ISA is the HW/SW interface

SW

ISA

HW

ISA choice determines:

• program size (& memory size)
• complexity of hardware (CPI and f)
• execution time for different applications and domains
• power consumption
• die area (cost)
• Backward compatibility
Stored program concept

- Von Neumann model: Instructions represented in binary numbers, just like data → both in memory

ISA design issues:
- What is the instruction size?
- How is it encoded?
- Where are the operands located? What are their sizes and values?
- Where should the result be stored?
- How to determine the successor instruction?
1. Operand storage choices

- Memory is slow \(\rightarrow\) need fast internal (but smaller) storage
- The international storage is one of the basic differentiation between ISAs.
- For register machines, how many registers are sufficient?
- What are the pros and cons of each method?

\[ C = A + B \]
1. Pros and Cons of different register ISAs

<table>
<thead>
<tr>
<th>Number of memory addresses</th>
<th>Maximum number of operands allowed</th>
<th>Type of architecture</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>3</td>
<td>Load-store</td>
<td>Alpha, ARM, MIPS, PowerPC, SPARC, SuperH, TM32</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>Register-memory</td>
<td>IBM 360/370, Intel 80x86, Motorola 68000, TI TMS320C54x</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>Memory-memory</td>
<td>VAX (also has three-operand formats)</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>Memory-memory</td>
<td>VAX (also has two-operand formats)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Type</th>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register-register (0, 3)</td>
<td>Simple, fixed-length instruction encoding. Simple code generation model. Instructions take similar numbers of clocks to execute (see Appendix C).</td>
<td>Higher instruction count than architectures with memory references in instructions. More instructions and lower instruction density lead to larger programs.</td>
</tr>
<tr>
<td>Register-memory (1, 2)</td>
<td>Data can be accessed without a separate load instruction first. Instruction format tends to be easy to encode and yields good density.</td>
<td>Operands are not equivalent since a source operand in a binary operation is destroyed. Encoding a register number and a memory address in each instruction may restrict the number of registers. Clocks per instruction vary by operand location.</td>
</tr>
<tr>
<td>Memory-memory (2, 2) or (3, 3)</td>
<td>Most compact. Doesn’t waste registers for temporaries.</td>
<td>Large variation in instruction size, especially for three-operand instructions. In addition, large variation in work per instruction. Memory accesses create memory bottleneck. (Not used today.)</td>
</tr>
</tbody>
</table>
2. Memory addressing choices

• Data addressing modes

<table>
<thead>
<tr>
<th>Addressing mode</th>
<th>Example instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register</td>
<td>Add R4, R3</td>
</tr>
<tr>
<td>Immediate</td>
<td>Add R4, #3</td>
</tr>
<tr>
<td>Register Indirect</td>
<td>Add R4, (R1)</td>
</tr>
<tr>
<td>Displacement</td>
<td>Add R4, 100(R1)</td>
</tr>
<tr>
<td>Indexed</td>
<td>Add R3, (R1+R2)</td>
</tr>
<tr>
<td>Direct or absolute</td>
<td>Add R1, (1001)</td>
</tr>
<tr>
<td>Memory indirect</td>
<td>Add R4, @(R1)</td>
</tr>
</tbody>
</table>

What is the impact on instruction size, decoding and execution time?

• Big Endian vs. little Endian

Increasing memory addresses
3. Type and size of operands

- Character
- integer
- single-precision floating point
- double-precision floating point.
- Scalar / vector
- Types supported lead to variations of individual instructions
4. Operations in ISA

<table>
<thead>
<tr>
<th>Operations</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arithmetic &amp; logical</td>
<td>Integer arithmetic, logical operations: add, and, multiply, etc</td>
</tr>
<tr>
<td>Data transfer</td>
<td>Load-stores</td>
</tr>
<tr>
<td>Control</td>
<td>Branch, jump, procedure call and return, traps</td>
</tr>
<tr>
<td>System</td>
<td>Operating system call, virtual memory management instructions</td>
</tr>
<tr>
<td>Floating point</td>
<td>Floating point operations: add, multiple, divide, compare</td>
</tr>
<tr>
<td>String</td>
<td>String move, string compare, string search</td>
</tr>
<tr>
<td>Graphics</td>
<td>Pixel and vertex operations, compression/decompression, etc</td>
</tr>
<tr>
<td>Signal processing</td>
<td>MAC units, vector (SIMD) processing</td>
</tr>
</tbody>
</table>
4. Operations supported

