Instruction Set

- How to tell a computer what to do?

- Bits are the letters of the computer
- Instructions are the words of the computer
  - An instruction is a collection of bits
- The instruction set is the vocabulary of the computer
  - An instruction set is a collection of instructions

Chapter 2

Instructions: Language of the Computer

Instruction Set

- The repertoire(全部本领;可表演项目;) of instructions of a computer
- Different computers have different instruction sets
  - But with many aspects in common
- Early computers had very simple instruction sets
  - Simplified implementation
- Many modern computers also have simple instruction sets

Agenda

- **Instruction Set**
- Operations and Operands
- Representing Instructions in the Computer
- Logical Operations
- Instructions for Making Decisions
- Supporting Procedures in Computer Hardware
- LEGv8 Addressing for Wide Immediates and Addresses
- A C Sort Example to Put It All Together
- Arrays versus Pointers

Agenda

- Instruction Set
- **Operations and Operands**
- Representing Instructions in the Computer
- Logical Operations
- Instructions for Making Decisions
- Supporting Procedures in Computer Hardware
- LEGv8 Addressing for Wide Immediates and Addresses
- A C Sort Example to Put It All Together
- Arrays versus Pointers

The ARMv8 Instruction Set

- A subset, called LEGv8, used as the example throughout the book
- Commercialized by ARM Holdings (www.arm.com)
- Large share of embedded core market
  - Applications in consumer electronics, network/storage equipment, cameras, printers, …
- Typical of many modern ISAs
  - See ARM Reference Data tear-out card

Arithmetic Operations

- Add and subtract, three operands
  - Two sources and one destination

```
ADD a, b, c  // a gets b + c
```

- All arithmetic operations have this form
- *Design Principle 1:* Simplicity favors regularity
  - Regularity makes implementation simpler
  - Simplicity enables higher performance at lower cost

Design Principles applied to the ARMv8

- Design principles
  1. Simplicity favors regularity
  2. Smaller is faster
  3. Make the common case fast
  4. Good design demands good compromises
- We will regularly see these principles when exploring the ISA

Register Operands

- Arithmetic instructions use register operands
- LEGv8 has a 32 × 64-bit register file
  - Use for frequently accessed data
  - 64-bit data is called a “doubleword”
    - 31 x 64-bit general purpose registers X0 to X30
  - 32-bit data called a “word”
    - 31 x 32-bit general purpose sub-registers W0 to W30
- *Design Principle 2:* Smaller is faster
  - Compare to main memory: millions of locations

Arithmetic Example

- C code:

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

- Compiled LEGv8 code:

```
ADD t0, g, h   // temp t0 = g + h
ADD t1, i, j   // temp t1 = i + j
SUB f, t0, t1  // f = t0 - t1
```

Processor does not work with variables but with registers

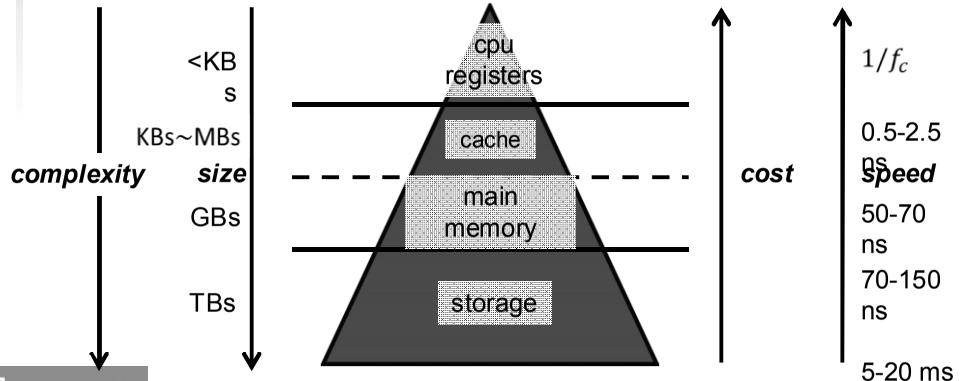
LEGv8 Registers

- X0 – X7: procedure arguments/results
- X8: indirect result location register
- X9 – X15: temporaries
- X16 – X17 (IP0 – IP1): may be used by linker as a scratch register, other times as temporary register
- X18: platform register for platform independent code; otherwise a temporary register
- X19 – X27: saved
- X28 (SP): stack pointer
- X29 (FP): frame pointer
- X30 (LR): link register (return address)
- XZR (register 31): the constant value 0

X0 – X8, X16 – X18, X28 – X30 will be explained later

Memory Hierarchy

- The ALU can only operate on values that are inside the register
- Not directly from the main memory!
- Need to LOAD stuff(原料) from main memory
- Memory Hierarchy
  - Will be discussed later



Memory Operand Example

- C code:

```
A[12] = h + A[8];
```

  - h in X21, base address of A in X22
- Compiled LEGv8 code:
  - Index 8 requires offset of ?

```
LDUR    X9,[X22,#64] // U for "unscaled"
ADD     X9,X21,X9
STUR    X9,[X22,#96]
```


Register Operand Example

- C code:

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

  - f, …, j in X19, X20, …, X23
- Compiled LEGv8 code:

```
ADD X9, X20, X21
ADD X10, X22, X23
SUB X19, X9, X10
```


Registers vs. Memory

- Registers are faster to access than memory
- Operating on memory data requires loads and stores
  - More instructions to be executed
- Compiler must use registers for variables as much as possible
  - Only spill(溢出) to memory for less frequently used variables
  - Register optimization is important!

