Chapter 2

ARM Processor Core and Instruction Sets

Prof. Tian-Sheuen Chang
Outline

- Processor programming model
- 32-bit instruction set
- 16-bit instruction set
- ARM processor core
- Software development
Processor Programming Model
ARM Ltd

- ARM was originally developed at Acron Computer Limited of Cambridge, England between 1983 and 1985
  - 1980, RISC concept at Stanford and Berkeley universities
  - first RISC processor for commercial use
- 1990 Nov, ARM Ltd was founded
- ARM cores
  licensed to partners who fabricate and sell to customers
- Technologies assist to design in the ARM application
  - Software tools, boards, debug hardware, application software, bus architectures, peripherals etc…
ARM Architecture vs. Berkeley RISC

- Features used
  - load/store architecture
  - fixed-length 32-bit instructions
  - 3-address instruction formats

- Features unused
  - register windows \(\Rightarrow\) costly
    - use shadow registers in ARM
  - delayed branch \(\Rightarrow\) not well to superscalar
    - badly with branch prediction
  - single-cycle execution of all instructions
    - most single-cycle
  - memory access
    - multiple cycles when no separate data and instruction memory support
    - auto-indexing addressing modes
Data Size and Instruction set

- ARM processor is a 32-bit architecture
- Most ARM’s implement two instruction sets
  - 32-bit ARM instruction set
  - 16-bit Thumb instruction set
Data Types

- ARM processor supports 6 data types
  - 8-bits signed and unsigned bytes
  - 16-bits signed and unsigned half-words, aligned on 2-byte boundaries
  - 32-bits signed and unsigned words, aligned on 4-byte boundaries
- ARM instructions are all 32-bit words, word-aligned
  Thumb instructions are half-words, aligned on 2-byte boundaries
- Internally all ARM operations are on 32-bit operands; the shorter data types are only supported by data transfer instructions. When a byte is loaded from memory, it is zero- or sign-extended to 32 bits
- ARM coprocessor supports floating-point values
Programming Model

• Each instruction can be viewed as performing a defined transformation of the states
  visible registers
  invisible registers
  system memory
  user memory
Processor Modes

- ARM has seven basic operating modes
- Mode changes by software control or external interrupts

<table>
<thead>
<tr>
<th>CPRS[4:0]</th>
<th>Mode</th>
<th>Use</th>
<th>Registers</th>
</tr>
</thead>
<tbody>
<tr>
<td>10000</td>
<td>User</td>
<td>Normal user code</td>
<td>User</td>
</tr>
<tr>
<td>10001</td>
<td>FIQ</td>
<td>Processing fast interrupts</td>
<td>_fiq</td>
</tr>
<tr>
<td>10010</td>
<td>IRQ</td>
<td>Processing standard interrupts</td>
<td>_irp</td>
</tr>
<tr>
<td>10011</td>
<td>SVC</td>
<td>Processing software interrupts (SWIs)</td>
<td>_svc</td>
</tr>
<tr>
<td>10111</td>
<td>Abort</td>
<td>Processing memory faults</td>
<td>_abt</td>
</tr>
<tr>
<td>11011</td>
<td>Undef</td>
<td>Handling undefined instruction traps</td>
<td>_und</td>
</tr>
<tr>
<td>11111</td>
<td>System</td>
<td>Running privileged operating system</td>
<td>user</td>
</tr>
</tbody>
</table>
Privileged Modes

- Most programs operate in user mode. ARM has other privileges operating modes which are used to handle exceptions, supervisor calls (software interrupts), and system mode.
- More access rights to memory systems and coprocessors.
- Current operating mode is defined by CPSR[4:0].
Supervisor Mode

- Having some protective privileges
- System-level function (transaction with the outside world) can be accessed through specified supervisor calls
- Usually implemented by software interrupt (SWI)
The Registers

- ARM has 37 registers, all of which are 32 bits long
  - 1 dedicated program counter
  - 1 dedicated current program status register
  - 5 dedicated saved program status registers
  - 30 general purpose registers
- The current processor mode governs which bank is accessible
  - each mode can access
    - a particular set of r0-r12 registers
    - a particular r13 (stack pointer, SP) and r14 (link register, LR)
    - the program counter, r15 (PC)
    - the current program status register, CPSR
  - privileged modes (except System) can access
    - a particular SPSR (saved program status register)
Register Banking

Current Visible Registers

User Mode
r0
r1
r2
r3
r4
r5
r6
r7
r8
r9
r10
r11
r12
r13 (sp)
r14 (lr)
r15 (pc)
cpsr

Banked out Registers

FIQ
IRQ
SVC
Undef
Abort
r8
r9
r10
r11
r12
r13 (sp)
r14 (lr)
r13 (sp)
r14 (lr)
r13 (sp)
r14 (lr)
r13 (sp)
r14 (lr)
r13 (sp)
r14 (lr)
# Registers Organization Summary

<table>
<thead>
<tr>
<th>User</th>
<th>FIQ</th>
<th>IRQ</th>
<th>SVC</th>
<th>Undef</th>
<th>Abort</th>
</tr>
</thead>
<tbody>
<tr>
<td>r0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>r1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>r2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>r3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>r4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>r5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>r6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>r7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>r8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>r9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>r10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>r11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>r12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>r13 (sp)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>r14 (lr)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>r15 (pc)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>r8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>r9</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>r10</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>r11</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>r12</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>r13 (sp)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>r14 (lr)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>r13 (sp)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>r14 (lr)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>r13 (sp)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>r14 (lr)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>r13 (sp)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>r14 (lr)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>r13 (sp)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>r14 (lr)</td>
</tr>
</tbody>
</table>

**Note:** System mode uses the User mode register set.
Program Counter (r15)

• When the processor is executing in ARM state:
  – all instructions are 32 bits wide
  – all instructions must be word-aligned
  – therefore the PC value is stored in bits [31:2] with bits [1:0] undefined (as instruction cannot be halfword or byte aligned)

• When the processor is executing in Thumb state:
  – all instructions are 16 bits wide
  – all instructions are must be halfword-aligned
  – therefore the PC value is stored in bits [31:1] with bit [0] undefined (as instruction cannot be byte-aligned)
### Program Status Registers (CPSR)

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>N</td>
</tr>
<tr>
<td>30</td>
<td>Z</td>
</tr>
<tr>
<td>29</td>
<td>C</td>
</tr>
<tr>
<td>28</td>
<td>V</td>
</tr>
<tr>
<td>27</td>
<td>Q</td>
</tr>
<tr>
<td>26</td>
<td>undefined</td>
</tr>
<tr>
<td>25</td>
<td>I</td>
</tr>
<tr>
<td>24</td>
<td>F</td>
</tr>
<tr>
<td>23</td>
<td>T</td>
</tr>
<tr>
<td>22</td>
<td>mode</td>
</tr>
</tbody>
</table>

- **Condition code flags**
  - N: Negative result from ALU
  - Z: Zero result from ALU
  - C: ALU operation Carried out
  - V: ALU operation overflowed

- **Sticky overflow flag – Q flag**
  - architecture 5TE only
  - indicates if saturation has occurred during certain operations

- **Interrupt disable bits**
  - I = 1, disables the IRQ
  - F = 1, disables the FIQ

- **T Bit**
  - architecture xT only
  - T = 0, processor in ARM state
  - T = 1, processor in Thumb state

- **Mode bits**
  - specify the processor mode
SPSRs

- Each privileged mode (except system mode) has associated with it a Save Program Status Register, or SPSR
- This SPSR is used to save the state of CPSR (Current Program Status Register) when the privileged mode is entered in order that the user state can be fully restored when the user process is resumed
- Often the SPSR may be untouched from the time the privileged mode is entered to the time it is used to restore the CPSR, but if the privileged supervisor calls to itself) then the SPSR must be copied into a general register and saved
Exceptions

- Exceptions are usually used to handle unexpected events which arise during the execution of a program, such as interrupts or memory faults, also cover software interrupts, undefined instruction traps, and the system reset.
- Three groups:
  1. generated as the direct effect of executing an instruction: software interrupts, undefined instructions, prefetch abort (memory fault)
  2. generated as the side-effect of an instruction: data aborts
  3. generated externally: reset, IRQ, FIQ
Exception Entry (1/2)

- When an exception arises, ARM completes the current instruction as best it can (except that reset exception terminates the current instruction immediately) and then departs from the current instruction sequence to handle the exception which starts from a specific location (exception vector)

- Processor performs the following sequence
  - change to the operating mode corresponding to the particular exception
  - save the address of the instruction following the exception entry instruction in r14 of the new mode
  - save the old value of CPSR in the SPSR of the new mode
  - disable IRQs by setting bit of the CPSR, and if the exception is a fast interrupt, disable further faster interrupt by setting bit of the CPSR
Exception Entry (2/2)

- force the PC to begin executing at the relevant vector address

<table>
<thead>
<tr>
<th>Exception</th>
<th>Mode</th>
<th>Vector address</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reset</td>
<td>SVC</td>
<td>0x00000000</td>
</tr>
<tr>
<td>Undefined instruction</td>
<td>UND</td>
<td>0x00000004</td>
</tr>
<tr>
<td>Software interrupt (SWI)</td>
<td>SVC</td>
<td>0x00000008</td>
</tr>
<tr>
<td>Prefetch abort (instruction fetch memory fault)</td>
<td>Abort</td>
<td>0x0000000C</td>
</tr>
<tr>
<td>Data abort (data access memory fault)</td>
<td>Abort</td>
<td>0x00000010</td>
</tr>
<tr>
<td>IRQ (normal interrupt)</td>
<td>IRQ</td>
<td>0x00000018</td>
</tr>
<tr>
<td>FIQ (fast interrupt)</td>
<td>FIQ</td>
<td>0x0000001C</td>
</tr>
</tbody>
</table>

- Normally the vector address contains a branch to the relevant routine, though the FIQ code can start immediately
- Two banked registers in each of the privilege modes are used to hold the return address and stack pointer
Exception Return (1/3)

• Once the exception has been handled, the user task is normally resumed

• The sequence is
  – any modified user registers must be restored from the handler’s stack
  – CPSR must be restored from the appropriate SPSR
  – PC must be changed back to the relevant instruction address

• The last two steps happen atomically as part of a single instruction
Exception Return (2/3)

• When the return address has been kept in the banked r14
  – to return from a SWI or undefined instruction trap
    MOVS PC, r14
  – to return from an IRQ, FIQ or prefetch abort
    SUBS PC, r14, #4
  – To return from a data abort to retry the data access
    SUBS PC, r14, #8
  – ‘S’ signifies when the destination register is the PC
Exception Return (3/3)

• When the return address has been saved onto a stack

```
LDMFD r13!,{r0-r3,PC}^ ;restore and return
```

- ‘^’ indicates that this is a special form of the instruction the CPSR is restored at the same time that the PC is loaded from memory, which will always be the last item transferred from memory since the registers are loaded in increasing order.
Exception Priorities

- Priority order
  1. reset (highest priority)
  2. data abort
  3. FIQ
  4. IRQ
  5. prefetch abort
  6. SWI, undefined instruction
Memory Organization

- Word, half-word alignment (xxxxoo or xxxxxxo)
- ARM can be set up to access data in either little-endian or big-endian format
Features of the ARM Instruction Set

- Load-store architecture
  process values which are in registers
  load, store instructions for memory data accesses
- 3-address data processing instructions
- Conditional execution of every instruction
- Load and store multiple registers
- Shift, ALU operation in a single instruction
- Open instruction set extension through the coprocessor instruction
- Very dense 16-bit compressed instruction set (Thumb)
Coprocessors

- Up to 16 coprocessor can be defined
- Expands the ARM instruction set
- ARM uses them for “internal functions” so as not to enforce a particular memory map (e.g., cp15 is the ARM cache controller)
- Usually better for system designers to use memory mapped peripherals
  - easier to implement
Thumb

- Thumb is a 16-bit instruction set
  - optimized for code density from C code
  - improved performance from narrow memory
  - subset of the functionality of the ARM instruction set
- Core has two execution states – ARM and Thumb
  - switch between them using BX instruction

For most instruction generated by compiler:
- Conditional execution is not used
- Source and destination registers identical
- Only Low registers used
- Constants are of limited size
- Inline barrel shifter not used
Average Thumb Code Sizes
ARM and Thumb Performance

Dhrystone 2.1/sec at 20MHz

32-bit | 16-bit | 16-bit with 32-bit stack

Memory Width (Zero Wait State)

- ARM
- Thumb
I/O System

• ARM handles input/output peripherals as memory-mapped with interrupt support.
• Internal registers in I/O devices as addressable locations within ARM’s memory map. Read and written using load-store instructions.
• Interrupt by normal interrupt (IRQ) or fast interrupt (FIQ) with higher priority. Input signals are level-sensitive and maskable.
• May include Direct Memory Access (DMA) hardware.
ARM Exceptions

• Supports interrupts, traps, supervisor calls
• When an exception occurs, the ARM:
  – copies CPSR into SPSR_<mode>
  – sets appropriate CPSR bits
    • if core currently in Thumb state then
      ARM state is entered
    • mode field bits
    • interrupt disable bits (if appropriate)
  – stores the return address in LR_<mode>
  – set pc to vector address
• To return, exception handler needs to:
  – restore CPSR from SPSR_<mode>
  – restore PC from LR_<mode>
This can only be done in ARM state

<table>
<thead>
<tr>
<th>Vector Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vector table can be at 0xffff0000 on ARM720T and on ARM9/10 family devices</td>
</tr>
</tbody>
</table>
ARM Exceptions

- Exception handler use r13_<mode> which will normally have been initialized to point to a dedicated stack in memory, to save some user registers for use as work registers
ARM Processor Cores