<table>
<thead>
<tr>
<th>Rank</th>
<th>Instruction</th>
<th>Integer Average</th>
<th>Percent total executed</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>load</td>
<td></td>
<td>22%</td>
</tr>
<tr>
<td>2</td>
<td>conditional branch</td>
<td></td>
<td>20%</td>
</tr>
<tr>
<td>3</td>
<td>compare</td>
<td></td>
<td>16%</td>
</tr>
<tr>
<td>4</td>
<td>store</td>
<td></td>
<td>12%</td>
</tr>
<tr>
<td>5</td>
<td>add</td>
<td></td>
<td>8%</td>
</tr>
<tr>
<td>6</td>
<td>and</td>
<td></td>
<td>6%</td>
</tr>
<tr>
<td>7</td>
<td>sub</td>
<td></td>
<td>5%</td>
</tr>
<tr>
<td>8</td>
<td>move register-register</td>
<td></td>
<td>4%</td>
</tr>
<tr>
<td>9</td>
<td>call</td>
<td></td>
<td>1%</td>
</tr>
<tr>
<td>10</td>
<td>return</td>
<td></td>
<td>1%</td>
</tr>
<tr>
<td></td>
<td>Total</td>
<td></td>
<td>96%</td>
</tr>
</tbody>
</table>

Simple instructions dominate instruction frequency

- In Intel x86 ISA, 10 simple instructions account for 96% of integer programs \(\rightarrow\) make the common case fast
What makes a good ISA?

- Efficiency of hardware implementation
- Convenience of programming / compiling
- Matches target applications (or alternatively generality)
- Compatibility and portability

Four design principles for ISA

1. Simplicity favors regularity
2. Smaller is faster
3. Make the common case fast
4. Good design demands good compromises

ISA design is an art!
Example: ARM ISA

- Billions of devices (e.g., smart phones, tables, etc) use ARM architecture.
- Example of register-register / load-store ISA
- 32 bit and 64 bit available
- All instructions are word aligned.
- All instructions could be conditionally executed.
- Most instructions execute in 1 cycle
- Will not cover all options in ISA but rather pick most used ones.
1. Register file

- ARM has 16 32-bit integer register file (*smaller is faster*)

<table>
<thead>
<tr>
<th>Name</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>R0</td>
<td>Argument / return value / temporary variable</td>
</tr>
<tr>
<td>R1-R3</td>
<td>Argument / temporary variables</td>
</tr>
<tr>
<td>R4-R11</td>
<td>Saved variables</td>
</tr>
<tr>
<td>R12</td>
<td>Temporary variable</td>
</tr>
<tr>
<td>R13 (SP)</td>
<td>Stack Pointer</td>
</tr>
<tr>
<td>R14 (LR)</td>
<td>Link Register</td>
</tr>
<tr>
<td>R15 (PC)</td>
<td>Program Counter</td>
</tr>
</tbody>
</table>

In addition is also a status register (CPSR) → holds flags: results of arithmetic and logical operations.
2. Memory instructions

- Each data byte has unique address
- 32-bit word = 4 bytes, so word address increments by 4 and aligned
- Use little endian numbering
- Instructions:
  - **Loads**: LDR, LDRB
  - **Stores**: STR, STRB
- Addresses are in bytes: can be written in decimal or hexadecimal (prefix address with 0x)
### Addressing offset and indexing methods