Memory Operands

- Main memory used for composite(复合的) data
  - Arrays, structures, dynamic data
- To apply arithmetic operations
  - Load values from memory into registers
  - Store result from register to memory
- Memory is byte addressed
  - Each address identifies an 8-bit byte
- LEGv8 does not require words to be aligned in memory, except for instructions and the stack

2s-Complement Signed Integers

- Given an n-bit number

$$x = -x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \cdots + x_1 2^1 + x_0 2^0$$

- Range: -2^{n-1} to $+2^{n-1} - 1$
- Example
  - $1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1100_2$
    $= -1 \times 2^{31} + 1 \times 2^{30} + \ldots + 1 \times 2^2 + 0 \times 2^1 + 0 \times 2^0$
    $= -2{,}147{,}483{,}648 + 2{,}147{,}483{,}644 = -4_{10}$
- Using 32 bits
  - –2,147,483,648 to +2,147,483,647

Immediate Operands

- Constant data specified(明确规定) in an instruction

```
ADDI X22, X22, #4
```

- *Design Principle 3:* Make the common case fast
  - Small constants are common
  - Immediate operand avoids a load instruction

2s-Complement Signed Integers

- Bit 31 is sign bit
  - 1 for negative numbers
  - 0 for non-negative numbers
- $-(-2^{n-1})$ can't be represented
- Non-negative numbers have the same unsigned and 2s-complement representation
- Some specific numbers
  - 0: 0000 0000 … 0000
  - –1: 1111 1111 … 1111
  - Most-negative: 1000 0000 … 0000
  - Most-positive: 0111 1111 … 1111

Unsigned Binary Integers

- Given an n-bit number

$$x = x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \cdots + x_1 2^1 + x_0 2^0$$

- Range from 0 to $+2^n - 1$
- Example
  - $0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1011_2$
    $= 0 + \ldots + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0$
    $= 0 + \ldots + 8 + 0 + 2 + 1 = 11_{10}$
- Using 32 bits
  - 0 to +4,294,967,295 $(2^{32} - 1)$

Agenda

- Instruction Set
- Operations and Operands
- **Representing Instructions in the Computer**
- Logical Operations
- Instructions for Making Decisions
- Supporting Procedures in Computer Hardware
- LEGv8 Addressing for Wide Immediates and Addresses
- A C Sort Example to Put It All Together
- Arrays versus Pointers

Signed Negation

- Complement(取反) and add 1
  - Complement means 1 → 0, 0 → 1

$$x + \bar{x} = 1111...111_2 = -1$$

$$\bar{x} + 1 = -x$$

- Example: negate(取反数)+2
  - +2 = 0000 0000 … 0010_2
  - –2 = 1111 1111 … 1101_2 + 1
    = 1111 1111 … 1110_2

Representing Instructions

- Instructions are encoded in binary
  - Called machine code
- LEGv8 instructions
  - Encoded as 32-bit instruction words
  - Small number of formats encoding operation code (opcode), register numbers, …
  - Regularity(规整，端正)!

Sign Extension

- Representing a number using more bits
  - Preserve(保持，维持) the numeric value
- Replicate the sign bit to the left
  - c.f. unsigned values: extend with 0s
- Examples: 8-bit to 16-bit
  - +2: 0000 0010 => 0000 0000 0000 0010
  - –2: 1111 1110 => 1111 1111 1111 1110
- In LEGv8 instruction set
  - LDURSB: sign-extend loaded byte
  - LDURB: zero-extend loaded byte

R-format Example

opcode	Rm	shamt	Rn	Rd
11 bits	5 bits	6 bits	5 bits	5 bits

```
ADD X9,X20,X21
```

		0_{ten}		
10001011000_{two}	10101_{two}	000000_{two}	10100_{two}	01001_{two}

$1000\ 1011\ 0001\ 0101\ 0000\ 0010\ 1000\ 1001_{two}$ =

$8B150289_{16}$

Hexadecimal

- Base 16
  - Compact representation of bit strings
  - 4 bits per hex digit

0	0000	4	0100	8	1000	c	1100
1	0001	5	0101	9	1001	d	1101
2	0010	6	0110	a	1010	e	1110
3	0011	7	0111	b	1011	f	1111

- Example: ECA8 6420
  - 1110 1100 1010 1000 0110 0100 0010 0000

LEGv8 D-format Instructions

opcode	address	op2	Rn	Rt
11 bits	9 bits	2 bits	5 bits	5 bits

- Load/store instructions
  - Rn: base register
  - address: constant offset from contents of base register (+/- 32 doublewords)
  - Rt: destination (load) or source (store) register number
- *Design Principle 4:* Good design demands good compromises
  - Different formats complicate decoding, but allow 32-bit instructions uniformly(一致地)
  - Keep formats as similar as possible

LEGv8 R-format Instructions

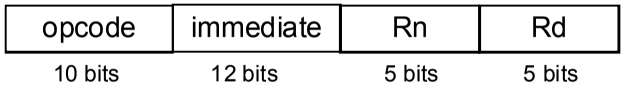
opcode	Rm	shamt	Rn	Rd
11 bits	5 bits	6 bits	5 bits	5 bits

- Instruction fields
  - opcode: operation code
  - Rm: the second register source operand
  - shamt: shift amount (000000 for now)
  - Rn: the first register source operand
  - Rd: the register destination

Agenda

- Instruction Set
- Operations and Operands
- Representing Instructions in the Computer
- **Logical Operations**
- Instructions for Making Decisions
- Supporting Procedures in Computer Hardware
- LEGv8 Addressing for Wide Immediates and Addresses
- A C Sort Example to Put It All Together
- Arrays versus Pointers

LEGv8 I-format Instructions



- Immediate instructions
  - Rn: source register
  - Rd: destination register
- Immediate field is zero-extended

