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 ARM Instruction Set

- Used as the example in chapters 2 and 3
- Most popular 32-bit instruction set in the world ([www.arm.com](http://www.arm.com))
- 6.1 Billion shipped in 2010
- Large share of embedded core market
  - Applications include mobile phones, consumer electronics, network/storage equipment, cameras, printers, …

- Typical of many modern RISC ISAs
  - See ARM Assembler instructions
  - Their encoding and instruction cycle timings in appendixes B1,B2 and B3 (CD-ROM)
Set Up for Development

1. Install Ubuntu
   - Use Linux as much as possible
   - If you want to stick to Windows, install Ubuntu in a virtual machine (Oracle Virtual Box)

2. Install cross-compiler
   - $ apt-get install gcc-4.6-arm-linux-gnueabihf libc6-dev-armel-cross

3. Install QEMU virtualizer
   - $ sudo apt-get install qemu-user-static
Hello World in ARM Assembly

```assembly
.data
msg: .asciiz "Hello, ARM!\n"
len = . - msg

.text
.globl _start
_start:
    /* syscall write(int fd, const void *buf, size_t count) */
    mov   %r0, $1   /* fd -> stdout */
    ldr   %r1, =msg /* buf -> msg */
    ldr   %r2, =len /* count -> len(msg) */
    mov   %r7, $4   /* write is syscall #4 */
    swi   $0        /* invoke syscall */

    /* syscall exit(int status) */
    mov   %r0, $0   /* status -> 0 */
    mov   %r7, $1   /* exit is syscall #1 */
    swi   $0        /* invoke syscall */
```
Test Dev. Environment

euireseong@UbuntuVB:~/.Temp$ ls
hello.s
euireseong@UbuntuVB:~/.Temp$ arm-linux-gnueabi-as -o hello.o hello.s

euireseong@UbuntuVB:~/.Temp$ arm-linux-gnueabi-ld -o hello hello.o

euireseong@UbuntuVB:~/.Temp$ file hello
hello: ELF 32-bit LSB executable, ARM, version 1 (SYSV), statically linked, not stripped

euireseong@UbuntuVB:~/.Temp$ ls
hello hello.o hello.s

euireseong@UbuntuVB:~/.Temp$ ./hello
Hello, ARM!
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 ARM code:**

  ```
  ADD t0, g, h ; temp t0 = g + h
  ADD t1, i, j ; temp t1 = i + j
  SUB f, t0, t1 ; f = t0 - t1
  ```
Register Operands

- Arithmetic instructions use register operands
- ARM has a $16 \times 32$-bit register file
  - Use for frequently accessed data
  - Registers numbered 0 to 15 (r0 to r15)
  - 32-bit data called a “word”
- **Design Principle 2: Smaller is faster**
  - c.f. main memory: millions of locations
Register Operand Example

- **C code:**
  
  \[ f = (g + h) - (i + j); \]
  
  - \( f, \ldots, j \) in registers \( r0, \ldots, r4 \)
  - \( r5 \) and \( r6 \) are temporary registers

- **Compiled ARM code:**

  
  ```
  ADD r5, r0, r1 ; register r5 contains g + h
  ADD r6, r2, r3 ; register r6 contains i + j
  SUB  r4, r5, r6 ; r4 gets r5-r6
  ```
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

- **Words are aligned in memory**
  - Address must be a multiple of 4

- **ARM is Little Endian**
  - Least-significant byte at least address
  - c.f. Big Endian: Most-significant byte at least address of a word
Memory Operand Example 1

- **C code:**
  
  \[ g = h + A[8]; \]
  
  - \( g \) in \( r1 \), \( h \) in \( r2 \), base address of \( A \) in \( r3 \)
  - \( r5 \) is temporary register

- **Compiled ARM code:**
  
  - Index 8 requires offset of 32
    - 4 bytes per word
  
  \[
  \begin{align*}
  &\text{LDR } r5, [r3, \#32] ; \text{ reg } r5 \text{ gets } A[8] \\
  &\text{ADD } r1, r2, r5 ; \text{ g } = h + A[8]
  \end{align*}
  \]
Memory Operand Example 2

- **C code:**
  \[
  \]
  - *h in r2, base address of A in r3*
  - *r5 is temporary register*

- **Compiled ARM code:**
  - Index 8 requires offset of 32
  
  LDR  r5,[r3,#32] ; reg r5 gets A[8]
  ADD  r5, r2, r5  ; reg r5 gets h+A[8]
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
  
  ADD r3, r3, #4 ; r3 = r3 + 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

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

- **Range**: 0 to \(+2^n - 1\)
- **Example**
  
  \[
  \begin{align*}
  0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1011_2 &= 0 + \cdots + 1\times2^3 + 0\times2^2 + 1\times2^1 + 1\times2^0 \\
  &= 0 + \cdots + 8 + 0 + 2 + 1 = 11_{10}
  \end{align*}
  \]