- ARM Processor core + cache + MMU → ARM CPU cores
- ARM6 → ARM7 (3V operation, 50-100MHz for .25µ or .18 µ)
  - T : Thumb 16-bit compressed instruction set
  - D : on-chip Debug support, enabling the processor to halt in response to a debug request
  - M : enhanced Multiplier, 64-bit result
  - I : embedded ICE hardware, give on-chip breakpoint and watchpoint support
ARM Processor Cores

• ARM 8 → ARM 9 → ARM 10

• ARM 9
  – 5-stage pipeline (130 MHz or 200MHz)
  – using separate instruction and data memory ports

• ARM 10 (1998. Oct.)
  – high performance, 300 MHz
  – multimedia digital consumer applications
  – optional vector floating-point unit
ARM Architecture Versions (1/5)

• Version 1
  – the first ARM processor, developed at Acorn Computers Limited 1983-1985
  – 26-bit addressing, no multiply or coprocessor support

• Version 2
  – sold in volume in the Acorn Archimedes
  – 26-bit addressing, including 32-bit result multiply and coprocessor

• Version 2a
  – coprocessor 15 as the system control coprocessor to manage cache
  – add the atomic load store (SWP) instruction
ARM Architecture Versions (2/5)

• Version 3
  – first ARM processor designed by ARM Limited (1990)
  – ARM6 (macro cell)
    ARM60 (stand-alone processor)
    ARM600 (an integrated CPU with on-chip cache, MMU, write buffer)
    ARM610 (used in Apple Newton)
  – 32-bit addressing, separate CPSR and SPSRs
  – add the undefined and abort modes to allow coprocessor emulation and virtual memory support in supervisor mode

• Version 3M
  – introduce the signed and unsigned multiply and multiply-accumulate instructions that generate the full 64-bit result
ARM Architecture Versions (3/5)

• Version 4
  – add the signed, unsigned half-word and signed byte load and store instructions
  – reserve some of SWI space for architecturally defined operations
  – system mode is introduced

• Version 4T
  – 16-bit Thumb compressed form of the instruction set is introduced
ARM Architecture Versions (4/5)

- **Version 5T**
  - introduced recently, a superset of version 4T adding the BLX, CLZ and BRK instructions

- **Version 5TE**
  - add the signal processing instruction set extension
## ARM Architecture Versions (5/5)

<table>
<thead>
<tr>
<th>Core</th>
<th>Architecture</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARM1</td>
<td>v1</td>
</tr>
<tr>
<td>ARM2</td>
<td>v2</td>
</tr>
<tr>
<td>ARM2as, ARM3</td>
<td>v2a</td>
</tr>
<tr>
<td>ARM6, ARM600, ARM610</td>
<td>v3</td>
</tr>
<tr>
<td>ARM7, ARM700, ARM710</td>
<td>v3</td>
</tr>
<tr>
<td>ARM7TDMI, ARM710T, ARM720T, ARM740T</td>
<td>v4T</td>
</tr>
<tr>
<td>StrongARM, ARM8, ARM810</td>
<td>v4</td>
</tr>
<tr>
<td>ARM9TDMI, ARM920T, ARM940T</td>
<td>V4T</td>
</tr>
<tr>
<td>ARM9E-S</td>
<td>v5TE</td>
</tr>
<tr>
<td>ARM10TDMI, ARM1020E</td>
<td>v5TE</td>
</tr>
</tbody>
</table>
32-bit Instruction Set
• ARM assembly language program
  – ARM development board or ARM emulator

• ARM instruction set
  – standard ARM instruction set
  – a compressed form of the instruction set, a subset of the full ARM instruction set is encoded into 16-bit instructions - Thumb instruction
  – some ARM cores support instruction set extensions to enhance signal processing capabilities
Instructions

- Data processing instructions
- Data transfer instructions
- Control flow instructions
### ARM Instruction Set Summary (1/4)

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADC</td>
<td>Add with carry</td>
<td>Rd:=Rn+Op2+Carry</td>
</tr>
<tr>
<td>ADD</td>
<td>Add</td>
<td>Rd:=Rn+Op2</td>
</tr>
<tr>
<td>AND</td>
<td>AND</td>
<td>Rd:=Rn AND Op2</td>
</tr>
<tr>
<td>B</td>
<td>Branch</td>
<td>R15:=address</td>
</tr>
<tr>
<td>BIC</td>
<td>Bit Clear</td>
<td>Rd:=Rn AND NOT Op2</td>
</tr>
<tr>
<td>BL</td>
<td>Branch with Link</td>
<td>R14:=R15</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R15:=address</td>
</tr>
<tr>
<td>BX</td>
<td>Branch and Exchange</td>
<td>R15:=Rn</td>
</tr>
<tr>
<td></td>
<td></td>
<td>T bit:=Rn[0]</td>
</tr>
<tr>
<td>CDP</td>
<td>Coprocessor Data Processing</td>
<td>(Coprocessor-specific)</td>
</tr>
<tr>
<td>CMN</td>
<td>Compare Negative</td>
<td>CPSR flags:=Rn+Op2</td>
</tr>
<tr>
<td>CMP</td>
<td>Compare</td>
<td>CPSR flags:=Rn-Op2</td>
</tr>
<tr>
<td>Mnemonic</td>
<td>Instruction</td>
<td>Action</td>
</tr>
<tr>
<td>----------</td>
<td>-------------</td>
<td>--------</td>
</tr>
<tr>
<td>EOR</td>
<td>Exclusive OR</td>
<td>Rd:=Rn^Op2</td>
</tr>
<tr>
<td>LDC</td>
<td>Load Coprocessor from memory</td>
<td>(Coprocessor load)</td>
</tr>
<tr>
<td>LDM</td>
<td>Load multiple registers</td>
<td>Stack Manipulation (Pop)</td>
</tr>
<tr>
<td>LDR</td>
<td>Load register from memory</td>
<td>Rd:=(address)</td>
</tr>
<tr>
<td>MCR</td>
<td>Move CPU register to coprocessor register</td>
<td>CRn:=rRn{&lt;op&gt;cRm}</td>
</tr>
<tr>
<td>MLA</td>
<td>Multiply Accumulate</td>
<td>Rd:=(Rm*Rs)+Rn</td>
</tr>
<tr>
<td>MOV</td>
<td>Move register or constant</td>
<td>Rd:=Op2</td>
</tr>
<tr>
<td>MRC</td>
<td>Move from coprocessor register to CPU register</td>
<td>rRn:=cRn{&lt;op&gt;cRm}</td>
</tr>
<tr>
<td>MRS</td>
<td>Move PSR status/flags to register</td>
<td>Rn:=PSR</td>
</tr>
<tr>
<td>MSR</td>
<td>Move register to PSR status/flags</td>
<td>PSR:=Rm</td>
</tr>
</tbody>
</table>
## ARM Instruction Set Summary (3/4)

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>MUL</td>
<td>Multiply</td>
<td>$R_d := R_m \times R_s$</td>
</tr>
<tr>
<td>MVN</td>
<td>Move negative register</td>
<td>$R_d := \neg R_{op2}$</td>
</tr>
<tr>
<td>ORR</td>
<td>OR</td>
<td>$R_d := R_n \lor R_{op2}$</td>
</tr>
<tr>
<td>RSB</td>
<td>Reverse Subtract</td>
<td>$R_d := R_{op2} - R_n$</td>
</tr>
<tr>
<td>RSC</td>
<td>Reverse Subtract with Carry</td>
<td>$R_d := R_{op2} - R_n - 1 + \text{Carry}$</td>
</tr>
<tr>
<td>SBC</td>
<td>Subtract with Carry</td>
<td>$R_d := R_n - R_{op2} - 1 + \text{Carry}$</td>
</tr>
<tr>
<td>STC</td>
<td>Store coprocessor register to memory</td>
<td>address := cRn</td>
</tr>
<tr>
<td>STM</td>
<td>Store Multiple</td>
<td>Stack manipulation (Push)</td>
</tr>
</tbody>
</table>
## ARM Instruction Set Summary (4/4)

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>STR</td>
<td>Store register to memory</td>
<td><code>&lt;address&gt;</code>:=Rd</td>
</tr>
<tr>
<td>SUB</td>
<td>Subtract</td>
<td>Rd:=Rn-Op2</td>
</tr>
<tr>
<td>SWI</td>
<td>Software Interrupt</td>
<td>OS call</td>
</tr>
<tr>
<td>SWP</td>
<td>Swap register with memory</td>
<td>Rd:=[Rn]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[Rn]:=Rm</td>
</tr>
<tr>
<td>TEQ</td>
<td>Test bitwise equality</td>
<td>CPSR flags:=Rn EOR Op2</td>
</tr>
<tr>
<td>TST</td>
<td>Test bits</td>
<td>CPSR flags:=Rn AND Op2</td>
</tr>
</tbody>
</table>
### ARM Instruction Set Format

#### Data Processing / PSR Transfer
- Multiply
- Multiply Long

#### Single Data Swap
- Branch and Exchange
- Halfword Data Transfer: register offset
- Halfword Data Transfer: immediate offset

#### Single Data Transfer
- Undefined
- Block Data Transfer
- Branch

#### Coprocessor Data Transfer
- Coprocessor Data Operation
- Coprocessor Register Transfer
- Software interrupt

<table>
<thead>
<tr>
<th>Cond</th>
<th>Opcode</th>
<th>S</th>
<th>Rn</th>
<th>Rd</th>
<th>Rs</th>
<th>Operand 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000000A</td>
<td>S</td>
<td>Rd</td>
<td>Rn</td>
<td>Rs</td>
<td>1001</td>
<td>Rm</td>
</tr>
<tr>
<td>0000001UAS</td>
<td>RdHi</td>
<td>RdLo</td>
<td>Rn</td>
<td>1001</td>
<td>Rm</td>
<td></td>
</tr>
<tr>
<td>00010B00</td>
<td>Rn</td>
<td>Rd</td>
<td>000010001</td>
<td>Rm</td>
<td></td>
<td></td>
</tr>
<tr>
<td>000100111111</td>
<td>11110001</td>
<td>Rn</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0000PU0WL</td>
<td>Rn</td>
<td>Rd</td>
<td>00001S H1</td>
<td>Rm</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0000P1WL</td>
<td>Rn</td>
<td>Rd</td>
<td>Offset 1S H1</td>
<td>Offset</td>
<td></td>
<td></td>
</tr>
<tr>
<td>011IPUBWL</td>
<td>Rn</td>
<td>Rd</td>
<td>Offset</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>011</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>100PUSW</td>
<td>Rn</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>101L</td>
<td></td>
<td>Offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>110PUNWL</td>
<td>Rn</td>
<td>CRd</td>
<td>CP#</td>
<td>Offset</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1110CP Opc</td>
<td>CRn</td>
<td>CRd</td>
<td>CP#</td>
<td>CP</td>
<td>CRm</td>
<td></td>
</tr>
<tr>
<td>1110CP Opc L</td>
<td>CRn</td>
<td>Rd</td>
<td>CP#</td>
<td>CP</td>
<td>CRm</td>
<td></td>
</tr>
<tr>
<td>1111</td>
<td></td>
<td></td>
<td></td>
<td>Ignored by processor</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0*
Data Processing Instructions

- Consist of
  - arithmetic (ADD, SUB, RSB)
  - logical (BIC, AND)
  - compare (CMP, TST)
  - register movement (MOV, MVN)

- All operands are 32-bit wide; come from registers or specified as literal in the instruction itself
- Second operand sent to ALU via barrel shifter
- 32-bit result placed in register; long multiply instruction produces 64-bit result
- 3-address instruction format
Conditional Execution

- Most instruction sets only allow branches to be executed conditionally.
- However by reusing the condition evaluation hardware, ARM effectively increases number of instructions.
  - all instructions contain a condition field which determines whether the CPU will execute them
  - non-executed instructions still take up 1 cycle
    - to allow other stages in the pipeline to complete
- This reduces the number of branches which would stall the pipeline
  - allows very dense in-line code
  - the time penalty of not executing several conditional instructions is frequently less than overhead of the branch or subroutine call that would otherwise be needed
## Conditional Execution

Each of the 16 values causes the instruction to be executed or skipped according to the N, Z, C, V flags in the CPSR.