Logical Operations

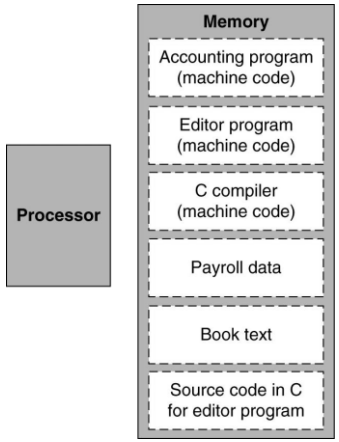
- Instructions for bitwise(按位) manipulation

Operation	C	Java	LEGv8
Shift left	<<	<<	LSL
Shift right	>>	>>>	LSR
Bit-by-bit AND	&	&	AND, ANDI
Bit-by-bit OR	|	|	ORR, ORRI
Bit-by-bit NOT	^	^	EOR, EORI

- Useful for extracting and inserting groups of bits in a word

Stored Program Computers

The BIG Picture



- Instructions represented in binary, just like data
- Instructions and data stored in memory
- Programs can operate on programs
  - e.g., compilers, linkers, …
- Binary compatibility allows compiled programs to work on different computers
  - Standardized ISAs

OR Operations

- Useful to include bits in a word
  - Set some bits to 1, leave others unchanged

```
ORR X9,X10,X11
```

X10 00000000 00000000 00000000 00000000 00000000 00000000 00001101 11000000

X11 00000000 00000000 00000000 00000000 00000000 00000000 00111100 00000000

X9 00000000 00000000 00000000 00000000 00000000 00000000 00111101 11000000

Shift Operations

opcode	Rm	shamt	Rn	Rd
11 bits	5 bits	6 bits	5 bits	5 bits

- shamt: how many positions to shift
- Shift left logical
  - Shift left and fill with 0 bits
  - `LSL` by *i* bits multiplies by 2^i
- Shift right logical
  - Shift right and fill with 0 bits
  - `LSR` by *i* bits divides by 2^i (unsigned only)

EOR Operations

- Differencing operation
  - Invert (flip) some bits, leave others unchanged

```
EOR X9,X10,X12  // NOT operation
```

X10 00000000 00000000 00000000 00000000 00000000 00000000 00001101 11000000

X12 11111111 11111111 11111111 11111111 11111111 11111111 11111111 11111111

X9 11111111 11111111 11111111 11111111 11111111 11111111 11110010 00111111

AND Operations

- Useful to mask bits in a word
  - Set some bits to 0, leave others unchanged

```
AND X9,X10,X11
```

X10 00000000 00000000 00000000 00000000 00000000 00000000 00001101 11000000

X11 00000000 00000000 00000000 00000000 00000000 00000000 00111100 00000000

X9 00000000 00000000 00000000 00000000 00000000 00000000 00001100 00000000

Compiling Two C Assignment Statements into LEGv8

This segment of a C program contains the five variables a, b, c, d, and e. Since Java evolved from C, this example and the next few work for either high-level programming language:

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

ANSWER:

ADD a, b, c
SUB d, a, e

LEGv8 Assembly language-1

LEGv8 assembly language

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 + 40]	Byte from 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, LSL 0	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, LSL 0	X1 = 20 or 20 * 2^{16} or 20 * 2^{32} or 20 * 2^{48}	Loads 16-bit constant, rest unchanged

FIGURE 2.1 LEGv8 assembly language revealed in this chapter. This information is also found in Column 1 of the LEGv8 Reference Data Card at the front of this book.

Translating a LEGv8 Assembly Instruction into a Machine

Let's do the next step in the refinement of the LEGv8 language as an example. We'll show the real LEGv8 language version of the instruction represented symbolically as

```
ADD X9,X20,X21
```

first as a combination of decimal numbers and then of binary numbers.

The decimal representation is

1112	21	0	20	9

This instruction can also be represented as fields of binary numbers instead of decimal:

10001011000	10101	000000	10100	01001
11 bits	5 bits	6 bits	5 bits	5 bits

LEGv8 Assembly language-2

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 + 100	Equal 0 test; PC-relative branch
	compare and branch on not equal 0	CBNZ X1, 25	if (X1 != 0) go to PC + 100	Not equal 0 test; PC-relative branch
	branch conditionally	B.cond 25	if (condition true) go to PC + 100	Test condition codes; if true, branch
Unconditional branch	branch	B 2500	go to PC + 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 + 10000	For procedure call PC-relative

FIGURE 2.1 (Continued).

Conditional Operations

- Branch to a labeled(示踪的) instruction if a condition is true
  - Otherwise, continue sequentially(循序地)
- `CBZ register, L1`
  - if (register == 0) branch to instruction labeled L1;

CBZ = Compare and Branch if Zero

- `CBNZ register, L1`
  - if (register != 0) branch to instruction labeled L1;
- `B L1`
  - branch unconditionally to instruction labeled L1;

20220309 Homework-2

1. For the following C statement, write the corresponding LEGv8 assembly code. Assume that the C variables f, g, and h, have already been placed in registers X0, X1, and X2 respectively. Use a minimal number of LEGv8 assembly instructions.

   f = g + (h − 5);

2. Translate the following LEGv8 code to C. Assume that the variables f, g, h, i, and j are assigned to registers X0,X1,X2,X3, and X4, respectively. Assume that the base address of the arrays A and B are in registers X6 and X7, respectively.

