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
  
  \[
  \text{ADD } a, b, c \quad \text{// } a \text{ gets } b + c
  \]

- 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 $32 \times 64$-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:
  \[ f = (g + h) - (i + j); \]
  - \( f, \ldots, j \) in X19, X20, ..., X23

- Compiled LEGv8 code:
  ```assembly
  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:**
  
  \[
  \]
  
  - `h` in X21, base address of `A` in X22

- **Compiled LEGv8 code:**
  
  - Index 8 requires offset of 64
  
  ```plaintext
  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
  \[ \text{ADDI X22, X22, #4} \]

- 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

\[ x = x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \ldots + x_12^1 + x_02^0 \]

- Range: 0 to +2^n – 1

- Example

\[
\begin{align*}
0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 1011 \_2 &= 0 + \ldots + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 \\
&= 0 + \ldots + 8 + 0 + 2 + 1 = 11_{10}
\end{align*}
\]

- Using 32 bits

- 0 to +4,294,967,295
2s-Complement Signed Integers

- Given an n-bit number

\[ x = -x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \cdots + x_12^1 + x_02^0 \]

- Range: \(-2^{n-1} \text{ to } +2^{n-1} - 1\)

- Example
  - \(1111 \ 1111 \ 1111 \ 1111 \ 1111 \ 1111 \ 1111 \ 1100_2\)
  - \(-1 \times 2^{31} + 1 \times 2^{30} + \cdots + 1 \times 2^2 + 0 \times 2^1 + 0 \times 2^0\)
  - \(-2,147,483,648 + 2,147,483,644 = -4_{10}\)

- Using 32 bits
  - \(-2,147,483,648 \text{ to } +2,147,483,647\)
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

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

- 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

<table>
<thead>
<tr>
<th></th>
<th>0000</th>
<th>0100</th>
<th>0101</th>
<th>0110</th>
<th>0111</th>
<th>1000</th>
<th>1001</th>
<th>1010</th>
<th>1011</th>
<th>1100</th>
<th>1101</th>
<th>1110</th>
<th>1111</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000</td>
<td>4</td>
<td>0100</td>
<td>8</td>
<td>1000</td>
<td>c</td>
<td>1100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0001</td>
<td>5</td>
<td>0101</td>
<td>9</td>
<td>1001</td>
<td>d</td>
<td>1101</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>0010</td>
<td>6</td>
<td>0110</td>
<td>a</td>
<td>1010</td>
<td>e</td>
<td>1110</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0011</td>
<td>7</td>
<td>0111</td>
<td>b</td>
<td>1011</td>
<td>f</td>
<td>1111</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **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 \(X9, X20, X21\)

<table>
<thead>
<tr>
<th>opcode</th>
<th>Rm</th>
<th>shamt</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>11 bits</td>
<td>5 bits</td>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
</tr>
</tbody>
</table>

Here is the R-format example:

\[
\begin{array}{cccccc}
1112_{\text{ten}} & 21_{\text{ten}} & 0_{\text{ten}} & 20_{\text{ten}} & 9_{\text{ten}} \\
10001011000_{\text{two}} & 10101_{\text{two}} & 000000_{\text{two}} & 10100_{\text{two}} & 01001_{\text{two}} \\
\end{array}
\]

\[
\begin{align*}
1000 & 1011 & 0001 & 0101 & 0000 & 0010 & 1000 & 1001_{\text{two}} = \\
& & & & & & & 8B150289_{16}
\end{align*}
\]
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
Logical Operations

