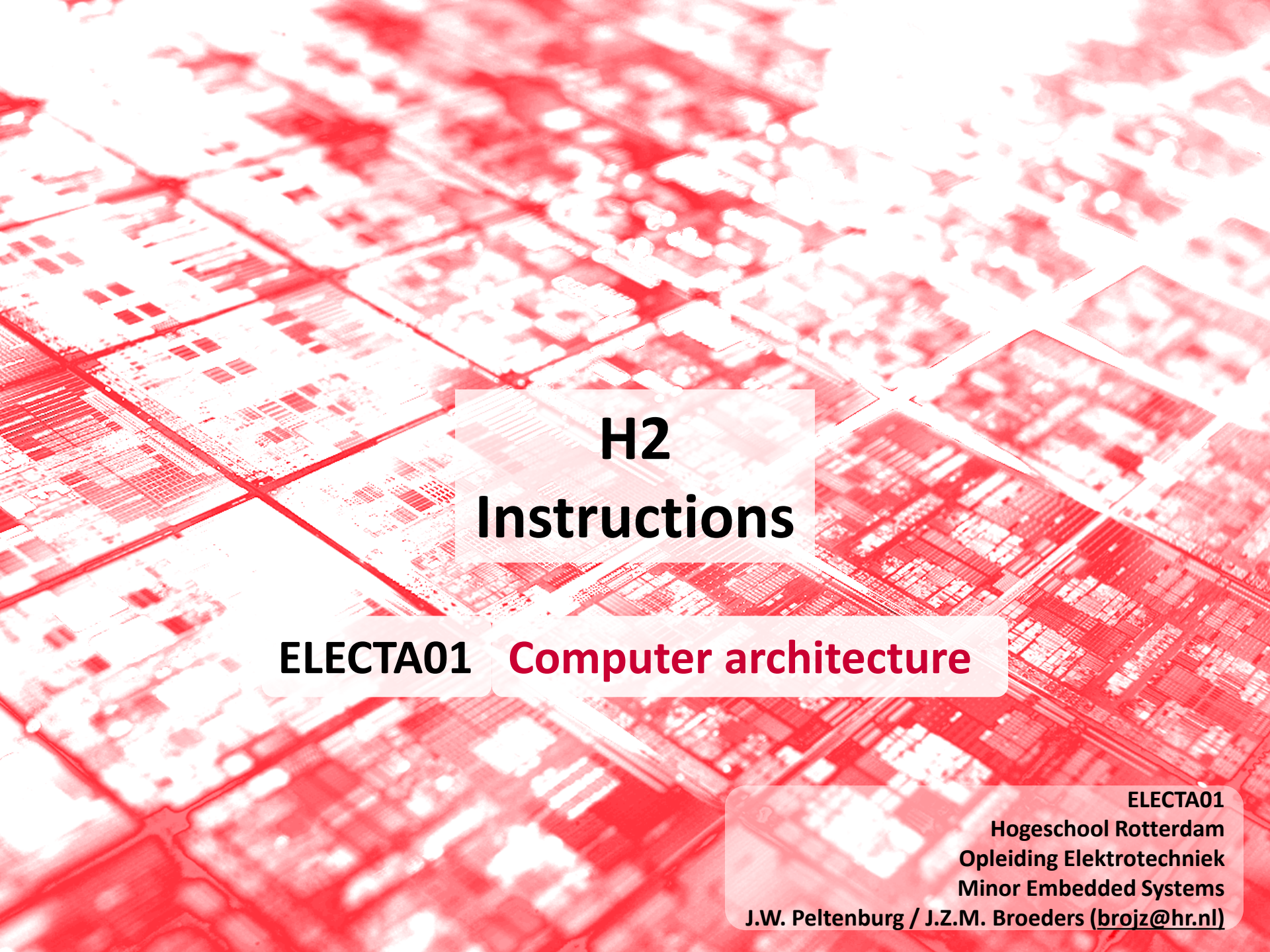


H2
Instructions

ELECTA01 Computer architecture

ELECTA01
Hogeschool Rotterdam
Opleiding Elektrotechniek
Minor Embedded Systems
J.W. Peltenburg / J.Z.M. Broeders (brojz@hr.nl)

INSTRUCTIONS: THE LANGUAGE OF THE COMPUTER

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

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

Instruction Set

- The repertoire 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

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

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

Design Principles applied to the ARMv8 ISA

- 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

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

Arithmetic Example

- C code:

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

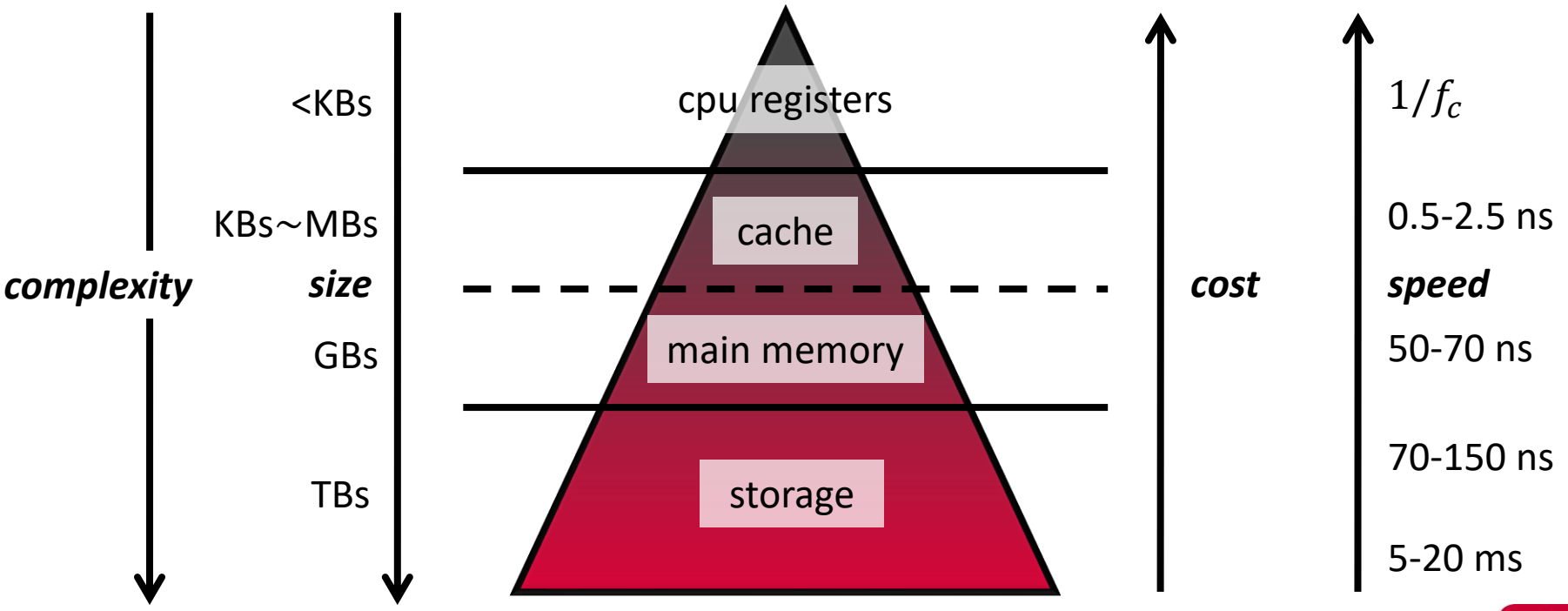
- Compiled LEGv8 code:

```
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

Memory Hierarchy

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



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

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

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
```

Memory Operands

- Main memory used for composite 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

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 64

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

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 to memory for less frequently used variables
  - Register optimization is important!

Immediate Operands

- Constant data specified in an instruction

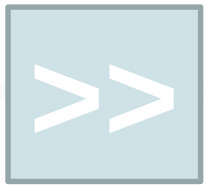
```
ADDI X22, X22, #4
```

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

Signed and Unsigned Numbers

- **I** skip this part of the lecture because you already now this stuff.
- **You** maybe have to study paragraph 2.4 yourself!

Unsigned Binary Integers

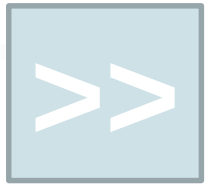


- Given an n-bit number

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

- Range: 0 to $+2^n - 1$
- Example
  - $0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1011_2$
    $= 0 + \ldots + 1\times2^3 + 0\times2^2 + 1\times2^1 + 1\times2^0$
    $= 0 + \ldots + 8 + 0 + 2 + 1 = 11_{10}$
- Using 32 bits
  - 0 to +4,294,967,295

2s-Complement Signed Integers

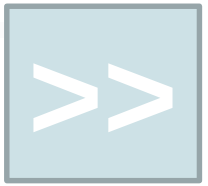


- Given an n-bit number

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

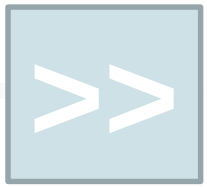
- Range: -2^{n-1} to $+2^{n-1} - 1$
- Example
  - $1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1100_2$
    $= -1\times2^{31} + 1\times2^{30} + \ldots + 1\times2^2 + 0\times2^1 + 0\times2^0$
    $= -2{,}147{,}483{,}648 + 2{,}147{,}483{,}644 = -4_{10}$
- Using 32 bits
  - –2,147,483,648 to +2,147,483,647

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

Signed Negation



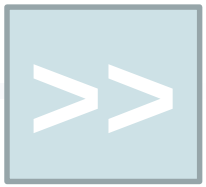
- Complement and add 1
  - Complement means 1 → 0, 0 → 1

$$x + \bar{x} = 1111\ldots111_2 = -1$$

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

- Example: negate +2
  - +2 = 0000 0000 … 0010_{two}
  - –2 = 1111 1111 … 1101_{two} + 1
    = 1111 1111 … 1110_{two}

Sign Extension



- Representing a number using more bits
  - Preserve 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

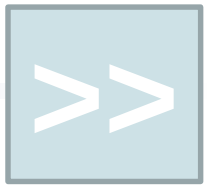
Agenda

- 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

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!