<table>
<thead>
<tr>
<th>Opcode  [31:28]</th>
<th>Mnemonic extension</th>
<th>Interpretation</th>
<th>Status flag state for execution</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>EQ</td>
<td>Equal / equals zero</td>
<td>Z set</td>
</tr>
<tr>
<td>0001</td>
<td>NE</td>
<td>Not equal</td>
<td>Z clear</td>
</tr>
<tr>
<td>0010</td>
<td>CS/HS</td>
<td>Carry set / unsigned higher or same</td>
<td>C set</td>
</tr>
<tr>
<td>0011</td>
<td>CC/LO</td>
<td>Carry clear / unsigned lower</td>
<td>C clear</td>
</tr>
<tr>
<td>0100</td>
<td>MI</td>
<td>Minus / negative</td>
<td>N set</td>
</tr>
<tr>
<td>0101</td>
<td>PL</td>
<td>Plus / positive or zero</td>
<td>N clear</td>
</tr>
<tr>
<td>0110</td>
<td>VS</td>
<td>Overflow</td>
<td>V set</td>
</tr>
<tr>
<td>0111</td>
<td>VC</td>
<td>No overflow</td>
<td>V clear</td>
</tr>
<tr>
<td>1000</td>
<td>HI</td>
<td>Unsigned higher</td>
<td>C set and Z clear</td>
</tr>
<tr>
<td>1001</td>
<td>LS</td>
<td>Unsigned lower or same</td>
<td>C clear or Z set</td>
</tr>
<tr>
<td>1010</td>
<td>GE</td>
<td>Signed greater than or equal</td>
<td>N equals V</td>
</tr>
<tr>
<td>1011</td>
<td>LT</td>
<td>Signed less than</td>
<td>N is not equal to V</td>
</tr>
<tr>
<td>1100</td>
<td>GT</td>
<td>Signed greater than</td>
<td>Z clear and N equals V</td>
</tr>
<tr>
<td>1101</td>
<td>LE</td>
<td>Signed less than or equal</td>
<td>Z set or N is not equal to V</td>
</tr>
<tr>
<td>1110</td>
<td>AL</td>
<td>Always</td>
<td>any</td>
</tr>
<tr>
<td>1111</td>
<td>NV</td>
<td>Never (do not use!)</td>
<td>none</td>
</tr>
</tbody>
</table>
Using and Updating the Condition Field

• To execute an instruction conditionally, simply postfix it with the appropriate condition:
  – for example an add instruction takes the form:
    • ADD r0,r1,r2 ;r0:=r1+r2 (ADDAL)
  – to execute this only if the zero flag is set:
    • ADDEQ r0,r1,r2 ;r0:=r1+r2 iff zero flag is set

• By default, data processing operations do not affect the condition flags
  – with comparison instructions this is the only effect

• To cause the condition flags to be updated, the S bit of the instruction needs to be set by postfixing the instruction (and any condition code) with an “S”.
  – for example to add two numbers and set the condition flags:
    • ADDS r0,r1,r2 ;r0:=r1+r2 and set flags
Data Processing Instructions

• Simple register operands
• Immediate operands
• Shifted register operands
• Multiply
Simple Register Operands (1/2)

• Arithmetic Operations
  ADD r0,r1,r2 ; r0 := r1 + r2
  ADC r0,r1,r2 ; r0 := r1 + r2 + C
  SUB r0,r1,r2 ; r0 := r1 - r2
  SBC r0,r1,r2 ; r0 := r1 - r2 + C - 1
  RSB r0,r1,r2 ; r0 := r2 - r1, reverse subtraction
  RSC r0,r1,r2 ; r0 := r2 - r1 + C - 1

  – by default, data processing operations do not affect the condition flags

• Bit-wise Logical Operations
  AND r0,r1,r2 ; r0 := r1 AND r2
  ORR r0,r1,r2 ; r0 := r1 OR r2
  EOR r0,r1,r2 ; r0 := r1 XOR r2
  BIC r0,r1,r2 ; r0 := r1 AND (NOT r2), bit clear
Simple Register Operands (2/2)

- **Register Movement Operations**
  - omit 1st source operand from the format

  - `MOV r0,r2 ; r0 := r2`
  - `MVN r0,r2 ; r0 := NOT r2, move 1's complement`

- **Comparison Operations**
  - not produce result; omit the destination from the format
  - just set the condition code bits (N, Z, C and V) in CPSR

  - `CMP r1,r2 ; set cc on r1-r2, compare`
  - `CMN r1,r2 ; set cc on r1+r2, compare negated`
  - `TST r1,r2 ; set cc on r1 AND r2, bit test`
  - `TEQ r1,r2 ; set cc on r1 XOR r2, test equal`
Immediate Operands

• Replace the second source operand with an immediate operand, which is a literal constant, preceded by “#”

\[
\begin{align*}
\text{ADD} & \ r3, r3, #1 ; r3 := r3 + 1 \\
\text{AND} & \ r8, r7, #&FF ; r8 := r7[7:0], &: \text{hexadecimal}
\end{align*}
\]

• Immediate = (0~255)\(\times 2^{2n}\)

where n is 0-15 4-bit value
Shifted Register Operands

- ADD r3, r2, r1, LSL#3
  \( r3 := r2 + 8 \times r1 \)
  - a single instruction executed in a single cycle
- LSL: Logical shift left by 0 to 31 places, 0 filled at the lsb end
- LSR, ASL (Arithmetic Shift Left), ASR, ROR (Rotate Right), RRX (Rotate Right eXtended by 1 place)
- ADD r5, r5, r3, LSL r2
  \( r5 := r5 + r3 \times 2^r2 \)
- MOV r12, r4, ROR r3
  \( r12 := r4 \) rotated right by value of r3
Using the Barrel Shifter: the 2nd Operand

- Register, optionally with shift operation applied.
  - Shift value can be either:
    - 5-bit unsigned integer
    - Specified in bottom byte of another register
  - Used for multiplication by constant

Immediate value
- 8-bit number, with a range of 0-255
  - Rotated right through even number of positions
- Allows increased range of 32-bit constants to be loaded directly into registers
Multiply

\[ \text{MUL } r4, r3, r2 \quad ;r4 := (r3 \times r2)_{[31:0]} \]

- Multiply-Accumulate

\[ \text{MLA } r4, r3, r2, r1 \quad ;r4 := (r3 \times r2 + r1)_{[31:0]} \]
Multiplication by a Constant

• Multiplication by a constant equals to a \((\text{power of 2}) \pm 1\) can be done in a single cycle
  – Using MOV, ADD or RSBs with an inline shift
• Example: \(r_0 = r_1 \times 5\)
• Example: \(r_0 = r_1 + (r_1 \times 4)\)
  
  ```
  ADD r0, r1, r1, LSL #2
  ```
• Can combine several instructions to carry out other multiplies
• Example: \(r_2 = r_3 \times 119\)
• Example: \(r_2 = r_3 \times 17 \times 7\)
• Example: \(r_2 = r_3 \times (16+1) \times (8-1)\)
  
  ```
  ADD r2, r3, r3, LSL #4 ; r2 := r3 \times 17
  RSB r2, r2, r2, LSL #3 ; r2 := r2 \times 7
  ```
Data Processing Instructions (1/3)

- `<op>行政部门{<cond>行政部门}>行政部门{S} Rd,Rn,#<32-bit immediate>`
- `<op>行政部门{<cond>行政部门}>行政部门{S} Rd,Rn,Rm,{<shift>行政部门}>行政部门`  
  - omit Rn when the instruction is monadic (MOV, MVN)  
  - omit Rd when the instruction is a comparison, producing only condition code outputs (CMP, CMN, TST, TEQ)  
  - `<shift>` specifies the shift type (LSL, LSR, ASL, ASR, ROR or RRX) and in all cases but RRX, the shift amount which may be a 5-bit immediate (`# < # shift行政部门>行政部门`) or a register Rs

- **3-address format**  
  - 2 source operands and 1 destination register  
  - one source is always a register, the second may be a register, a shifted register or an immediate value
## Data Processing Instructions (2/3)

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>AND</td>
<td>Logical bit-wise AND</td>
<td>Rd := Rn AND Op2</td>
</tr>
<tr>
<td>0001</td>
<td>EOR</td>
<td>Logical bit-wise exclusive OR</td>
<td>Rd := Rn EOR Op2</td>
</tr>
<tr>
<td>0010</td>
<td>SUB</td>
<td>Subtract</td>
<td>Rd := Rn – Op2</td>
</tr>
<tr>
<td>0011</td>
<td>RSB</td>
<td>Reverse subtract</td>
<td>Rd := Op2 – Rn</td>
</tr>
<tr>
<td>0100</td>
<td>ADD</td>
<td>Add</td>
<td>Rd := Rn + Op2</td>
</tr>
<tr>
<td>0101</td>
<td>ADC</td>
<td>Add with carry</td>
<td>Rd := Rn + Op2 + C</td>
</tr>
<tr>
<td>0110</td>
<td>SBC</td>
<td>Subtract with carry</td>
<td>Rd := Rn – Op2 + C – 1</td>
</tr>
<tr>
<td>0111</td>
<td>RSC</td>
<td>Reverse subtract with carry</td>
<td>Rd := Op2 – Rn + C – 1</td>
</tr>
<tr>
<td>1000</td>
<td>TST</td>
<td>Test</td>
<td>Scc on Rn AND Op2</td>
</tr>
<tr>
<td>1001</td>
<td>TEQ</td>
<td>Test equivalence</td>
<td>Scc on Rn EOR Op2</td>
</tr>
<tr>
<td>1010</td>
<td>CMP</td>
<td>Compare</td>
<td>Scc on Rn – Op2</td>
</tr>
<tr>
<td>1011</td>
<td>CMN</td>
<td>Compare negated</td>
<td>Scc on Rn + Op2</td>
</tr>
<tr>
<td>1100</td>
<td>ORR</td>
<td>Logical bit-wise OR</td>
<td>Rd := Rn OR Op2</td>
</tr>
<tr>
<td>1101</td>
<td>MOV</td>
<td>Move</td>
<td>Rd := Op2</td>
</tr>
<tr>
<td>1110</td>
<td>BIC</td>
<td>Bit clear</td>
<td>Rd := Rn AND NOT Op2</td>
</tr>
<tr>
<td>1111</td>
<td>MVN</td>
<td>Move negated</td>
<td>Rd := NOT Op2</td>
</tr>
</tbody>
</table>
Data Processing Instructions (3/3)

- Allows direct control of whether or not the condition codes are affected by S bit (condition code unchanged when S=0)
  - N=1 if the result is negative; 0 otherwise
  - (i.e. N=bit 31 of the result)
  - Z=1 if the result is zero; 0 otherwise
  - C= carry out from the ALU when ADD, ADC, SUB, SBC, RSB, RSC, CMP, CMN; carry out from the shifter
  - V=1 if overflow from bit 30 to bit 31; 0 if no overflow
    (V is preserved in non-arithmetic operations)

- PC may be used as a source operand (address of the instruction plus 8) except when a register-specified shift amount is used
- PC may be specified as the destination register, the instruction is a form of branch (return from a subroutine)
Examples

- ADD r5, r1, r3
- ADD Rs, PC, #offset ; PC is ADD address + 8
- **Decrement r2 and check for zero**
  
  SUBS r2, r2#1 ; dec r2 and set cc
  
  BEQ LABEL

  ...

- **Multiply r0 by 5**
  
  ADD r0, r0, r0, LSL #2

- **A subroutine to multiply r0 by 10**
  
  MOV r0, #3
  
  BL TIMES10

  ......

  TIMES10 MOV r0, r0, LSL #1 ; *2
  
  ADD r0, r0, r0, LSL #2 ; *5
  
  MOV PC, r14 ; return

- **Add a 64-bit integer in r1, r0 to one in r3, r2**
  
  ADDs r2, r2, r0
  
  ADC r3, r3, r1
Multiply Instructions (1/2)

- 32-bit Product (Least Significant)
  
  \[
  \text{MUL}\{\text{<cond>}\}\{S\} \ Rd, Rm, Rs \\
  \text{MLA}\{\text{<cond>}\}\{S\} \ Rd, Rm, Rs, Rn
  \]

- 64-bit Product
  
  \[
  \text{<mul>}\{\text{<cond>}\}\{S\} \ RdHi, RdLo, Rm, Rs \\
  \text{<mul>} \text{ is (UMULL, UMLAC, SMULL, SMLAL)}
  \]

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>MUL</td>
<td>Multiply (32-bit result)</td>
<td>(Rd := (Rm \times Rs) [31:0])</td>
</tr>
<tr>
<td>001</td>
<td>MLA</td>
<td>Multiply-accumulate (32-bit result)</td>
<td>(Rd := (Rm \times Rs + Rn) [31:0])</td>
</tr>
<tr>
<td>100</td>
<td>UMULL</td>
<td>Unsigned multiply long</td>
<td>(RdHi:RdLo := Rm \times Rs)</td>
</tr>
<tr>
<td>101</td>
<td>UMLAL</td>
<td>Unsigned multiply-accumulate long</td>
<td>(RdHi:RdLo := Rm \times Rs)</td>
</tr>
<tr>
<td>110</td>
<td>SMULL</td>
<td>Signed multiply long</td>
<td>(RdHi:RdLo := Rm \times Rs)</td>
</tr>
<tr>
<td>111</td>
<td>SMLAL</td>
<td>Signed multiply-accumulate long</td>
<td>(RdHi:RdLo := Rm \times Rs)</td>
</tr>
</tbody>
</table>
Multiply Instructions (2/2)

- Accumulation is denoted by “+=“
- Example: form a scalar product of two vectors

```
MOV r11,#20 ;initialize loop counter
MOV r10,#0 ;initialize total
Loop LDR r0,[r8],#4 ;get first component
    LDR r1,[r9],#4 ;get second component
    MLA r10,r0,r1,r10
    SUBS r11,r11,#1
    BNE Loop
```
Count Leading Zeros (CLZ-V5T only)

- **CLZ{<cond>}$ \text{Rd,Rm}$**
  - set Rd to the number of the bit position of the most significant 1 in Rm; If Rm is zero, Rd=32
  - useful for renormalizing numbers

- **Example**
  
  MOV r0,#&100  
  CLZ r1,r0   ; r1 := 23

- **Example**
  
  CLZ r1,r2  
  MOV r2,r2,LSL r1
Data Transfer Instructions

- Three basic forms to move data between ARM registers and memory
  - single register load and store instruction
    - a byte, a 16-bit half word, a 32-bit word
  - multiple register load and store instruction
    - to save or restore workspace registers for procedure entry and exit
    - to copy blocks of data
  - single register swap instruction
    - a value in a register to be exchanged with a value in memory
    - to implement semaphores to ensure mutual exclusion on accesses
Single Register Data Transfer