Instructions for bitwise manipulation

<table>
<thead>
<tr>
<th>Operation</th>
<th>C</th>
<th>Java</th>
<th>LEGv8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shift left</td>
<td>&lt;&lt;</td>
<td>&lt;&lt;</td>
<td>LSL</td>
</tr>
<tr>
<td>Shift right</td>
<td>&gt;&gt;</td>
<td>&gt;&gt;&gt;</td>
<td>LSR</td>
</tr>
<tr>
<td>Bit-by-bit AND</td>
<td>&amp;</td>
<td>&amp;</td>
<td>AND, ANDI</td>
</tr>
<tr>
<td>Bit-by-bit OR</td>
<td></td>
<td></td>
<td>OR, ORI</td>
</tr>
<tr>
<td>Bit-by-bit NOT</td>
<td>~</td>
<td>~</td>
<td>EOR, EORI</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>opcode</th>
<th>Rm</th>
<th>shamt</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>11 bits</td>
<td>5 bits</td>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
</tr>
</tbody>
</table>

- **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**

<table>
<thead>
<tr>
<th></th>
<th>X10</th>
<th>X11</th>
<th>X9</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>000000000 00000000 00000000 00000000 00000000 00000000 00001101 11000000</td>
<td>00000000 00000000 00000000 00000000 00000000 00000000 00001110 00000000</td>
<td>00000000 00000000 00000000 00000000 00000000 00000000 00001100 00000000</td>
</tr>
</tbody>
</table>
OR Operations

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

OR \(x_9, x_{10}, x_{11}\)

\[
\begin{array}{cccccccccccccccc}
X_{10} & 00000000 & 00000000 & 00000000 & 00000000 & 00000000 & 00000000 & 00000000 & 00011011 & 11000000 \\
X_{11} & 00000000 & 00000000 & 00000000 & 00000000 & 00000000 & 00000000 & 00111100 & 00000000 \\
X_{9} & 00000000 & 00000000 & 00000000 & 00000000 & 00000000 & 00000000 & 00111101 & 11000000
\end{array}
\]
### EOR Operations

- Differencing operation
  - Set some bits to 1, leave others unchanged

\[
\text{EOR } x_9, x_{10}, x_{12} \quad \text{// NOT operation}
\]

<table>
<thead>
<tr>
<th>X10</th>
<th>00000000 00000000 00000000 00000000 00000000 00000000 00001101 11000000</th>
</tr>
</thead>
<tbody>
<tr>
<td>X12</td>
<td>11111111 11111111 11111111 11111111 11111111 11111111 11111111 11111111</td>
</tr>
<tr>
<td>X9</td>
<td>11111111 11111111 11111111 11111111 11111111 11111111 11100010 00111111</td>
</tr>
</tbody>
</table>
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:**
  
  ```c
  if (i==j) f = g+h;
  else f = g-h;
  ```

- **f, g, ... in X22, X23, ...**

- **Compiled LEGv8 code:**
  ```asm
  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:
  ```c
  while (save[i] == k) i += 1;
  ```
  - i in x22, k in x24, address of save in x25

- Compiled LEGv8 code:
  ```assembly
  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
  - overflow (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

- \(X_{22} = 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\)
- \(X_{23} = 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001\)
- \(X_{22} < X_{23}\) # signed
  - \(-1 < +1\)
- \(X_{22} > X_{23}\) # 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:

```c
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
  - Restore X10, X9, X19 from stack
  - Return to caller

Chapter 2 — Instructions: Language of the Computer — 40
Local Data on the Stack

Chapter 2 — Instructions: Language of the Computer — 41
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:**
  ```c
  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:**

```assembly
fact:
  SUBI SP,SP,#16
  STUR LR,[SP,#8]
  STUR X0,[SP,#0]
  SUBIS XZR,X0,#1
  B.GE L1
  ADDI X1,XZR,#1
  ADDI SP,SP,#16
  BR LR
L1:  SUBI X0,X0,#1
    BL fact
    LDUR X0,[SP,#0]
    LDUR LR,[SP,#8]
    ADDI SP,SP,#16
    MUL X1,X0,X1
    BR LR
```