Hexadecimal

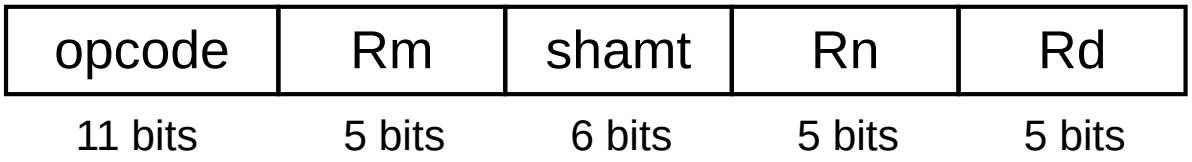


- Base 16
  - Compact representation of bit strings
  - 4 bits per hex digit

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

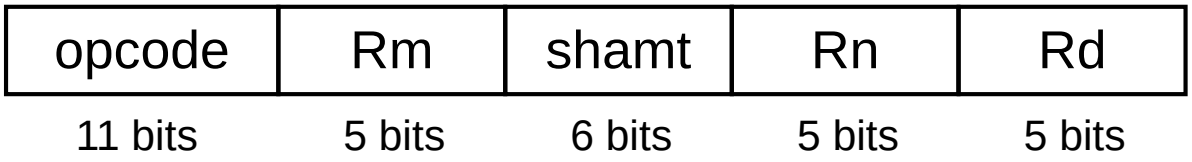
- Example: ECA8 6420
  - 1110 1100 1010 1000 0110 0100 0010 0000

LEGv8 R-format Instructions



- 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

R-format Example



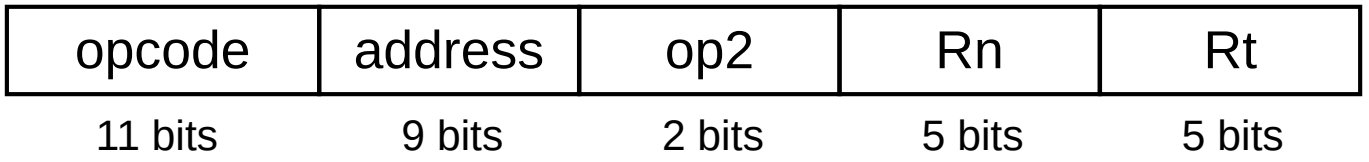
ADD X9,X20,X21

1112_{ten}	21_{ten}	0_{ten}	20_{ten}	9_{ten}
10001011000_{two}	10101_{two}	000000_{two}	10100_{two}	01001_{two}

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

$8B150289_{16}$

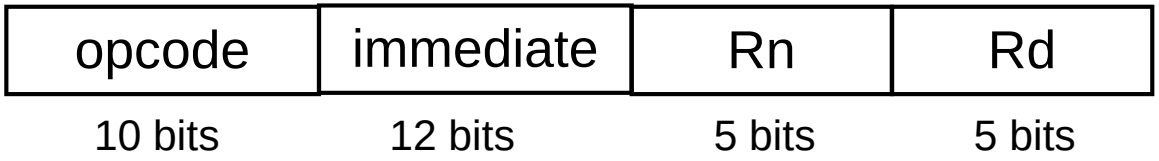
LEGv8 D-format Instructions



- 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 3:* Good design demands good compromises
  - Different formats complicate decoding, but allow 32-bit instructions uniformly
  - Keep formats as similar as possible

