Chapter 2

Instructions: Language of the Computer

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

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 favours 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
  ADD f, t0, t1  // f = t0 - t1
  ```

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
  - c.f. 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

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

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

- $0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1011_2$
  $= 0 + \ldots + 1{\times}2^3 + 0{\times}2^2 + 1{\times}2^1 + 1{\times}2^0$
  $= 0 + \ldots + 8 + 0 + 2 + 1 = 11_{10}$

■ Using 32 bits

- 0 to +4,294,967,295

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 $+2{,}147{,}483{,}647$

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 $1 \rightarrow 0$, $0 \rightarrow 1$

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

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

- 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

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

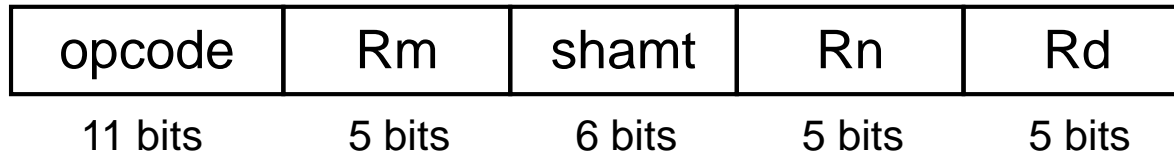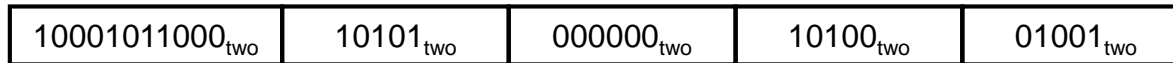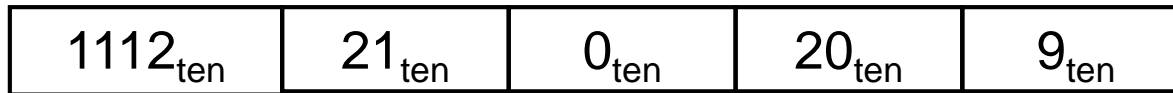| opcode	Rm	shamt	Rn	Rd
11 bits	5 bits	6 bits	5 bits	5 bits

- **Instruction fields**
  - opcode: operation code
  - Rm: the second register source operand
  - shamt: shift amount (00000 for now)
  - Rn: the first register source operand
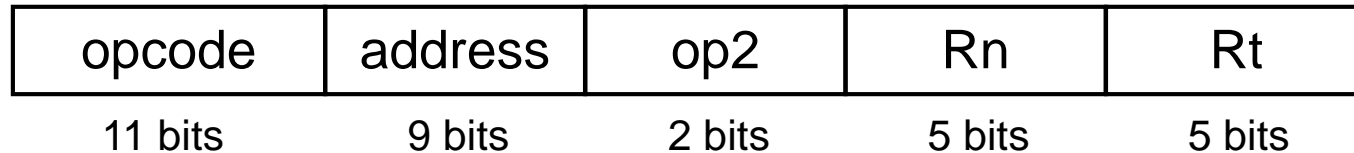 - Rd: the register destination

R-format Example

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

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

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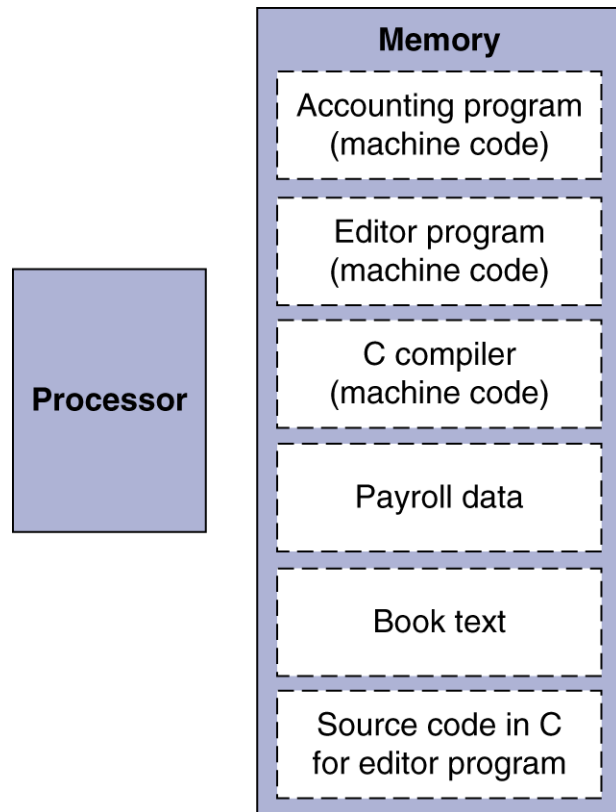 - Keep formats as similar as possible

LEGv8 I-format Instructions

opcode	immediate	Rn	Rd
10 bits	12 bits	5 bits	5 bits
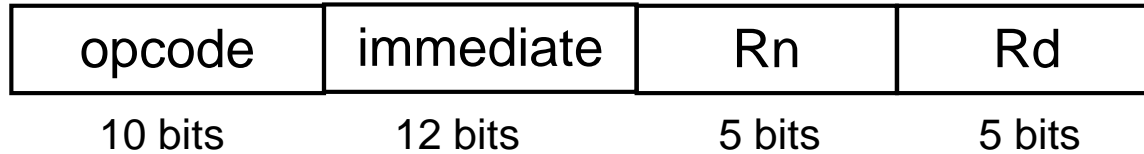
- **Immediate instructions**
  - Rn: source register
  - Rd: destination register

- **Immediate field is zero-extended**

Stored Program Computers

The BIG Picture

Memory

- Accounting program (machine code)
- Editor program (machine code)
- C compiler (machine code)
- Payroll data
- Book text
- Source code in C for editor program

Processor

- 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

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	|	|	OR, ORI
Bit-by-bit NOT	~	~	EOR, EORI

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

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)

AND Operations

- Useful to mask bits in a word
  - Select some bits, clear others to 0

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

OR 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**
  - Set some bits to 1, 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

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;

- `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;
  ```
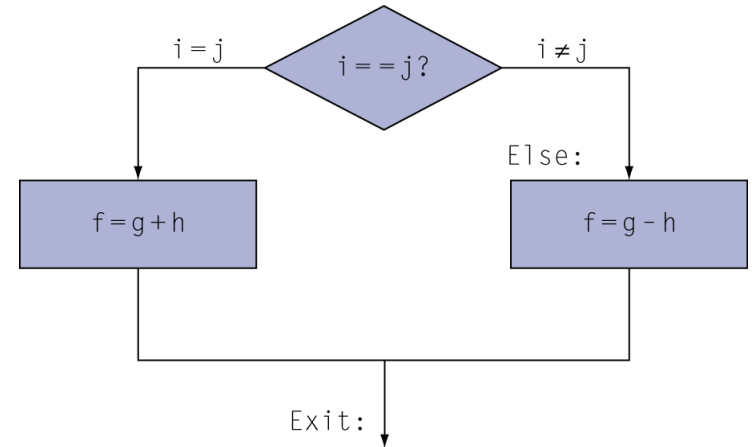
  - f, g, … in X22, X23, …