- Save return address and n on stack
- compare n and 1
  - if n >= 1, go to L1
  - Else, set return value to 1
- Pop stack, don’t bother restoring values
- Return
- n = n - 1
  - call fact(n-1)
- Restore caller’s n
- Restore caller’s return address
- Pop stack
- return n * fact(n-1)
- return

**Chapter 2 — Instructions: Language of the Computer — 45**
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
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 32 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
  ```c
  void strcpy (char x[], char y[])
  {
    size_t i;
    i = 0;
    while ((x[i]=y[i])!='\0')
      i += 1;
  }
  ```
String Copy Example

- **LEGv8 code:**
  
  ```asm
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
  ```

Chapter 2 — Instructions: Language of the Computer — 50
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 0000 0000 0000 0000 0000 0000 1111 1111 1111 1111
```

MOVK X9,255,LSL 0

```
0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 1111 1111 1111 1111
```
Branch Addressing

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

<table>
<thead>
<tr>
<th>5</th>
<th>$10000_{ten}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>26 bits</td>
</tr>
</tbody>
</table>

- **CB-type**
  - CBNZ X19, Exit // go to Exit if X19 \(!=\ 0\)

<table>
<thead>
<tr>
<th>181</th>
<th>Exit</th>
<th>19</th>
</tr>
</thead>
<tbody>
<tr>
<td>8 bits</td>
<td>19 bits</td>
<td>5 bits</td>
</tr>
</tbody>
</table>

- Both addresses are PC-relative
  - Address = PC + offset (from instruction)
1. Immediate addressing
   \[
   \begin{array}{cccc}
   \text{op} & \text{rs} & \text{rt} & \text{Immediate} \\
   \end{array}
   \]

2. Register addressing
   \[
   \begin{array}{cccc}
   \text{op} & \text{Rm} & \ldots & \text{Rn} & \text{Rd} \\
   \end{array}
   \]

3. Base addressing
   \[
   \begin{array}{cccc}
   \text{op} & \text{Address} & \text{op} & \text{Rn} & \text{Rt} \\
   \end{array}
   \]

4. PC-relative addressing
   \[
   \begin{array}{cc}
   \text{op} & \text{Address} & \text{Rt} \\
   \end{array}
   \]

Chapter 2 — Instructions: Language of the Computer — 53
# LEGv8 Encoding Summary

## Chapter 2 — Instructions: Language of the Computer — 54

<table>
<thead>
<tr>
<th>Name</th>
<th>Fields</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Field size</td>
<td>6 to 11 bits</td>
<td>5 to 10 bits</td>
</tr>
<tr>
<td>R-format</td>
<td>R</td>
<td>opcode</td>
</tr>
<tr>
<td>I-format</td>
<td>I</td>
<td>opcode</td>
</tr>
<tr>
<td>D-format</td>
<td>D</td>
<td>opcode</td>
</tr>
<tr>
<td>B-format</td>
<td>B</td>
<td>opcode</td>
</tr>
<tr>
<td>CB-format</td>
<td>CB</td>
<td>opcode</td>
</tr>
<tr>
<td>IW-format</td>
<td>IW</td>
<td>opcode</td>
</tr>
</tbody>
</table>
Synchronization

- Two processors sharing an area of memory
  - P1 writes, then P2 reads
  - Data race if P1 and P2 don’t synchronize
    - Result depends on 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
Many compilers produce object modules directly.

Diagram:
- C program
  - Compiler
    - Assembly language program
      - Assembler
        - Object: Machine language module
        - Object: Library routine (machine language)
          - Linker
            - Executable: Machine language program
              - Loader
                - Memory

Static linking
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
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
Lazy Linkage

Indirection table

Stub: Loads routine ID, Jump to linker/loader

Linker/loader code

Dynamically mapped code

(a) First call to DLL routine
(b) Subsequent calls to DLL routine
Starting Java Applications

Simple portable instruction set for the JVM

Compiles bytecodes of “hot” methods into native code for host machine
C Sort Example

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

```c
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

```assembly
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)