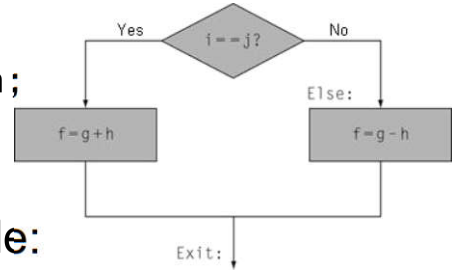
```
ADDI X9, X6, #8
ADD X10, X6, XZR
STUR X10, [X9, #0]
LDUR X9, [X9, #0]
ADD X0, X9, X10
```


Compiling If Statements

- C code:

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

  - f, …, j in X19, …, X23



- Compiled LEGv8 code:

```
        SUB   X9, X22, X23
        CBNZ  X9, Else
        ADD   X19, X20, X21
        B     Exit
Else:   SUB   X19, X20, X21
Exit: …
```

Assembler calculates offset

Agenda

- Instruction Set
- Operations and Operands
- Representing Instructions in the Computer
- Logical Operations
- **Instructions for Making Decisions**
- Supporting Procedures in Computer Hardware
- LEGv8 Addressing for Wide Immediates and Addresses
- A C Sort Example to Put It All Together
- Arrays versus Pointers

More Conditional Operations

- Condition codes, set from arithmetic instruction with S-suffix (ADDS, ADDIS, ANDS, ANDIS, SUBS, SUBIS)
  - negative (N): result had 1 in MSB
  - zero (Z): result was 0
  - overflow (V): result sign overflowed
  - carry (C): result had carryout from MSB

S = Set flags

- Use **subtract and set flags**, then conditionally branch:
  - **B.EQ** (equal)
  - **B.NE** (not equal)
  - **B.LT** (less than, < signed), **B.LO** (lower, < unsigned)
  - **B.LE** (less than or equal, ≤ signed), **B.LS** (lower or same, ≤ unsigned)
  - **B.GT** (greater than, > signed), **B.HI** (higher, > unsigned)
  - **B.GE** (greater than or equal, ≥ signed), **B.HS** (higher or same, ≥ unsigned)

Compiling Loop Statements

- C code:

```
while (save[i] == k) i += 1;
```

  - i in X22, k in X24, address of save in X25
- Compiled LEGv8 code:

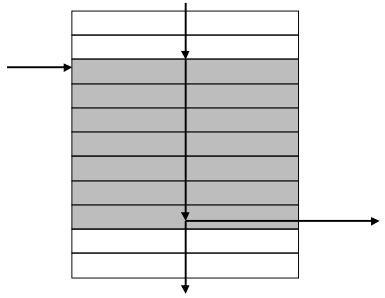
```
Loop: LSL    X9, X22, #3
      ADD    X10, X25, X9
      LDUR   X11, [X10, #0]
      SUB    X12, X11, X24
      CBNZ   X12, Exit
      ADDI   X22, X22, #1
      B      Loop
Exit: …
```


Signed vs. Unsigned

- Signed comparison ≠ Unsigned comparison
- Example
  - W22 = 1111 1111 1111 1111 1111 1111 1111 1111
  - W23 = 0000 0000 0000 0000 0000 0000 0000 0001
  - `W22 < W23 # signed`
    - –1 < +1
  - `W22 > W23 # unsigned`
    - +4,294,967,295 > +1

Basic Blocks

- A basic block is a sequence of instructions with
  - No embedded branches (except at end)
  - No branch targets (except at beginning)



- A compiler identifies basic blocks for optimization
- An advanced processor can accelerate execution of basic blocks

Pseudo instructions

- A pseudo(伪) instruction in assembler language is translated into an other instruction.
- Meant to make your live a little easier.
- See book page 129.

CMP	Xa, Xb	➔	SUBS XZR, Xa, Xb
CMPI	Xa,#c	➔	SUBIS XZR, Xa, #c
MOV	Xa, Xb	➔	ORR Xa, XZR, Xb

Conditional Example

- if (a > b) a += 1;
  - a in X22, b in X23
  - a and b are signed numbers

```
    SUBS    XZR, X22, X23
    B.LE    Exit
    ADDI    X22, X22, #1
Exit:
```

Note: condition in B is complement of condition in if

Agenda

- Instruction Set
- Operations and Operands
- Representing Instructions in the Computer
- Logical Operations
- Instructions for Making Decisions
- **Supporting Procedures in Computer Hardware**
- LEGv8 Addressing for Wide Immediates and Addresses
- A C Sort Example to Put It All Together
- Arrays versus Pointers

Conditional Example

- if (a > b) a += 1;
  - a in X22, b in X23
  - a and b are unsigned numbers

```
    SUBS    XZR, X22, X23
    B.LS    Exit
    ADDI    X22, X22, #1
Exit:
```

Register Usage

- X9 to X17: temporary registers
  - Not preserved by the callee(被召唤者)

  Callee = function which is called
- X19 to X28: saved registers
  - If used, the callee saves and restores them

ARM Call Convention(协定)

Should we follow this convention?

Procedure Calling

- Steps **required**
  1. Place parameters in registers X0 to X7
  2. **Transfer control to procedure**
  3. Acquire(获得) storage for procedure
  4. **Perform procedure's operations**
  5. Place result in register for caller
  6. **Return to place of call (address in X30)**

When the book writes "procedure" you may read "function"

Leaf Procedure Example

- C code:

```
long long int leaf_example(
    long long int g, long long int h,
    long long int i, long long int j)
{
    long long int f;
    f = (g + h) - (i + j);
    return f;
}
```