<table>
<thead>
<tr>
<th>Addressing Method</th>
<th>ARM Assembly</th>
<th>Memory Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rd←[Rn]</td>
<td>LDR R0, [R9]</td>
<td>R9</td>
</tr>
<tr>
<td>Rd←[Rn,±imm]</td>
<td>LDR R0, [R3, #4]</td>
<td>R3 + 4</td>
</tr>
<tr>
<td></td>
<td>LDR R0, [R5, #-16]</td>
<td>R5 – 16</td>
</tr>
<tr>
<td>Rd←[Rn,±Rm]</td>
<td>LDR R1, [R6, R7]</td>
<td>R6 + R7</td>
</tr>
<tr>
<td></td>
<td>LDR R2, [R8, -R9]</td>
<td>R8 – R9</td>
</tr>
<tr>
<td>Rd←[Rn,±Rm,shft]</td>
<td>LDR R3, [R10, R11, LSL #2]</td>
<td>R10 + (R11 &lt;&lt; 2)</td>
</tr>
<tr>
<td></td>
<td>LDR R4, [R1, -R12, ASR #4]</td>
<td>R1 – (R12 &gt;&gt;&gt; 4)</td>
</tr>
</tbody>
</table>

### Indexing:

1. **Preindex:** Use `!` to update Rn with memory access.

   ➔ LDR R3, [R5, #16]! ; R3 = mem[R5 + 16]; R5 = R5 + 16;

2. **Postindex:** Use `[ ]` on Rn to update after memory access.

   ➔ LDR R8, [R1], #8 ; R8 = mem[R1] ; R1 = R1 + 8
Memory instruction format

- \( op = 01 \)
- \( Rn = \) base register
- \( Rd = \) destination (load), source (store)
- \( Src2 = \) offset: register (optionally shifted) or immediate
- \( funct = \) 6 control bits

### Instruction Table

<table>
<thead>
<tr>
<th>( L )</th>
<th>( B )</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>STR</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>STRB</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>LDR</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>LDRB</td>
</tr>
</tbody>
</table>

### Indexing Mode

<table>
<thead>
<tr>
<th>( P )</th>
<th>( W )</th>
<th>Indexing Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>Not supported</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>Postindex</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>Offset</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>Preindex</td>
</tr>
</tbody>
</table>

### Value Table

<table>
<thead>
<tr>
<th>Value</th>
<th>( I )</th>
<th>( U )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Immediate</td>
<td>Subtract offset from base</td>
</tr>
<tr>
<td>1</td>
<td>Register</td>
<td>Add offset to base</td>
</tr>
</tbody>
</table>
Example 1 (register):

LDR R3, [R4, R5]

- **Operation**: R3 ← mem[R4 + R5]
- **cond** = 1110₂ (14) for unconditional execution
- **op** = 01₂ (1) for memory instruction
- **funct** = 111001₂ (57)
  - I = 1 (register offset), P = 1 (offset indexing),
  - U = 1 (add), B = 0 (load word),
  - W = 0 (offset indexing), L = 1 (load)
- **Rd** = 3, **Rn** = 4, **Rm** = 5 (**shamt5** = 0, **sh** = 0)

1110 01 111001 0100 0011 00000 00 0 0101 = 0xE7943005
Example 2 (immediate):

\texttt{STR R11, [R5], #-26}

- **Operation:** \(R11 \leftarrow \text{mem}[R5]; R5 \leftarrow R5 - 26\)
- **cond** = \(1110_2\) (14) for unconditional execution
- **op** = \(01_2\) (1) for memory instruction
- **funct** = \(00000002\) (0)
  - \(I = 0\) (immediate offset), \(P = 0\) (postindex),
  - \(U = 0\) (subtract), \(B = 0\) (store word),
  - \(W = 0\) (postindex), \(L = 0\) (store)
- **Rd** = 11, **Rn** = 5, **imm12** = 26