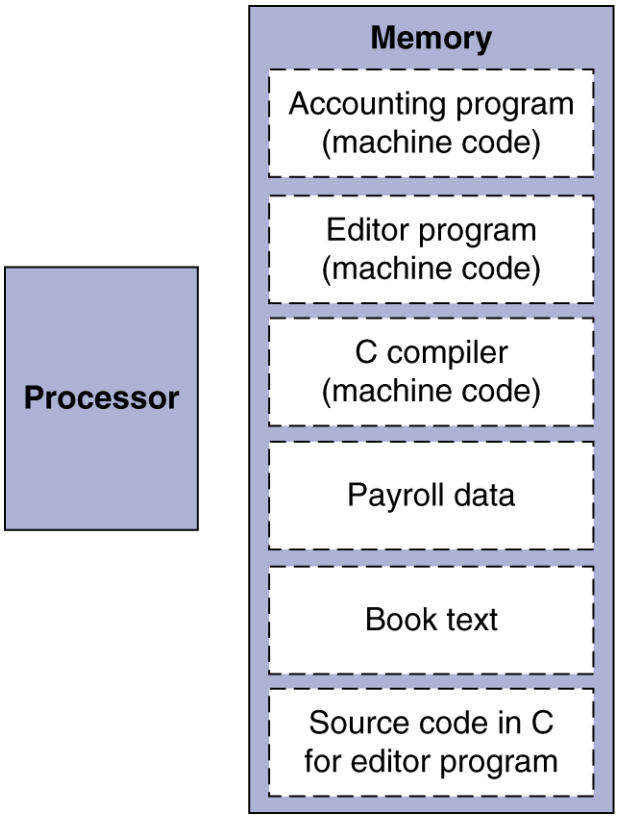
LEGv8 I-format Instructions



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

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

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

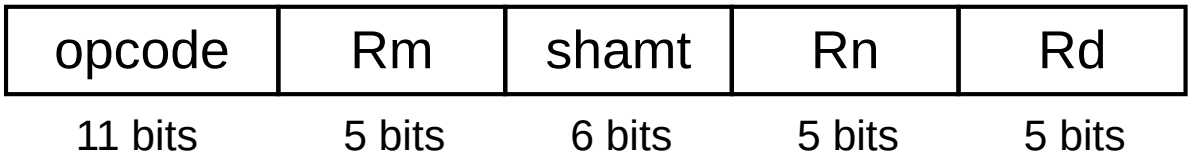
Logical Operations

- Instructions for bitwise 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

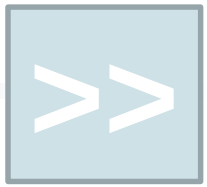
- Useful for extracting and inserting groups of bits in a word

Shift Operations



- 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)

AND Operations

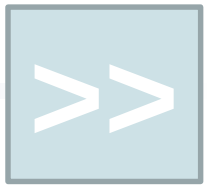


- Useful to mask bits in a word
  - Set some bits to 0, leave others unchanged

```
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

OR Operations

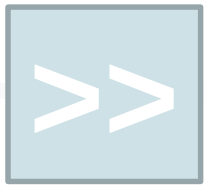


- Useful to include bits in a word
  - Set some bits to 1, leave others unchanged

```
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

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

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 Operations

- Branch to a labeled instruction if a condition is true
  - Otherwise, continue sequentially

- `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;

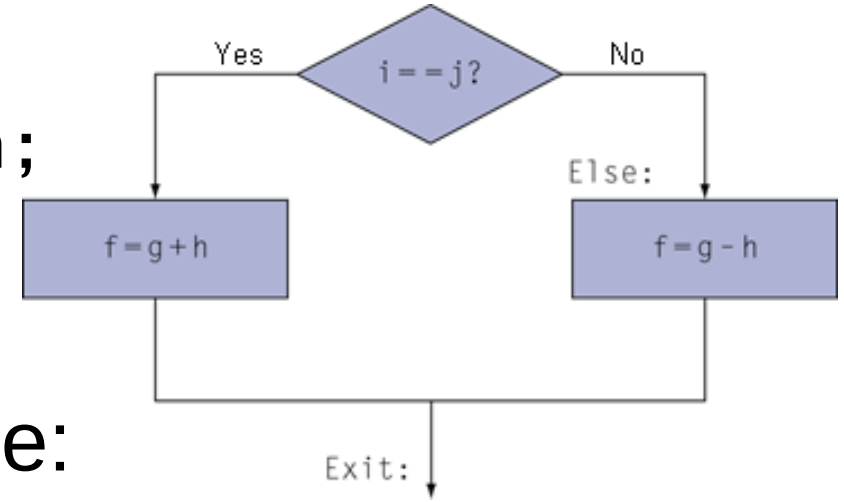
- `B L1`
  - branch unconditionally to instruction labeled L1;

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

Compiling Loop Statements

- C code:

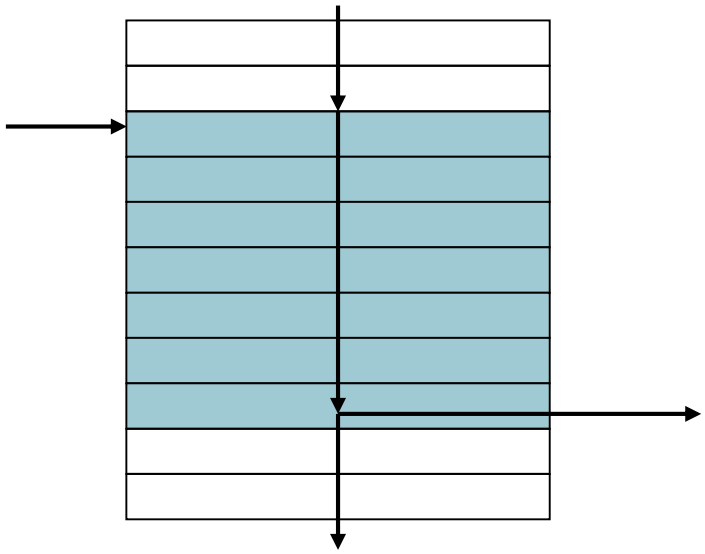
```
while (save[i] == k) i += 1;
```

  - 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: …
```

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

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)

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

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

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:
```