```c
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 \geq X1 (i \geq 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
Effect of Compiler Optimization

Compiled with gcc for Pentium 4 under Linux

- Relative Performance
- Instruction count
- Clock Cycles
- CPI

Chapter 2 — Instructions: Language of the Computer — 71
Effect of Language and Algorithm

Bubblesort Relative Performance

Quicksort Relative Performance

Quicksort vs. Bubblesort Speedup
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;
}

Example: Clearing an Array

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
loop2: STUR XZR,0[X9,#0] // array[0]
        LSL X10,X1,#3  // X10 = size * 8
        ADD X11,X0,X10 // X11 = address
        // of array[size]
        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

Chapter 2 — Instructions: Language of the Computer — 75
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
ARM & MIPS Similarities

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

<table>
<thead>
<tr>
<th></th>
<th>ARM</th>
<th>MIPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Date announced</td>
<td>1985</td>
<td>1985</td>
</tr>
<tr>
<td>Instruction size</td>
<td>32 bits</td>
<td>32 bits</td>
</tr>
<tr>
<td>Address space</td>
<td>32-bit flat</td>
<td>32-bit flat</td>
</tr>
<tr>
<td>Data alignment</td>
<td>Aligned</td>
<td>Aligned</td>
</tr>
<tr>
<td>Data addressing modes</td>
<td>9</td>
<td>3</td>
</tr>
<tr>
<td>Registers</td>
<td>$15 \times 32$-bit</td>
<td>$31 \times 32$-bit</td>
</tr>
<tr>
<td>Input/output</td>
<td>Memory mapped</td>
<td>Memory mapped</td>
</tr>
</tbody>
</table>
## Instruction Encoding

**LEGv8**
- **Register-register**
  - Opcode: 31
  - Register: 2120
  - Constant: 1615
  - Rm: 10
  - Constant: 5
  - Rd: 4

**MIPS**
- **Register-register**
  - Opcode: 31
  - Register: 2625
  - Constant: 2120
  - Rs: 16
  - Constant: 1110
  - Rd: 6

**ARMv7**
- **Register-register**
  - Opcode: 31
  - Register: 2827
  - Constant: 2019
  - Rs: 16
  - Constant: 1211

**Data transfer**
- **LEGv8**
  - Opcode: 31
  - Register: 2423
  - Constant: 1615

**MIPS**
- **Register-register**
  - Opcode: 31
  - Register: 2625
  - Constant: 2120
  - Rs: 16

**ARMv7**
- **Register-register**
  - Opcode: 31
  - Register: 2827
  - Constant: 2423

**Branch**
- **LEGv8**
  - Opcode: 31
  - Register: 2625

**MIPS**
- **Register-register**
  - Opcode: 31
  - Register: 2625

**ARMv7**
- **Register-register**
  - Opcode: 31
  - Register: 2827

### Diagram
- **Legend**
  - Opcode
  - Register
  - Constant

---

Chapter 2 — Instructions: Language of the Computer — 78
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
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
    - 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
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
## Basic x86 Registers

<table>
<thead>
<tr>
<th>Name</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>EAX</td>
<td>GPR 0</td>
</tr>
<tr>
<td>ECX</td>
<td>GPR 1</td>
</tr>
<tr>
<td>EDX</td>
<td>GPR 2</td>
</tr>
<tr>
<td>EBX</td>
<td>GPR 3</td>
</tr>
<tr>
<td>ESP</td>
<td>GPR 4</td>
</tr>
<tr>
<td>EBP</td>
<td>GPR 5</td>
</tr>
<tr>
<td>ESI</td>
<td>GPR 6</td>
</tr>
<tr>
<td>EDI</td>
<td>GPR 7</td>
</tr>
<tr>
<td>CS</td>
<td>Code segment pointer</td>
</tr>
<tr>
<td>SS</td>
<td>Stack segment pointer (top of stack)</td>
</tr>
<tr>
<td>DS</td>
<td>Data segment pointer 0</td>
</tr>
<tr>
<td>ES</td>
<td>Data segment pointer 1</td>
</tr>
<tr>
<td>FS</td>
<td>Data segment pointer 2</td>
</tr>
<tr>
<td>GS</td>
<td>Data segment pointer 3</td>
</tr>
<tr>
<td>EIP</td>
<td>Instruction pointer (PC)</td>
</tr>
<tr>
<td>EFLAGS</td>
<td>Condition codes</td>
</tr>
</tbody>
</table>
Basic x86 Addressing Modes