**Field Values**

<table>
<thead>
<tr>
<th>Field</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cond</td>
<td>1110</td>
</tr>
<tr>
<td>Op</td>
<td>01</td>
</tr>
<tr>
<td>Rn</td>
<td>0000000</td>
</tr>
<tr>
<td>Rd</td>
<td>0101</td>
</tr>
<tr>
<td>imm12</td>
<td>1011</td>
</tr>
<tr>
<td>E</td>
<td>0000, 0001</td>
</tr>
<tr>
<td>4</td>
<td>1010</td>
</tr>
<tr>
<td>0</td>
<td>A</td>
</tr>
</tbody>
</table>
Example 3 (register shifted):

```
STR R9, [R1, R3, LSL #2]
```

- **Operation:** $R9 \leftarrow \text{mem}[R1 + (R3 << 2)]$
- **cond** = 1110₂ (14) for unconditional execution
- **op** = 01₂ (1) for memory instruction
- **funct** = 111000₂ (0)
  - $\bar{I} = 1$ (register offset), $P = 1$ (offset indexing),
  - $U = 1$ (add), $B = 0$ (store word), $W = 0$ (offset indexing),
  - $L = 0$ (store)
- **Rd** = 9, **Rn** = 1, **Rm** = 3, **shamt** = 2, **sh** = 00₂ (LSL)

```
1110 01 111000 0001 1001 00010 00 0 0011 = 0xE7819103
```
3.A Data processing operations

- **Movement**
  
  MOV R1, #0x45
  MOV R1, #0xFF0
  MOV R1, R0, LSL #3; R1 ← (R0 << 3)
  MVN R7, R2

- **Arithmetic**: ADD, SUB, MUL

- **Logical**: AND, ORR, EOR, BIC (bit clear)

- **Shift/Rotation**: LSL, LSR, ASR, ROR
  
  LSL R0, R7, #5 ; R0 ← R7 << 5
  ADD R0, R1, R2 ; R0 ← R1 + R2
  ASR R9, R11, R4 ; R9 ← R11 >>> R47:0
  ORR R9, R5, R3, LSR #2 ; R9 ← R5 + (R3 >> 5)
  EOR R8, R9, R10, ROR R12 ; R8←R9+(R10 ROR R12)

- First operand must be a register.
- Second operand may be: immediate or a register (just register, register shifted by immediate or register shifted a register).
- Immediate can only be 8 bits.
Encoding of data operation instructions

- **Rn**: first source register
- **Rd**: destination register
- **Op**: 00 for data operations
- **cmd**: is code of operation. (e.g., 0100₂ for ADD, 0010₂ for SUB, 1100₂ for ORR, 0001₂ for EOR, 1101₂ for all shift operations)
- **S-bit**: 1 if sets condition flags
  - **S = 0**: SUB R0, R5, R7
  - **S = 1**: ADDS R8, R2, R4

Notice similarity to memory format: simplicity require regularity!
Example 1 (immediate):

```
ADD R0, R1, #42
```

- **cond** = $1110_2$ (14) for unconditional execution
- **op** = $00_2$ (0) for data-processing instructions
- **cmd** = $0100_2$ (4) for ADD
- **Src2** is an immediate so $I = 1$
- **Rd** = 0, **Rn** = 1
- **imm8** = 42, **rot** = 0

### Field Values

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1110_2</td>
<td>00_2</td>
<td>1</td>
<td>0100_2</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>42</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>cond</th>
<th>op</th>
<th>I</th>
<th>cmd</th>
<th>S</th>
<th>Rn</th>
<th>Rd</th>
<th>shamt5</th>
<th>sh</th>
<th>Rm</th>
</tr>
</thead>
<tbody>
<tr>
<td>1110_2</td>
<td>00_2</td>
<td>1</td>
<td>0100_2</td>
<td>0001</td>
<td>0000</td>
<td>0000</td>
<td>00101010</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

0xE281002A
Example 2 (immediate rotated):

SUB R2, R3, #0xFF0

- **cond** = 1110₂ (14) for unconditional execution
- **op** = 00₂ (0) for data-processing instructions
- **cmd** = 0010₂ (2) for SUB
- **Src2** is an immediate so \( I = 1 \)
- **Rd** = 2, **Rn** = 3
- **imm8** = 0xFF
- **imm8** must be rotated right by 28 to produce 0xFF0, so \( \text{rot} \) = 14