Pseudo instructions

- A pseudo 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

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

Procedure Calling

- Steps **required**
  1. Place parameters in registers X0 to X7
  2. **Transfer control to procedure**
  3. Acquire 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"

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

Register Usage

- X9 to X17: temporary registers
  - Not preserved by the callee

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

ARM Call Convention

Should we follow this convention?

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

Leaf Procedure Example

- LEGv8 code:

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

Leaf Procedure Example

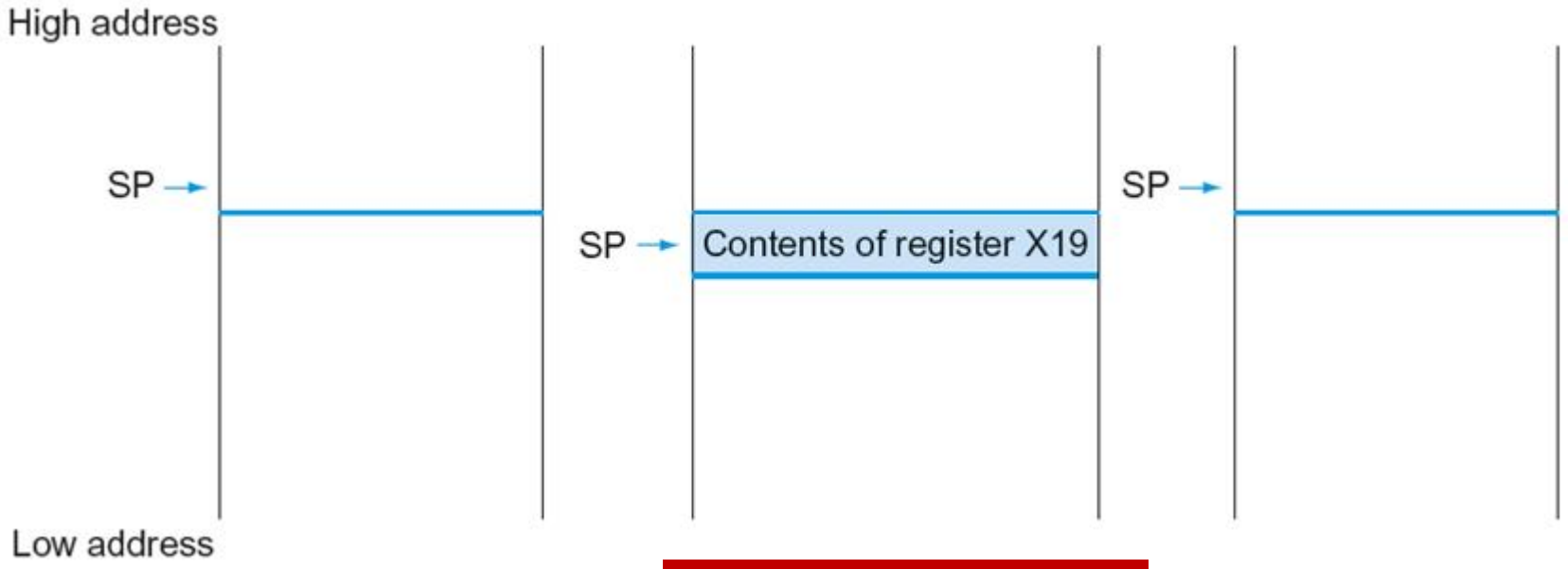
- C code:

```
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

Local Data on the Stack



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
```

Non-Leaf Procedures

- Procedures that call other procedures
- For nested 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

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! 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

