Instruction Set Architecture (ISA)

Encoding of instructions raises some interesting choices...

- **Tradeoffs**: performance, compactness, programmability
- **Uniformity**: Should different instructions
  - Be the same size (number of bits)?
  - Take the same amount of time to execute?
- **Complexity**: How many different instructions? What level operations?
  - Level of support for particular software operations: array indexing, procedure calls, "polynomial evaluate", etc
  - "Reduced Instruction Set Computer" (RISC) philosophy: simple instructions, optimized for speed
- **Mix of Engineering & Art...**
In Comp 411 we’ll use a subset of the ARM7 core instruction set as an example ISA.

ARM7 uses byte memory addresses. However, each instruction is 32-bits wide, and *must* be aligned on a multiple of 4 (word) address. Each word contains four 8-bit bytes. Addresses of consecutive instructions (words) differ by 4.
ARM7 Memory Nits

- Memory locations are addressable in different sized chunks
  - 8-bit chunks (bytes)
  - 16-bit chunks (shorts)
  - 32-bit chunks (words)
  - 64-bit chunks (longs/doubles)
- We also frequently need access to individual bits! (Instructions help with this)
- Every BYTE has a unique address (ARM is a byte-addressable machine)
- Most instructions are one word
- We will consider the predominant "little-endian" ARM.
ARM REGISTER NITS

- There are 16 named registers [R0, R1, ..., R15]
- The operands of most instructions are registers
- This means to operate on a variable in memory you must:
  - Load the value/values from memory into a register
  - Perform the instruction
  - Store the result back into memory
- Going to and from memory can be expensive
  (4x to 20x slower than operating on a register)
- Net effect: Keep variables in registers as much as possible!
- 3 registers are dedicated to specific tasks
  (SP=R13, LR=R14, PC=R15), 13 are available for general use
Basic ARM Instructions

- Instructions include various "fields" that encode combinations of Opcodes and arguments
- special fields enable extended functions (more in a minute)
- several 4-bit OPERAND fields, for specifying the sources and destination of the operation, usually one of the 16 registers
- Embedded constants ("immediate" values) of various sizes,

The "basic" data-processing instruction formats:

<table>
<thead>
<tr>
<th>R type:</th>
<th>1110</th>
<th>000</th>
<th>Opcode</th>
<th>0</th>
<th>Rn</th>
<th>Rd</th>
<th>00000000</th>
<th>Rm</th>
</tr>
</thead>
<tbody>
<tr>
<td>I type:</td>
<td>1110</td>
<td>001</td>
<td>Opcode</td>
<td>0</td>
<td>Rn</td>
<td>Rd</td>
<td>Shift</td>
<td>Imm</td>
</tr>
</tbody>
</table>

The 4-bit Opcode field encodes the type of operation, the operand fields specify the sources and destinations, and the Immediate field provides an immediate value if needed.
R-type Data Processing

Instructions that process three-register arguments:

Simple R-type instructions follow the following template:

\[ \text{OP} \quad \text{Rd}, \text{Rn}, \text{Rm} \]

Later on we'll introduce more complex variants of these 'simple' R-type instructions.

Is encoded as:

\[ 1110 \ 0000 \ 1000 \ 0001 \ 0000 \ 0000 \ 0000 \ 0011 \]

0xE0810003
**I-type Data Processing**

Instructions that process two registers and a constant:

<table>
<thead>
<tr>
<th>I type:</th>
<th>1110</th>
<th>001</th>
<th>Opcode</th>
<th>S</th>
<th>Rn</th>
<th>Rd</th>
<th>Rotate</th>
<th>Imm8</th>
</tr>
</thead>
</table>

Simple I-type instructions follow the following template:

```
OP       Rd, Rn, #constant
```

In the I-type instructions the second register operand is replaced by a constant that is encoded in the instruction.

RSB      R7, R10, #49

Is encoded as:

```
1110 0010 0110 1010 0111 0000 0011 0001
```

0xDE26A7031
I-TYPE CONSTANTS

ARM7 provides only 8-bits for specifying an immediate constant value. Given that ARM7 is a 32-bit architecture, this may appear to be a severe limitation. However, by allowing for a rotating shift to be applied to the constant.

\[ \text{imm32} = (\text{imm8} \gg (2 \times \text{rotate})) | (\text{imm8} \ll (32 - (2 \times \text{rotate}))) \]

Example: 1920 is encoded as:

<table>
<thead>
<tr>
<th>Rotate</th>
<th>Imm8</th>
</tr>
</thead>
<tbody>
<tr>
<td>1101</td>
<td>00011110</td>
</tr>
</tbody>
</table>

\[ = (30 \gg (2 \times 13)) | (30 \ll (32 - (2 \times 13))) \]

\[ = 0 | 30 \times 64 \]

\[ = 1920 \]

How would 256 be encoded?

