National Technical University of Athens
Microprocessors and Digital Systems Laboratory

ARM Programmer's model

Masouros Dimosthenis
Manolis Katsaragakis
Ioannis Oroutzoglou
Aggelos Ferikoglou

ARM ltd

- Founded in November 1990 as **Advanced RISC Machines** Ltd
- Designs the ARM range of RISC processor cores
- Licenses ARM core designs to semiconductor partners who fabricate and sell to their customers
  - **ARM does not fabricate silicon itself!**
- Also develop technologies to assist with the design of the ARM architecture
  - Software tools
  - Boards
  - Debug hardware
  - Application software
  - Graphics
  - Bus architectures
  - Peripherals
  - Cell libraries
- Bought by Softbank in 2016 for $31.4 billion!

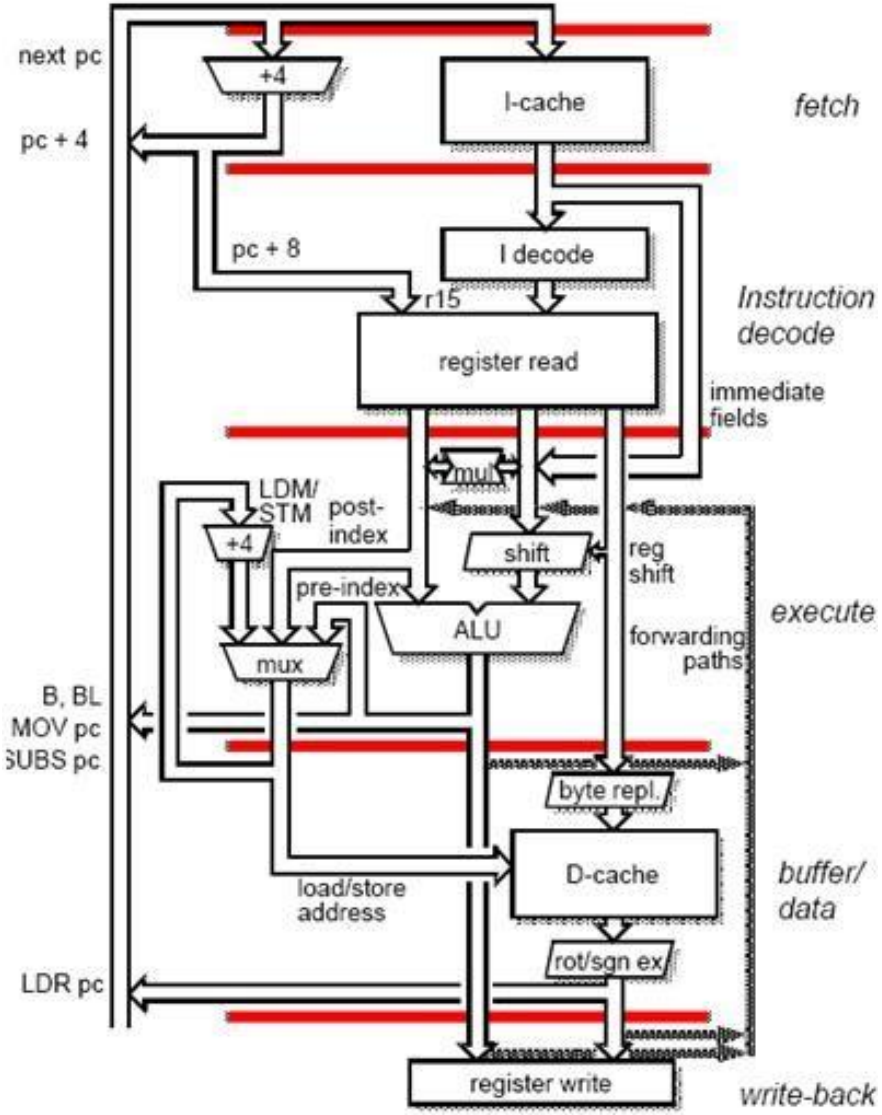
ARM as a RISC architecture

- ARM confronts to the Reduced Instruction Set Computer (RISC) architecture.

- A typical RISC system is defined:
  - Load / Store model
  - Operations on registers and not directly on memory
  - All data must be loaded into registers before they can be operated on.
  - Fixed instruction length
  - Small number of addressing modes
  - A large set of general-purpose registers that can hold either data or an address.

- However, ARM is not a pure RISC architecture:
  - Conditional execution of most instructions
  - Arithmetic instructions alter condition codes only when desired.
  - Addition of a 32-bit barrel shifter before instruction execution

ARM organization & implementation



ARM organization & implementation - AMBA

- The important aspect of a SoC is not only which **components** or blocks it houses, but also **how they are interconnected**.