**Field Values**

\[
\begin{array}{cccccccc}
1110₂ & 00₂ & 1 & 0010₂ & 0 & 3 & 2 & 14 & 255 \\
\end{array}
\]

<table>
<thead>
<tr>
<th>cond</th>
<th>op</th>
<th>I</th>
<th>cmd</th>
<th>S</th>
<th>Rn</th>
<th>Rd</th>
<th>rot</th>
<th>imm8</th>
</tr>
</thead>
<tbody>
<tr>
<td>1110₂</td>
<td>00₂</td>
<td>1</td>
<td>0010₂</td>
<td>0</td>
<td>3</td>
<td>2</td>
<td>14</td>
<td>255</td>
</tr>
</tbody>
</table>

\[\text{ROR by 28} = \text{ROL by (32-28)} = 4\]

\[\text{0xE2432EFF}\]
Example 3 (register):

\[ \text{ADD} \ R5, \ R6, \ R7 \]

- \textit{cond} = 1110_2 (14) for unconditional execution
- \textit{op} = 00_2 (0) for data-processing instructions
- \textit{cmd} = 0100_2 (4) for \textit{ADD}
- \textit{Src2} is a register so \textit{I}=0
- \textit{Rd} = 5, \textit{Rn} = 6, \textit{Rm} = 7
- \textit{shamt} = 0, \textit{sh} = 0

\begin{center}
\begin{tabular}{c|c|c|c|c|c|c|c|c|c}
\hline
\hline
1110_2 & 00_2 & 0 & 0100_2 & 0 & 6 & 5 & 0 & 0 & 0 & 7 \\
\hline
\end{tabular}
\end{center}

Field Values

\textit{cond} \quad \textit{op} \quad \textit{I} \quad \textit{cmd} \quad \textit{S} \quad \textit{Rn} \quad \textit{Rd} \quad \textit{shamt5} \quad \textit{sh} \quad \textit{Rm}

\begin{center}
\begin{tabular}{c|c|c|c|c|c|c|c|c|c|c}
\hline
1110 & 00 & 0 & 0100 & 0 & 0110 & 0101 & 00000 & 00 & 0 & 0111 \\
\hline
\end{tabular}
\end{center}

\textbf{0xE0865007}
Example 3 (register shifted):

```plaintext
ORR R9, R5, R3, LSR #2
```

- **Operation:** R9 = R5 OR (R3 >> 2)
- **cond** = \(1110_2\) (14) for unconditional execution
- **op** = \(00_2\) (0) for data-processing instructions
- **cmd** = \(1100_2\) (12) for ORR
- **Src2** is a register so \(I=0\)
- \(Rd = 9, Rn = 5, Rm = 3\)
- **shamt5** = 2, **sh** = \(01_2\) (LSR)

**Data-processing**

```
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>cond</td>
<td><strong>op</strong></td>
<td>I</td>
<td>cmd</td>
<td>S</td>
<td>Rn</td>
<td>Rd</td>
<td>Src2</td>
</tr>
</tbody>
</table>
```

**Register**

```
<table>
<thead>
<tr>
<th>11:7</th>
<th>6:5</th>
<th>4</th>
<th>3:0</th>
</tr>
</thead>
<tbody>
<tr>
<td>shamt5</td>
<td>sh</td>
<td>0</td>
<td>Rm</td>
</tr>
</tbody>
</table>
```

<table>
<thead>
<tr>
<th>funct</th>
<th>I = 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1110)</td>
<td>(00)</td>
</tr>
</tbody>
</table>

\(0\times1859123\)
3.B Control flow operations

- Branches enable out of sequence instruction execution

- Types of branches:
  - Branch (B)
    - branches to another instruction
  - Branch and link (BL)
    - discussed later

- Both can be conditional or unconditional
Unconditional branching

ARM assembly

```assembly
MOV R2, #17 ; R2 = 17
B   TARGET ; branch to target
ORR R1, R1, #0x4 ; not executed
```