<table>
<thead>
<tr>
<th>Rotate</th>
<th>Imm8</th>
</tr>
</thead>
<tbody>
<tr>
<td>1100</td>
<td>00000001</td>
</tr>
</tbody>
</table>
NEXT TIME

- We will examine more of the "basic" instruction types and capabilities
- Result flags
- Program Status Registers
Read the Instructions

.. when all else fails

- What do instructions do?
- How are instructions decoded?
- Uniformity and Symmetry
- Cramming stuff in
- CPU state
  - Condition codes
  - Program Status Register (PSR)
A closer look at the opcodes

The **Opcode** field is common to both of the basic instruction types

<table>
<thead>
<tr>
<th></th>
<th>4</th>
<th>3</th>
<th>4</th>
<th>1</th>
<th>4</th>
<th>4</th>
<th>8</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>R type:</strong></td>
<td>1110</td>
<td>000</td>
<td>Opcode</td>
<td>S</td>
<td>Rn</td>
<td>Rd</td>
<td>00000000</td>
<td>Rm</td>
</tr>
<tr>
<td><strong>I type:</strong></td>
<td>1110</td>
<td>001</td>
<td>Opcode</td>
<td>S</td>
<td>Rn</td>
<td>Rd</td>
<td>Rotate</td>
<td>Imm8</td>
</tr>
</tbody>
</table>

ARM data processing instructions can be broken into four basic groups:
- Arithmetic (6)
- Logic (4)
- Comparison (4)
- Register transfer (2)

We haven't discussed the 'S' field yet. If set, it tells the processor to retain some 'state' after the instruction has executed. This 'state' is in the form of 5-flags.

Many instructions (all we've seen thus far) have a special variant that sets the state flags. In these variants the opcode has an 'S' appended.
**ARITHMETIC INSTRUCTIONS**

**ADD** R3, R2, R12

R3 ← R2 + R12

Registers can contain either 32-bit unsigned values or 32-bit 2's-complement signed values.

**SUB** R0, R4, R6

R0 ← R4 - R6

Once more, either 32-bit unsigned values or 32-bit 2's-complement signed values.

**RSB** R0, R4, R2

R0 ← - R4 + R2

The operands of the subtraction are in reversed order. It is called 'Reverse Subtract'. Why? The I-type version makes more sense.

**ADC** R1, R5, R8

R1 ← R5 + R8 + C

Where 'C' is the Carry-out from some earlier instruction (usually an ADDS or ADCS) as saved in the Program Status Register (PSR)

**SBC** R2, R5, R7

R2 ← R5 - R7 - 1 + C

Where 'C' is the Carry-out from some earlier instruction (usually a SUBS or SUBCS) as saved in the PSR

**RSC** R1, R5, R3

R1 ← - R5 + R3 - 1 + C

'Reverse Subtract' with a Carry. Usually a carry generated from a previous RSBS or RSCS instruction.

A byte-sized example: 411 = 00000001 00000001

-42 = 00000000 00101010

\[ \begin{array}{c}
1=1-1+C \\
1=01111111 + 00101010 + 00000001 \\
10011011 \end{array} \]

\[ C=1 \]

\[ 00000001 + 00000001 + 11111111 = 00000001 + 11010101 + 01110001 = 10011011 \]

= 256 + 113 = 369
**Logic Instructions**

Logical operations on words operate "bitwise", that is they are applied to corresponding bits of both source operands.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Example</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AND R0, R1, R2</td>
<td>R1: 0000 0000 0000 0000 1111 1111 0000 0000&lt;br&gt;R2: 0000 0000 0000 0000 1111 0000 1111 0000&lt;br&gt;R0: 0000 0000 0000 0000 1111 0000 0000 0000</td>
<td>Commonly called &quot;exclusive-or&quot;</td>
</tr>
<tr>
<td>ORR R0, R1, R2</td>
<td>R0: 0000 0000 0000 0000 1111 0000 1111 0000</td>
<td>Called &quot;Bit-clear&quot;</td>
</tr>
<tr>
<td>EOR R0, R1, R2</td>
<td>R0: 0000 0000 0000 0000 1111 1111 1111 0000</td>
<td></td>
</tr>
<tr>
<td>BIC R0, R1, R2</td>
<td>R0: 0000 0000 0000 0000 0000 1111 1111 1111 0000</td>
<td></td>
</tr>
</tbody>
</table>
Now it is time to discuss what status flags are available. These five status flags are kept in a special register called the Program Status Register (PSR). The PSR also contains other important bits that control the processor.