- The **A**dvanced **M**icrocontroller **B**us **A**rchitecture was introduced in 1996 and is widely used as the on-chip bus in System-on-a-chip (SoC) designs processors.

- AMBA Goals:
  - Technology independence
  - To encourage modular system design

- The AMBA 3.0 specification define five buses/interfaces:
  - Advanced eXtensible Interface (AXI)
  - Advanced High-performance Bus (AHB)
  - Advanced System Bus (ASB)
  - Advanced Peripheral Bus (APB)
  - Advanced Trace Bus (ATB)

- ARM provides ARMA Design Kit (ADK) as a generic, stand-alone environment to enable the rapid creation of AMBA-based components and System-on-Chip (SoC) designs.

ARM data sizes and instructions

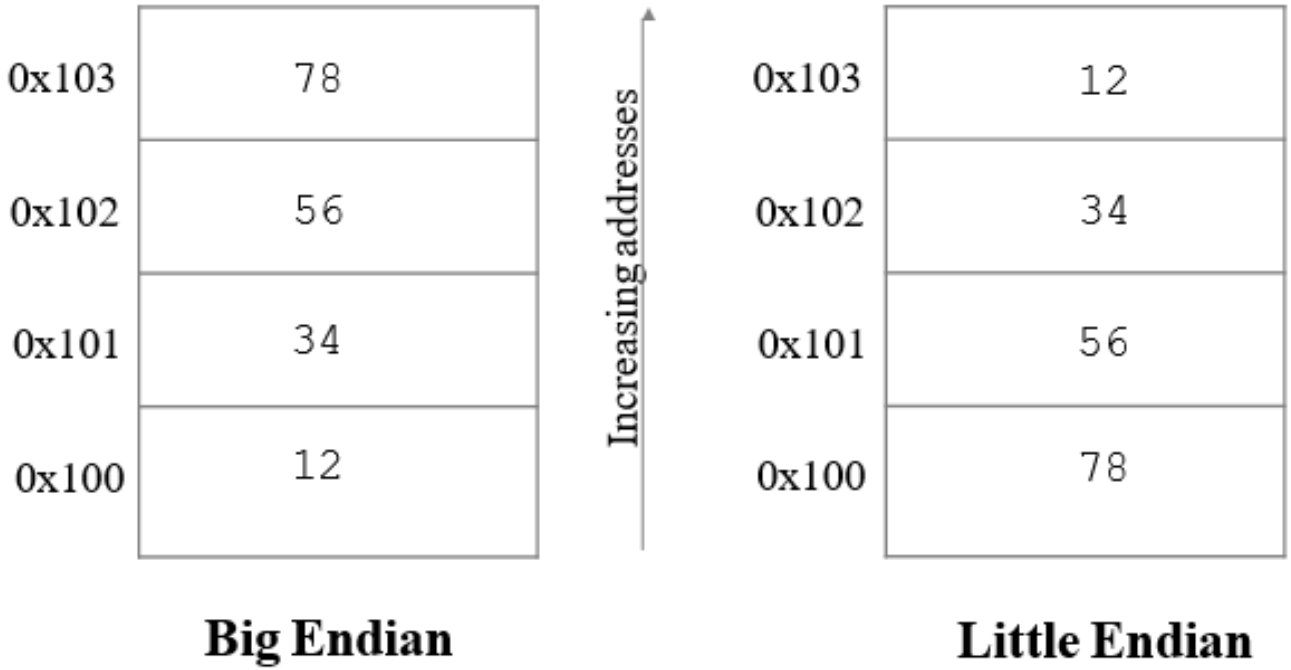
- The ARM is a 32-bit RISC architecture, so in relation to that:
  - Byte means 8 bits
  - Halfword means 16 bits (two bytes)
  - Word means 32 bits (four bytes)

- Most ARM cores implement two instruction sets:
  - 32-bit ARM Instruction Set
  - 16-bit Thumb Instruction Set
  - 64-bit (v8)

- ARM cores can be configured to view words stored in memory as either Big-Endian or Little-Endian format.

Big Endian vs. Little Endian

- How 0x12345678 would be stored in a 32-bit memory?