- Compiled LEGv8 code:

  ```
          SUB X9,X22,X23
          CBNZ X9,Else
          ADD X19,X20,X21
          B Exit
  Else:   SUB X9,X22,x23
  Exit:  …
  ```

Assembler calculates addresses

Compiling Loop Statements

- C code:

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

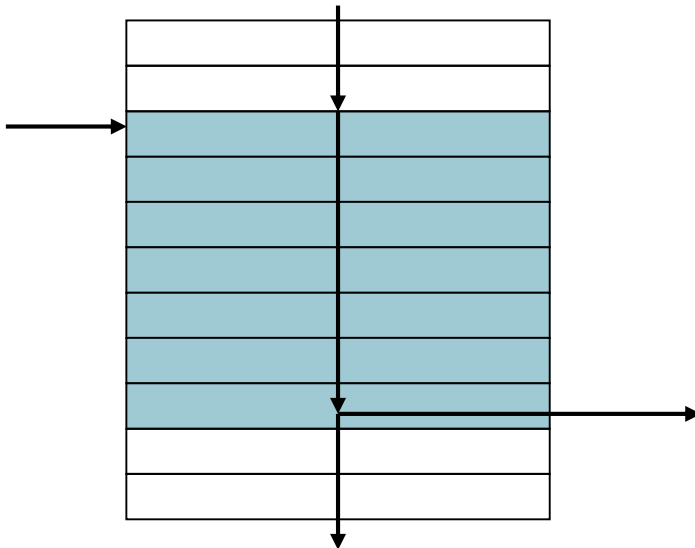 - i in x22, k in x24, address of save in x25

- Compiled LEGv8 code:

  ```
  Loop: LSL    X10,X22,#3
        ADD    X10,X10,X25
        LDUR   X9,[X10,#0]
        SUB    X11,X9,X24
        CBNZ   X11,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
    - overlow (V): result overflowed
    - carry (C): result had carryout from MSB
- Use subtract to set flags, then conditionally branch:
    - **B.EQ**
    - **B.NE**
    - **B.LT** (less than, signed), **B.LO** (less than, unsigned)
    - **B.LE** (less than or equal, signed), **B.LS** (less than or equal, unsigned)
    - **B.GT** (greater than, signed), **B.HI** (greater than, unsigned)
    - **B.GE** (greater than or equal, signed),
    - **B.HS** (greater than or equal, unsigned)

Conditional Example

- if (a > b) a += 1;
  - a in X22, b in X23

```
SUBS X9,X22,X23  // use subtract to make comparison
B.LTE Exit       // conditional branch
ADDI X22,X22,#1
Exit:
```

Signed vs. Unsigned

- Signed comparison
- Unsigned comparison
- Example
  - X22 = 1111 1111 1111 1111 1111 1111 1111 1111
  - X23 = 0000 0000 0000 0000 0000 0000 0000 0001
  - X22 < X23 # signed
    - −1 < +1
  - X22 > X23 # unsigned
    - +4,294,967,295 > +1

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)

Procedure Call Instructions

- Procedure call: jump and link

  `BL ProcedureLabel`

  - Address of following instruction put in X30
  - Jumps to target address

- Procedure return: jump register

  `BR LR`

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

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
  - f in X19 (hence, need to save $s0 on stack)

Leaf Procedure Example

- ## LEGv8 code:

```
leaf_example:
SUBI SP,SP,#24
STUR X10,[SP,#16]
STUR X9,[SP,#8]
STUR X19,[SP,#0]
ADD X9,X0,X1
ADD X10,X2,X3
SUB X19,X9,X10
ADD X0,X19,XZR
LDUR X10,[SP,#16]
LDUR X9,[SP,#8]
LDUR X19,[SP,#0]
ADDI SP,SP,#24
BR LR
```

Save X10, X9, X19 on stack

X9 = g + h

X10 = i + j

f = X9 − X10
copy f to return register

Resore X10, X9, X19 from stack

Return to caller

Local Data on the Stack

High address

SP →

SP → | Contents of register X10 |
| Contents of register X9 |
| Contents of register X19 |

SP →

Low address
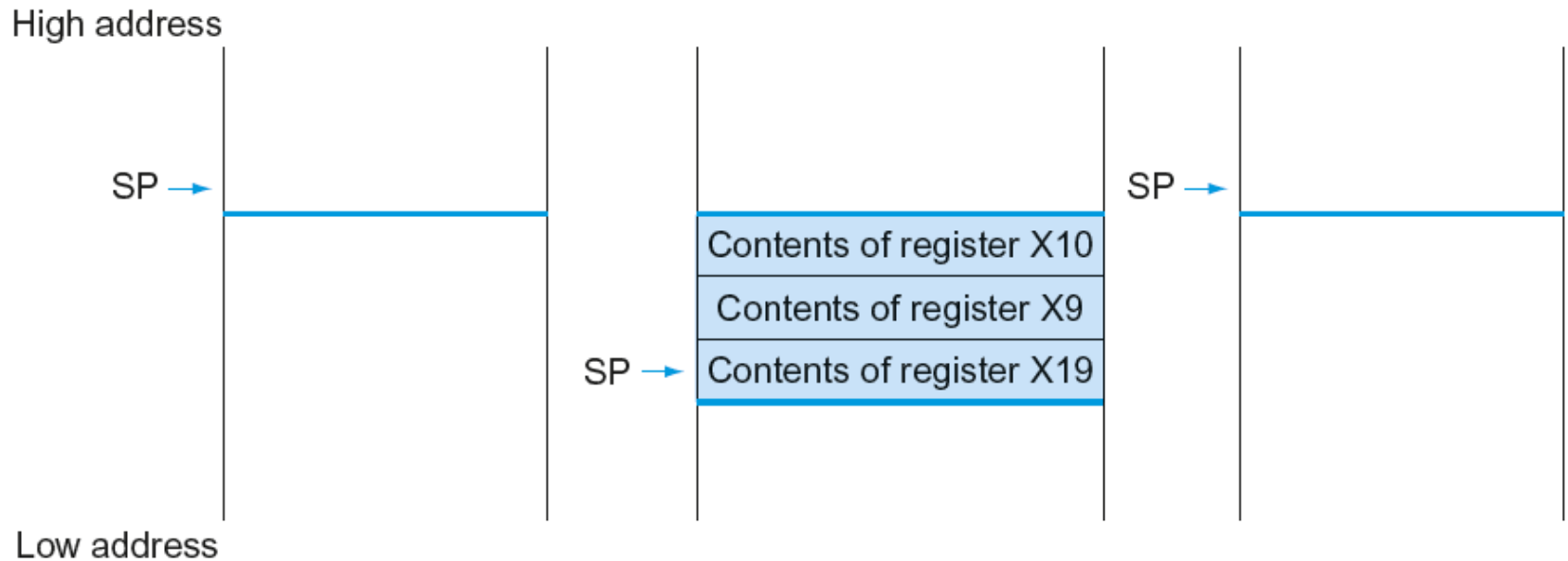
Register Usage

- ## X9 to X17: temporary registers

  - ### Not preserved by the callee

- ## X19 to X28: saved registers

  - ### If used, the callee saves and restores them

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:

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

- Argument n in X0
- Result in X1

Leaf Procedure Example

- ## LEGv8 code:

```
fact:
    SUBI SP,SP,#16              Save return address and n on stack
    STUR LR,[SP,#8]
    STUR X0,[SP,#0]
    SUBIS XZR,X0,#1            compare n and 1
    B.GE L1                    if n >= 1, go to L1
    ADDI X1,XZR,#1            Else, set return value to 1
    ADDI SP,SP,#16           Pop stack, don't bother restoring values
    BR LR                     Return
L1: SUBI X0,X0,#1             n = n - 1
    BL fact                   call fact(n-1)
    LDUR X0,[SP,#0]           Restore caller's n
    LDUR LR,[SP,#8]           Restore caller's return address
    ADDI SP,SP,#16           Pop stack
    MUL X1,X0,X1             return n * fact(n-1)
    BR LR                     return
```
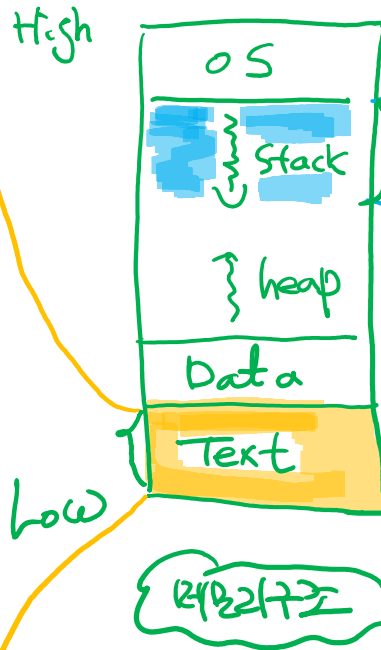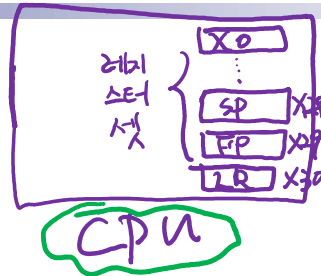
Memory Layout

- 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: automatic storage

SP → 0000 007f ffff fffc$_{hex}$

0000 0000 1000 0000$_{hex}$

PC → 0000 0000 0040 0000$_{hex}$

0

Stack
↓
↑
Dynamic data
Static data
Text
Reserved
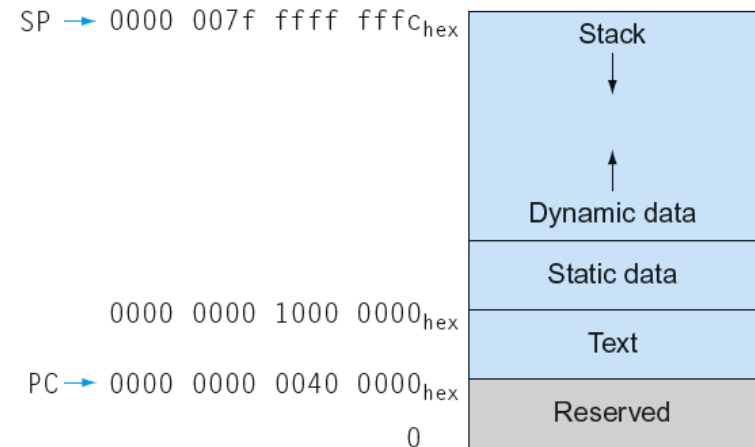
Character Data

- Byte-encoded character sets
  - ASCII: 128 characters
    - 95 graphic, 33 control
  - Latin-1: 256 characters
    - ASCII, +96 more graphic characters
- Unicode: 32-bit character set
  - Used in Java, C++ wide characters, …
  - Most of the world's alphabets, plus symbols
  - UTF-8, UTF-16: variable-length encodings

Byte/Halfword Operations

- LEGv8 byte/halfword load/store
  - Load byte:
    - LDURB Rt, [Rn, offset]
    - Sign extend to <u>56</u> bits in rt
  - Store byte:
    - STURB Rt, [Rn, offset]
    - Store just rightmost byte
  - Load halfword:
    - LDURH Rt, [Rn, offset]
    - Sign extend to 32 bits in rt
  - Store halfword:
    - STURH Rt, [Rn, offset]
    - Store just rightmost halfword

String Copy Example

- C code:
  - Null-terminated string

```
void strcpy (char x[], char y[])
{ size_t i;
  i = 0;
  while ((x[i]=y[i])!='\0')
    i += 1;
}
```

String Copy Example

- **LEGv8 code:**

```
strcpy:
    SUBI SP,SP,8            // push X19
    STUR X19,[SP,#0]
    ADD X19,XZR,XZR         // i=0
L1: ADD X10,X19,X1          // X10 = addr of y[i]
    LDURB X11,[X10,#0]      // X11 = y[i]
    ADD X12,X19,X0          // X12 = addr of x[i]
    STURB X11,[X12,#0]      // x[i] = y[i]
    CBZ X11,L2              // if y[i] == 0 then exit
    ADDI X19,X19,#1         // i = i + 1
    B L1                    // next iteration of loop
L2: LDUR X19,[SP,#0]        // restore saved $s0
    ADDI SP,SP,8            // pop 1 item from stack
    BR LR                   // and return
```

32-bit Constants

■ Most constants are small

- 12-bit immediate is sufficient

■ For the occasional 32-bit constant

