ARM History

  - Acorn Computers Limited, Cambridge, England
- ARM - Advanced RISC Machine 1990
  - ARM Limited, 1990
  - ARM has been licensed to many semiconductor manufacturers

Outline

- ARM Architecture
- ARM Organization and Implementation
- ARM Instruction Set
- Architectural Support for High-level Languages
- Thumb Instruction Set
- Architectural Support for System Development
- ARM Processor Cores
- Memory Hierarchy
- Architectural Support for Operating Systems
- ARM CPU Cores
- Embedded ARM Applications

ARM's visible registers

- User level
  - 15 GPRs, PC, CPSR (current program status register)
- Remaining registers are used for system-level programming and for handling exceptions

- ARM's visible registers
  - User mode
  - FIQ mode
  - SVCE mode
  - AHB mode
  - IRQ mode
  - Undefined mode
**ARM CPSR format**
- **N** (Negative), **Z** (Zero), **C** (Carry), **V** (Overflow)
- mode – control processor mode
- **T** – control instruction set
  - **T = 1** – instruction stream is 16-bit Thumb instructions
  - **T = 0** – instruction stream is 32-bit ARM instructions
- **I F** – interrupt enables

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>N</td>
<td>Z</td>
<td>C</td>
<td>V</td>
<td><strong>unused</strong></td>
<td><strong>IF</strong></td>
<td><strong>T</strong></td>
<td><strong>mode</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**ARM memory organization**
- Linear array of bytes numbered from 0 to $2^{32} - 1$
- Data items
  - bytes (8 bits)
  - half-words (16 bits) – always aligned to 2-byte boundaries (start at an even byte address)
  - words (32 bits) – always aligned to 4-byte boundaries (start at a byte address which is multiple of 4)

**ARM instruction set**
- Load-store architecture
  - operands are in GPRs
  - load/store – only instructions that operate with memory
- Instructions
  - Data Processing – use and change only register values
  - Data Transfer – copy memory values into registers (load) or copy register values into memory (store)
  - Control Flow
    - branch
    - branch-and-link – save return address to resume the original sequence
    - trapping into system code – supervisor calls

**ARM instruction set (cont’d)**
- Three-address data processing instructions
- Conditional execution of every instruction
- Powerful load/store multiple register instructions
- Ability to perform a general shift operation and a general ALU operation in a single instruction that executes in a single clock cycle
- Open instruction set extension through coprocessor instruction set, including adding new registers and data types to the programmer’s model
- Very dense 16-bit compressed representation of the instruction set in the Thumb architecture
I/O system

- I/O is memory mapped
  - internal registers of peripherals (disk controllers, network interfaces, etc.) are addressable locations within the ARM's memory map and may be read and written using the load-store instructions
- Peripherals may use either the normal interrupt (IRQ) or fast interrupt (FIQ) input
  - normally most interrupt sources share the IRQ input, while just one or two time-critical sources are connected to the FIQ input
- Some systems may include external DMA hardware to handle high-bandwidth I/O traffic

ARM exceptions

- ARM supports a range of interrupts, traps, and supervisor calls – all are grouped under the general heading of exceptions
- Handling exceptions
  - current state is saved by copying the PC into r14_exc and CPSR into SPSR_exc (exc stands for exception type)
  - processor operating mode is changed to the appropriate exception mode
  - PC is forced to a value between 0016 and 1C16, the particular value depending on the type of exception
  - instruction at the location PC is forced to (the vector address) usually contains a branch to the exception handler; the exception handler will use r13_exc, which is normally initialized to point to a dedicated stack in memory, to save some user registers
  - return: restore the user registers and then restore PC and CPSR atomically

ARM cross-development toolkit

- Software development
  - tools developed by ARM Limited
  - public domain tools (ARM back end for gcc C compiler)
- Cross-development
  - tools run on different architecture from one for which they produce code