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

- Example
  - 1111 1111 1111 1111 1111 1111 1111 1100₂
    \[ = -1\times2^{31} + 1\times2^{30} + \cdots + 1\times2^2 + 0\times2^1 + 0\times2^0 \]
    \[ = -2,147,483,648 + 2,147,483,644 = -4_{10} \]

- 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

\[
x + x = 1111\ldots111_2 = -1
\]

\[
x + 1 = -x
\]

- Example: negate +2
  - \( +2 = 0000\ 0000 \ldots\ 0010_2 \)
  - \( -2 = 1111\ 1111 \ldots\ 1101_2 + 1 \\
    = 1111\ 1111 \ldots\ 1110_2 \)
Sign Extension

- Representing a number using more bits
  - Preserve the numeric value

- In ARM instruction set
  - LDRSB, LDRSH: extend loaded byte/halfword

- 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
Representing Instructions

- Instructions are encoded in binary
  - Called machine code
- ARM instructions
  - Encoded as 32-bit instruction words
  - Small number of formats encoding operation code (opcode), register numbers, ...
  - Regularity!
- Register numbers – r0 to r15
### ARM Data Processing (DP) Instructions

#### Instruction fields

- **Opcode**: Basic operation of the instruction
- **Rd**: The destination register operand
- **Rn**: The first register source operand
- **Operand2**: The second source operand
- **I**: Immediate. If I is 0, the second source operand is a register, else the second source is a 12-bit immediate
- **S**: Set Condition Code
- **Cond**: Condition
- **F**: Instruction Format

<table>
<thead>
<tr>
<th>Cond</th>
<th>F</th>
<th>I</th>
<th>Opcode</th>
<th>S</th>
<th>Rn</th>
<th>Rd</th>
<th>Operand2</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 bits</td>
<td>2 bits</td>
<td>1 bits</td>
<td>4 bits</td>
<td>1 bits</td>
<td>4 bits</td>
<td>4 bits</td>
<td>12 bits</td>
</tr>
</tbody>
</table>

![Instruction Format Diagram](image-url)
### DP Instruction Example

<table>
<thead>
<tr>
<th>Cond</th>
<th>F</th>
<th>I</th>
<th>Opcode</th>
<th>S</th>
<th>Rn</th>
<th>Rd</th>
<th>Operand2</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 bits</td>
<td>2 bits</td>
<td>1 bits</td>
<td>4 bits</td>
<td>1 bits</td>
<td>4 bits</td>
<td>4 bits</td>
<td>12 bits</td>
</tr>
</tbody>
</table>

**ADD** \( r5, r1, r2 \); \( r5 = r1 + r2 \)

\[
\begin{array}{cccccccc}
14 & 0 & 0 & 4 & 0 & 1 & 5 & 2 \\
4 bits & 2 bits & 1 bits & 4 bits & 1 bits & 4 bits & 4 bits & 12 bits \\
\end{array}
\]

\[1110000001000000101010000000000010_2\]
Hexadecimal

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

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></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>
</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>
</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>
</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>
</tr>
</tbody>
</table>

- **Example:** eca8 6420
  - 1110 1100 1010 1000 0110 0100 0010 0000
**Design Principle 4:**
**Good design demands good compromises**

- Different formats complicate decoding, but allow 32-bit instructions uniformly
- Keep formats as similar as possible

### ARM Data Transfer (DT) Instruction

<table>
<thead>
<tr>
<th>Cond</th>
<th>F</th>
<th>Opcode</th>
<th>Rn</th>
<th>Rd</th>
<th>Offset12</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 bits</td>
<td>2 bits</td>
<td>6 bits</td>
<td>4 bits</td>
<td>4 bits</td>
<td>12 bits</td>
</tr>
</tbody>
</table>

LDR r5, [r3, #32] ; Temporary reg r5 gets A[8]

```
14  1  24  3  5  32
```

4 bits 2 bits 6 bits 4 bits 4 bits 12 bits
## Logical Operations

### Instructions for bitwise manipulation

<table>
<thead>
<tr>
<th>Operation</th>
<th>C</th>
<th>Java</th>
<th>ARM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shift left</td>
<td><code>&lt;&lt;</code></td>
<td><code>&lt;&lt;</code></td>
<td>LSL</td>
</tr>
<tr>
<td>Shift right</td>
<td><code>&gt;&gt;</code></td>
<td><code>&gt;&gt;&gt;</code></td>
<td>LSR</td>
</tr>
<tr>
<td>Bitwise AND</td>
<td><code>&amp;</code></td>
<td><code>&amp;</code></td>
<td>AND</td>
</tr>
<tr>
<td>Bitwise OR</td>
<td>`</td>
<td>`</td>
<td>`</td>
</tr>
<tr>
<td>Bitwise NOT</td>
<td><code>~</code></td>
<td><code>~</code></td>
<td>MVN</td>
</tr>
</tbody>
</table>