- **Word transfer**
  \[ \text{LDR / STR} \]
- **Byte transfer**
  \[ \text{LDRB / STRB} \]
- **Halfword transfer**
  \[ \text{LDRH / STRH} \]
- **Load signed byte or halfword-load value and sign extended to 32 bits**
  \[ \text{LDRSB / LDRSH} \]
- **All of these can be conditionally executed by inserting the appropriate condition code after STR/LDR**
  \[ \text{LDREQB} \]
Addressing

• Register-indirect addressing
• Base-plus-offset addressing
  – base register
    r0-r15
  – offset, add or subtract an unsigned number
    immediate
    register (not PC)
    scaled register (only available for word and unsigned byte instructions)
• Stack addressing
• Block-copy addressing
Register-indirect Addressing

- Use a value in one register (base register) as a memory address
  
  LDR r0, [r1] ; r0 := mem32[r1]
  
  STR r0, [r1] ; mem32[r1] := r0

- Other forms
  - adding immediate or register offsets to the base address
Initializing an Address Pointer

• A small offset to the program counter, r15
  – ARM assembler has a “pseudo” instruction, `ADR`

• As an example, a program which must copy data
  from TABLE1 to TABLE2, both of which are near
  to the code

```assembly
COPY       ADR r1,TABLE1 ; r1 points to TABLE1
          ADR r2,TABLE2 ; r2 points to TABLE2

...  

TABLE1

...  ;<source>

TABLE2

...  ;<destination>
```
Single Register Load and Store

- A base register, an offset which may be another register or an immediate value

<table>
<thead>
<tr>
<th>Copy</th>
<th>ADR r1, TABLE1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ADR r2, TABLE2</td>
</tr>
<tr>
<td>Loop</td>
<td>LDR r0, [r1]</td>
</tr>
<tr>
<td></td>
<td>STR r0, [r2]</td>
</tr>
<tr>
<td></td>
<td>ADD r1, r1, #4</td>
</tr>
<tr>
<td></td>
<td>ADD r2, r2, #4</td>
</tr>
<tr>
<td></td>
<td>???</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
<tr>
<td>TABLE1</td>
<td>...</td>
</tr>
<tr>
<td>TABLE2</td>
<td>...</td>
</tr>
</tbody>
</table>
Base-plus-offset Addressing (1/3)

- **Pre-indexing**
  
  ```
  LDR r0, [r1, #4] ; r0 := mem32[r1+4]
  ```
  
  - offset up to 4K, added or subtracted, (#-4)

- **Post-indexing**
  
  ```
  LDR r0, [r1], #4 ; r0 := mem32[r1], r1 := r1+4
  ```
  
  - equivalent to a simple register-indirect load, but faster, less code space

- **Auto-indexing**
  
  ```
  LDR r0, [r1, #4]! ; r0 := mem32[r1+4], r1 := r1+4
  ```
  
  - no extra time, auto-indexing performed while the data is being fetched from memory
Base-plus-offset Addressing (2/3)

*Pre-indexed: STR r0, [r1, #12]

Auto-update from: STR r0, [r1, #12]!

*Post-indexed: STR r0, [r1], #12
Base-plus-offset Addressing (3/3)

- Copy
  ADR r1, TABLE1
  ADR r2, TABLE2

- Loop
  LDR r0, [r1], #4
  STR r0, [r2], #4
  ??
  ...

- TABLE1
  ...

- TABLE2
  ...

- A single unsigned byte load
  LDRB r0, [r1] ; r0 := mem_8[r1]
  – also support signed bytes, 16-bit half-word
Loading Constants (1/2)

• No single ARM instruction can load a 32-bit immediate constant directly into a register
  – all ARM instructions are 32-bit long
  – ARM instructions do not use the instruction stream as data

• The data processing instruction format has 12 bits available for operand 2
  – if used directly, this would only give a range of 4096

• Instead it is used to store 8-bit constants, give a range of 0~255

• These 8 bits can then be rotated right through an even number of positions (i.e. RORs by 0,2,4,…,30)

• This gives a much larger range of constants that can be directly loaded, though some constants will still need to be loaded from memory
Loading Constants (2/2)

- This gives us:
  - 0~255 \([0-0xff]\)
  - 256, 260, 264, ..., 1020 \([0x100-0x3fc, \text{step}4, 0x40-0xff \text{ ror} 30]\)
  - 1024, 1240, ..., 4080 \([0x400-0xff0, \text{step}16, 0x40-0xff \text{ ror} 28]\)
  - 4096, 4160, ..., 16320 \([0x1000-0x3fc0, \text{step}64, 0x40-0xff \text{ ror} 26]\)

- To load a constant, simply move the required value into a register - the assembler will convert to the rotate form for us
  - MOV r0,#4096 ;MOV r0,#0x1000 (0x40 ror 26)

- The bitwise complements can also be formed using MVN:
  - MOV r0,#0xFFFFFFFF ;MVN r0,#0

- Values that cannot be generated in this way will cause an error
Loading 32-bit Constants

• To allow larger constants to be loaded, the assembler offers a pseudo-instruction:
  LDR rd,=const

• This will either:
  – produce a MOV or MVN instruction to generate the value (if possible) or
  – generate a LDR instruction with a PC-relative address to read the constant from a literal pool (Constant data area embedded in the code)

• For example
  LDR r0,=0xFF ;MOV r0,#0xFF
  LDR r0,=0x55555555 ;LDR r0,[PC,#Imm10]

• As this mechanism will always generate the best instruction for a given case, it is the recommended way of loading constants.
Multiple Register Data Transfer (1/2)

- The load and store multiple instructions (LDM/STM) allow between 1 and 16 registers to be transferred to or from memory
  - order of register transfer cannot be specified, order in the list is insignificant
  - lowest register number is always transferred to/from lowest memory location accessed

- The transferred registers can be either
  - any subset of the current bank of registers (default)
  - any subset of the user mode bank of registers when in a privileged mode
    (postfix instruction with a “^”)

- Base register used to determine where memory access should occur
  - 4 different addressing modes
  - base register can be optionally updated following the transfer (using “!”)

- These instructions are very efficient for
  - moving blocks of data around memory
  - saving and restoring context - stack
Multiple Register Data Transfer (2/2)

- Allow any subset (or all, r0 to r15) of the 16 registers to be transferred with a single instruction

\[
\text{LDMIA } r1,\{r0,r2,r5\} \\
; r0 := \text{mem}_{32}[r1] \\
; r2 := \text{mem}_{32}[r1+4] \\
; r5 := \text{mem}_{32}[r1+8]
\]
Stack Processing

• A stack is usually implemented as a linear data structure which grows up (an ascending stack) or down (a descending stack) memory
• A stack pointer holds the address of the current top of the stack, either by pointing to the last valid data item pushed onto the stack (a full stack), or by pointing to the vacant slot where the next data item will be placed (an empty stack)
• ARM multiple register transfer instructions support all four forms of stacks
  – **full ascending**: grows up; base register points to the highest address containing a valid item
  – **empty ascending**: grows up; base register points to the first empty location above the stack
  – **full descending**: grows down; base register points to the lowest address containing a valid data
  – **empty descending**: grows down, base register points to the first empty location below the stack
Block Copy Addressing (1/2)

- Addressing modes

<table>
<thead>
<tr>
<th>Addressing modes</th>
<th>Ascending</th>
<th>Descending</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Full</td>
<td>Empty</td>
</tr>
<tr>
<td>Before</td>
<td>STMIB</td>
<td>LDMIB</td>
</tr>
<tr>
<td>Increment</td>
<td>STMIA</td>
<td>LDMIA</td>
</tr>
<tr>
<td>After</td>
<td>STMEA</td>
<td>LDMFD</td>
</tr>
<tr>
<td>Before</td>
<td>LDMDB</td>
<td>STMDB</td>
</tr>
<tr>
<td>Decrement</td>
<td>LDMEA</td>
<td>STMFD</td>
</tr>
<tr>
<td>After</td>
<td>LDMDA</td>
<td>STMDA</td>
</tr>
<tr>
<td></td>
<td>LDMFA</td>
<td>STMED</td>
</tr>
</tbody>
</table>
Block Copy Addressing (2/2)

- Copy 8 words from the location r0 points to to the location r1 points to
  
  LDMIA r0!,{r2-r9}
  STMIA r1,{r2-r9}
  
  - r0 increased by 32, r1 unchanged

- If r2 to r9 contained useful values, preserve them by pushing them onto a stack
  
  STMFD r13!,{r2-r9}
  LDMIA r0!,{r2-r9}
  STMIA r1,{r2-r9}
  LDMFD r13,{r2-r9}
  
  - FD postfix: full descending stack addressing mode
Memory Block Copy

- The direction that the base pointer moves through memory is given by the postfix to the STM/LDM instruction
  - STMIA/LDMIA: Increment After
  - STMIB/LDMIB: Increment Before
  - STMDA/LDMDA: Decrement After
  - STMDB/LDMDB: Decrement Before

- For Example
  ;r12 points to start of source data
  ;r14 points to end of source data
  ;r13 points to start of destination data
  Loop LDMIA r12!,{r0-r11} ;load 48 bytes
      STMIA r13!,{r0-r11} ;and store them
      CMP r12,r14 ;check for the end
      BNE Loop ;and loop until done

  - this loop transfers 48 bytes in 31 cycles
  - over 50Mbytes/sec at 33MHz
Single Word and Unsigned Byte Data Transfer Instructions

- **Pre-indexed form**
  \[ \text{LDR|STR} \{<\text{cond}>\}{B} \text{ Rd, [Rn,<offset>]}\{!\} \]

- **Post-index form**
  \[ \text{LDR|STR} \{<\text{cond}>\}{B}{T} \text{ Rd, [Rn],<offset>} \]

- **PC-relative form**
  \[ \text{LDR|STR} \{<\text{cond}>\}{B} \text{ Rd, LABEL} \]
  
  - LDR ‘load register’; STR ‘store register’
  - ‘B’ unsigned byte transfer, default is word;
  - <offset> may be \# +/-<12-bit immediate> or +/- Rm{,shift}
  - ! auto-indexing
  - T flag selects the user view of the memory translation and protection system
Example

- Store a byte in r0 to a peripheral

```assembly
LDR r1, UARTADD
STRB r0, [r1] ; store data to UART
...
UARTADD & &10000000
```
Half-word and Signed Byte Data Transfer Instructions

- **Pre-indexed form**
  \[
  \text{LDR|STR\{<cond>|H|SH|SB\ \text{Rd, [Rn,<offset>]\{!\}}} \\
  \]

- **Post-indexed form**
  \[
  \text{LDR|STR\{<cond>|H|SH|SB\ \text{Rd, [Rn],<offset>}} \\
  \]

  - `<offset>` is # +/-<8-bit immediate> or +/-Rm
  - H|SH|SB selects the data type - unsigned half-word, signed half-word and signed byte. Otherwise is for word and unsigned byte transfer
Example

• Expand an array of signed half-words into an array of words

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADR r1,ARRAY1</td>
<td>half-word array start</td>
</tr>
<tr>
<td>ADR r2,ARRAY2</td>
<td>word array start</td>
</tr>
<tr>
<td>ADR r3,ENDARR1</td>
<td>ARRAY1 end+2</td>
</tr>
<tr>
<td>Loop</td>
<td></td>
</tr>
<tr>
<td>LDRSH r0,[r1],#2</td>
<td>get signed half-word</td>
</tr>
<tr>
<td>STR r0,[r2],#4</td>
<td>save word</td>
</tr>
<tr>
<td>CMP r1,r3</td>
<td>check for end of array</td>
</tr>
<tr>
<td>BLT Loop</td>
<td>if not finished, loop</td>
</tr>
</tbody>
</table>
Multiple Register Transfer Instructions

- **LDM|STM {<cond>}<add mode> Rn{!},<registers>**
  - `<add mode>` specifies one of the addressing modes; ‘!’: auto-indexing; `<registers>` a list of registers, e.g. `{r0,r3-r7,pc}`

- **In non-user mode, the CPSR may be restored by**
  - **LDM{<cond>}<add mode> Rn{!},<registers+PC>**

- **In non-user mode, the user registers may be saved or restored by**
  - **LDM|STM{<cond>}<add mode> Rn,<registers-PC>**
    - The register list must not contain PC and write-back is not allowed
Example

- Save 3 work registers and the return address upon entering a subroutine (assume r13 has been initialized for use as a stack pointer)
  
  \[
  \text{STMFD r13!,}\{r0-r2,r14\}
  \]

- Restore the work registers and return
  
  \[
  \text{LDMFD r13!,}\{r0-r2,pc\}
  \]
Swap Memory and Register Instructions

- \( \text{SWP}\{<\text{cond}>\}\{B\} \ Rd, Rm, [Rn] \)
- \( Rd \leftarrow [Rn], [Rn] \leftarrow Rm \)

- Combine a load and a store of a word or an unsigned byte in a single instruction

- Example
  
  \[
  \begin{align*}
  &\text{ADR} \ r0, \text{SEMAPHORE} \\
  &\text{SWPB} \ r1, r1, [r0] \quad ;\text{exchange byte}
  \end{align*}
  \]
Status Register to General Register Transfer Instructions

- MRS{<cond>} Rd, CPSR|SPSR

- The CPSR or the current mode SPSR is copied into the destination register. All 32 bits are copied.

- Example
  
  MRS r0, CPSR
  MRS r3, SPSR
General Register to Status Register Transfer Instructions

• MSR{<cond>} CPSR_<field>|SPSR_<field>,#<32-bit immediate>
  MSR{<cond>} CPSR_<field>|SPSR_<field>,Rm

  – <field> is one of
    • c - the control field PSR[7:0]
    • x - the extension field PSR[15:8]
    • s - the status field PSR[23:16]
    • f - the flag field PSR[31:24]

• Example
  – set N, X, C, V flags
    MSR CPSR_f,#&f0000000
  – set just C, preserving N, Z, V
    MRS r0,CPSR
    ORR r0,r0,#&20000000 ;set bit29 of r0
    MSR CPSR_f,r0
Control Flow Instructions

- Branch instructions
- Conditional branches
- Conditional execution
- Branch and link instructions
- Subroutine return instructions
- Supervisor calls
- Jump tables
Branch Instructions

B LABEL

... 