Outline

- ARM Architecture
- ARM Assembly Language Programming
- ARM Organization and Implementation
- ARM Instruction Set
- Architectural Support for High-level Languages
- Thumb Instruction Set
- Architectural Support for System Development
- ARM Processor Cores
- Memory Hierarchy
- Architectural Support for Operating Systems
- ARM CPU Cores
- Embedded ARM Applications
ARM Instruction Set

- Data Processing Instructions
- Data Transfer Instructions
- Control flow Instructions

Data Processing Instructions

- Classes of data processing instructions
  - Arithmetic operations
  - Bit-wise logical operations
  - Register-movement operations
  - Comparison operations
- Operands: 32-bits wide; there are 3 ways to specify operands
  - come from registers
  - the second operand may be a constant (immediate)
  - shifted register operand
- Result: 32-bits wide, placed in a register
  - long multiply produces a 64-bit result

Data Processing Instructions (cont'd)

### Arithmetic Operations

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD r0, r1, r2</td>
<td>r0 := r1 + r2</td>
</tr>
<tr>
<td>ADC r0, r1, r2</td>
<td>r0 := r1 + r2 + C</td>
</tr>
<tr>
<td>SUB r0, r1, r2</td>
<td>r0 := r1 - r2</td>
</tr>
<tr>
<td>SBC r0, r1, r2</td>
<td>r0 := r1 - r2 - C</td>
</tr>
<tr>
<td>RSB r0, r1, r2</td>
<td>r0 := r2 - r1</td>
</tr>
<tr>
<td>ADC r0, r1, r2</td>
<td>r0 := r1 + r2 + C</td>
</tr>
<tr>
<td>ADD r0, r1, r2</td>
<td>r0 := r1 + r2</td>
</tr>
</tbody>
</table>

### Bit-wise Logical Operations

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AND r0, r1, r2</td>
<td>r0 := r1 and r2</td>
</tr>
<tr>
<td>ORR r0, r1, r2</td>
<td>r0 := r1 or r2</td>
</tr>
<tr>
<td>EOR r0, r1, r2</td>
<td>r0 := r1 xor r2</td>
</tr>
<tr>
<td>BIC r0, r1, r2</td>
<td>r0 := r1 and (not) r2</td>
</tr>
</tbody>
</table>

### Register Movement

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV r0, r2</td>
<td>r0 := r2</td>
</tr>
<tr>
<td>MVN r0, r2</td>
<td>r0 := not r2</td>
</tr>
</tbody>
</table>

### Comparison Operations

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CMP r1, r2</td>
<td>set cc on r1 - r2</td>
</tr>
<tr>
<td>CMN r1, r2</td>
<td>set cc on r1 + r2</td>
</tr>
<tr>
<td>TST r1, r2</td>
<td>set cc on r1 and r2</td>
</tr>
<tr>
<td>TEQ r1, r2</td>
<td>set cc on r1 xor r2</td>
</tr>
</tbody>
</table>

Data Processing Instructions (cont'd)

### Immediate Operands

Immediate = \((0 - 255) \times 2^n\), \(0 \leq n \leq 12\)

### Shifted Register Operands

- the second operand is subject to a shift operation before it is combined with the first operand

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD r3, r2, r1, LSL #3</td>
<td>r3 := r2 - 8 x r1</td>
</tr>
<tr>
<td>ADD r5, r3, r3, LSL #2</td>
<td>r5 := r5 - 2 x r3</td>
</tr>
</tbody>
</table>
**ARM shift operations**

- LSL – Logical Shift Left
- LSR – Logical Shift Right
- ASR – Arithmetic Shift Right
- ROR – Rotate Right
- RRX – Rotate Right
  - Extended by 1 place

**Setting the condition codes**

- Any DPI can set the condition codes (N, Z, V, and C)
  - for all DPIs except the comparison operations a specific request must be made
  - at the assembly language level this request is indicated by adding an ‘S’ to the opcode
  - Example (r3-r2 := r1-r0 + r3-r2)
    ```
    ADDS r2, r2, r0
    ADDC r3, r3, r1
    ```
  - Arithmetic operations set all the flags (N, Z, C, and V)
  - Logical and move operations set N and Z
    - preserve V and either preserve C when there is no shift operation, or set C according to shift operation (fall off bit)