  - Arguments g, …, j in X0, …, X3
  - Return value in X0

Procedure Call Instructions

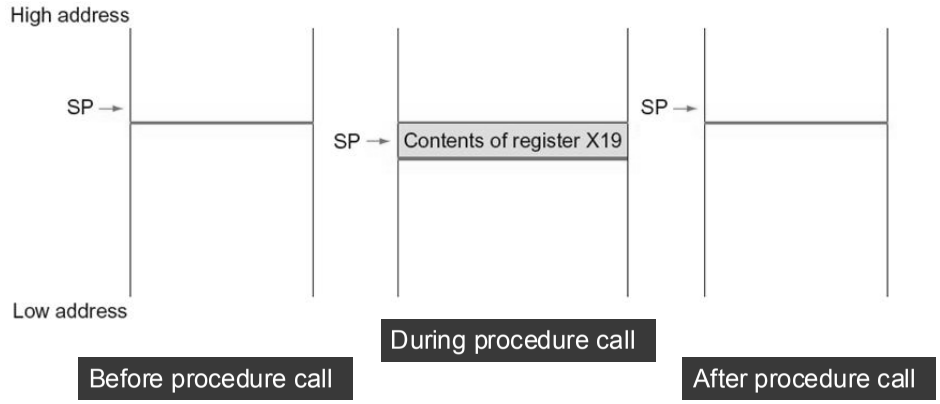
- Procedure call: branch with link

  `BL ProcedureLabel`
  - Address of following instruction put in X30 = LR (Link Register)
  - Jumps to target address
- Procedure return: branch to register

  `BR LR`
  - Copies LR to program counter
  - Can also be used for computed jumps
    - e.g., for case/switch statements

Can also be used for jump to function pointers

Local Data on the Stack



Leaf Procedure Example

- LEGv8 code:

```
leaf_example:
        ADD     X9, X0, X1
        ADD     X10, X2, X3
        SUB     X0, X9, X10
        BR      LR
```


Leaf Procedure Example

- LEGv8 code:

```
leaf_example:
        SUBI    SP, SP, #8
        STUR    X19, [SP, #0]
        ADD     X9, X0, X1
        ADD     X10, X2, X3
        SUB     X19, X9, X10
        ADD     X0, X19, XZR
        LDUR    X19, [SP, #0]
        ADDI    SP, SP, #8
        BR      LR
```


Leaf Procedure Example

- C code:

```
long long int leaf_example(
    long long int g, long long int h,
    long long int i, long long int j)
{
    long long int f;
    f = (g + h) - (i + j);
    return f;
}
```

  - Arguments g, …, j in X0, …, X3
  - Return value in X0
  - f in X19 (hence, need to save X19 on stack)

Strange requirement! Let's just do it as an exercise

Leaf Procedure Example

- LEGv8 code:

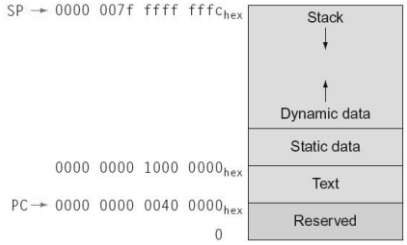
```
fact:
        SUBIS   XZR, X0, #1
        B.GT    else
        BR      LR
else:
        SUBI    SP, SP, #16
        STUR    LR, [SP, #8]
        STUR    X0, [SP, #0]
        SUBI    X0, X0, #1
        BL      fact
        LDUR    X9, [SP, #0]
        LDUR    LR, [SP, #8]
        ADDI    SP, SP, #16
        MUL     X0, X9, X0
        BR      LR
```


Non-Leaf Procedures

- Procedures that call other procedures
- For nested(嵌套) call, caller needs to save on the stack:
  - Its return address
  - Any arguments and temporaries needed after the call
- Restore from the stack after the call

Memory Layout

- Text: program code
- Static data: global variables
  - e.g., static variables in C, constant arrays and strings
- Dynamic data: heap(堆)
  - E.g., malloc in C, new in Java
- Stack(栈): local (automatic) variables (spilled(溢出) registers)



Non-Leaf Procedure Example

- C code:

```
long long int fact(long long int n)
{
  if (n <= 1) return n;
  else return n * fact(n - 1);
}
```

Strange implementation of factorial(阶乘)! Let's just do it as an exercise

- Argument n in X0
- Result in X0

Use MUL instruction for multiplication (see Chapter 3)

Assume result fits in long long int

Branch Addressing

- B-type
  - `B Label // go to location Label:`

5	Instruction offset to Label:
6 bits	26 bits

- CB-type
  - `CBNZ X19, Exit // go to Exit if X19 != 0`

181	Instruction offset to Exit:	19
8 bits	19 bits	5 bits

- Both addresses are PC-relative
  - Address = PC + offset (from instruction) x 4

Agenda

- Instruction Set
- Operations and Operands
- Representing Instructions in the Computer
- Logical Operations
- Instructions for Making Decisions
- Supporting Procedures in Computer Hardware
- **LEGv8 Addressing for Wide Immediates and Addresses**
- A C Sort Example to Put It All Together
- Arrays versus Pointers

LEGv8 Addressing Summary

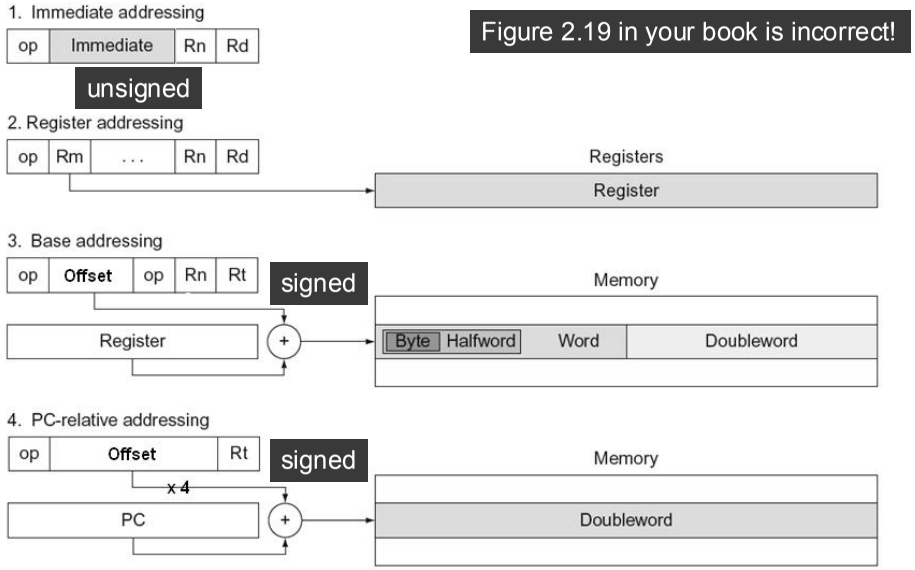


Figure 2.19 in your book is incorrect!