```
MOVZ:   move wide with zeros
MOVK:   move with with keep
```

■ Use with flexible second operand (shift)

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

Branch Addressing

- ## B-type
  - B 1000 // go to location 10000_{ten}

5	10000_{ten}
6 bits	26 bits
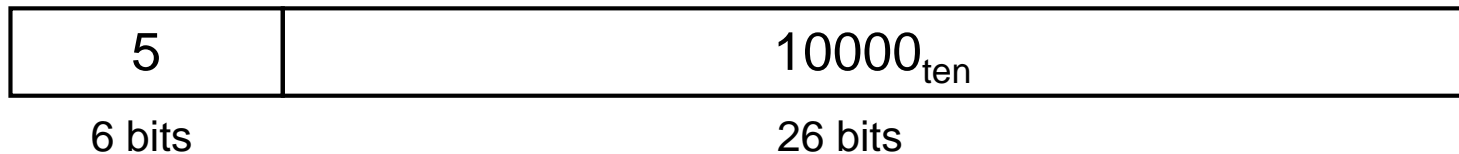
- ## CB-type
  - CBNZ X19, Exit // go to Exit if X19 != 0

181	Exit	19
8 bits	19 bits	5 bits

- ## Both addresses are PC-relative
  - Address = PC + offset (from instruction)
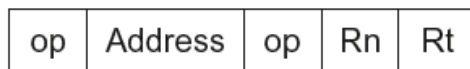
LEGv8 Addressing Summary

1. Immediate addressing

op	rs	rt	Immediate

2. Register addressing

op	Rm	. . .	Rn	Rd
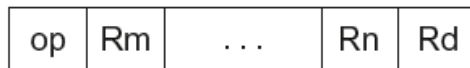
Registers

Register

3. Base addressing

op	Address	op	Rn	Rt

Register +

Memory

| Byte | Halfword | Word | Doubleword |

4. PC-relative addressing

op	Address	Rt

PC +

Memory

Doubleword
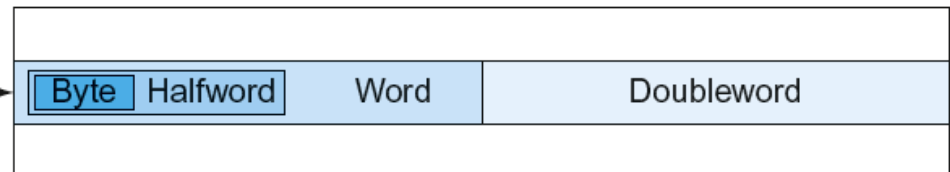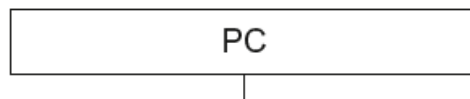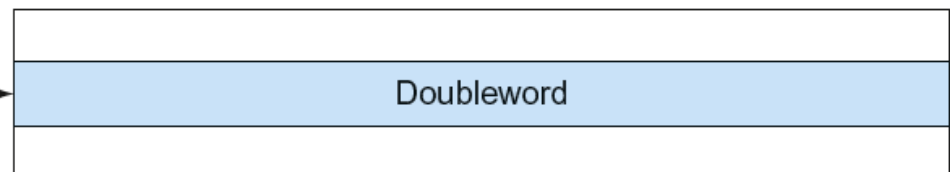
LEGv8 Encoding Summary

Name		6 to 11 bits	5 to 10 bits	5 or 4 bits	2 bits	5 bits	5 bits	Comments
		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

R: 11   5   6   5   5
D: 11   9   2   5   5
i: 10   12   5 5
CB: 8   19   5
B: 6   26
IW: 9   2   16   5

Synchronization

- Two processors sharing an area of memory
  - P1 writes, then P2 reads
  - Data race if P1 and P2 don't synchronize
    - Result depends of order of accesses

- Hardware support required
  - Atomic read/write memory operation
  - No other access to the location allowed between the read and write

- Could be a single instruction
  - E.g., atomic swap of register ↔ memory
  - Or an atomic pair of instructions

Synchronization in LEGv8

- Load exclusive register: LDXR
- Store exclusive register: STXR
- To use:
  - Execute LDXR then STXR with same address
  - If there is an intervening change to the address, store fails (communicated with additional output register)
  - Only use register instruction in between

Synchronization in LEGv8

- Example 1: atomic swap (to test/set lock variable)

```
again:  LDXR X10,[X20,#0]
        STXR X23,X9,[X20] // X9 = status
        CBNZ X9, again
        ADD X23,XZR,X10 // X23 = loaded value
```
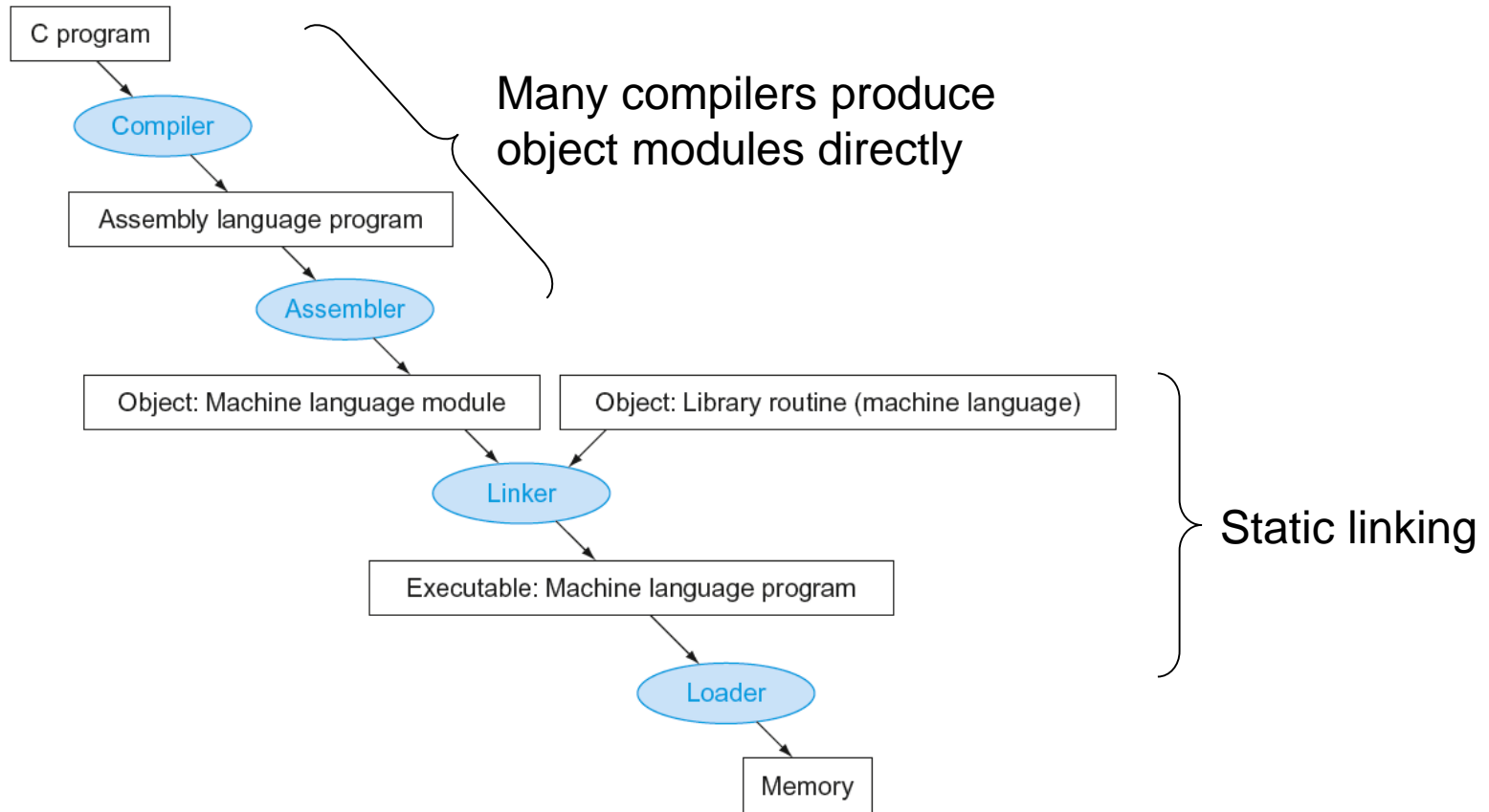
- Example 2: lock