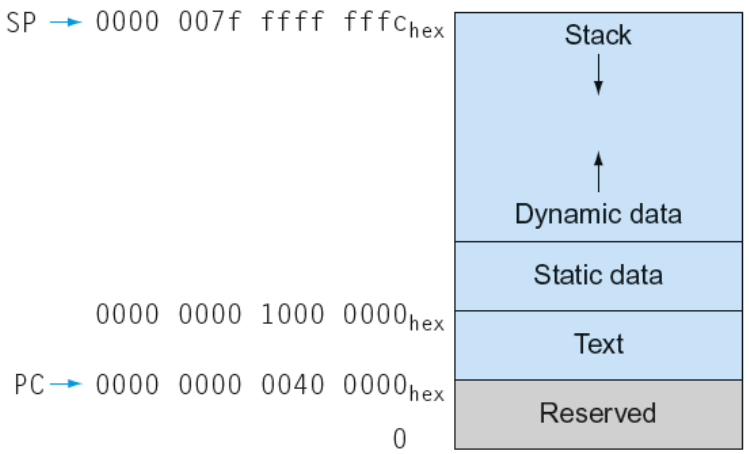
Leaf Procedure Example

- 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
```

Memory Layout

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



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

64-bit Constants

- Most constants are small
  - 12-bit immediate is sufficient
- For the occasional 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

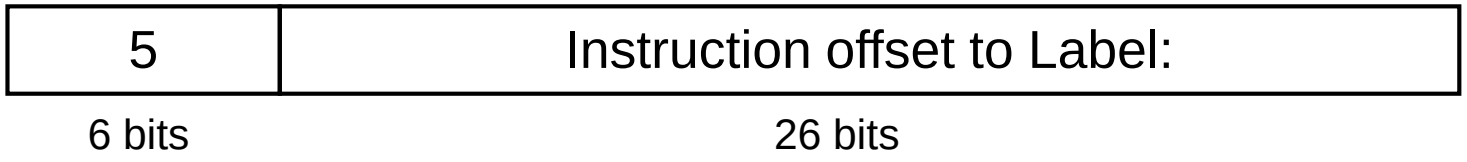
Pseudo instructions

- A pseudo instruction in assembler language is translated into an other instruction.
- Meant to make your live a little easier.

LDA Xa, Label ➔ proper set of MOVZ and MOVK instructions (max 4)

Branch Addressing

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



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

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

LEGv8 Addressing Summary

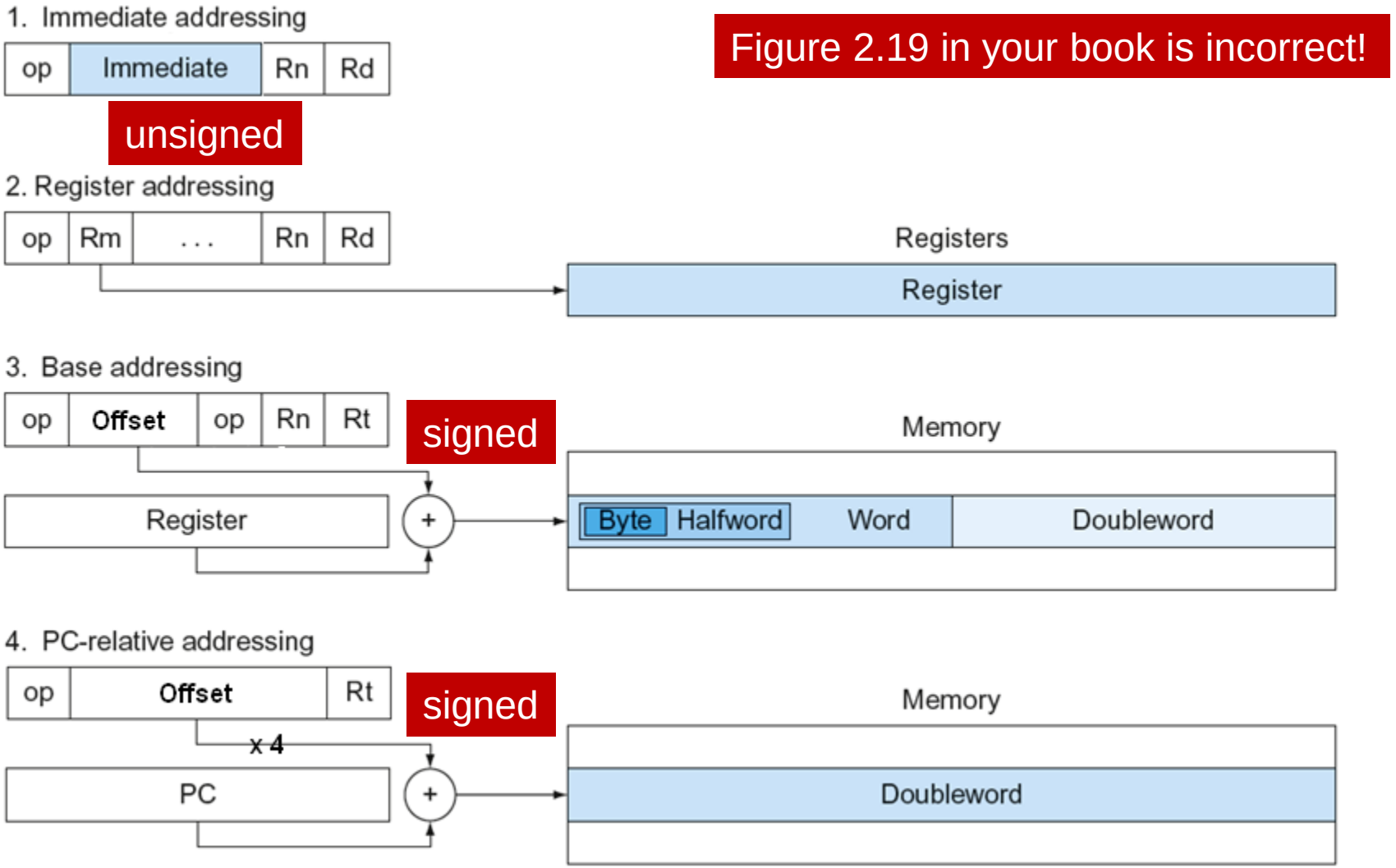


Figure 2.19 in your book is incorrect!

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

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

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?

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
```