Two operands per instruction

<table>
<thead>
<tr>
<th>Source/dest operand</th>
<th>Second source operand</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register</td>
<td>Register</td>
</tr>
<tr>
<td>Register</td>
<td>Immediate</td>
</tr>
<tr>
<td>Register</td>
<td>Memory</td>
</tr>
<tr>
<td>Memory</td>
<td>Register</td>
</tr>
<tr>
<td>Memory</td>
<td>Immediate</td>
</tr>
</tbody>
</table>

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
### x86 Instruction Encoding

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

#### a. JE EIP + displacement

<table>
<thead>
<tr>
<th>JE</th>
<th>Condition</th>
<th>Displacement</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>4</td>
<td>8</td>
</tr>
</tbody>
</table>

#### b. CALL

<table>
<thead>
<tr>
<th>CALL</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td></td>
</tr>
</tbody>
</table>

#### c. MOV EBX, [EDI + 45]

<table>
<thead>
<tr>
<th>MOV</th>
<th>d</th>
<th>w</th>
<th>r/m</th>
<th>Postbyte</th>
<th>Displacement</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>1</td>
<td>1</td>
<td>8</td>
<td></td>
<td>8</td>
</tr>
</tbody>
</table>

#### d. PUSH ESI

<table>
<thead>
<tr>
<th>PUSH</th>
<th>Reg</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>3</td>
</tr>
</tbody>
</table>

#### e. ADD EAX, #6765

<table>
<thead>
<tr>
<th>ADD</th>
<th>Reg</th>
<th>Immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>3</td>
<td>1</td>
</tr>
</tbody>
</table>

#### f. TEST EDX, #42

<table>
<thead>
<tr>
<th>TEST</th>
<th>Postbyte</th>
<th>Immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>1</td>
<td>8</td>
</tr>
</tbody>
</table>

---

Chapter 2 — Instructions: Language of the Computer — 84
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
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

![Graph showing the increase in the number of instructions over time (1978-2012), with a linear trend indicating the accretion of instructions over time. The x86 instruction set is mentioned as an example.]
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
Concluding Remarks

Additional ARMv8 features:
- Flexible second operand
- Additional addressing modes
- Conditional instructions (e.g. CSET, CINC)

<table>
<thead>
<tr>
<th>Class</th>
<th>Loads/Stores</th>
<th>Operations</th>
<th>Branches</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>AL</td>
<td>ML</td>
<td>AL</td>
<td>ML</td>
</tr>
<tr>
<td>Integer</td>
<td>49</td>
<td>145</td>
<td>74</td>
<td>105</td>
</tr>
<tr>
<td>Floating Point &amp; Int Mul/Div</td>
<td>0</td>
<td>18</td>
<td>63</td>
<td>156</td>
</tr>
<tr>
<td>SIMD/Vector</td>
<td>16</td>
<td>166</td>
<td>229</td>
<td>371</td>
</tr>
<tr>
<td>System/Special</td>
<td>11</td>
<td>55</td>
<td>52</td>
<td>40</td>
</tr>
<tr>
<td></td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>Total</td>
<td>76</td>
<td>384</td>
<td>418</td>
<td>672</td>
</tr>
</tbody>
</table>

Chapter 2 — Instructions: Language of the Computer — 90