```
        ADDI X11,XZR,#1        // copy locked value
again:  LDXR X10,[X20,#0]      // read lock
        CBNZ X10, again        // check if it is 0 yet
        STXR X11, X9, [X20]    // attempt to store
        BNEZ X9,again          // branch if fails
```

- Unlock:

```
        STUR XZR, [X20,#0]     // free lock
```

Translation and Startup

C program

Compiler

Assembly language program

Assembler

Object: Machine language module   Object: Library routine (machine language)

Linker

Executable: Machine language program

Loader

Memory

Many compilers produce object modules directly

Static linking

어셈블러의 의사명령어(Pseudo Instruction)

LEGv8 기계어에는 move 명령어가 없지만 LEGv8 어셈블러는 이 명령을 받아들인다.

```
MOV X9, X10        // register X9 gets register X10
```

어셈블러는 이 명령어를 다음 명령에 해당하는 기계어로 바꾼다.

```
ORR X9, XZR, X10   // register X9 gets 0 OR register X10
```

LEGv8 어셈블러는 CMP(compare) 명령어를 뺄셈 명령어로 바꾸는데 이 뺄셈 명령어는 컨디션 코드를 설정하고 XZR을 목적지로 갖는다. 따라서

```
CMP X9, X10        // compare X9 to X10 and set condition codes
```

는

```
SUBS XZR, X9, X10 // use X9 - X10 to set condition codes
```

가 된다.

또한 먼 거리로 분기하는 명령어는 두 개의 분기 명령어로 바꾸기도 한다. LEGv8에서는 명령어의 수치 필드는 제한된 크기를 갖지만 LEGv8 어셈블러는 레지스터에 큰 상수를 넣는 일도 해 줄 수 있다. 이렇게 어셈블러는 LDA(load address)를 받아들여 필요한 명령어 시퀀스로 바꿔 준다. 마지막으로 어셈블러는 프로그래머가 원하는 명령어 변형을 처리하여 명령어 집합을 좀 더 간단하게 해 준다. 예를 들어 LEGv8 어셈블러는 산술 논리 연산 명령어에서 수치값을 사용할 때 프로그래머가 굳이 수치 명령어를 사용하지 않아도 되게 해 준다. 대신 어셈블러가 알맞은 opcode를 생성해 준다. 예를 들어

```
AND X9, X10, #15   // register X9 gets X10 AND 15
```

는

```
ANDI X9, X10, #15 // register X9 gets X10 AND 15
```

가 된다.