TARGET:

```assembly
SUB R1, R1, #78 ; R1 = R1 + 78
```

Labels (like TARGET) indicate instruction location. Labels can’t be reserved words (like ADD, ORR, etc.)
Conditional branches

ARM Assembly

MOV  R0, #4       ; R0 = 4
ADD  R1, R0, R0   ; R1 = R0+R0 = 8
CMP  R0, R1       ; sets flags with R0-R1
BEQ  THERE        ; branch not taken (Z=0)
ORR  R1, R1, #1   ; R1 = R1 OR R1 = 9

THERE:
ADD  R1, R1, 78   ; R1 = R1 + 78 = 87
Example if-else code

**C Code**

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

**ARM Assembly Code**

```asm
;R0=f, R1=g, R2=h, R3=i, R4=j

CMP R3, R4        ; set flags with R3-R4
    BNE L1          ; if i!=j, skip if block
ADD R0, R1, R2    ; f = g + h
    B   L2          ; branch past else block
L1
SUB R0, R0, R2    ; f = f - i
    L2
```

```
Example: while loops

C Code

// determines the power
// of x such that \(2^x = 128\)
int pow = 1;
int x = 0;

while (pow != 128) {
    pow = pow * 2;
    x = x + 1;
}

ARM Assembly Code

; R0 = pow, R1 = x
MOV R0, #1 ; pow = 1
MOV R1, #0 ; x = 0

WHILE:
    CMP R0, #128 ; R0-128
    BEQ DONE ; if (pow==128)
               ; exit loop
    LSL R0, R0, #1 ; pow=pow*2
    ADD R1, R1, #1 ; x=x+1
    B WHILE ; repeat loop

DONE:
Branch format

Encodes \( B \) and \( BL \)

- \( op = 10_2 \)
- \( imm24 \): 24-bit immediate; # of words BTA is away from PC+8
- \( funct = 1L_2 \): \( L = 0 \) for \( B \), \( L = 1 \) for \( BL \) (branch & link)

Some condition codes:

0000: EQ: equal
0001: NE: not equal
0100: MI: negative
0101: PL: positive or zero
1010: GE: greater than or equal
1011: LT: less than
1100: GT: greater than
1101: LE: less than or equal
Conditional execution

Encode in \textit{cond} bits of machine instruction

\begin{tabular}{|l|}
\hline
\textbf{C Code} & \textbf{ARM Assembly Code} \\
\hline
if (i == j) & \texttt{;R0=f, R1=g, R2=h, R3=i, R4=j} \\
f = g + h; & \texttt{CMP R3, R4} ; set flags with R3-R4 \\
else & \texttt{ADDEQ R0, R1, R2} ; if (i==j) f = g + h \\
f = f - i; & \texttt{SUBNE R0, R0, R2} ; else f = f - i \\
\hline
\end{tabular}

When to use?

\begin{center}
\textbf{Data-processing}
\end{center}

\begin{tabular}{|c|c|c|c|c|c|}
\hline
\hline
\texttt{cond} & 4 bits & 2 bits & 6 bits & 4 bits & 4 bits & 12 bits \\
\texttt{op} & & & & & & \\
\texttt{funct} & & & & & & \\
\texttt{Rn} & & & & & & \\
\texttt{Rd} & & & & & & \\
\texttt{Src2} & & & & & & \\
\hline
\end{tabular}

\begin{center}
\texttt{EORLT R9, R5, R6} \\
\texttt{ADDEQ R4, R5, R6}
\end{center}

\begin{tabular}{|c|c|c|c|c|c|c|c|c|c|}
\hline
& 11 & 0 & 0 & 1 & 0 & 5 & 9 & 0 & 0 & 0 & 6 \\
\hline
\end{tabular}

\begin{tabular}{|c|c|c|c|c|c|c|c|c|c|}
\hline
& 0 & 0 & 0 & 4 & 0 & 5 & 4 & 0 & 0 & 0 & 6 \\
\hline
\end{tabular}
Procedure/ function calls

- **Caller:**
  - passes *arguments* to callee (R0-R3)
  - branches to callee using branch and link (BL)

- **Callee:**
  - *performs* the function
  - *returns* result to caller in R0
  - *returns* to point of call by moving the link register to PC: MOV PC, LR
  - *must not overwrite* registers or memory needed by caller
Example

High-level code