64-bit Constants

- Most constants are small
  - 12-bit immediate is sufficient
- For the occasional(偶然的) 64-bit constant

  `MOVZ:` move 16-bit wide with zeros

  `MOVK:` move 16-bit wide with keep
- Use with second operand (LSL 0, 16, 32 or 48)

`MOVZ X9,255,LSL 16`

0000 0000 0000 0000	0000 0000 0000 0000	0000 0000 1111 1111	0000 0000 0000 0000

`MOVK X9,255,LSL 0`

0000 0000 0000 0000	0000 0000 0000 0000	0000 0000 1111 1111	0000 0000 1111 1111

C Sort Example

- Illustrates use of assembly instructions for a C bubble sort function
- Swap procedure (leaf)

```
void swap(long long int v[],
     size_t k)
{
     long long int temp;
     temp = v[k];
     v[k] = v[k+1];
     v[k+1] = temp;
}
```

  - v in X0, k in X1, temp in X9

Why do we use X9?

LEGv8 Encoding Summary

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
IW-format	IW	opcode	immediate				Rd	Wide Immediate format

Address field actually contains an offset

The Procedure Swap

```
swap: LSL   X10, X1, #3       // X10 = k * 8
      ADD   X10, X0, X10      // X10 = address of v[k]
      LDUR  X9, [X10, #0]     // X9 = v[k]
      LDUR  X11, [X10, #8]    // X11 = v[k+1]
      STUR  X11, [X10, #0]    // v[k] = X11 (v[k+1])
      STUR  X9, [X10, #8]     // v[k+1] = X9 (v[k])
      BR    LR                // return to calling
                              // routine
```


Agenda

- Instruction Set
- Operations and Operands
- Representing Instructions in the Computer
- Logical Operations
- Instructions for Making Decisions
- Supporting Procedures in Computer Hardware
- LEGv8 Addressing for Wide Immediates and Addresses
- **A C Sort Example to Put It All Together***
- Arrays versus Pointers

The Inner Loop

- Skeleton of inner loop:
  - for (j = i – 1; j >= 0 && v[j] > v[j + 1]; j – = 1) {

```
        SUBI X20, X19, #1      // j = i - 1
for2tst: CMP X20,XZR           // compare X20 to 0 (j to 0)
        B.LT exit2             // go to exit2 if X20 < 0 (j < 0)
        LSL X10, X20, #3       // reg X10 = j * 8
        ADD X11, X0, X10       // reg X11 = v + (j * 8)
        LDUR X12, [X11,#0]     // reg X12 = v[j]
        LDUR X13, [X11,#8]     // reg X13 = v[j + 1]
        CMP X12, X13           // compare X12 to X13
        B.LE exit2             // go to exit2 if X12 ≤ X13
        MOV X0, X21            // first swap parameter is v
        MOV X1, X20            // second swap parameter is j
        BL swap                // call swap
        SUBI X20, X20, #1      // j -= 1
        B for2tst              // branch to test of inner loop
    exit2:
```


The Sort Procedure in C

- Non-leaf (calls swap)

```
void sort (long long int v[], size_t n)
{
    size_t i, j;
    for (i = 0; i < n; i += 1) {
        for (j = i – 1;
             j >= 0 && v[j] > v[j + 1];
             j -= 1)
        {
            swap(v, j);
        }
    }
}
```

  - v in X0, n in X1, i in X19, j in X20

Why do we use X19 and X20?

Preserving Registers

- Preserve saved registers:

```
SUBI SP,SP,#40        // make room on stack for 5 regs
STUR LR,[SP,#32]      // save LR on stack
STUR X22,[SP,#24]     // save X22 on stack
STUR X21,[SP,#16]     // save X21 on stack
STUR X20,[SP,#8]      // save X20 on stack
STUR X19,[SP,#0]      // save X19 on stack
MOV X21, X0           // copy parameter X0 into X21
MOV X22, X1           // copy parameter X1 into X22
```

- Restore saved registers:

```
exit1: LDUR X19, [SP,#0]  // restore X19 from stack
   LDUR X20, [SP,#8]      // restore X20 from stack
   LDUR X21,[SP,#16]      // restore X21 from stack
   LDUR X22,[SP,#24]      // restore X22 from stack
   LDUR X30,[SP,#32]      // restore LR from stack
   ADDI SP,SP,#40                 // restore stack pointer
```

The code in your book is incorrect!