```
CBZ X19, L1

CBNZ X19, L2
B L1
L2:
```
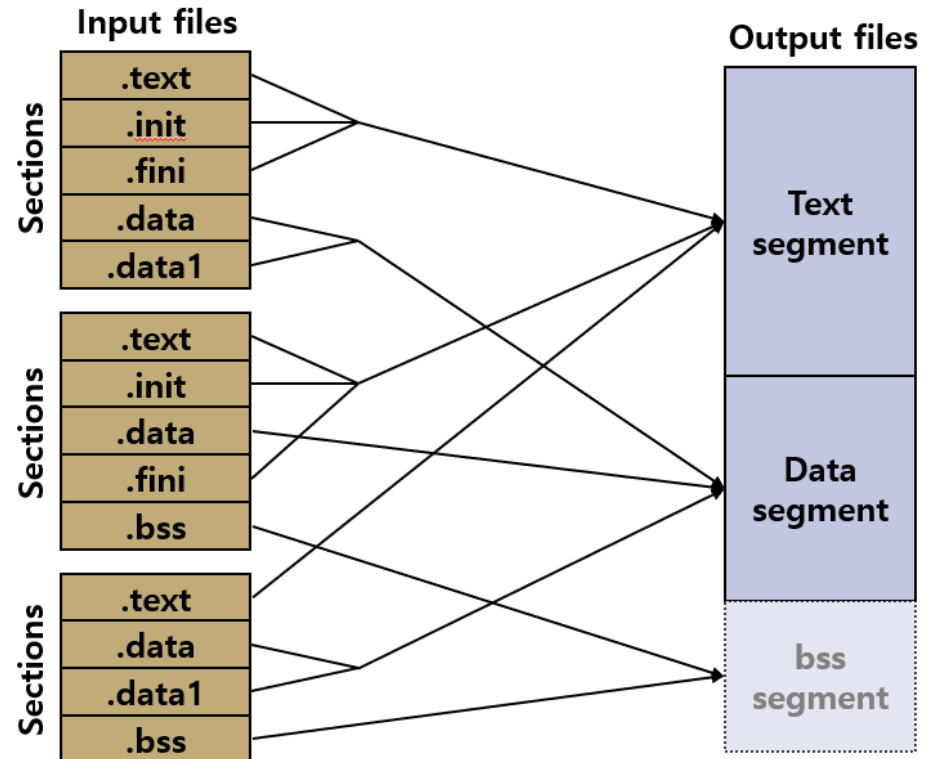
Producing an Object Module

- Assembler (or compiler) translates program into machine instructions

- Provides information for building a complete program from the pieces
  - Header: described contents of object module
  - Text segment: translated instructions
  - Static data segment: data allocated for the life of the program
  - Relocation info: for contents that depend on absolute location of loaded program
  - Symbol table: global definitions and external refs
  - Debug info: for associating with source code

Linking Object Modules

- Produces an executable image
  1. Merges segments
  2. Resolve labels (determine their addresses)
  3. Patch location-dependent and external refs

- Could leave location dependencies for fixing by a relocating loader
  - But with virtual memory, no need to do this
  - Program can be loaded into absolute location in virtual memory space

ELF File Format & Linking

ELF Header
segment table
.text
.data
.rodata
.bss
.sym
.rel.text
.rel.data
.line
.debug
.strtab
section table

Input files

Sections
.text
.init
.fini
.data
.data1

Sections
.text
.init
.data
.fini
.bss

Sections
.text
.data
.data1
.bss

Output files

Text segment

Data segment

bss segment

Object file header			
	Name	Procedure A	
	Text size	100_{hex}	
	Data size	20_{hex}	
Text segment	Address	Instruction	
	0	LDUR X0, [X27,#0]	
	4	BL 0	
	. . .	. . .	
Data segment	0	(X)	
	. . .	. . .	
Relocation information	Address	Instruction type	Dependency
	0	LDUR	X
	4	BL	B
Symbol table	Label	Address	
	X	–	
	B	–	
	Name	Procedure B	
	Text size	200_{hex}	
	Data size	30_{hex}	
Text segment	Address	Instruction	
	0	STUR X1, [X27,#0]	
	4	BL 0	
	. . .	. . .	
Data segment	0	(Y)	
	. . .	. . .	
Relocation information	Address	Instruction type	Dependency
	0	STUR	Y
	4	BL	A
Symbol table	Label	Address	
	Y	–	
	A	–	

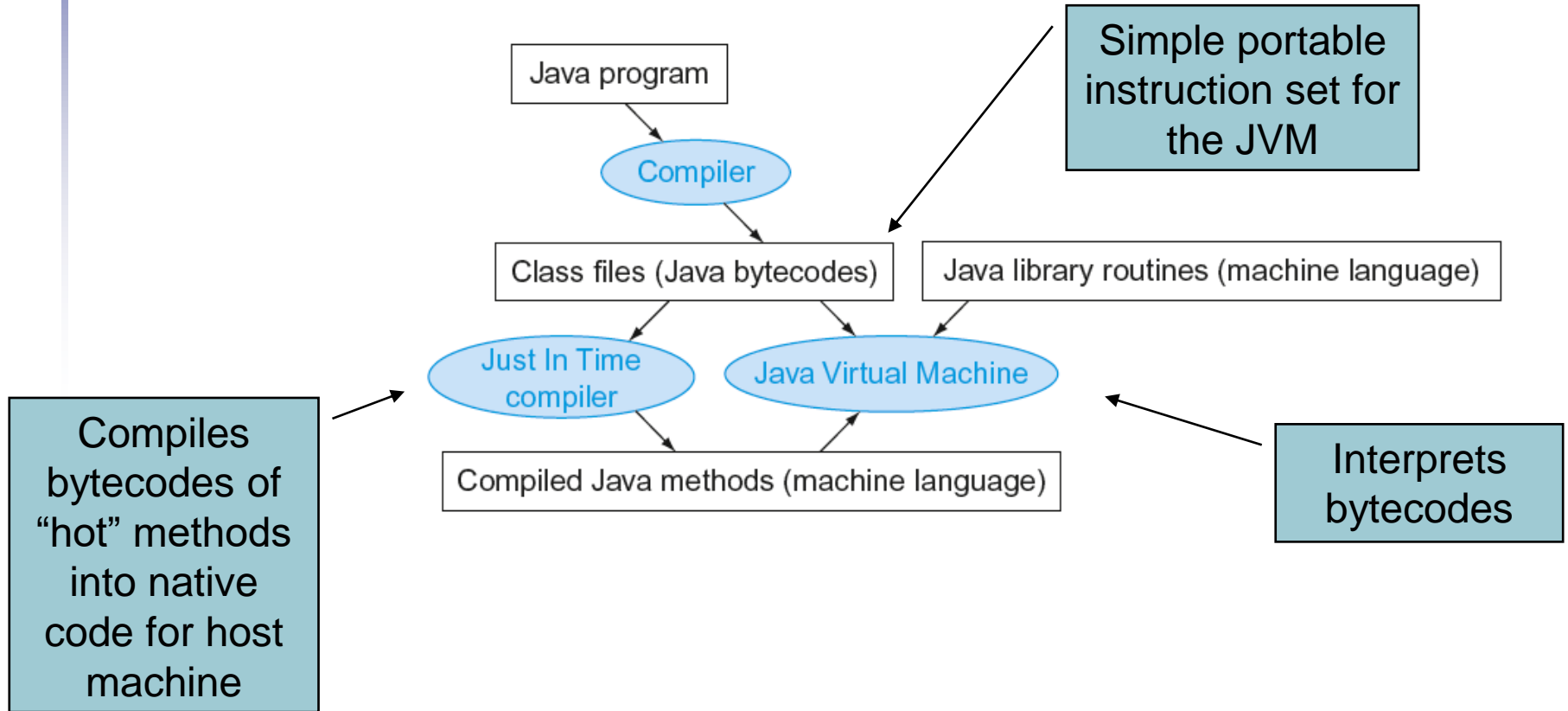| Executable file header		
	Text size	300$_{hex}$
	Data size	50$_{hex}$
Text segment	Address	Instruction
	0000 0000 0040 0000$_{hex}$	LDUR X0, [X27,#0$_{hex}$]
	0000 0000 0040 0004$_{hex}$	BL 000 00FC$_{hex}$
	. . .	. . .
	0000 0000 0040 0100$_{hex}$	STUR X1, [X27,#20$_{hex}$]
	0000 0000 0040 0104$_{hex}$	BL 3FF FEFC$_{hex}$
	. . .	. . .
Data segment	Address	
	0000 0000 1000 0000$_{hex}$	(X)
	. . .	. . .
	0000 0000 1000 0020$_{hex}$	(Y)
	. . .	. . .

Loading a Program

- Load from image file on disk into memory
    1. Read header to determine segment sizes
    2. Create virtual address space
    3. Copy text and initialized data into memory
        - Or set page table entries so they can be faulted in
    4. Set up arguments on stack
    5. Initialize registers (including SP, FP)
    6. Jump to startup routine
        - Copies arguments to X0, … and calls main
        - When main returns, do exit syscall

Dynamic Linking

- Only link/load library procedure when it is called
  - Requires procedure code to be relocatable
  - Avoids image bloat caused by static linking of all (transitively) referenced libraries
  - Automatically picks up new library versions

Starting Java Applications

Java program

↓

Compiler

↓

Class files (Java bytecodes)

Java library routines (machine language)

Simple portable instruction set for the JVM

Just In Time compiler

Java Virtual Machine

Compiled Java methods (machine language)

Compiles bytecodes of "hot" methods into native code for host machine

Interprets bytecodes

C Sort Example

- Illustrates use of assembly instructions for a C bubble sort function

- Swap procedure (leaf)