LABEL ... 

- LABEL comes after or before the branch instruction
Conditional Branches

- The branch has a condition associated with it and it is only executed if the condition codes have the correct value - taken or not taken

```
MOV r0,#0 ;initialize counter
Loop ...
ADD r0, r0, #1 ;increment loop counter
CMP r0, #10 ;compare with limit
BNE Loop ;repeat if not equal
;else fall through
```
## Conditional Branch

<table>
<thead>
<tr>
<th>Branch</th>
<th>Interpretation</th>
<th>Normal uses</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>Unconditional</td>
<td>Always take this branch</td>
</tr>
<tr>
<td>BAL</td>
<td>Always</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 minus or negative</td>
</tr>
<tr>
<td>BCC</td>
<td>Carry clear</td>
<td>Arithmetic operation did not give carry-out</td>
</tr>
<tr>
<td>BLO</td>
<td>Lower</td>
<td>Unsigned comparison gave lower</td>
</tr>
<tr>
<td>BCS</td>
<td>Carry set</td>
<td>Arithmetic operation gave 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; no overflow occurred</td>
</tr>
<tr>
<td>BVS</td>
<td>Overflow set</td>
<td>Signed integer operation; overflow occurred</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 or equal</td>
</tr>
<tr>
<td>BLT</td>
<td>Less than</td>
<td>Signed integer comparison gave less than</td>
</tr>
<tr>
<td>BLE</td>
<td>Less 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

- An unusual feature of the ARM instruction set is that conditional execution applies not only to branches but to all ARM instructions

```
CMP r0,#5
BEQ Bypass ;if(r0!=5)    CMP r0,#5
ADD r1,r1,r0 ;{r1=r1+r0-r2} ADDNE r1,r1,r0
SUB r1,r1,r2
```

- Whenever the conditional sequence is 3 instructions or fewer it is better (smaller and faster) to exploit conditional execution than to use a branch

```
if((a==b)&&(c==d)) e++;       CMP r0,r1
   CMP r0,#5
   CMPEQ r2,r3
   ADDEQ r4,r4,#1
```
Branch and Link Instructions

• Perform a branch, save the address following the branch in the link register, r14
  
  BL SUBR ;branch to SUBR
  ...
  SUBR ...
  MOV PC, r14 ;return

• For nested subroutine, push r14 and some work registers required to be saved onto a stack in memory
  
  BL SUB1
  ...
  SUB1 STMFD r13!, {r0-r2,r14}; save work and link regs
  BL SUB2
  ...
  SUB2 ...

Subroutine Return Instructions

```
SUB ... 
    MOV PC, r14 ; copy r14 into r15 to return
```

- Where the return address has been pushed onto a stack

```
SUB1   STMFD r13!, {r0-r2,r14} ; save work regs and link
BL SUB2
...
LDMFD r13!, {r0-e12,PC} ; restore work regs & return
```
Supervisor Calls

• The supervisor is a program which operates at a privileged level, which means that it can do things that a user-level program cannot do directly (e.g. input or output)

• SWI instruction
  – software interrupt or supervisor call

  SWI SWI_WriteC ;output r0[7:0]
  SWI SWI.Exit ;return to monitor program
Jump Tables

- To call one of a set of subroutines, the choice depending on a value computed by the program

```
BL JUMPTAB
...
JUMPTAB CMP r0,#0 JUMPTAB ADR r1, SUBTAB ; r1 -> SUBTAB
BEQ SUB0 CMP r0,#1 LDRLS PC, [r1, r0, LSL#2]; if OK, table jump
CMP r0,#2 SUB0 BEQ SUB2
BEQ SUB1
... SUBTAB DCD SUB0
... DCD SUB1
... DCD SUB2
```

- The ‘DCD’ directive instructs the assembler to reserve a word of store and to initialize it to the value of the expression to the right, which in these cases is just the address of the label.
Branch and Branch with Link (B,BL)

- \( \text{B}\{\text{L}\}\{\langle \text{cond} \rangle \} \ <\text{target address}> \)
  - \(<\text{target address}>\) is normally a label in the assembler code.

\[
\begin{array}{cccccc}
31 & 28 & 27 & 25 & 24 & 23 \\
\text{cond} & 1110 & L & & & 0
\end{array}
\]

24-bit offset, sign-extended, shift left 2 places
+ PC (address of branch instruction + 8)

\[
\text{target address}
\]
Examples

- **Unconditional jump**
  
  ```
  B LABEL
  ...
  LABEL ...
  ```

- **Loop ten times**
  
  ```
  MOV r0,#10
  Loop...
  SUBS r0,#1
  BNE Loop
  ...
  ```

- **Call a subroutine**
  
  ```
  BL SUB
  ...
  SUB ...
  MOV PC,r14
  ```

- **Conditional subroutine call**
  
  ```
  CMP r0,#5
  BLLT SUB1 ; if r0<5, call SUB1
  BLGE SUB2 ; else call SUB2
  ```
Branch, Branch with Link and eXchange

- \( B\{L\}X\{<\text{cond}>\} \ Rm \)
  - the branch target is specified in a register, \( Rm \)
  - bit[0] of \( Rm \) is copied into the T bit in CPSR; bit[31:1] is moved into PC
  - if \( Rm[0] \) is 1, the processor switches to execute Thumb instructions and begins executing at the address in \( Rm \) aligned to a half-word boundary by clearing the bottom bit
  - if \( Rm[0] \) is 0, the processor continues executing ARM instructions and begins executing at the address in \( Rm \) aligned to a word boundary by clearing \( Rm[1] \)

- \( BLX \ <\text{target address}> \)
  - call Thumb subroutine from ARM
  - the H bit (bit 24) is also added into bit 1 of the resulting address, allowing an odd half-word address to be selected for the target instruction which will always be a Thumb instruction
Example

- A call to a Thumb subroutine

\[
\begin{align*}
\text{CODE32} \\
\ldots \\
\text{BLX TSUB} \quad ;\text{call Thumb subroutine} \\
\ldots \\
\text{CODE16} \quad ;\text{start of Thumb code} \\
\text{TSUB} \quad \ldots \\
\text{BX r14} \quad ;\text{return to ARM code}
\end{align*}
\]
Software Interrupt (SWI)

- SWI\{<\text{cond}>\} <24\text{-bit immediate}>
  - used for calls to the operating system and is often called a “supervisor call”
  - it puts the processor into supervisor mode and begins executing instruction from address 0x08
    - Save the address of the instruction after SWI in r14_svc
    - Save the CPSR in SPSR_svc
    - Enter supervisor mode and disable IRQs by setting CPSR[4:0] to 10011_2 and CPSR[7] to 1
    - Set PC to 08_{16} and begin executing the instruction there
  - the 24-bit immediate does not influence the operation of the instruction but may be interpreted by the system code
Examples

- Output the character ‘A’
  
  MOV r0,#'A'
  SWI SWI_WriteC

- Finish executing the user program and return to the monitor
  
  SWI SWI.Exit

- A subroutine to output a text string
  
  BL STROUT
  = "Hello World",&0a,&0d,0
  
  ... 

  STROUT LDRB r0,[r14],#1 ;get character
  CMP r0,#0 ;check for end marker
  SWINSE SWI_WriteC ;if not end, print
  BNE STROUT ;   … , loop
  ADD r14,#3 ;align to next word
  BIC r14,#3
  MOV PC,r14 ;return
Coprocessor Instructions

• Extend the instruction set through the addition of coprocessors
  – **System Coprocessor**: control on-chip function such as cache and memory management unit
  – **Floating-point Coprocessor**
  – **Application-Specific Coprocessor**

• Coprocessors have their own private register sets and their state is controlled by instructions that mirror the instructions that control ARM registers
Coprocessor Data Operations

- **CDP**{$<\text{cond}>$, $<\text{CP#}>$, $<\text{Cop1}>$, CRd, CRn, CRm{, $<\text{Cop2}>$}}

- Use to control internal operations on data in coprocessor registers
- CP# identifies the coprocessor number
- Cop1, Cop2 operation
- Examples
  
  - CDP $P2, 3, C0, C1, C2$
  - CDPEQ $P3, 6, C1, C5, C7, 4$
Coproprocessor Data Transfers

- **Pre-indexed form**
  \[ \text{LDC|STC\{<cond>\}{L}<CP#},\text{CRd},[Rn,<offset>]{}! \]

- **Post-indexed form**
  \[ \text{LDC|STC\{<cond>\}{L}<CP#},\text{CRd},[Rn],<offset> \]
  - L flag, if present, selects the long data type
  - <offset> is \# +/-<8-bit immediate>
  - the number of words transferred is controlled by the coprocessor
  - address calculated within ARM; number of words transferred controlled by the coprocessor

- **Examples**
  - LDC P6,c0,[r1]
  - STCEQL P5,c1,[r0],#4
Coprocessor Register Transfers

• Move to ARM register from coprocessor
  \[ \text{MRC}\{\text{<cond>}}\ <\text{CP#}>,<\text{Cop1}>,Rd,\text{CRn},\text{CRm},\{,<\text{Cop2}>\} \]

• Move to coprocessor from ARM register
  \[ \text{MCR}\{\text{<cond>}}\ <\text{CP#}>,<\text{Cop1}>,Rd,\text{CRn},\text{CRm},\{,<\text{Cop2}>\} \]

Examples

- \[ \text{MCR P14,3,r0,c1,c2} \]
- \[ \text{MRCCS P2,4,r3,c3,c4,6} \]
Breakpoint Instructions (BKPT-v5T only)

- BKPT <16-bit immediate>

- Used for software debugging purposes; they cause the processor to break from normal instruction execution and enter appropriate debugging procedures
- BKPT is unconditional
- Handled by an exception handler installed on the prefetch abort vector
Unused Instruction Space

- Unused Arithmetic Instructions

- Unused Control Instructions

- Unused Load/Store Instructions

- Unused Coprocessor Instructions

- Undefined Instruction Space
16-bit Instruction Set
<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>Lo Register</th>
<th>Hi Register</th>
<th>Condition Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADC</td>
<td>Add with carry</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADD</td>
<td>Add</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AND</td>
<td>AND</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ASR</td>
<td>Arithmetic Shift Right</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>Branch</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bxx</td>
<td>Conditional Branch</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BIC</td>
<td>Bit Clear</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BL</td>
<td>Branch with Link</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BX</td>
<td>Branch and Exchange</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CMN</td>
<td>Compare Negative</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CMP</td>
<td>Compare</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EOR</td>
<td>EOR</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LDMIA</td>
<td>Load Multiple</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LDR</td>
<td>Load Word</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Thumb Instruction Set (2/3)

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>Lo Register</th>
<th>Hi Register</th>
<th>Condition Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDRB</td>
<td>Load Byte</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LDRH</td>
<td>Load Halfword</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LSL</td>
<td>Logical Shift Left</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LDSB</td>
<td>Load Signed Byte</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LDSH</td>
<td>Load Signed Halfword</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LSR</td>
<td>Logical Shift Right</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MOV</td>
<td>Move Register</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MUL</td>
<td>Multiply</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MVN</td>
<td>Move Negative Register</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NEG</td>
<td>Negate</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ORR</td>
<td>OR</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>POP</td>
<td>Pop Registers</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PUSH</td>
<td>Push Registers</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ROR</td>
<td>Rotate Right</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mnemonic</td>
<td>Instruction</td>
<td>Lo Register</td>
<td>Hi Register</td>
<td>Condition Code</td>
</tr>
<tr>
<td>----------</td>
<td>---------------------------</td>
<td>-------------</td>
<td>-------------</td>
<td>----------------</td>
</tr>
<tr>
<td>SBC</td>
<td>Subtract with Carry</td>
<td></td>
<td></td>
<td>○</td>
</tr>
<tr>
<td>STMIA</td>
<td>Store Multiple</td>
<td></td>
<td></td>
<td>○</td>
</tr>
<tr>
<td>STR</td>
<td>Store Word</td>
<td></td>
<td></td>
<td>○</td>
</tr>
<tr>
<td>STRB</td>
<td>Store Byte</td>
<td></td>
<td></td>
<td>○</td>
</tr>
<tr>
<td>STRH</td>
<td>Store Halfword</td>
<td></td>
<td></td>
<td>○</td>
</tr>
<tr>
<td>SWI</td>
<td>Software Interrupt</td>
<td></td>
<td></td>
<td>○</td>
</tr>
<tr>
<td>SUB</td>
<td>Subtract</td>
<td></td>
<td></td>
<td>○</td>
</tr>
<tr>
<td>TST</td>
<td>Test Bits</td>
<td></td>
<td></td>
<td>○</td>
</tr>
</tbody>
</table>
### Thumb Instruction Format