The ARM programmer's model

- What is a mode?
  - Characterized by specific behavior, privileges, associated registers
  - Triggered by some action (e.g. exception, interrupt)

- The ARM has seven basic operating modes:
  - **User**: normal program execution mode
  - **FIQ**: used for handling a high priority (fast) interrupt
  - **IRQ**: used for handling a low priority (normal) interrupt
  - **Supervisor**: entered on reset and when a Software Interrupt instruction is executed
  - **Abort**: used for handling memory access violations
  - **Undefined**: used for handling undefined instructions
  - **System**: a privileged mode that uses the same registers as the user mode

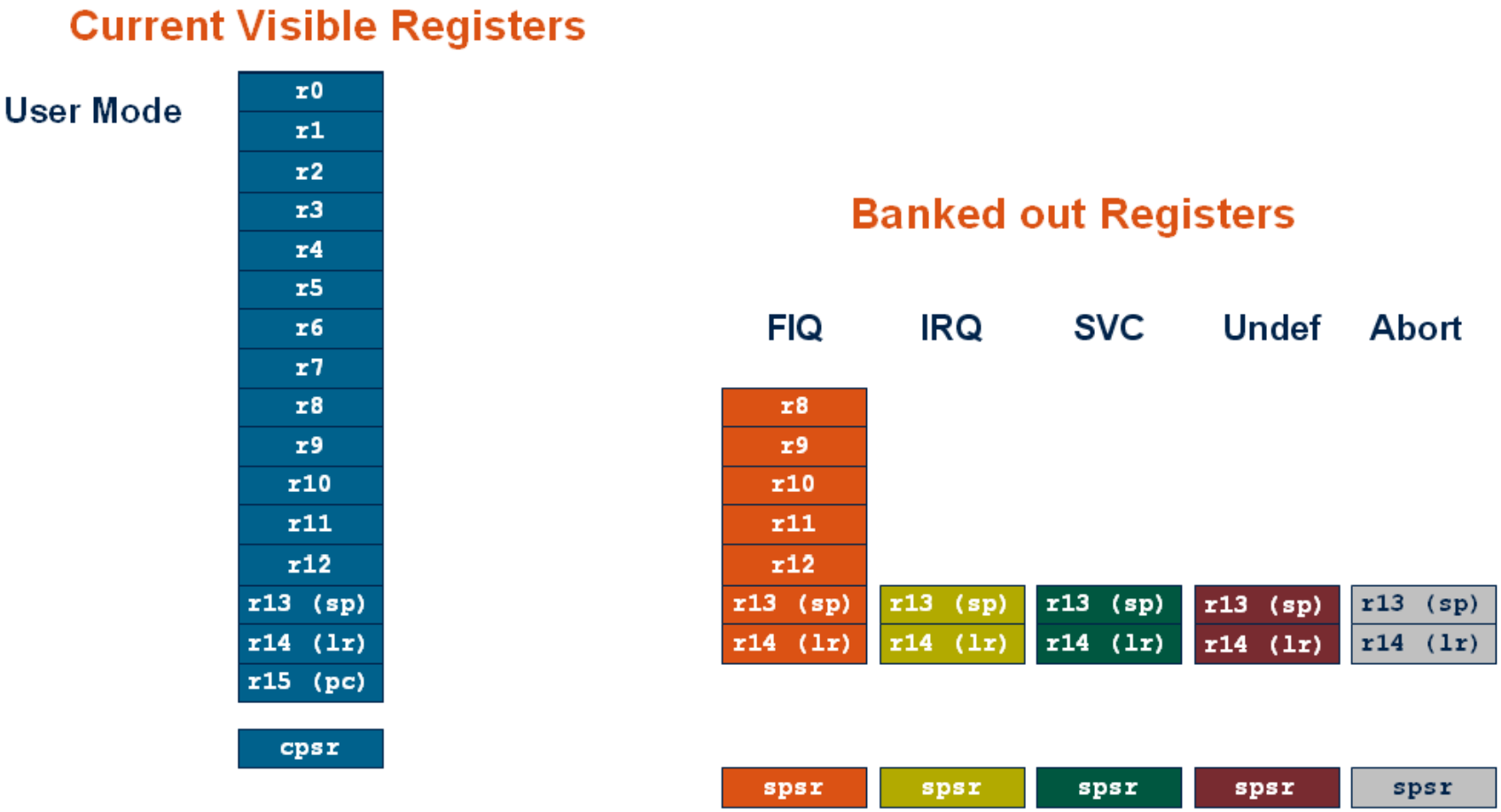
ARM's Register Set

- ARM has a total of 37 registers, all of which are 32-bit long
  - 30 general purpose registers
  - 1 dedicated program counter (pc)
  - 1 dedicated current program status register (cpsr)
  - 5 dedicated saved program status registers (spsr)
- In any mode only a subset of the 37 registers are visible
  - The hidden registers are called banked registers.
  - The current processor mode governs which registers are accessible.