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?

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:
```

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:
```

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
```

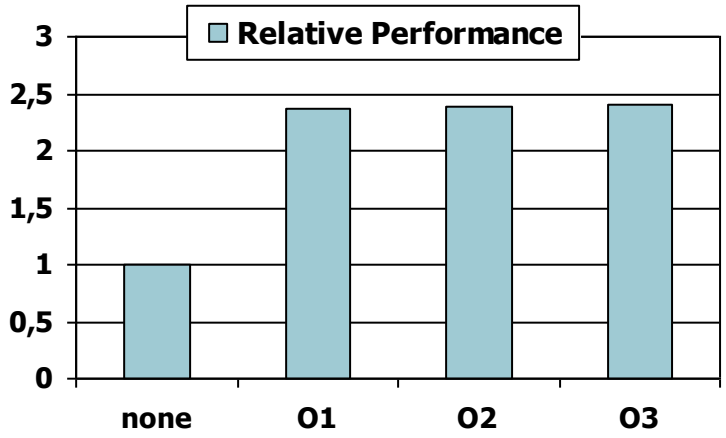
- Restore saved registers:

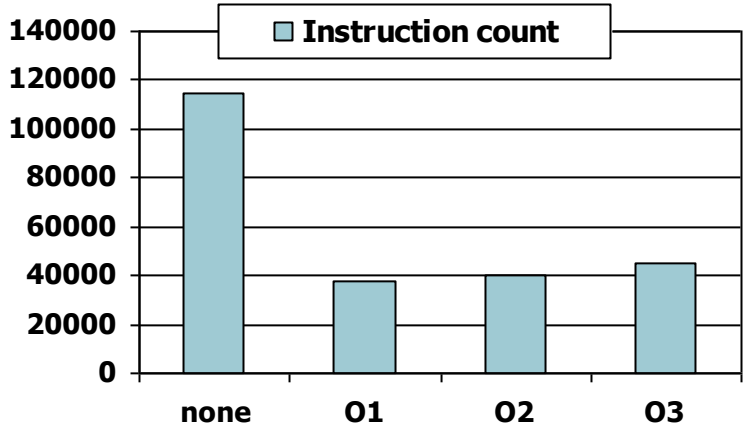
```
exit1: LDUR X19, [SP,#0]  // restore X19 from stack
  LDUR X20, [SP,#8]       // restore X20 from stack
  LDUR X21,[SP,#16]       // restore X21 from stack
  LDUR X22,[SP,#24]       // restore X22 from stack
  LDUR X30,[SP,#32]       // restore LR from stack
  ADDI SP,SP,#40          // restore stack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

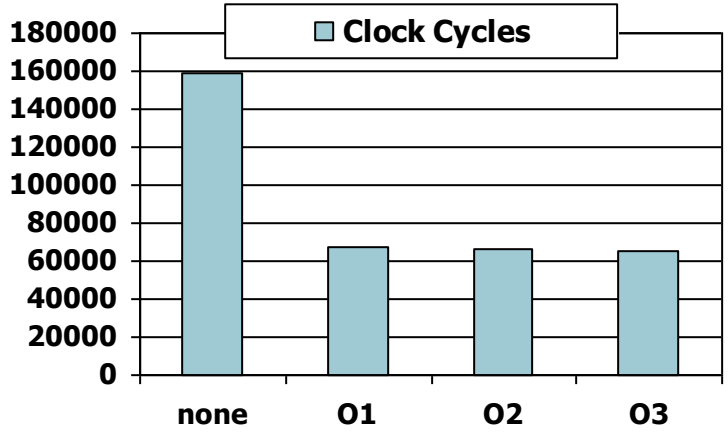
The code in your book is incorrect!