The Outer Loop

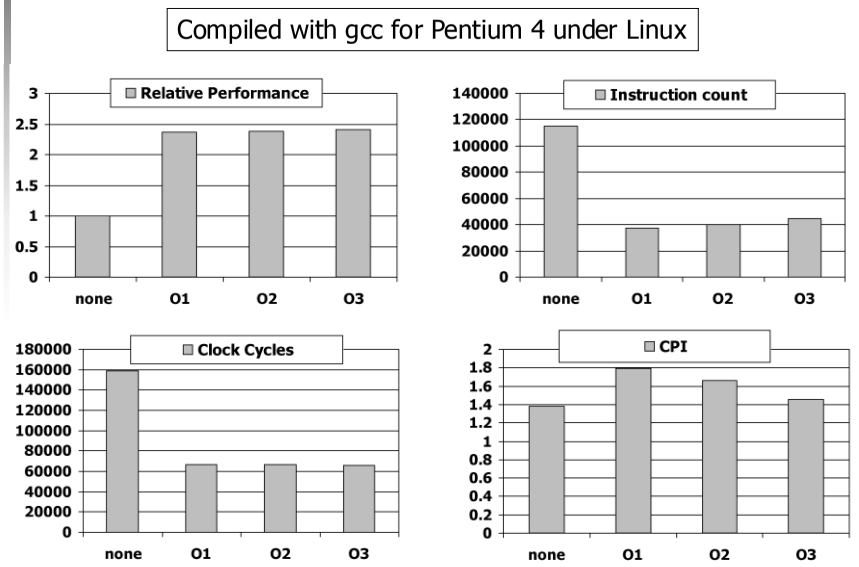
- Skeleton of outer loop:
  - for (i = 0; i <n; i += 1) {

```
  MOV X19,XZR              // i = 0
for1tst:
  CMP X19, X1              // compare X19 to X1 (i to n)
  B.GE exit1         // go to exit1 if X19 ≥ X1 (i≥n)

  (body of outer for-loop)

  ADDI X19,X19,#1          // i += 1
  B for1tst                // branch to test of outer loop
exit1:
```

Lessons Learnt

- Instruction count and CPI are not good performance indicators in isolation(孤独)
- Compiler optimizations are sensitive to the algorithm
- Java/JIT compiled code is significantly faster than JVM interpreted
  - Comparable to optimized C in some cases
- Nothing can fix a dumb algorithm!

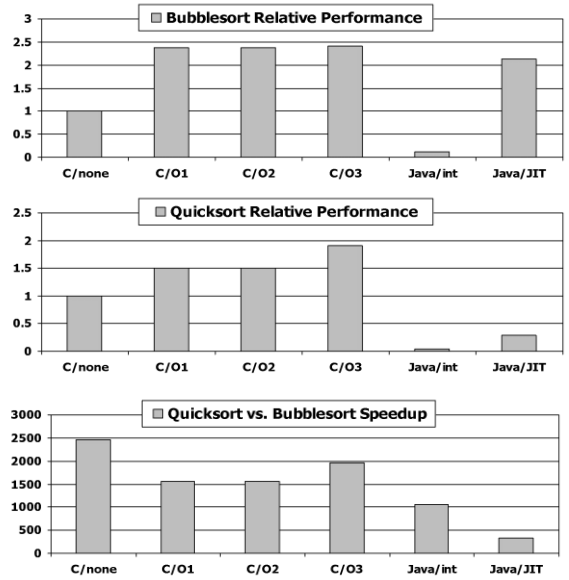
Effect of Compiler Optimization



Agenda

- Instruction Set
- Operations and Operands
- Representing Instructions in the Computer
- Logical Operations
- Instructions for Making Decisions
- Supporting Procedures in Computer Hardware
- LEGv8 Addressing for Wide Immediates and Addresses
- A C Sort Example to Put It All Together
- **Arrays versus Pointers**

Effect of Language and Algorithm



Comparison of Array vs. Ptr

- Multiply “strength reduced” to shift
- Array version requires shift to be inside loop
  - Part of index calculation for incremented i
  - c.f. incrementing pointer
- Compiler can achieve same effect as manual use of pointers
  - Induction variable elimination
  - Better to make program clearer and safer

Arrays vs. Pointers

- Array indexing involves(牵涉)
  - Multiplying index by element size
  - Adding to array base address
- Pointers correspond directly to memory addresses
  - Can avoid indexing complexity

ARM & MIPS Similarities

- ARM: the most popular embedded core
- Similar basic set of instructions to MIPS

	ARM	MIPS
Date announced	1985	1985
Instruction size	32 bits	32 bits
Address space	32-bit flat	32-bit flat
Data alignment	Aligned	Aligned
Data addressing modes	9	3
Registers	15 × 32-bit	31 × 32-bit
Input/output	Memory mapped	Memory mapped

Example: Clearing an Array

```
clear1(long long array[], size_t size)
{
  size_t i;
  for (i = 0; i < size; i++)
    array[i] = 0;
}
```

```
clear2(long long *array, size_t size)
{
  long long *p;
  for (p = &array[0]; p < &array[size];
       p++)
    *p = 0;
}
```

```
       MOV X9,XZR      // i = 0
loop1: LSL X10,X9,#3   // X10 = i * 8
       ADD X11,X0,X10  // X11 = address
                       // of array[i]
       STUR XZR,[X11,#0]
                       // array[i] = 0
       ADDI X9,X9,#1   // i = i + 1
       CMP X9,X1       // compare i to
                       // size
       B.LT loop1      // if (i < size)
                       // go to loop1
```

```
       MOV X9,X0       // p = address of
                       // array[0]
       LSL X10,X1,#3   // X10 = size * 8
       ADD X11,X0,X10  // X11 = address
                       // of array[size]
loop2: STUR XZR,[X9,#0]
                       // Memory[p] = 0
       ADDI X9,X9,#8   // p = p + 8
       CMP X9,X11      // compare p to <
                       // &array[size]
       B.LT loop2      // if (p <
                       // &array[size])
                       // go to loop2
```