```c
int main()
{
    int y;
    ...
    y = diffofsums(2, 3, 4, 5); // 4 arguments
    ...
}

int diffofsums(int f, int g, int h, int i)
{
    int result;
    result = (f + g) - (h + i);
    return result; // return value
}
```
Example

; R4 = y
MAIN
...

MOV R0, #2    ; argument 0 = 2
MOV R1, #3    ; argument 1 = 3
MOV R2, #4    ; argument 2 = 4
MOV R3, #5    ; argument 3 = 5
BL DIFFOFSUMS ; call function
MOV R4, R0    ; y = returned value
...

; R4 = result
DIFFOFSUMS:
ADD R8, R0, R1 ; R8 = f + g
ADD R9, R2, R3 ; R9 = h + i
SUB R4, R8, R9 ; result = (f + g) - (h + i)
MOV R0, R4    ; put return value in R0
MOV PC, LR    ; return to caller

• diffofsums overwrote 3 registers: R4, R8, R9
• diffofsums can use stack to temporarily store registers
**The stack**

- Memory used to temporarily save variables
- Like stack of dishes, last-in-first-out (LIFO) queue
- **Expands:** uses more memory when more space needed
- **Contracts:** uses less memory when the space no longer needed
- Grows down (from higher to lower memory addresses)
- Stack pointer: SP points to top of the stack

<table>
<thead>
<tr>
<th>Address</th>
<th>Data</th>
<th>Address</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEFFFAE8</td>
<td>AB000001</td>
<td>BEFFFAE8</td>
<td>AB000001</td>
</tr>
<tr>
<td>BEFFFAE4</td>
<td></td>
<td>BEFFFAE4</td>
<td>12345678</td>
</tr>
<tr>
<td>BEFFFAE0</td>
<td></td>
<td>BEFFFAE0</td>
<td>FFEEDDCC</td>
</tr>
<tr>
<td>BEFFFADC</td>
<td></td>
<td>BEFFFADC</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Stack expands by 2 words
### Storing register values on the stack

**DIFFOFSUMS**