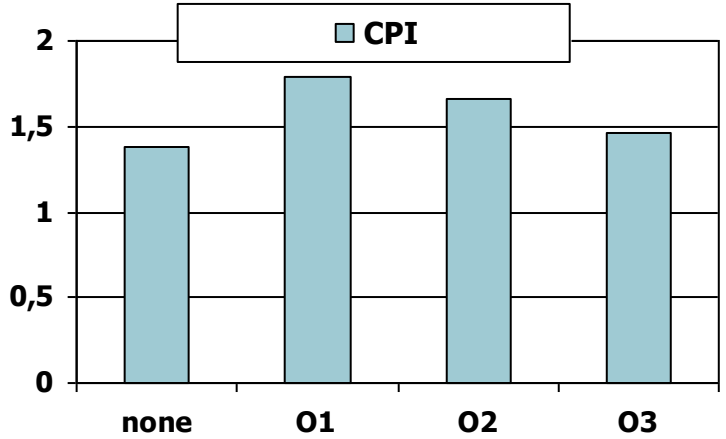
Effect of Compiler Optimization

Compiled with gcc for Pentium 4 under Linux

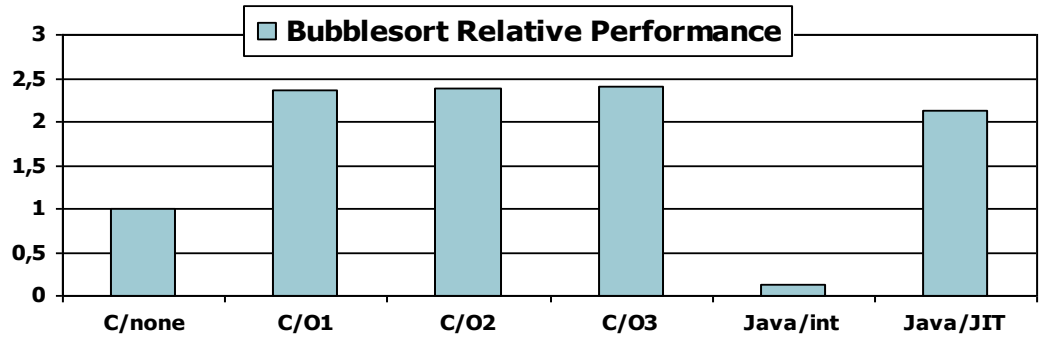


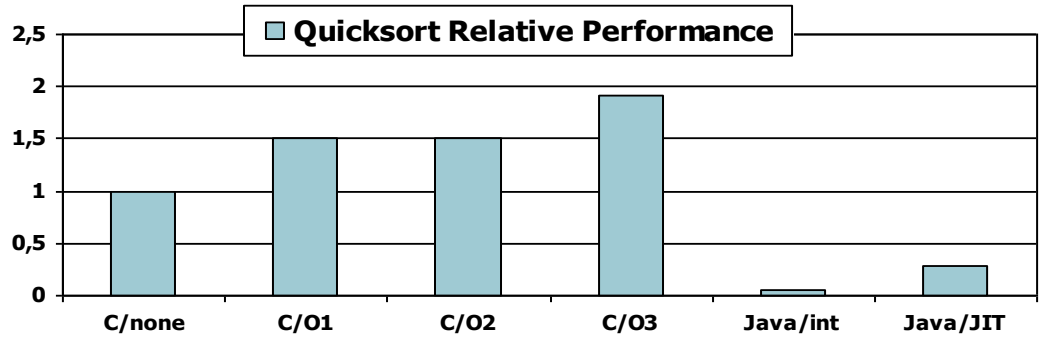


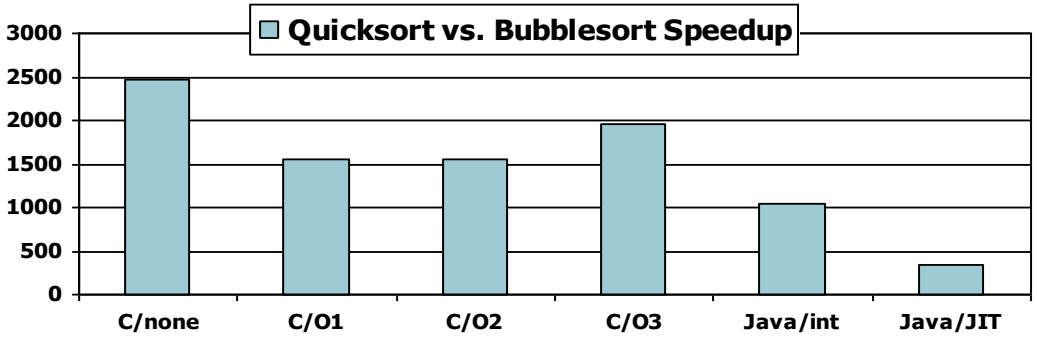




Effect of Language and Algorithm







Lessons Learnt

- Instruction count and CPI are not good performance indicators in isolation
- 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!

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**

Arrays vs. Pointers

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

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
```

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

NIET VERGETEN

- Mail sturen over met wie je practicum wilt doen.