- **N** - set if the result of an operation is negative (Most Significant Bit (MSB) is a 1)
- **Z** - set if the result of an operation is "0"
- **C** - set if the result of an operation has a carry out of its MSB
- **V** - set if a sum of two positive operands gives a negative result, or if the sum of two negative operands gives a positive result
- **Q** - a sticky version of overflow created by instructions that generate multiple results (more on this later on).
**Comparison Instructions**

These instructions modify the status flags, but leave the contents of the registers unchanged. They are used to test register contents, and they **must** have their "S" bit set to "1". They also don't modify their Rd, and by convention, Rd is set to "0000".

<table>
<thead>
<tr>
<th>R type:</th>
<th>1110</th>
<th>000</th>
<th>Opcode</th>
<th>1</th>
<th>Rn</th>
<th>0000</th>
<th>00000000</th>
<th>Rm</th>
</tr>
</thead>
<tbody>
<tr>
<td>I type:</td>
<td>1110</td>
<td>001</td>
<td>Opcode</td>
<td>1</td>
<td>Rn</td>
<td>0000</td>
<td>Rotate</td>
<td>Imm8</td>
</tr>
</tbody>
</table>

**CMP R0, R1**  
PSR flags set for the result  
R2 - R3

**CMN R2, R3**  
PSR flags set for the result  
R2 + R3

**TST R4, #8**  
PSR flags set for the result  
R4 & 8

**TEQ R5, #1024**  
PSR flags set for the result  
R5 ^ 1024
Register Transfer

These instructions are used to transfer the contents of one register to another, or simply to initialize the contents of a register. They make use of only one operand, and, by convention, have their Rn field set to "0000".

**MOV R0, R3**  
R0 ← R3

**MOV R1, #4096**  
R1 ← 4096

**MVN R2, R4**  
R2 ← -R4

**MVN R3, #1**  
R3 ← -1

<table>
<thead>
<tr>
<th>R type:</th>
<th>1110</th>
<th>000</th>
<th>Opcode</th>
<th>S</th>
<th>0000</th>
<th>Rd</th>
<th>00000000</th>
<th>Rm</th>
</tr>
</thead>
<tbody>
<tr>
<td>I type:</td>
<td>1110</td>
<td>001</td>
<td>Opcode</td>
<td>S</td>
<td>0000</td>
<td>Rd</td>
<td>Rotate</td>
<td>Imm8</td>
</tr>
</tbody>
</table>

09/11/2017  
Comp 411 - Fall 2017
ARM Shift Operations

A novel feature of ARM is that all data-processing instructions can include an optional "shift", whereas most other architectures have separate shift instructions. This is actually very useful as we will see later on. The key to shifting is that 8-bit field between Rd and Rm.
Left Shifts

Left shifts effectively multiply the contents of a register by $2^s$ where $s$ is the shift amount.

**MOV R0, R0, LSL 7**

R0 before: 0000 0000 0000 0000 0000 0000 0000 0111 = 7

R1 after: 0000 0000 0000 0000 0000 0011 1000 0000 = 7 * $2^7$ = 896

Shifts can also be applied to the second operand of any data processing instruction

**ADD R1, R1, R0, LSL 7**
Right Shifts

Right Shifts behave like dividing the contents of a register by $2^s$ where s is the shift amount, if you assume the contents of the register are unsigned.

MOV R0, R0, LSR 2

R0 before: 0000 0000 0000 0000 0000 0100 0000 0000 = 1024
R1 after: 0000 0000 0000 0000 0000 0000 0001 0000 = 1024 / $2^2$ = 256
Arithmetic Right Shifts

Arithmetic right shifts behave like dividing the contents of a register by $2^s$ where $s$ is the shift amount, if you assume the contents of the register are signed.

MOV R0, R0, ASR 2

R0 before: 1111 1111 1111 1111 1111 1100 0000 0000 = -1024

R1 after: 1111 1111 1111 1111 1111 1111 1111 0000 0000 = -1024 / 2^2 = -256
Rotate Right Shifts

Rotating shifts have no arithmetic analogy. However, they don’t lose bits like both logical and arithmetic shifts. We saw rotate right shift used for the I-type "immediate" value earlier.

MOVE R0, R0, ROR 2

R0 before: 0000 0000 0000 0000 0000 0000 0000 0111
R1 after: 1100 0000 0000 0000 0000 0000 0000 0001 = 7

= -1,073,741,823

Why no rotate left shift?

- Ran out of encodings?
- Almost anything Rotate lefts can do ROR can do as well!
Next time

Instructions still missing

- Access to memory
- Branches and Calls
- Control
- Multiplication?
- Division?
- Floating point?