```assembly
SUB SP, SP, #12 ; make space on stack for 3 registers
STR R4, [SP, #-8] ; save R4 on stack
STR R8, [SP, #-4] ; save R8 on stack
STR R9, [SP] ; save R9 on stack
ADD R8, R0, R1 ; R8 = f + g
ADD R9, R2, R3 ; R9 = h + i
SUB R4, R8, R9 ; result = (f + g) - (h + i)
MOV R0, R4 ; put return value in R0
LDR R9, [SP] ; restore R9 from stack
LDR R8, [SP, #-4] ; restore R8 from stack
LDR R4, [SP, #-8] ; restore R4 from stack
ADD SP, SP, #12 ; deallocate stack space
MOV PC, LR ; return to caller
```

<table>
<thead>
<tr>
<th>Address</th>
<th>Data</th>
<th>Address</th>
<th>Data</th>
<th>Address</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEFOFOFC</td>
<td>?</td>
<td>BEFOFOFC</td>
<td>?</td>
<td>BEFOFOFC</td>
<td>?</td>
</tr>
<tr>
<td>BEFOFOF6</td>
<td></td>
<td>BEFOFOF0</td>
<td></td>
<td>BEFOFOF0</td>
<td></td>
</tr>
<tr>
<td>BEFOFOF4</td>
<td></td>
<td>R9</td>
<td></td>
<td>R8</td>
<td></td>
</tr>
<tr>
<td>BEFOFOF0</td>
<td></td>
<td>R4</td>
<td></td>
<td>R4</td>
<td></td>
</tr>
</tbody>
</table>

- **Before call**
- **During call**
- **After call**

**Can code be reduced?**
Protocol for preserving registers

<table>
<thead>
<tr>
<th>Preserved</th>
<th>Nonpreserved</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Callee-Saved</strong></td>
<td><strong>Caller-Saved</strong></td>
</tr>
<tr>
<td>R4-R11</td>
<td>R12</td>
</tr>
<tr>
<td>R14 (LR)</td>
<td>R0-R3</td>
</tr>
<tr>
<td>R13 (SP)</td>
<td>CPSR</td>
</tr>
<tr>
<td>stack above SP</td>
<td>stack below SP</td>
</tr>
</tbody>
</table>
Recursive procedure calls

High-level code

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

What is the potential problem with recursive or nested function calls?
# Recursive procedure calls

## C Code

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

## ARM Assembly Code

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x94</td>
<td>FACTORIAL</td>
<td>STR R0, [SP, #-4]!</td>
</tr>
<tr>
<td>0x98</td>
<td></td>
<td>STR LR, [SP, #-4]!</td>
</tr>
<tr>
<td>0x9C</td>
<td></td>
<td>CMP R0, #2</td>
</tr>
<tr>
<td>0xA0</td>
<td></td>
<td>BHS ELSE</td>
</tr>
<tr>
<td>0xA4</td>
<td></td>
<td>MOV R0, #1</td>
</tr>
<tr>
<td>0xA8</td>
<td></td>
<td>ADD SP, SP, #8</td>
</tr>
<tr>
<td>0xAC</td>
<td></td>
<td>MOV PC, LR</td>
</tr>
<tr>
<td>0xB0</td>
<td>ELSE</td>
<td>SUB R0, R0, #1</td>
</tr>
<tr>
<td>0xB4</td>
<td></td>
<td>BL FACTORIAL</td>
</tr>
<tr>
<td>0xB8</td>
<td></td>
<td>LDR LR, [SP], #4</td>
</tr>
<tr>
<td>0xBC</td>
<td></td>
<td>LDR R1,[SP], #4</td>
</tr>
<tr>
<td>0xC0</td>
<td></td>
<td>MUL R0, R1, R0</td>
</tr>
<tr>
<td>0xC4</td>
<td></td>
<td>MOV PC, LR</td>
</tr>
</tbody>
</table>
```
Stack during recursive calls

### Before call
- **Address**: BEFF0FF0
- **Data**: SP
- **Address**: BEFF0FEC
- **Data**: SP
- **Address**: BEFF0FE8
- **Data**: SP
- **Address**: BEFF0FE4
- **Data**: SP
- **Address**: BEFF0FE0
- **Data**: SP
- **Address**: BEFF0FDC
- **Data**: SP
- **Address**: BEFF0FD8
- **Data**: SP

### During call
- **Address**: BEFF0FF0
  - **Data**: LR
- **Address**: BEFF0FEC
  - **Data**: R0 (3)
- **Address**: BEFF0FE8
  - **Data**: LR (0x8520)
- **Address**: BEFF0FE4
  - **Data**: R0 (2)
- **Address**: BEFF0FE0
  - **Data**: LR (0x8520)
- **Address**: BEFF0FDC
  - **Data**: R0 (1)

### After call
- **Address**: BEFF0FF0
  - **Data**: LR
- **Address**: BEFF0FEC
  - **Data**: R0 (3)
- **Address**: BEFF0FE8
  - **Data**: LR (0x8520)
- **Address**: BEFF0FE4
  - **Data**: R0 (2)
- **Address**: BEFF0FE0
  - **Data**: LR (0x8520)
- **Address**: BEFF0FDC
  - **Data**: R0 (1)

R0 values:
- **Before call**: 6
- **During call**: 3
- **After call**: 1
Summary

• Overviewed most relevant instructions in ARM ISA.

• Additional instructions exist for various versions: Thumb extensions, floating point and SIMD NEON extensions.