<table>
<thead>
<tr>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>Op</th>
<th>Offset5</th>
<th>Rs</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>Op</td>
<td>Rn/offset3</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>Op</td>
<td>Rd</td>
<td>Offset8</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Op</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>Op</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>Rd</td>
<td>Word8</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>L</td>
<td>B</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>H</td>
<td>S</td>
<td>1</td>
</tr>
<tr>
<td>9</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>B</td>
<td>L</td>
<td>Offset5</td>
<td>Rb</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>L</td>
<td>Offset5</td>
<td>Rb</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>L</td>
<td>Rd</td>
<td>Word8</td>
</tr>
<tr>
<td>12</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>SP</td>
<td>Rd</td>
<td>Word8</td>
</tr>
<tr>
<td>13</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>14</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>L</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>15</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>L</td>
<td>Rb</td>
<td>Rlist</td>
</tr>
<tr>
<td>16</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>Cond</td>
<td>Offset8</td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>18</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>Offset11</td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>H</td>
<td>Offset</td>
<td></td>
</tr>
</tbody>
</table>

- **Move shifted register**
- **Add/subtract**
- **Move/compare/add/subtract immediate**
- **ALU operations**
- **Hi register operations/branch exchange**
- **PC-relative load**
- **Load/store with register offset**
- **Load/store sign-extended byte/halfword**
- **Load/store with immediate offset**
- **Load/store halfword**
- **SP-relative load/store**
- **Load address**
- **Add offset to stack pointer**
- **Push/pop registers**
- **Multiple load/store**
- **Conditional branch**
- **Software Interrupt**
- **Unconditional branch**
- **Long branch with link**
Thumb-ARM Difference

- Thumb instruction set is a subset of the ARM instruction set and the instructions operate on a restricted view of the ARM registers
- Most Thumb instructions are executed unconditionally (All ARM instructions are executed conditionally)
- Many Thumb data processing instructions use a 2-address format, i.e. the destination register is the same as one of the source registers (ARM data processing instructions, with the exception of the 64-bit multiplies, use a 3-address format)
- Thumb instruction formats are less regular than ARM instruction formats => dense encoding
Register Access in Thumb

- Not all registers are directly accessible in Thumb
- Low register $r0$-$r7$: fully accessible
- High register $r8$-$r12$: only accessible with MOV, ADD, CMP; only CMP sets the condition code flags
- $SP$ (stack pointer), $LR$ (link register) & $PC$ (program counter): limited accessibility, certain instructions have implicit access to these
- $CPSR$: only indirect access
- $SPSR$: no access
Thumb Accessible Registers

- Lo registers
- Hi registers
- Shaded registers have restricted access

- r0
- r1
- r2
- r3
- r4
- r5
- r6
- r7
- r8
- r9
- r10
- r11
- r12
- SP (r13)
- LR (r14)
- PC (r15)
- CPSR
Branches

- Thumb defines three PC-relative branch instructions, each of which have different offset ranges
  - Offset depends upon the number of available bits
- Conditional Branches
  - B<cond> label
  - 8-bit offset: range of -128 to 127 instructions (+/- 256 bytes)
  - Only conditional Thumb instructions
- Unconditional Branches
  - B label
  - 11-bit offset: range of -1024 to 1023 instructions (+/- 2Kbytes)
- Long Branches with Link
  - BL subroutine
  - Implemented as a pair of instructions
  - 22-bit offset: range of -2097152 to 2097151 instructions (+/- 4Mbytes)
Data Processing Instructions

• Subset of the ARM data processing instructions
• Separate shift instructions (e.g. LSL, ASR, LSR, ROR)
  
  \[
  \text{LSL Rd, Rs, #Imm5 } \quad ; \text{Rd:=Rs <shift> #Imm5} \\
  \text{ASR Rd, Rs } \quad ; \text{Rd:=Rd <shift> Rs}
  \]

• Two operands for data processing instructions
  – act on low registers
  
  \[
  \text{BIC Rd, Rs } \quad ; \text{Rd:=Rd AND NOT Rs} \\
  \text{ADD Rd, #Imm8 } \quad ; \text{Rd:=Rd+#Imm8}
  \]
  – also three operand forms of add, subtract and shifts
  
  \[
  \text{ADD Rd, Rs, #Imm3 } \quad ; \text{Rd:=Rs+#Imm3}
  \]

• Condition code always set by low register operations
Load or Store Register

- Two pre-indexed addressing modes
  - base register+offset register
  - base register+5-bit offset, where offset scaled by
    - 4 for word accesses (range of 0-124 bytes / 0-31 words)
      - STR Rd, [Rb,#Imm7]
    - 2 for halfword accesses (range of 0-62 bytes / 0-31 halfwords)
      - LDRH Rd, [Rb,#Imm6]
    - 1 for byte accesses (range of 0-31 bytes)
      - LDRB Rd, [Rb,#Imm5]

- Special forms:
  - load with PC as base with 1Kbyte immediate offset (word aligned)
    - used for loading a value from a literal pool
  - load and store with SP as base with 1Kbyte immediate offset (word aligned)
    - used for accessing local variables on the stack
Block Data Transfers

• Memory copy, incrementing base pointer after transfer
  - STMIA Rb!, {Low Reg list}
  - LDMIA Rb!, {Low Reg list}

• Full descending stack operations
  - PUSH {Low Reg list}
  - PUSH {Low Reg list, LR}
  - POP {Low Reg list}
  - POP {Low Reg list, PC}

• The optional addition of the LR/PC provides support for subroutine entry/exit.
Miscellaneous

• Thumb SWI instruction format
  – same effect as ARM, but SWI number limited to 0~255
  – syntax:
    • SWI <SWI number>

• Indirect access to CPSR and no access to SPSR, so no MRS or MSR instructions

• No coprocessor instruction space
Thumb Instruction Entry and Exit

• T bit, bit 5 of CPSR
  – if T=1, the processor interprets the instruction stream as 16-bit Thumb instruction
  – if T=0, the processor interprets it as standard ARM instructions

• Thumb Entry
  – ARM cores startup, after reset, executing ARM instructions
  – executing a Branch and Exchange instruction (BX)
    • set the T bit if the bottom bit of the specified register was set
    • switch the PC to the address given in the remainder of the register

• Thumb Exit
  – executing a Thumb BX instruction
The Need for Interworking

- The code density of Thumb and its performance from narrow memory make it ideal for the bulk of C code in many systems. However there is still a need to change between ARM and Thumb state within most applications:
  - ARM code provides better performance from wide memory
    • therefore ideal for speed-critical parts of an application
  - some functions can only be performed with ARM instructions, e.g.
    • access to CPSR (to enable/disable interrupts & to change mode)
    • access to coprocessors
  - exception Handling
    • ARM state is automatically entered for exception handling, but system specification may require usage of Thumb code for main handler
  - simple standalone Thumb programs will also need an ARM assembler header to change state and call the Thumb routine
Interworking Instructions

• Interworking is achieved using the Branch Exchange instructions
  – in Thumb state
    BX Rn
  – in ARM state (on Thumb-aware cores only)
    BX<condition> Rn
    where Rn can be any registers (r0 to r15)

• This performs a branch to an absolute address in 4GB address space by copying Rn to the program counter

• Bit 0 of Rn specifies the state to change to
Switching between States

ARM/Thumb Selection
0- ARM State
1- Thumb State

Destination Address

BX
Example

; start off in ARM state
CODE32
ADR r0, Into_Thumb+1 ; generate branch target
; address & set bit 0,
; hence arrive Thumb state
BX r0 ; branch exchange to Thumb
...
CODE16 ; assemble subsequent as
; Thumb
Into_Thumb ...
ADR r5, Back_to_ARM ; generate branch target to
; word-aligned address,
; hence bit 0 is cleared.
BX r5 ; branch exchange to ARM
...
CODE32 ; assemble subsequent as
; ARM
Back_to_ARM ...

...
ARM Processor Core
3-Stage Pipeline ARM Organization

- **Register Bank**
  - 2 read ports, 1 write ports, access any register
  - 1 additional read port, 1 additional write port for r15 (PC)

- **Barrel Shifter**
  - Shift or rotate the operand by any number of bits

- **ALU**

- **Address register and incrementer**

- **Data Registers**
  - Hold data passing to and from memory

- **Instruction Decoder and Control**
Data Processing Instructions

- All Operations take place in a single clock cycle
3-Stage Pipeline (1/2)

• Fetch
  – the instruction is fetched from memory and placed in the instruction pipeline

• Decode
  – the instruction is decoded and the datapath control signals prepared for the next cycle

• Execute
  – the register bank is read, an operand shifted, the ALU result generated and written back into a destination register
3-Stage Pipeline (2/2)

- At any time slice, 3 different instructions may occupy each of these stages, so the hardware in each stage has to be capable of independent operations.
- When the processor is executing data processing instructions, the latency = 3 cycles and the throughput = 1 instruction/cycle.
- When accessing r15 (PC), r15=address of current instruction + 8.
Data Transfer Instructions

- Computes a memory address similar to a data processing instruction
- Load instruction follow a similar pattern except that the data from memory only gets as far as the ‘data in’ register on the 2nd cycle and a third cycle is needed to transfer the data from there to the destination register
Multi-cycle Instruction

- Memory access (fetch, data transfer) in every cycle
- Datapath used in every cycle (execute, address calculation, data transfer)
- Decode logic generates the control signals for the data path use in next cycle (decode, address calculation)
- The third cycle, which is required to complete the pipeline refilling, is also used to make a small correction to the value stored in the link register in order that it points directly at the instruction which follows the branch.
Branch Pipeline Example

Breaking the pipeline

Note that the core is executing in ARM state
Interrupt Pipeline Example

<table>
<thead>
<tr>
<th>Cycle</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>address</td>
<td>operation</td>
<td>fetch</td>
<td>decode</td>
<td>execute</td>
<td>fetch</td>
<td>decode IRQ</td>
<td>execute IRQ</td>
<td>linkret</td>
</tr>
<tr>
<td>0x8000</td>
<td>ADD</td>
<td>fetch</td>
<td>decode</td>
<td>execute</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x8004</td>
<td>SUB</td>
<td></td>
<td>fetch</td>
<td>decode IRQ</td>
<td>execute IRQ</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x8008</td>
<td>MOV</td>
<td></td>
<td></td>
<td>fetch</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x800C</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x001C</td>
<td>B(to 0xAF00)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x0018</td>
<td>XX</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x0020</td>
<td>XXX</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0xAF00</td>
<td>STMFD</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0xAF04</td>
<td>MOV</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0xAF08</td>
<td>LDR</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

IRQ interrupt minimum latency = 7 cycles
5-Stage Pipelined ARM Organization

- $T_{prog}=N_{inst} \times CPI \times \text{cycle}_{time}$
  - $N_{inst}$, compiler dependent
  - CPI, hazard $\Rightarrow$ pipeline stalls
  - $\text{cycle}_{time}$, frequency
- Separate instruction and data memories $\Rightarrow$ 5 stage pipeline
- Used in ARM9TDMI
ARM9TDMI 5-stage Pipeline Organization

- **Fetch**
  - The instruction is fetched from memory and placed in the instruction pipeline

- **Decode**
  - The instruction is decoded and register operands read from the register file. There are 3 operand read ports in the register file so most ARM instructions can source all their operands in one cycle

- **Execute**
  - An operand is shifted and the ALU result generated. If the instruction is a load or store, the memory address is computed in the ALU

- **Buffer/Data**
  - Data memory is accessed if required. Otherwise the ALU result is simply buffered for one cycle

- **Write Back**
  - The results generated by the instruction are written back to the register file, including any data loaded from memory
Data Forwarding

• Data dependency arises when an instruction needs to use the result of one of its predecessors before the result has returned to the register file => pipeline hazards
• Forwarding paths allow results to be passed between stages as soon as they are available
• 5-stage pipeline requires each of the three source operands to be forwarded from any of the intermediate result registers
• Still one load stall

    LDR rN,[...]
    ADD r2,r1,rN ;use rN immediately

    – one stall
    – compiler rescheduling
ARM7TDMI Processor Core

- Current low-end ARM core for applications like digital mobile phones
- TDMI
  - **T**: Thumb, 16-bit compressed instruction set
  - **D**: on-chip Debug support, enabling the processor to halt in response to a debug request
  - **M**: enhanced Multiplier, yield a full 64-bit result, high performance
  - **I**: Embedded ICE hardware
- Von Neumann architecture
- 3-stage pipeline, CPI ~1.9
ARM7TDMI Block Diagram

ICEBreaker

Core

Scan Chain 0

Scan Chain 1

Scan Chain 2

TAP controller

D[31:0]

DIN[31:0]

DOUT[31:0]

RANGEOUT0

RANGEOUT1

EXTERN0

EXTERN1

nOPC

nRW

MAS[1:0]

nTRANS

nMREQ

A[31:0]

Bus Splitter

TCK

TMS

nTRST

TDI

TDO

TAPSM[3:0]

IR[3:0]

SCREG[3:0]

All Other Signals
ARM7TDMI Core Diagram
ARM7TDMI Interface Signals (2/4)

- **Clock control**
  - all state change within the processor are controlled by mclk, the memory clock
  - internal clock = mclk AND \wait
  - eclk clock output reflects the clock used by the core

- **Memory interface**
  - 32-bit address A[31:0], bidirectional data bus D[31:0], separate data out Dout[31:0], data in Din[31:0]
  - \mreq indicates a processor cycle which requires a memory access
  - seq indicates that the memory address will be sequential to that used in the previous cycle

<table>
<thead>
<tr>
<th>\mreq</th>
<th>seq</th>
<th>Cycle</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>N</td>
<td>Non-sequential memory access</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>S</td>
<td>Sequential memory access</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>I</td>
<td>Internal cycle – bus and memory inactive</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>C</td>
<td>Coprocessor register transfer – memory inactive</td>
</tr>
</tbody>
</table>
lock indicates that the processor should keep the bus to ensure the atomicity of the read and write phase of a SWAP instruction

\( r/w \), read or write

\( \text{mas}[1:0] \), encode memory access size - byte, half-word or word

\( \text{bl}[3:0] \), externally controlled enables on latches on each of the 4 bytes on the data input bus

**MMU interface**

\( \text{trans} \) (translation control), 0:user mode, 1:privileged mode

\( \text{mode}[4:0] \), bottom 5 bits of the CPSR (inverted)

abort, disallow access

**State**

T bit, whether the processor is currently executing ARM or Thumb instructions

**Configuration**

bigend, big-endian or little-endian
ARM7TDMI Interface Signals (4/4)

- **Interrupt**
  - \texttt{fiq}, fast interrupt request, higher priority
  - \texttt{irq}, normal interrupt request
  - \texttt{isync}, allow the interrupt synchronizer to be passed

- **Initialization**
  - \texttt{reset}, starts the processor from a known state, executing from address \texttt{00000000}\_16

- **ARM7TDMI characteristics**

<table>
<thead>
<tr>
<th>Process</th>
<th>0.35 μm</th>
<th>Transistors</th>
<th>74,209</th>
<th>MIPS</th>
<th>60</th>
</tr>
</thead>
<tbody>
<tr>
<td>Metal layers</td>
<td>3</td>
<td>Core area</td>
<td>2.1 mm\textsuperscript{2}</td>
<td>Power</td>
<td>87 mW</td>
</tr>
<tr>
<td>Vdd</td>
<td>3.3 V</td>
<td>Clock</td>
<td>0–66 MHz</td>
<td>MIPS/W</td>
<td>690</td>
</tr>
</tbody>
</table>
External Address Generation
Memory Access

* The ARM7 is a Von Neumann, load/store architecture, i.e.
  - One 32 bit data bus for both instructions and data.
  - Only the load/store instructions (and SWP) access memory.