```
void swap(long long int v[],
long long int 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

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

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
    SUBI SP,SP,#40                // restore stack pointer
```

|				
Saving registers				
	sort:	SUBI	SP,SP,#40	// make room on stack for 5 registers
		STUR	X30,[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
Procedure body				
Move parameters		MOV	X21, X0	// copy parameter X0 into X21
		MOV	X22, X1	// copy parameter X1 into X22
Outer loop		MOV	X19, XZR	// i = 0
	for1tst:CMP		X19, X22	// compare X19 to X22(i to n)
		B.GE	exit1	// go to exit1 if X19 ≥ X1 (i≥n)
Inner loop		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, X21, 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
Pass parameters and call		MOV	X0, X21	// first swap parameter is v
		MOV	X1, X20	// second swap parameter is j
		BL	swap	
Inner loop		SUBI	X20, X20, #1	// j -= 1
		B	for2tst	// branch to test of inner loop
Outer loop	exit2:	ADDI	X19, X19, #1	// i += 1
		B	for1tst	// branch to test of outer loop
Restoring 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
Procedure return				
		BR	LR	// return to calling routine

그림 2.28 그림 2.27의 sort 프로시저의 LEGv8 어셈블리 코드.

Effect of Compiler Optimization

Compiled with gcc for Pentium 4 under Linux