**Multiplies**

- Example
  ```
  MUL r4, r3, r2
  MLA r4, r3, r2, r1
  ```

- Note
  - least significant 32-bits are placed in the result register, the rest are ignored
  - immediate second operand is not supported
  - result register must not be the same as the first source register
  - if ‘S’ bit is set the V is preserved and the C is rendered meaningless

- Example (r0 = r0 x 35)
  ```
  ADD r0, r0, r0, LSL #2
  RSB r3, r3, r1
  ```

**Data transfer instructions**

- Single register load and store instructions
  - transfer of a data item (byte, half-word, word) between ARM registers and memory
- Multiple register load and store instructions
  - enable transfer of large quantities of data
  - used for procedure entry and exit, to save/restore workspace registers, to copy blocks of data around memory
- Single register swap instructions
  - allow exchange between a register and memory in one instruction
  - used to implement semaphores to ensure mutual exclusion on accesses to shared data in multipla
Data Transfer Instructions (cont'd)

Single register load and store

Register-indirect addressing

LDR r0, [r1] ; r0 := mem32[r1]

ST R r0, [r1] ; mem32[r1] := r0

Note: r1 keeps a word address (2 LSBs are 0)

Base+offset addressing

(offset of up to 4Kbytes)

LDR r0, [r1, #4] ; r0 := mem32[r1 + 4]

Auto-indexing addressing

LDR r0, [r1] ; r0 := mem32[r1]

r1 := r1 + 4

Post-indexed addressing

LDR r0, [r1, #4] ; r0 := mem32[r1]

r1 := r1 + 4

Data Transfer Instructions (cont'd)

COPY: ADR r1, TABLE1 ; r1 points to TABLE1
ADR r2, TABLE2 ; r2 points to TABLE2
LOOP: LDR r0, [r1]
STR r0, [r2]
ADD r1, r1, #4
ADD r2, r2, #4

... TABLE1: ...

TABLE2: ...

COPY: ADR r1, TABLE1 ; r1 points to TABLE1
ADR r2, TABLE2 ; r2 points to TABLE2
LOOP: LDR r0, [r1, #4]
STR r0, [r2, #4]

... TABLE1: ...

TABLE2: ...

Multiple register data transfers

LDMIA r1, {r0, r2, r5}

r0 := mem32[r1]
r2 := mem32[r1 + 4]
r5 := mem32[r1 + 8]

Note: any subset (or all) of the registers may be transferred with a single instruction

Note: the order of registers within the list is insignificant

Note: including r15 in the list will cause a change in the control flow

- Stack organizations
  - FA - full ascending
  - EA - empty ascending
  - FD - full descending
  - ED - empty descending

Multiple register transfer addressing modes

Stack organization

- FA - full ascending
- EM - empty ascending
- FD - full descending
- ED - empty descending

Block copy view

- data is to be stored above or below the the address held in the base register
- address incrementing or decrementing begins before or after storing the first value

- FA - full ascending
- EM - empty ascending
- FD - full descending
- ED - empty descending

Stack organization

- FA - full ascending
- EM - empty ascending
- FD - full descending
- ED - empty descending

Block copy view

- data is to be stored above or below the the address held in the base register
- address incrementing or decrementing begins before or after storing the first value

- FA - full ascending
- EM - empty ascending
- FD - full descending
- ED - empty descending
The mapping between the stack and block copy views