* Memory is addressed as a 32 bit address space.

* Data types can be 8 bit bytes, 16 bit half-words or 32 bit words, and may be seen as a byte line folded into 4-byte words.

* Words must be aligned to 4 byte boundaries, and half-words to 2 byte boundaries.

* Always ensure that memory controller supports all three access sizes!
ARM Memory Interface

* Sequential (S cycle)
  - nMREQ = 0, SEQ = 1
  - The ARM core requests a transfer to or from an address which is either the same, or one word or one half-word greater than the preceding address.

* Non-sequential (N cycle)
  - nMREQ = 0, SEQ = 0
  - The ARM core requests a transfer to or from an address which is unrelated to the address used in the preceding cycle.

* Internal (I cycle)
  - nMREQ = 1, SEQ = 0
  - The ARM core does not require a transfer, as it is performing an internal function, and no useful prefetching can be performed at the same time.

* Coprocessor register transfer (C cycle)
  - nMREQ = 1, SEQ = 1
  - The ARM core wishes to use the data bus to communicate with a coprocessor, but does not require any action by the memory system.
## Instruction Execution Cycles (1/2)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Qualifier</th>
<th>Cycle count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Any unexecuted</td>
<td>Condition codes fail</td>
<td>+S</td>
</tr>
<tr>
<td>Data processing</td>
<td>Single-cycle</td>
<td>+S</td>
</tr>
<tr>
<td>Data processing</td>
<td>Register-specified shift</td>
<td>+I +S</td>
</tr>
<tr>
<td>Data processing</td>
<td>R15 destination</td>
<td>+N +2S</td>
</tr>
<tr>
<td>Data processing</td>
<td>R15, register-specified shift</td>
<td>+I +N +2S</td>
</tr>
<tr>
<td>MUL</td>
<td></td>
<td>+(m)I +S</td>
</tr>
<tr>
<td>MLA</td>
<td></td>
<td>+I +(m)I +S</td>
</tr>
<tr>
<td>MULL</td>
<td></td>
<td>+(m)I +I +S</td>
</tr>
<tr>
<td>MLAL</td>
<td></td>
<td>+I +(m)I +I</td>
</tr>
<tr>
<td>B, BL</td>
<td></td>
<td>+N +2S</td>
</tr>
<tr>
<td>LDR</td>
<td>Non-R15 destination</td>
<td>+N +I +S</td>
</tr>
</tbody>
</table>
### Instruction Execution Cycles (2/2)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Qualifier</th>
<th>Cycle count</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDR</td>
<td>R15 destination</td>
<td>+N +I +N +2S</td>
</tr>
<tr>
<td>STR</td>
<td></td>
<td>+N +N</td>
</tr>
<tr>
<td>SWP</td>
<td></td>
<td>+N +N +I +S</td>
</tr>
<tr>
<td>LDM</td>
<td>Non-R15 destination</td>
<td>+N +(n-1)S +I +S</td>
</tr>
<tr>
<td>LDM</td>
<td>R15 destination</td>
<td>+N +(n-1)S +I +N +2S</td>
</tr>
<tr>
<td>STM</td>
<td></td>
<td>+N +(n-1)S +I +N</td>
</tr>
<tr>
<td>MSR, MRS</td>
<td></td>
<td>+S</td>
</tr>
<tr>
<td>SWI, trap</td>
<td></td>
<td>+N +2S</td>
</tr>
<tr>
<td>CDP</td>
<td></td>
<td>+(b)I +S</td>
</tr>
<tr>
<td>MCR</td>
<td></td>
<td>+(b)I +C +N</td>
</tr>
<tr>
<td>MRC</td>
<td></td>
<td>+(b)I +C +I +S</td>
</tr>
<tr>
<td>LDC, STC</td>
<td></td>
<td>+(b)I +N +(n-1)S +N</td>
</tr>
</tbody>
</table>
Effect of T bit
Cached ARM7TDMI Macrocells

* ARM710T
  - 8K Unified write through cache
  - Full Memory Management Unit supporting virtual memory and memory protection
  - Write Buffer

* ARM720T
  - As ARM710T but with WinCE support

* ARM740T
  - 8K Unified write through cache
  - Memory Protection Unit
  - Write Buffer
ARM 8

- Higher performance than ARM7
  - by increasing the clock rate
  - by reducing the CPI
    - higher memory bandwidth, 64-bit wide memory
    - Separate memories for instruction and data accesses

- ARM8 \(\rightarrow\) ARM9TDMI
  \(\rightarrow\) ARM10TDMI

- Core Organization
  - the prefetch unit is responsible for fetching instructions from memory and buffering them (exploiting the double bandwidth memory)
  - it is also responsible for branch prediction and use static prediction based on the branch prediction (backward: predicted ‘taken’, forward: predicted ‘not taken’)

![Diagram of ARM Architecture](image-url)
Pipeline Organization

- 5-stage, prefetch unit occupies the 1st stage, integer unit occupies the remainder

1. Instruction prefetch
2. Instruction decode and register read
3. Execute (shift and ALU)
4. Data memory access
5. Write back results
Integer Unit Organization
ARM9TDMI

- Harvard architecture
  - increases available memory bandwidth
    - instruction memory interface
    - data memory interface
  - simultaneous accesses to instruction and data memory can be achieved

- 5-stage pipeline

- Changes implemented to
  - improve CPI to ~1.5
  - improve maximum clock frequency
ARM9TDMI Organization
• Not sufficient slack time to translate Thumb instructions into ARM instructions and then decode, instead the hardware decode both ARM and Thumb instructions directly
ARM9TDMI Datapath (1/2)
LDR Interlock

<table>
<thead>
<tr>
<th>Cycle</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operation</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADD R1, R1, R2</td>
<td>F</td>
<td>D</td>
<td>E</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SUB R3, R4, R1</td>
<td>F</td>
<td>D</td>
<td>E</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LDR R4, [R7]</td>
<td></td>
<td>F</td>
<td>D</td>
<td>E</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ORR R8, R3, R4</td>
<td>F</td>
<td>D</td>
<td>E</td>
<td></td>
<td>I</td>
<td>E</td>
<td>W</td>
<td></td>
<td></td>
</tr>
<tr>
<td>AND R6, R3, R1</td>
<td>F</td>
<td>I</td>
<td>D</td>
<td>E</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EOR R3, R1, R2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>F</td>
<td>D</td>
<td>E</td>
<td>W</td>
</tr>
</tbody>
</table>

F - Fetch  D - Decode  E - Execute  I - Interlock  M - Memory  W - Writeback

In this example it takes 7 clock cycles to execute 6 instructions, CPI of 1.2. The LDR instruction immediately followed by a data operation using the same register causes an interlock.
## Optimal Pipelining

<table>
<thead>
<tr>
<th>Cycle</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Operation</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>ADD</strong></td>
<td>R1, R1, R2</td>
<td>F</td>
<td>D</td>
<td>E</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>SUB</strong></td>
<td>R3, R4, R1</td>
<td>F</td>
<td>D</td>
<td>E</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>LDR</strong></td>
<td>R4, [R7]</td>
<td>F</td>
<td>D</td>
<td>E</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>AND</strong></td>
<td>R6, R3, R1</td>
<td>F</td>
<td>D</td>
<td>E</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>ORR</strong></td>
<td>R8, R3, R4</td>
<td>F</td>
<td>D</td>
<td>E</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>EOR</strong></td>
<td>R3, R1, R2</td>
<td>F</td>
<td>D</td>
<td>E</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*F - Fetch  D - Decode  E - Execute  I - Interlock  M - Memory  W - Writeback*

* In this example it takes 6 cycles to execute 6 instructions, CPI of 1.
* The LDR instruction does not cause the pipeline to interlock
LDM Interlock (1/2)

<table>
<thead>
<tr>
<th>Cycle</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Operation</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LDMIA</td>
<td>R13!, {R0-R3}</td>
<td>F</td>
<td>D</td>
<td>E</td>
<td>L</td>
<td>M</td>
<td>MW</td>
<td>MW</td>
<td>MW</td>
</tr>
<tr>
<td>SUB</td>
<td>R9, R7, R2</td>
<td>F</td>
<td>D</td>
<td>E</td>
<td>I</td>
<td>I</td>
<td>I</td>
<td>E</td>
<td>P</td>
</tr>
<tr>
<td>STR</td>
<td>R4, [R9]</td>
<td>F</td>
<td>I</td>
<td>I</td>
<td>I</td>
<td>I</td>
<td>D</td>
<td>E</td>
<td>M</td>
</tr>
<tr>
<td>ORR</td>
<td>R8, R4, R3</td>
<td>F</td>
<td>D</td>
<td>E</td>
<td>I</td>
<td>I</td>
<td>W</td>
<td></td>
<td></td>
</tr>
<tr>
<td>AND</td>
<td>R6, R3, R1</td>
<td>F</td>
<td>D</td>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

* F - Fetch  D - Decode  E - Execute  I - Interlock  M - Memory  
  MW - Simultaneous Memory and Writeback  W - Writeback

* In this example it takes 8 clock cycles to execute 5 instructions, CPI of 1.6
* During the LDM there are parallel memory and writeback cycles
LDM Interlock (2/2)

<table>
<thead>
<tr>
<th>Cycle</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operation</td>
<td>F</td>
<td>D</td>
<td>E</td>
<td>M</td>
<td>MW</td>
<td>MW</td>
<td>MW</td>
<td>W</td>
<td></td>
</tr>
<tr>
<td>LDMIA R13*, {R0-R3}</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SUB R9, R7, R3</td>
<td>F</td>
<td>D</td>
<td>I</td>
<td>I</td>
<td>I</td>
<td>I</td>
<td>E</td>
<td>W</td>
<td></td>
</tr>
<tr>
<td>STR R4, [R9]</td>
<td>F</td>
<td>I</td>
<td>I</td>
<td>I</td>
<td>I</td>
<td>D</td>
<td>E</td>
<td>M</td>
<td>W</td>
</tr>
<tr>
<td>ORR R8, R4, R3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>F</td>
<td>D</td>
<td>E</td>
<td>W</td>
</tr>
<tr>
<td>AND R6, R3, R1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>F</td>
<td>D</td>
<td>E</td>
<td></td>
</tr>
</tbody>
</table>

* In this example it takes 9 clock cycles to execute 5 instructions, of 1.8
* The sub incurs a further cycle of interlock due to it using the highest specified register in the LDM instruction
  - This would occur for any of the LDM variants, e.g. IA, DB, FD, etc.
Example ARM9TDMI System

**Note:**

The data interface must have read access to the instruction memory for literal pool data. For debug purposes it is recommended that the data interface have read and write access to instruction memory.
Cached ARM9TDMI Macrocell

- **ARM920T**
  - 2x 16K caches
  - Full Memory Management Unit supporting virtual addressing and memory protection
  - Write Buffer

- **ARM940T**
  - 2x 4K caches
  - Memory Protection Unit
  - Write Buffer
ARM9TDMI Pipeline Operations (2/2)

• Coprocessor support
  – coprocessors: floating-point, digital signal processing, special-purpose hardware accelerator

• On-chip debug
  – additional features compared to ARM7TDMI
    • hardware single stepping
    • breakpoint can be set on exceptions

• ARM9TDMI characteristics

<table>
<thead>
<tr>
<th>Process</th>
<th>0.25 μm</th>
<th>Transistors</th>
<th>111,000</th>
<th>MIPS</th>
<th>220</th>
</tr>
</thead>
<tbody>
<tr>
<td>Metal layers</td>
<td>3</td>
<td>Core area</td>
<td>2.1 mm²</td>
<td>Power</td>
<td>150 mW</td>
</tr>
<tr>
<td>Vdd</td>
<td>2.5 V</td>
<td>Clock</td>
<td>0–200 MHz</td>
<td>MIPS/W</td>
<td>1500</td>
</tr>
</tbody>
</table>
ARM9E-S Family Overview