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

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

\[
\text{AND } r5, r1, r2 \ ; \ \text{reg } r5 = \text{reg } r1 \ & \ \text{reg } r2
\]

<table>
<thead>
<tr>
<th>r2</th>
<th>0000 0000 0000 0000 0000 0000 1101 1100 0000</th>
</tr>
</thead>
<tbody>
<tr>
<td>r1</td>
<td>0000 0000 0000 0000 0011 1100 0000 0000</td>
</tr>
<tr>
<td>r5</td>
<td>0000 0000 0000 0000 0000 0000 1100 0000 0000</td>
</tr>
</tbody>
</table>
OR Operations

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

\[
\text{ORR } r5, \, r1, \, r2 \ ; \ \text{reg } r5 = \text{reg } r1 \ | \ \text{reg } r2
\]

<table>
<thead>
<tr>
<th>r2</th>
<th>0000 0000 0000 0000 0000 0000 1101 1100 0000</th>
</tr>
</thead>
<tbody>
<tr>
<td>r1</td>
<td>0000 0000 0000 0000 0011 1100 0000 0000</td>
</tr>
<tr>
<td>r5</td>
<td>0000 0000 0000 0000 0011 1101 1100 0000</td>
</tr>
</tbody>
</table>
NOT Operations

- Useful to invert bits in a word
  - Change 0 to 1, and 1 to 0
- ARM has Move Not (MVN)

```assembly
MVN r5, r1 ;  reg r5 = ~ reg r1
```

<table>
<thead>
<tr>
<th>r1</th>
<th>0000 0000 0000 0000 0011 1100 0000 0000</th>
</tr>
</thead>
<tbody>
<tr>
<td>r5</td>
<td>1111 1111 1111 1111 1100 0011 1111 1111</td>
</tr>
</tbody>
</table>
Shift Operations

- **shift_imm**: how many positions to shift
- **Logical shift left (LSL)**
  - Shift left and fill with 0 bits
  - LSL by \( i \) bits multiplies by \( 2^i \)
    - \( rm, LSL \#<shift\_imm> \)
    - \( ADD r5,r1,r2 \ LSL \#2 \ ; \ r5 = r1 + (r2 \ll 2) \)
- **Logical shift right (LSR)**
  - Shift right and fill with 0 bits
  - LSR by \( i \) bits divides by \( 2^i \) (unsigned only)
    - \( rm, LSR \#<shift\_imm> \)
    - \( MOV r6,r5, LSR \#4 \ ; \ r6 = r5 \gg 4 \)
Shift Operations

- **MOV** r6, r5, LSR r3 ; r6 = r5 >> r3
  - shi = 1
  - Rs instead of shift_imm

- **shift field determines shift operation type**

<table>
<thead>
<tr>
<th>Cond</th>
<th>000</th>
<th>Opcode</th>
<th>S</th>
<th>Rn</th>
<th>Rd</th>
<th>Rs</th>
<th>0</th>
<th>shift</th>
<th>shi</th>
<th>Rm</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 bits</td>
<td>3 bits</td>
<td>4 bits</td>
<td>1 bit</td>
<td>4 bits</td>
<td>4 bits</td>
<td>4 bits</td>
<td>2 bits</td>
<td>1 bit</td>
<td>4 bits</td>
<td></td>
</tr>
</tbody>
</table>
Conditional Operations

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

- **CMP** reg1, reg2
  - **BEQ** L1
    - if (reg1 == reg2) branch to instruction labeled L1;

- **CMP** reg1, reg2
  - **BNE** L1
    - if (reg1 != reg2) branch to instruction labeled L1;

- **B exit** ; go to exit
  - unconditional jump to instruction labeled exit
Compiling If Statements

- **C code:**
  
  ```c
  if (i==j) f = g+h;
  else f = g-h;
  ```
  
  - *f, g, ... in r0, r1,...r4*

- **Compiled ARM code:**

  ```assembly
  CMP r3, r4
  BNE Else ; go to Else if i != j
  
  ADD r1, r1, r2 ; f = g + h (skipped if i != j)
  B exit
  
  Else: SUB r0, r1, r2 ; f = g + h (skipped if i = j)
  Exit:
  
  Assembler calculates addresses
  ```
Compiling Loop Statements

- **C code:**

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

  - `i in r3, k in r5, base address of save in r6`

- **Compiled ARM code:**

  ```
  Loop:  ADD r12, r6, r3, LSL # 2  ; r12 = address of save[i]  
         LDR r0, [r12, #0]  ; Temp reg r0 = save[i]  
         CMP r0, r5  
         BNE Exit  ; go to Exit if save[i] ≠ k  
         ADD r3, r3, #1  ; i = i + 1  
         B Loop  ; go to Loop  
  Exit:
  ```
Branch Instruction format

<table>
<thead>
<tr>
<th>Condition</th>
<th>12</th>
<th>address</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 bits</td>
<td>24 bits</td>
<td></td>
</tr>
</tbody>
</table>