Control flow instructions

<table>
<thead>
<tr>
<th>Branch</th>
<th>Interpretation</th>
<th>Normal uses</th>
</tr>
</thead>
<tbody>
<tr>
<td>BAL</td>
<td>Unconditional</td>
<td>Always take this branch</td>
</tr>
<tr>
<td>BEQ</td>
<td>Equal</td>
<td>Comparison equal or zero result</td>
</tr>
<tr>
<td>BNE</td>
<td>Not equal</td>
<td>Comparison not equal or non-zero result</td>
</tr>
<tr>
<td>BPL</td>
<td>Plus</td>
<td>Result positive or zero</td>
</tr>
<tr>
<td>BMI</td>
<td>Minus</td>
<td>Result negative or non-zero</td>
</tr>
<tr>
<td>BCS</td>
<td>Carry clear</td>
<td>Arithmetic operation did not give carry-out</td>
</tr>
<tr>
<td>BHS</td>
<td>Higher or same</td>
<td>Unsigned comparison gave higher or same</td>
</tr>
<tr>
<td>BVC</td>
<td>Overflow clear</td>
<td>Signed integer operation did not give overflow</td>
</tr>
<tr>
<td>BVS</td>
<td>Overflow set</td>
<td>Signed integer operation gave overflow</td>
</tr>
<tr>
<td>BGT</td>
<td>Greater than</td>
<td>Signed integer comparison gave greater than</td>
</tr>
<tr>
<td>BGE</td>
<td>Greater or equal</td>
<td>Signed integer comparison gave greater than or equal</td>
</tr>
<tr>
<td>BLT</td>
<td>Less than</td>
<td>Signed integer comparison gave less than</td>
</tr>
<tr>
<td>BLSE</td>
<td>Less than or equal</td>
<td>Signed integer comparison gave less than or equal</td>
</tr>
<tr>
<td>BHI</td>
<td>Higher</td>
<td>Unsigned comparison gave higher</td>
</tr>
<tr>
<td>BLS</td>
<td>Lower or same</td>
<td>Unsigned comparison gave lower or same</td>
</tr>
</tbody>
</table>

Conditional execution

- Conditional execution to avoid branch instructions used to skip a small number of non-branch instructions
- Example

```assembly
CMP r0, #5
BEQ BYPASS ; if (r0=5) {
    ADD r1, r1, r0 ; r1:=r1+r0-r2
    SUB r1, r1, r2 ;
    JMP END
}
BYPASS: ...

CMP r0, #5
ADDNE r1, r1, r0 ;
SUBNE r1, r1, r2 ;
...```

With conditional execution

```assembly
CMP r0, #5
ADDNE r1, r1, r0 ;
SUBNE r1, r1, r2 ;
```

Branch and link instructions

- Branch to subroutine (r14 serves as a link register)
- Nested subroutines
- BL SUB1
  - SUB1: ; save work and link register
  - STMFD r13, {r0-r2, r14}
  - BL SUB2
  - LDMFD r13, {r0-r2, pc}
  - SUB2: MOV pc, r14 ; copy r14 into r15
**Supervisor calls**

- Supervisor is a program which operates at a privileged level – it can do things that a user-level program cannot do directly
  - Example: send text to the display
- ARM ISA includes SWI (SoftWare Interrupt)

```
output r0
SWI WriteC
SWI Exit
```

**Jump tables**

- Call one of a set of subroutines depending on a value computed by the program

```
BL JTAB ...
JTAB: CMP r0, #0
BEQ SUB0
CMP r0, #1
BEQ SUB1
CMP r0, #2
BEQ SUB2
...
```

Note: slow when the list is long, and all subroutines are equally frequent

---

**Hello ARM World!**

```
AREA HelloW, CODE, READONLY ; declare code area
SWI_WriteC EQU &0 ; output character in r0
SWI_Exit EQU &11 ; finish program
ENTRY ; code entry point
START: ADR r1, TEXT ; r1 <- Hello ARM World!
LOOP: LDRB r0, [r1], #1 ; get the next byte
CMP r0, #0 ; check for text end
SWI WriteC ; if not end of string, print
BNE LOOP
SWI Exit ; end of execution
TEXT = "Hello ARM World", #0a, #0d, 0
END
```