- ARM9E is based on an ARM9TDMI with the following extensions
  - single cycle 32*16 multiplier implementation
  - EmbeddedICE Logic RT
  - improved ARM/Thumb interworking
  - new 32*16 and 16*16 multiply instructions
  - new count leading zeros instruction
  - new saturated maths instructions

- ARM946E-S
  - ARM9E-S core
  - instruction and data caches, selectable sizes
  - instruction and data RAMs, selectable sizes
  - protection unit
  - AHB bus interface

- ARM966E-S
  - similar to ARM946-S, but with no cache
ARM1020T Overview

• Architecture v5T
  – ARM1020E will be v5TE
• CPI ~ 1.3
• 6-stage pipeline
• Static branch prediction
• 32KB instruction and 32KB data caches
  – ‘hit under miss’ support
• 64 bits per cycle LDM/STM operations
• EmbeddedICE Logic RT-II
• Support for new VFPv1 architecture
• ARM10200 test chip
  – ARM1020T
  – VFP10
  – SDRAM memory interface
  – PLL
ARM10TDMI (1/2)

- Current high-end ARM processor core
- Performance on the same IC process
  
  \[
  \begin{array}{ccc}
  \text{ARM10TDMI} & \overset{*2}{\longrightarrow} & \text{ARM9TDMI} & \overset{*2}{\longrightarrow} & \text{ARM7TDMI}
  \end{array}
  \]

- 300MHz, 0.25uM CMOS
- Increase clock rate
Software Development
- ARM software development - ADS
- ARM system development - ICE and trace
- ARM-based SoC development – modeling, tools, design flow

![Diagram of ARM development process]

- C source
- C libraries
- asm source
- C compiler
- .aof
- linker
- .aif / debug
- object libraries
- system model
- ARMsd
- ARMulator
- development board
ARM Development Suite (ADS), ARM Software Development Toolkit (SDT) (1/3)

- Develop and debug C, C++ or assembly language program
- **armcc** ARM C compiler
  - **armcpp** ARM C++ compiler
  - **tcc** Thumb C compiler
  - **tcpp** Thumb C++ compiler
  - **armasm** ARM and Thumb assembler
  - **armlink** ARM linker
    - combine the contents of one or more object files with selected parts of one or more object libraries to produce an executable program
    - ARM linker creates ELF executable images
- **armsd** ARM and Thumb symbolic debugger
  - can single-step through C or assembly language sources, set break-points and watch-points, and examine program variables or memory
ARM Development Suite (ADS), ARM Software Development Toolkit (SDT) (2/3)

- `.aof` ARM object format file
- `.aif` ARM image format file

- The `.aif` file can be built to include the debug tables
  => ARM symbolic debugger, ARMsd

- ARMsd can load, run and debug programs either on hardware such as the ARM development board or using the software emulation of the ARM (ARMulator)

- AxD (ADW, ADU)
  - ARM debugger for Windows and Unix with graphics user interface
  - debug C, C++, and assembly language source

Code Warrior IDE
  - project management tool for windows
Institute of Electronics, National Chiao Tung University

ARM Development Suite (ADS),
ARM Software Development Toolkit (SDT) (3/3)

- Utilities
  - `armprof`: ARM profiler
  - `Flash downloader`: download binary images to Flash memory on a development board

- Supporting software
  - `ARMulator`: ARM core simulator
    - provide instruction accurate simulation of ARM processors and enable ARM and Thumb executable programs to be run on non-native hardware
    - integrated with the ARM debugger
  - `Angel`: ARM debug monitor
    - run on target development hardware and enable you to develop and debug applications on ARM-based hardware
ARM C Compiler

- Compiler is compliant with the ANSI standard for C
- Supported by the appropriate library of functions
- Use ARM Procedure Call Standard, APCS for all external functions
  - for procedure entry and exit
- May produce assembly source output
  - can be inspected, hand optimized and then assembled sequentially
- Can also produce Thumb codes
Linker

• Take one or more object files and combine them
• Resolve symbolic references between the object files and extract the object modules from libraries
• Normally the linker includes debug tables in the output file
ARM Symbolic Debugger

- A front-end interface to debug program running either under emulation (on the ARMulator) or remotely on an ARM development board (via a serial line or through JTAG test interface)
- ARMsd allows an executable program to be loaded into the ARMulator or a development board and run. It allows the setting of
  - breakpoints, addresses in the code
  - watchpoints, memory address if accessed as data address
  => cause exception to halt so that the processor state can be examined
ARM Emulator

- ARMulator is a suite of programs that models the behavior of various ARM processor cores in software on a host system.
- It operates at various levels of accuracy:
  - instruction accurate
  - cycle accurate
  - timing accurate
  => instruction count or number of cycles can be measured for a program
  => performance analysis
- Timing accurate model is used for cache, memory management unit analysis, and so on.
ARM Development Board

• A circuit board including an ARM core (e.g. ARM7TDMI), memory components, I/O and electrically programmable devices
• It can support both hardware and software development before the final application-specific hardware is available
Writing Assembly Language Programs

- The following tools are needed
  - a text editor to type the program into
  - an assembler to translate the program into ARM binary code
  - an ARM system or emulator to execute the binary code
  - a debugger to see what is happening inside the code
Program Design

• Start with understanding the requirements, translate the requirements into an unambiguous specifications
• Define a program structure, the data structure and the algorithms that are used to perform the required operations on the data
• The algorithms may be expressed in pseudo-code
• Individual modules should be coded, tested and documented
• Nearly all programming is based on high-level languages, however it may be necessary to develop small software components in assembly language to get the best performance
System Architecture (1/2)

- ARM processor, memory system, buses, and the ARM reference peripheral specification
- The reference peripheral specification defines a basic set of components, providing a framework within which an operating system can run but leaving full scope for application-specific system
- Components include
  - a memory map
  - an interrupt control
  - a counter timer
  - a reset controller with defined boot behavior, power-on reset detection, a “wait for interrupt” pause mode
- The system must define
  - the base address of the interrupt controller (ICBase)
  - the base address of the counter-timer (CTBase)
  - the base address of the reset and pause controller (RPCBase)
All the address of the registers are relative to one of the base addresses
System Architecture (2/2)

- Interrupt controller provides a way of enabling, disabling (by mask) and examining the status of up to 32 level-sensitive IRQ sources and one FIQ source
- Two 16-bit counter-timers, controlled by registers. The counters operate from the system clock with selectable pre-scaling
- Reset and pause controller includes some registers
  - the readable registers give identification and reset status information
  - the writable registers can set or clear the reset status, clear the reset map and put the system into pause mode where it uses minimal power until an interrupt wakes it up again
- Add application-specific peripherals
Hardware System Prototype

- Verifying the function correctness of hardware blocks, software modules (on-developing) and speed performance is acceptable
- Simulating the system using software tools => slower, can’t verify the full system
- Hardware Prototyping
  - building a hardware platform by pre-existing or on-developing components for system verification and software development
  - “ARM Integrator” or “Rapid Silicon Prototyping”
ARM Integrator

- A motherboard with some extensions to support the development of applications
- Provide core modules, logic modules (Xilinx Virtex FPGA), OS, input/output resources, bus arbitration, interrupt handling
Rapid Silicon Prototyping (VLSI Tech. Inc.)

- Specially developed reference chips + off-chip extensions
ARMulator (1/2)

- ARMulator is a collection of programs that emulate the instruction sets and architecture of various ARM processors. It is an instruction set simulator.
- ARMulator is suited to software development and benchmarking ARM-targeted software. It models the instruction set and counts cycles.
- ARMulator supports a C library to allow complete C programs to run on the simulated system.
- To run software on ARMulator, through ARM symbolic debugger or ARM GUI debuggers, AxD
ARMulator (2/2)

- It includes
  - processor core models which can emulate any ARM core
  - a memory interface which allows the characteristics of the target memory system to be modeled
  - a coprocessor interface that supports custom coprocessor models
  - an OS interface that allows individual system calls to be handled
- The processor core model incorporates the remote debug interface, so the processor and the system state are visible from the ARM symbolic debugger
- ARMulator => a cycle accurate model of a system including a cache, MMU, physical memory, peripheral devices, OS, software
- Once the design is OK,
  hardware -> design or synthesis by CAD
  software -> still use ARMulator model, but instruction accurate
JTAG Boundary Scan (1/2)

- IEEE 1149, Standard Test Access Port and Boundary Scan Architecture or called JTAG boundary scan (by Joint Test Action Group)
JTAG Boundary Scan (2/2)

• Test signals
  – TRST: a test reset input
  – TCK: test clock which controls the timing of the test interface independently from any system clock
  – TMS: test mode select which controls the operation of the test interface state machine
  – TDI: test data input line
  – TDO: test data output line

• TAP controller (Test Access Port)
  A state machine whose state transitions are controlled by TMS
TAP Controller (1/2)

- **TMS=0**
  - run test/idle
  - select DR scan
  - capture DR
  - shift DR
  - exit1 DR
  - pause DR
  - exit2 DR
  - update DR

- **TMS=1**
  - run test/idle
  - select IR scan
  - capture IR
  - shift IR
  - exit1 IR
  - pause IR
  - exit2 IR
  - update IR

- test logic reset
TAP Controller (2/2)

- Test instruction selects various data registers
  - device ID register, bypass register, boundary scan register
- Some public instructions
  - **BYPASS**: connect TDI to TDO with 1-clock delay
  - **EXTEST**: test the board-level connectivity, boundary scan register is connected
    - **capture DR**: captured by the boundary scan register
    - **shift DR**: shift out via TDO
    - **update DR**: new data applied to the boundary scan register via TDI
  - **TNTEST**: test the core logic
  - **INCODE**: ID register is connect
Macrocell Testing

- System chip is composed of the pre-designed macrocells with application-specific custom logic
- Various approaches to test the macrocells
  - test mode provided which multiplexes the signals in turn onto the chip
  - on-chip bus may support direct test access to macrocell pins
  - each macrocell may have a boundary scan path using JTAG extensions
ARM Debug Architecture (1/2)

- Two basic approaches to debug
  - from the outside, use a logic analyzer
  - from the inside, tools supporting single stepping, breakpoint setting

- **Breakpoint**: replacing an instruction with a call to the debugger

- **Watchpoint**: a memory address which halts execution if it is accessed as a data transfer address

- **Debug Request**: through ICEBreaker programming or by DBGRQ pin asynchronously
ARM Debug Architecture (2/2)

- In debug state, the core’s internal state and the system’s external state may be examined. Once examination is complete, the core and system state may be restored and program execution is resumed.

- The internal state is examined via a JTAG-style serial interface, which allows instructions to be serially inserted into the core’s pipeline without using the external data bus.

- When in debug state, a store-multiple (STM) could be inserted into the instruction pipeline and this would dump the contents of ARM’s registers.
A debugger is software that enables you to make use of a debug agent in order to examine and control the execution of software running on a debug target.

Different forms of the debug target:
- early stage of product development, software
- prototype, on a PCB including one or more processors
- final product

The debugger issues instructions that can:
- load software into memory on the target
- start and stop execution of that software
- display the contents of memory, registers, and variables
- allow you to change stored values

A debug agent performs the actions requested by the debugger, such as:
- setting breakpoints
- reading from / writing to memory
Examples of debug agents

- Multi-ICE
- Embedded ICE
- ARMulator
- BATS
- Angle

- Remote Debug Interface (RDI) is an open ARM standard procedural interface between a debugger and the debug agent
In Circuit Emulator (ICE)

- The processor in the target system is removed and replaced by a connection to an emulator.
- The emulator may be based around the same processor chip, or a variant with more pins, but it will also incorporate buffers to copy the bus activity to a “trace buffer” and various hardware resources which can watch for particular events, such as execution passing through a breakpoint.
Multi-ICE and Embedded ICE

• Multi-ICE and Embedded ICE are JTAG-based debugging systems for ARM processors
• They provide the interface between a debugger and an ARM core embedded within an ASIC
• It provides
  – real time address-dependent and data-dependent breakpoints
  – single stepping
  – full access to, and control of the ARM core
  – full access to the ASIC system
  – full memory access (read and write)
  – full I/O system access (read and write)
Basic Debug Requirements

• Control of program execution
  – set watchpoints on interesting data accesses
  – set breakpoints on interesting instructions
  – single step through code

• Examine and change processor state
  – read and write register values

• Examine and change system state
  – access to system memory
    • download initial code
Debugging with Multi-ICE

- The system being debugged may be the final system
- Third party protocol converters are also available at http://www.arm.com/DevSupp/ICE_Analyz/
ICEBreaker (EmbeddedICE macrocell)

- ICEBreaker is programmed in a serial fashion using the TAP controller.
- It consists of 2 real-time watch-point units, together with a control and status register.
- Either watch-point unit can be configured to be a watch-point or a breakpoint.
Real-Time Trace (1/2)

• Debugging uses the breakpoint and single-step to run application code to a given point, and then stop the processor to examine or change memory or register contents, and then step or restart the code

• Some bugs occur while the system is running at full clock speed => need non-intrusive trace of instruction flow and data accesses

• Using Trace Debug Tool (TDT), you can set up the trace filter facility to collect trace data only during the interrupt routine, and use a trigger to stop tracing
Real-Time Trace (2/2)

- Embedded trace macrocell
  - monitor the ARM core buses, passed compressed information through the trace port to Trace Port Analyzer (TPA)
  - the on-chip cell contains the trigger and filter logic
- Trace port analyzer
  - an external device that stores the information from the trace port
- Trace debug tool
  - set up the trigger and filter logic, retrieve the data from the analyzer and reconstruct a historical view of processor activity
ARM10TDMI (2/2)

- Reduce CPI
  - branch prediction
  - non-blocking load and store execution
  - 64-bit data memory => transfer 2 registers in each cycle