- Encoding of options for Condition field

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>EQ (EQual)</td>
<td>8</td>
<td>HI (unsigned Higher)</td>
</tr>
<tr>
<td>1</td>
<td>NE (Not Equal)</td>
<td>9</td>
<td>LS (unsigned Lower or Same)</td>
</tr>
<tr>
<td>2</td>
<td>HS (unsigned Higher or Same)</td>
<td>10</td>
<td>GE (signed Greater than or Equal)</td>
</tr>
<tr>
<td>3</td>
<td>LO (unsigned LOWer)</td>
<td>11</td>
<td>LT (signed Less Than)</td>
</tr>
<tr>
<td>4</td>
<td>MI (MInus, &lt;0)</td>
<td>12</td>
<td>GT (signed Greater Than)</td>
</tr>
<tr>
<td>5</td>
<td>PL - (PPlus, &gt;=0)</td>
<td>13</td>
<td>LE (signed Less Than or Equal)</td>
</tr>
<tr>
<td>6</td>
<td>VS (oVerflow Set, overflow)</td>
<td>14</td>
<td>AL (Always)</td>
</tr>
<tr>
<td>7</td>
<td>VC (oVerflow Clear, no overflow)</td>
<td>15</td>
<td>NV (reserved)</td>
</tr>
</tbody>
</table>

- Address is translated to the PC-relative address value
  - Current branch instruction address + 8 + (address << 2)
Conditional Execution

- ARM code for executing *if* statement
  - Code on right does not use branch
  - This can help performance of pipelined computers (Chapter 4)

```assembly
CMP r3, r4       ; reg r3 and r4 contain i,j
BNE Else        ; go to Else if i <> j
ADD r0,r1,r2   ; f = g + h
B Exit          ; go to Exit
Else: SUB r0,r1,r2 ; f = g - h
Exit:
```

```
CMP r3,r4
ADDEQ r0,r1,r2 ; f = g+h
SUBNE r0,r1,r2 ; f = g-h
```
## ARM instruction format summary

<table>
<thead>
<tr>
<th>Name</th>
<th>Format</th>
<th>Example</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Field size</td>
<td>4 bits</td>
<td>2 bits</td>
<td>1 bit</td>
</tr>
<tr>
<td>DP format</td>
<td>DP</td>
<td>Cond</td>
<td>F</td>
</tr>
<tr>
<td>DT format</td>
<td>DT</td>
<td>Cond</td>
<td>F</td>
</tr>
<tr>
<td>Field size</td>
<td>4 bits</td>
<td>2 bits</td>
<td>2 bits</td>
</tr>
<tr>
<td>BR format</td>
<td>BR</td>
<td>Cond</td>
<td>F</td>
</tr>
</tbody>
</table>
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
Signed vs. Unsigned

- **Signed comparison:** BGE, BLT, BGT, BLE
- **Unsigned comparison:** BHS, BLO, BHI, BLS
- **Example**
  
  - \( r_0 = 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111 \)
  - \( r_1 = 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001 \)
  - CMP \( r_0, r_1 \)
  - BLO L1 ; unsigned branch
    - Branch not taken since \( 4,294,967,295_{10} > 1_{10} \)
  - BLT L2 ; signed branch
    - Branch taken to L2 since \(-1_{10} < 1_{10}\)
Procedure Calling

Steps required

1. Place parameters in registers
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
# ARM register conventions

<table>
<thead>
<tr>
<th>Name</th>
<th>Register number</th>
<th>Usage</th>
<th>Preserved on call?</th>
</tr>
</thead>
<tbody>
<tr>
<td>a1-a2</td>
<td>0–1</td>
<td>Argument / return result / scratch register</td>
<td>no</td>
</tr>
<tr>
<td>a3-a4</td>
<td>2–3</td>
<td>Argument / scratch register</td>
<td>no</td>
</tr>
<tr>
<td>v1-v8</td>
<td>4–11</td>
<td>Variables for local routine</td>
<td>yes</td>
</tr>
<tr>
<td>ip</td>
<td>12</td>
<td>Intra-procedure-call scratch register</td>
<td>no</td>
</tr>
<tr>
<td>sp</td>
<td>13</td>
<td>Stack pointer</td>
<td>yes</td>
</tr>
<tr>
<td>lr</td>
<td>14</td>
<td>Link Register (Return address)</td>
<td>yes</td>
</tr>
<tr>
<td>pc</td>
<td>15</td>
<td>Program Counter</td>
<td>n.a.</td>
</tr>
</tbody>
</table>
Procedure Call Instructions

- Procedure call: Branch and link
  
  **BL  ProcedureAddress**
  - Address of following instruction put in *lr*
  - Jumps to target address

- Procedure return:
  