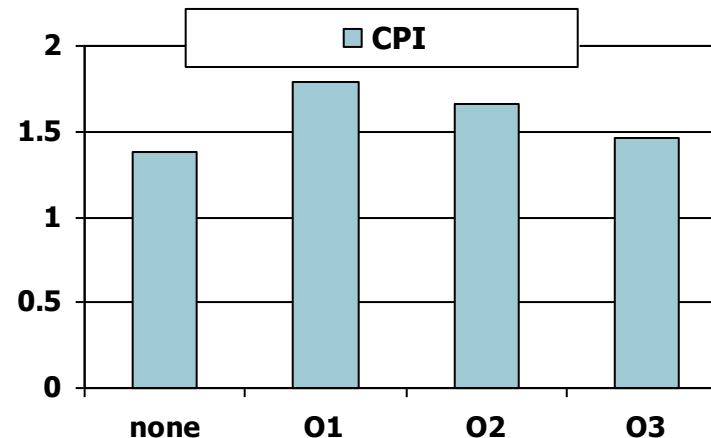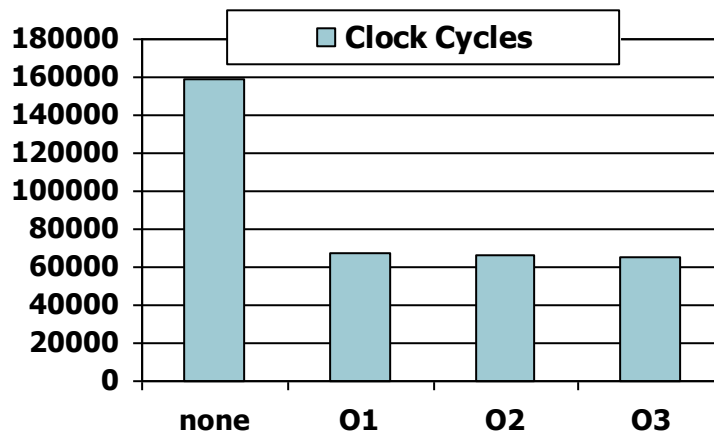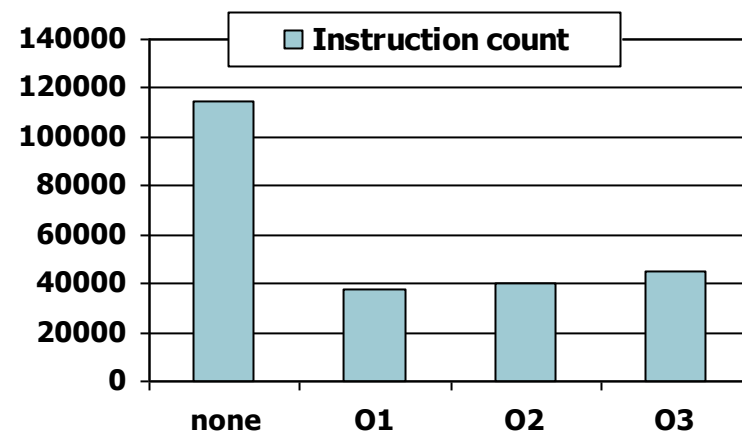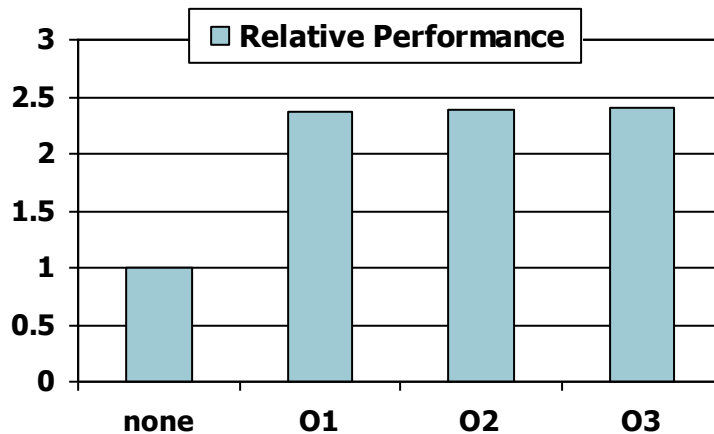
Effect of Language and Algorithm
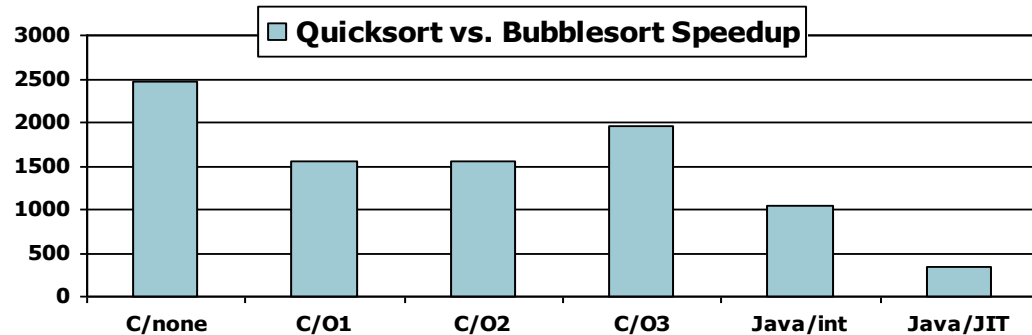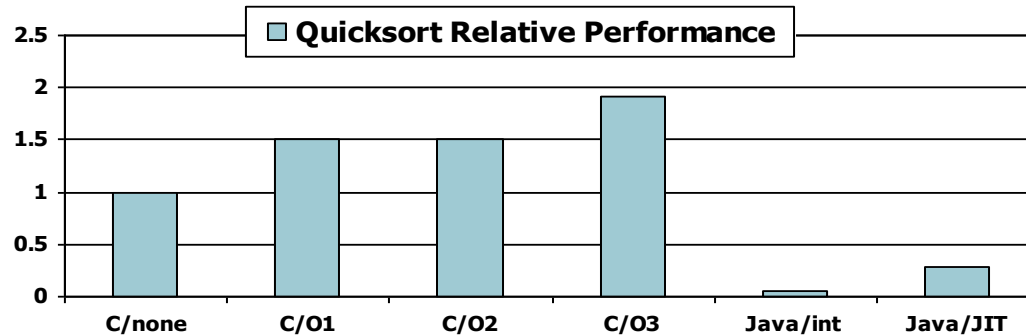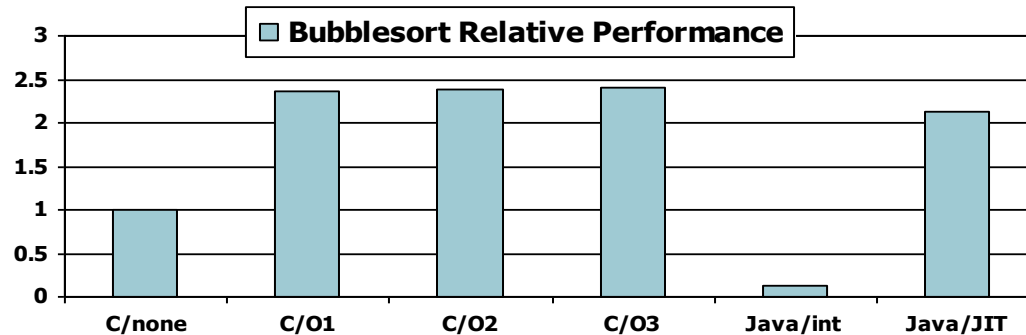
Lessons Learnt

- 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!

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(int array[], int size) {
  int i;
  for (i = 0; i < size; i += 1)
    array[i] = 0;
}
```

```
clear2(int *array, int size) {
  int *p;
  for (p = &array[0]; p < &array[size];
       p = p + 1)
    *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,0[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

Fallacies

- Powerful instruction $\Rightarrow$ 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 $\Rightarrow$ more errors and less productivity

Fallacies

- Backward compatibility $\Rightarrow$ instruction set doesn't change
  - But they do accrete more instructions
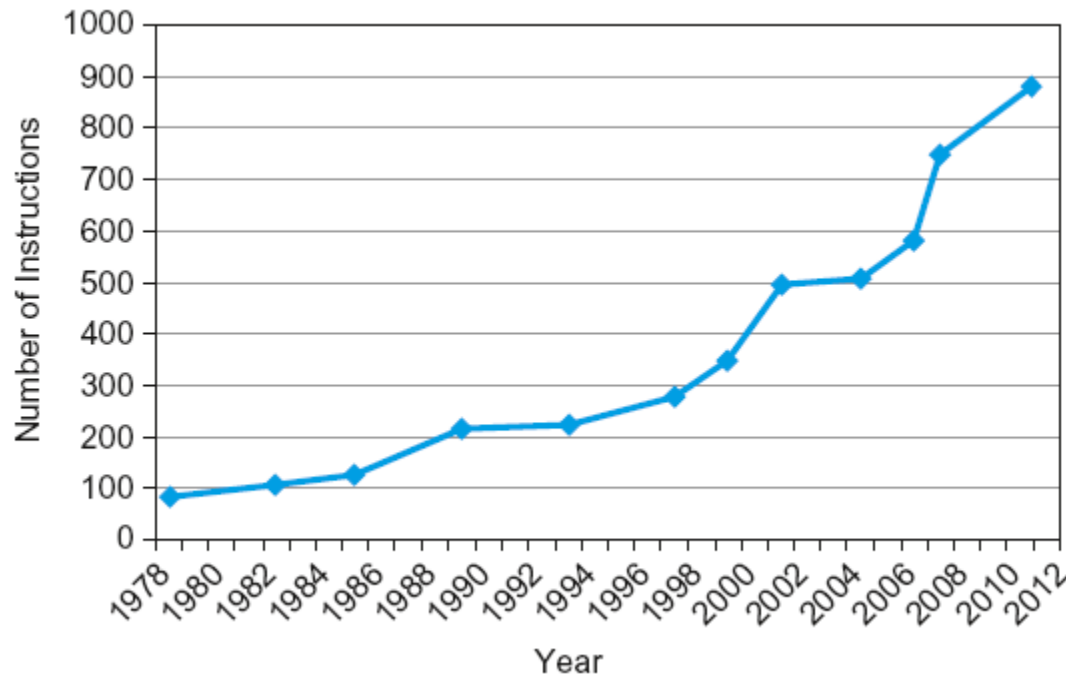
x86 instruction set

Pitfalls

- Sequential words are not at sequential addresses
  - Increment by 4, not by 1!
- Keeping a pointer to an automatic variable after procedure returns
  - e.g., passing pointer back via an argument
  - Pointer becomes invalid when stack popped

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
- LEGv8: typical of RISC ISAs
    - c.f. x86