M

Each category of ARMv8 instructions is associated with constructs that appear in programming languages:

- Arithmetic instructions correspond to the operations found in assignment statements.
- Transfer instructions are most likely to occur when dealing with data structures like arrays or structures.
- Conditional branches are used in *if* statements and in loops.
- Unconditional branches are used in procedure calls and returns and for *case/switch* statements.

These instructions are not born equal; the popularity of the few dominates the many. For example, the figure below shows the popularity of each class of instructions for SPEC CPU2006. The varying popularity of instructions plays an important role in the chapters about datapath, control, and pipelining.

Figure 2.19.2: LEGv8 instruction classes, examples, correspondence to high-level program language constructs, and percentage of LEGv8 instructions executed by category for the average integer and floating point SPEC CPU2006 benchmarks (COD Figure 2.46).



Instruction class	LEGv8 examples	HLL correspondence	Frequency	
			Integer	Ft. pt.
Arithmetic	`ADD, SUB, ADDI, SUBI`	Operations in assignment statements	16%	48%
Data transfer	`LDUR, STUR, LDURSW, STURW, LDURH, STURH, LDURB, STURB, MOVZ, MOVK`	References to data structures, such as arrays	35%	36%
Logical	`AND, ORR, EOR, ANDI, ORRI, EORI, LSL, LSR`	Operations in assignment statements	12%	4%
Conditional branch	`CBZ, CBNZ, B.cond, CMP, CMPI`	*If* statements and loops	34%	8%
Jump	`B, BR, BL`	Procedure calls, returns, and *case/switch* statements	2%	0%

After we explain computer arithmetic in COD Chapter 3 (Arithmetic for Computers), we reveal more of the ARMv8 instruction set architecture.

PARTICIPATION ACTIVITY 2.19.1: ARMv8 Instructions.

1) Keeping all instructions the same size is a compromise the ARMv8 instruction set makes to maintain simplicity and regularity.
- True
- False

2) Adding more registers to ARMv8 would likely improve performance.
- True
- False

3) PC-relative addressing likely has no influence on the performance of the ARMv8 architecture.
- True
- False

4) Assembly language adds a level of abstraction to a computer by allowing programmers to write code that is translated into binary numbers that the computer can read.
- True
- False

5) ARMv8 instructions and pseudoinstructions correspond to operations and constructs used in higher-level programming languages.
- True
- False

Provide feedback on this section