  **MOV  pc, 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
int leaf_example (int g, h, i, j)
{
    int f;
    f = (g + h) - (i + j);
    return f;
}
```

- Arguments $g$, $h$, $i$, $j$ in $r0$, $r1$, $r2$, $r3$
- $f$ in $r4$ (hence, need to save $r4$ on stack)
- Result in $r0$
Leaf Procedure Example

- **ARM code:**

  ```
  leaf_example:
  SUB sp, sp, #12
  STR r6, [sp,#8]
  STR r5, [sp,#4]
  STR r4, [sp,#0]
  ADD r5, r0, r1
  ADD r6, r2, r3
  SUB r4, r5, r6
  MOV r0, r4
  LDR r4, [sp,#0]
  LDR r5, [sp,#4]
  LDR r6, [sp,#8]
  ADD sp, sp, #12
  MOV pc, lr
  ```

  - Make room for 3 items
  - Save r4,r5,r6 on stack
  - \( r5 = (g+h), \ r6 = (i+j) \)
  - Result in r4
  - Result moved to return value register r0.
  - Restore r4,r5,r6
  - Return
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 register `r0`
  - Result in register `r0`
Non-Leaf Procedure Example

- ARM code:

```
fact:
    SUB sp,sp,#8     ; Adjust stack for 2 items
    STR lr,[sp,#8]   ; Save return address
    STR r0,[sp,#0]   ; Save argument n
    CMP r0,#1        ; compare n to 1
    BGE L1
    MOV r0,#1        ; if so, result is 1
    ADD sp,sp,#8     ; Pop 2 items from stack
    MOV pc,lr        ; Return to caller
L1:    SUB r0,r0,#1     ; else decrement n
    BL fact          ; Recursive call
    MOV r12,r0       ; Restore original n
    LDR r0,[sp,#0]   ; and return address
    LDR lr,[sp,#0]   ; pop 2 items from stack
    ADD sp,sp,#8     ; Multiply to get result
    MUL r0,r0,r12    ; and return
    MOV pc,lr
```
Local Data on the Stack

- **Local data allocated by callee**
  - e.g., C automatic variables

- **Procedure frame (activation record)**
  - Used by some compilers to manage stack storage

Diagram:

- High address
  - sp
  - Saved argument registers (if any)
  - Saved return address
  - Saved saved registers (if any)
  - Local arrays and structures (if any)

- Low address
  - sp

[Diagram showing stack layout with sp pointers and various sections for data and registers]
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

- Could use bitwise operations
- ARM byte load/store
  - String processing is a common case
    \[
    \text{LDRB } r0, \ [sp, #0] \ ; \text{Read byte from source}
    \]
    \[
    \text{STRB } r0, \ [r10, #0] \ ; \text{Write byte to destination}
    \]
  - Sign extend to 32 bits
    \[
    \text{LDRSB} \ ; \text{Sign extends to fill leftmost 24 bits}
    \]
- ARM halfword load/store
  \[
  \text{LDRH } r0, \ [sp, #0] \ ; \text{Read halfword (16 bits) from source}
  \]
  \[
  \text{STRH } r0, \ [r12, #0] \ ; \text{Write halfword (16 bits) to destination}
  \]
  - Sign extend to 32 bits
    \[
    \text{LDRSH} \ ; \text{Sign extends to fill leftmost 16 bits}
    \]
String Copy Example

- **C code (naïve):**
  - Null-terminated string
  ```c
  void strcpy (char x[], char y[]) {
    int i;
    i = 0;
    while ((x[i]=y[i])!='\0')
      i += 1;
  }
  ```
  - Addresses of \( x, y \) in registers \( r0, r1 \)
  - \( i \) in register \( r4 \)
String Copy Example

- ARM code:

```assembly
strcpy:
  SUB  sp,sp, #4 ; adjust stack for 1 item
  STR  r4,[sp,#0] ; save r4
  MOV  r4,#0 ; i = 0 + 0
L1:  ADD  r2,r4,r1 ; addr of y[i] in r2
    LDRBS r3, [r2, #0] ; r3 = y[i]
    ADD  r12,r4,r0 ; Addr of x[i] in r12
    STRB  r3 [r12, #0] ; x[i] = y[i]
    BEQ  L2 ; exit loop if y[i] == 0
    ADD  r4,r4,#1 ; i = i + 1
    B    L1 ; next iteration of loop
L2:  LDR  r4, [sp,#0] ; restore saved r4
    ADD  sp,sp, #4 ; pop 1 item from stack
    MOV  pc,lr ; return
```
32-bit Constants

- Most constants are small
  - 16-bit immediate is sufficient
- For the occasional 32-bit constant
- Load 32 bit constant to r4

\[
\begin{array}{cccccccccccc}
\text{Cond} & F & I & \text{Opcode} & S & Rn & Rd & \text{rotate-imm} & \text{mm}_8 \\
14 & 0 & 1 & 13 & 0 & 0 & 4 & 8 & 217 \\
4 \text{ bits} & 2 \text{ bits} & 1 \text{ bit} & 4 \text{ bits} & 1 \text{ bit} & 4 \text{ bits} & 4 \text{ bits} & 4 \text{ bits} & 8 \text{ bits}
\end{array}
\]

The 8 non-zerobits (1101 1101, 217) of the constant is rotated by 16 bits and MOV instruction (opcode -13) loads the 32 bit value.
Addressing Mode Summary (1-3 of 12)

1. Immediate: ADD r2, r0, #5

| cond | f | opcode | m | rd | Immediate |

2. Register: ADD r2, r0, r1

| cond | f | opcode | mn | rd | \ldots | rm |

3. Scaled register: ADD r2, r0, r1, LSL #2

| cond | f | opcode | mn | rd | \ldots | rm |

Diagrams showing the flow of data from registers and a shifter.
Addressing Mode Summary (4-6 of 12)

4. PC-relative: BEQ 1000

<table>
<thead>
<tr>
<th>cond</th>
<th>opcode</th>
<th>offset</th>
</tr>
</thead>
</table>

PC +

5. Immediate offset: LDR r2, [r0, #8]

<table>
<thead>
<tr>
<th>cond</th>
<th>f</th>
<th>opcode</th>
<th>m</th>
<th>rd</th>
<th>address</th>
</tr>
</thead>
</table>

register +

Memory

6. Register offset: LDR r2, [r0, r1]

<table>
<thead>
<tr>
<th>cond</th>
<th>f</th>
<th>opcode</th>
<th>m</th>
<th>rd</th>
<th>...</th>
<th>rm</th>
</tr>
</thead>
</table>

register +

register +

Memory
Addressing Mode Summary (7-8 of 12)

7. Scaled register offset: LDR r2, [r0, r1, LSL #2]

   | cond | f | opcode | rm | rd | ... | rm |

   register

   shifter

   +

   register

8. Immediate offset pre-indexed: LDR r2, [r0, #4]!

   | cond | f | opcode | rm | rd | address |

   register

   +

   register

   Memory

   Byte  Half  Word
Addressing Mode Summary (9-10 of 12)

9. Immediate offset post-indexed: LDR r2, [r0], #4

10. Register offset pre-indexed: LDR r2, [r0, r1]!
11. Scaled register offset pre-indexed: LDR r2, [r0, r1, LSL #2]!

12. Register offset post-indexed: LDR r2, [r0], r1
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

- **Single instruction for atomic exchange or swap**
  - Atomic swap of register ↔ memory
  - ARM instruction: SWP
Stored Program Computers

- 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
Many compilers produce object modules directly.

Static linking

C program

Compiler

Assembly language program

Assembler

Object: Machine language module

Object: Library routine (machine language)

Linker

Executable: Machine language program

Loader

Memory
Assembler Pseudoinstructions

- Most assembler instructions represent machine instructions one-to-one
- Pseudoinstructions: figments of the assembler’s imagination

LDR r0, #constant

- The assembler determines which instructions to use to create the constant in the most efficient way
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
  6. Jump to startup routine
     - Copies arguments to r0, … and calls main
     - When main returns, startup terminates with exit system call
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

Java program

Compiler

Class files (Java bytecodes)

Interprets bytecodes

Java Virtual Machine

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

Compiled Java methods (machine language)

Just In Time compiler

Java library routines (machine language)

Simple portable instruction set for the JVM

Interprets bytecodes

Compiler

Class files (Java bytecodes)

Java Virtual Machine

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

Compiled Java methods (machine language)

Just In Time compiler

Java library routines (machine language)

Simple portable instruction set for the JVM
C Sort Example

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

```c
void swap(int v[], int k)
{
    int temp;
    temp = v[k];
    v[k] = v[k+1];
    v[k+1] = temp;
}
```
The Procedure Swap

Assembler directive

- \( v \) RN 0 ; 1st argument address of \( v \)
- \( k \) RN 1 ; 2nd argument index \( k \)
- temp RN 2 ; local variable
- temp2 RN 3 ; temporary variable for \( v[k+1] \)
- vkAddr RN 12 ; to hold address of \( v[k] \)

Procedure body

\[
\text{swap: } \text{ADD} \quad \text{vkAddr, v, k, LSL \#2} \quad ; \text{reg } \text{vkAddr} = v + (k \times 4) \\
\text{LDR} \quad \text{temp, [vkAddr, \#0]} \quad ; \text{temp (temp) = v[k]} \\
\text{LDR} \quad \text{temp2, [vkAddr, \#4]} \quad ; \text{temp2 = v[k + 1]} \\
\text{STR} \quad \text{temp2, [vkAddr, \#0]} \quad ; \text{v[k] = temp2} \\
\text{STR} \quad \text{temp, [vkAddr, \#4]} \quad ; \text{v[k+1] = temp}
\]

Procedure return

\[
\text{MOV} \quad \text{pc, lr} \quad ; \text{return to calling routine}
\]
The Sort Procedure in C

- Non-leaf (calls swap)