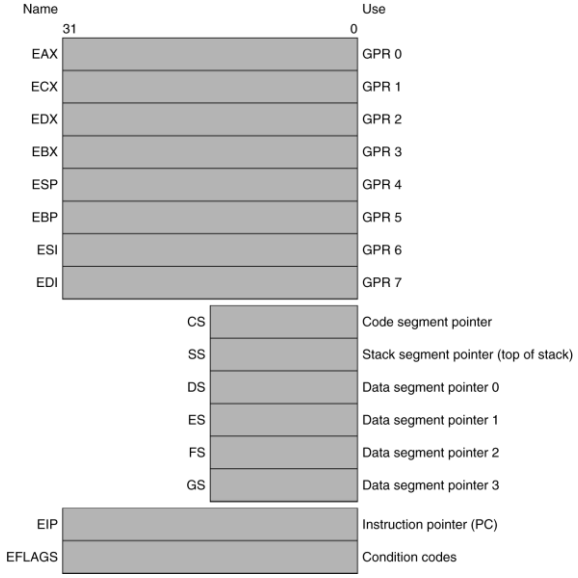
The Intel x86 ISA

- And further…
  - AMD64 (2003): extended architecture to 64 bits
  - EM64T – Extended Memory 64 Technology (2004)
    - AMD64 adopted by Intel (with refinements)
    - Added SSE3 instructions
  - Intel Core (2006)
    - Added SSE4 instructions, virtual machine support
  - AMD64 (announced 2007): SSE5 instructions
    - Intel declined to follow, instead…
  - Advanced Vector Extension (announced 2008)
    - Longer SSE registers, more instructions
- If Intel didn't extend with compatibility, its competitors would!
  - Technical elegance ≠ market success

The Intel x86 ISA

- Evolution with backward compatibility
  - 8080 (1974): 8-bit microprocessor
    - Accumulator, plus 3 index-register pairs
  - 8086 (1978): 16-bit extension to 8080
    - Complex instruction set (CISC)
  - 8087 (1980): floating-point coprocessor
    - Adds FP instructions and register stack
  - 80286 (1982): 24-bit addresses, MMU
    - Segmented memory mapping and protection
  - 80386 (1985): 32-bit extension (now IA-32)
    - Additional addressing modes and operations
    - Paged memory mapping as well as segments

Basic x86 Registers



The Intel x86 ISA

- Further evolution…
  - i486 (1989): pipelined, on-chip caches and FPU
    - Compatible competitors: AMD, Cyrix, …
  - Pentium (1993): superscalar, 64-bit datapath
    - Later versions added MMX (Multi-Media eXtension) instructions
    - The infamous FDIV bug
  - Pentium Pro (1995), Pentium II (1997)
    - New microarchitecture (see Colwell, *The Pentium Chronicles*)
  - Pentium III (1999)
    - Added SSE (Streaming SIMD Extensions) and associated registers
  - Pentium 4 (2001)
    - New microarchitecture
    - Added SSE2 instructions

Implementing IA-32

- Complex instruction set makes implementation difficult
  - Hardware translates instructions to simpler microoperations
    - Simple instructions: 1–1
    - Complex instructions: 1–many
  - Microengine similar to RISC
  - Market share makes this economically viable
- Comparable performance to RISC
  - Compilers avoid complex instructions

Basic x86 Addressing Modes

- Two operands per instruction

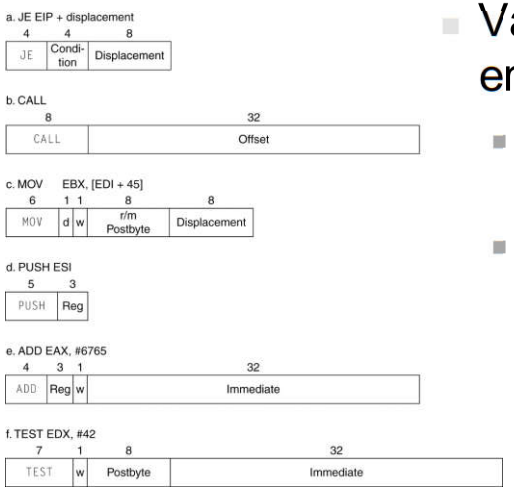
Source/dest operand	Second source operand
Register	Register
Register	Immediate
Register	Memory
Memory	Register
Memory	Immediate

- Memory addressing modes
  - Address in register
  - Address = R_{base} + displacement
  - Address = $R_{base} + 2^{scale} \times R_{index}$ (scale = 0, 1, 2, or 3)
  - Address = $R_{base} + 2^{scale} \times R_{index}$ + displacement

ARM v8 Instructions

- In moving to 64-bit, ARM did a complete overhaul
- ARM v8 resembles MIPS
  - Changes from v7:
    - No conditional execution field
    - Immediate field is 12-bit constant
    - Dropped load/store multiple
    - PC is no longer a GPR
    - GPR set expanded to 32
    - Addressing modes work for all word sizes
    - Divide instruction
    - Branch if equal/branch if not equal instructions

x86 Instruction Encoding



- Variable length encoding
  - Postfix bytes specify addressing mode
  - Prefix bytes modify operation
    - Operand length, repetition, locking, …

DISCUSSION SESSION 1
RISC VS. CISC

Fallacies

- Powerful instruction ⇒ higher performance
  - Fewer instructions required
  - But complex instructions are hard to implement
    - May slow down all instructions, including simple ones
  - Compilers are good at making fast code from simple instructions
- Use assembly code for high performance
  - But modern compilers are better at dealing with modern processors
  - More lines of code ⇒ more errors and less productivity

Concluding Remarks

- Design principles
  1. Simplicity favors regularity
  2. Smaller is faster
  3. Make the common case fast
  4. Good design demands good compromises
- Layers of software/hardware
  - Compiler, assembler, hardware
- ARM: typical of RISC ISAs
  - c.f. x86