```c
void sort (int v[], int n)
{
    int 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);
        }
    }
}
```
Register allocation and saving registers for sort

Register allocation

<table>
<thead>
<tr>
<th>Variable</th>
<th>Register</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>v</td>
<td>RN 0</td>
<td>1st argument address of v</td>
</tr>
<tr>
<td>n</td>
<td>RN 1</td>
<td>2nd argument index n</td>
</tr>
<tr>
<td>i</td>
<td>RN 2</td>
<td>local variable i</td>
</tr>
<tr>
<td>j</td>
<td>RN 3</td>
<td>local variable j</td>
</tr>
<tr>
<td>vjAddr</td>
<td>RN 12</td>
<td>to hold address of v[j]</td>
</tr>
<tr>
<td>vj</td>
<td>RN 4</td>
<td>to hold a copy of v[j]</td>
</tr>
<tr>
<td>vj1</td>
<td>RN 5</td>
<td>to hold a copy of v[j+1]</td>
</tr>
<tr>
<td>vcopy</td>
<td>RN 6</td>
<td>to hold a copy of v</td>
</tr>
<tr>
<td>ncopy</td>
<td>RN 7</td>
<td>to hold a copy of n</td>
</tr>
</tbody>
</table>

Saving registers

sort:  

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>SUB sp,sp,#20</td>
<td>make room on stack for 5 registers</td>
</tr>
<tr>
<td>STR lr, [sp,#16]</td>
<td>save lr on stack</td>
</tr>
<tr>
<td>STR ncopy, [sp,#12]</td>
<td>save ncopy on stack</td>
</tr>
<tr>
<td>STR vcopy, [sp,#8]</td>
<td>save vcopy on stack</td>
</tr>
<tr>
<td>STR j, [sp,#4]</td>
<td>save j on stack</td>
</tr>
<tr>
<td>STR i, [sp,#0]</td>
<td>save i on stack</td>
</tr>
</tbody>
</table>


## Procedure body - sort

| Move parameters | MOV vcopy, v : copy parameter v into vcopy (save r0)  
|                 | MOV ncopy, n : copy parameter n into ncopy (save r1) |
| Outer loop      | MOV i, #0 ; i = 0  
|                 | CMP i, n ; if i ≥ n  
|                 | BGE exit1 ; go to exit1 if i ≥ n |
| Inner loop      | SUB j, i, #1 ; j = i - 1  
|                 | CMP j, #0 ; if j < 0  
|                 | BLT exit2 ; go to exit2 if j < 0  
|                 | ADD vjAddr, v, j, LSL #2 ; reg vjAddr = v + (j * 4)  
|                 | LDR vj, [vjAddr,#0] ; reg vj = v[j]  
|                 | LDR vj1, [vjAddr,#4] ; reg vj1 = v[j + 1]  
|                 | CMP vj, vj1 ; if vj ≤ vj1  
|                 | BLE exit2 ; go to exit2 if vj ≤ vj1 |
| Pass parameters and call | MOV r0, vcopy ; first swap parameter is v  
|                 | MOV r1, j ; second swap parameter is j  
|                 | BL swap ; swap code shown in Figure 2.23 |
| Inner loop      | SUB j, j, #1 ; j = j - 1  
|                 | B for2tst ; branch to test of inner loop |
| Outer loop      | exit2: ADD i, i, #1 ; i = i + 1  
|                 | B for1tst ; branch to test of outer loop |
Restoring registers and return - sort

exit1:  

LDR    i, [sp, #0]  ; restore i from stack  
LDR    j, [sp, #4]  ; restore j from stack  
LDR    vcopy, [sp, #8]  ; restore vcopy from stack  
LDR    ncopy, [sp, #12]  ; restore ncopy from stack  
LDR    lr, [sp, #16]  ; restore lr from stack  
ADD    sp,sp,#20  ; restore stack pointer

---

Procedure return

MOV  pc, lr  ; return to calling routine
Effect of Compiler Optimization

Compiled with gcc for Pentium 4 under Linux

- **Relative Performance**
  - none
  - O1
  - O2
  - O3

- **Instruction count**
  - none
  - O1
  - O2
  - O3

- **Clock Cycles**
  - none
  - O1
  - O2
  - O3

- **CPI**
  - none
  - O1
  - O2
  - O3
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 and Array

```c
example: Clearing and 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;
}

```assembly

```MOV  i,#0 ; i = 0
MOV  zero,#0 ;zero = 0

loop1: STR  zero, [array,i, LSL #2]; array[i] = 0
ADD  i,i,#1 ; i = i + 1
CMP  i,size ; i < size
BLT  loop1 ; if (i < size) go to loop1
``` clear2(int *array, int size) 
{  
    int *p;
    for (p = &array[0]; p < &array[size]; p = p + 1)
        *p = 0;
}

```assembly

```MOV  p,array ; p = & array[0]
MOV  zero,#0 ;zero = 0
ADD  arraySize,array,size,LSL #2
        ; arraySize = &array[size]

loop2: STR  zero,[p],#4 ; Memory[p] = 0,p = p + 4
CMP  p,arraySize; p<&array[size]
BLT  loop2 ; if () 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
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>
Conditional branches
- Setting a register depending on arithmetic result
- BEQ and BNE with two register values

Lack of conditional execution

$zero register

16-bit immediate field
- Zero extending for unsigned
- Sign extending for signed

32 registers
# Instruction Encoding

<table>
<thead>
<tr>
<th>Category</th>
<th>ARM</th>
<th>MIPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register-register</td>
<td></td>
<td></td>
</tr>
<tr>
<td>31 28 27 20 19 16 15 12 11 4 3 0</td>
<td>31 26 25 21 20 16 15 11 10 6 5 0</td>
<td></td>
</tr>
<tr>
<td>Data transfer</td>
<td></td>
<td></td>
</tr>
<tr>
<td>31 28 27 20 19 16 15 12 11 0</td>
<td>31 26 25 21 20 16 15 0</td>
<td></td>
</tr>
<tr>
<td>Branch</td>
<td></td>
<td></td>
</tr>
<tr>
<td>31 28 27 24 23 20 16 15 0</td>
<td>31 26 25 21 20 16 15 0</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction Type</th>
<th>ARM</th>
<th>MIPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Opcode</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Constant</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

## ARM
- `Op`: Opcode
- `Rs`: Register
- `Rd`: Result Register
- `Const`: Constant

## MIPS
- `Op`: Opcode
- `Rs`: Register
- `Rd`: Result Register
- `Const`: Constant
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
- **Pentium Pro (1995), Pentium II (1997)**
  - New microarchitecture (see Colwell, *The Pentium Chronicles*)
- **Pentium III (1999)**
  - Added SSE (Streaming SIMD Extensions) and associated registers
- **Pentium 4 (2001)**
  - New microarchitecture
  - Added SSE2 instructions
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 \(= R_{\text{base}} + \text{displacement} \)
  - Address \(= R_{\text{base}} + 2^{\text{scale}} \times R_{\text{index}} \) (scale = 0, 1, 2, or 3)
  - Address \(= R_{\text{base}} + 2^{\text{scale}} \times R_{\text{index}} + \text{displacement} \)
### x86 Instruction Encoding

#### Variable length encoding

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

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Encoding</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>a. JE EIP + displacement</td>
<td>4 4 8</td>
<td>JE Condition Displacement</td>
</tr>
<tr>
<td>b. CALL</td>
<td>8 32</td>
<td>CALL Offset</td>
</tr>
<tr>
<td>c. MOV EBX, [EDI + 45]</td>
<td>6 1 1 8 8</td>
<td>MOV d w r/m Postbyte Displacement</td>
</tr>
<tr>
<td>d. PUSH ESI</td>
<td>5 3</td>
<td>PUSH Reg</td>
</tr>
<tr>
<td>e. ADD EAX, #6765</td>
<td>4 3 1 32</td>
<td>ADD Reg w Immediate</td>
</tr>
<tr>
<td>f. TEST EDX, #42</td>
<td>7 8 32</td>
<td>TEST w Postbyte Immediate</td>
</tr>
</tbody>
</table>
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 ⇒ higher performance**
  - Fewer instructions required
  - But complex instructions are hard to implement
    - May slow down all instructions, including simple ones
  - Compilers are good at making fast code from simple instructions

- **Use assembly code for high performance**
  - But modern compilers are better at dealing with modern processors
  - More lines of code ⇒ more errors and less productivity
Fallacies

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

[Graph showing the increase in the number of instructions from 1978 to 2008 for the 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

- **ARM: typical of RISC ISAs**
  - c.f. x86
Concluding Remarks

- Measure ARM instruction executions in benchmark programs
  - Consider making the common case fast
  - Consider compromises

<table>
<thead>
<tr>
<th>Instruction class</th>
<th>ARM examples</th>
<th>SPEC2006 Int</th>
<th>SPEC2006 FP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arithmetic</td>
<td>ADD, SUB, MOV</td>
<td>16%</td>
<td>48%</td>
</tr>
<tr>
<td>Data transfer</td>
<td>LDR, STR, LDRB, LDRSB, LDRH, LDRSH, STRB, STRH</td>
<td>35%</td>
<td>36%</td>
</tr>
<tr>
<td>Logical</td>
<td>AND, ORR, MNV, LSL, LSR</td>
<td>12%</td>
<td>4%</td>
</tr>
<tr>
<td>Conditional Branch</td>
<td>B_, CMP</td>
<td>34%</td>
<td>8%</td>
</tr>
<tr>
<td>Jump</td>
<td>B, BL</td>
<td>2%</td>
<td>0%</td>
</tr>
</tbody>
</table>