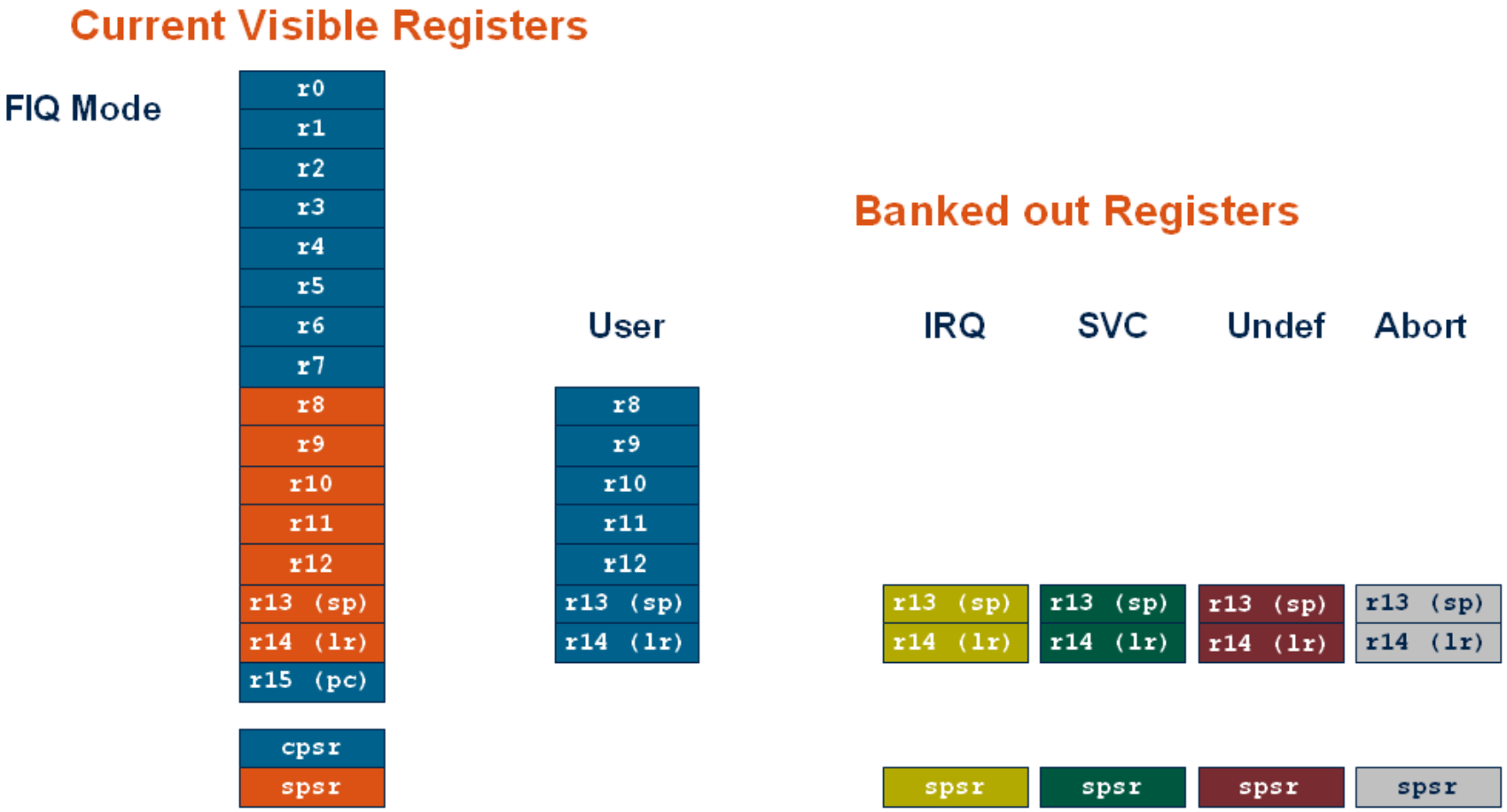
ARM's Register Set

- Each mode can access
  - A particular set of r0-r12 registers
  - A particular r13 (the stack pointer, sp) and r14 (the link register, lr)
  - The program counter, r15 (pc)
  - The current program status register, cpsr
- Privileged modes (except System) can also access
  - A particular saved program status register (spsr)

ARM's Register Organization



ARM's Register Organization



Banked Registers

- Banking of registers implies that the specific register depends not only on the number (r0, r1, r2 ... r15) but also on the processor mode
- The values stored in banked registers are preserved across mode changes.

Example

- Assume that the processor is executing in user mode
- Assume that the processor writes 0 in r0 and 8 in r8.
- Processor changes to FIQ mode
  - In FIQ mode the value of r0 is
  - If processor overwrites both r0 and r8 with 1 in FIQ mode and changes back to user mode
    - The new value stored in r0 (in user mode) is
    - The new value stored in r8 (in user mode) is

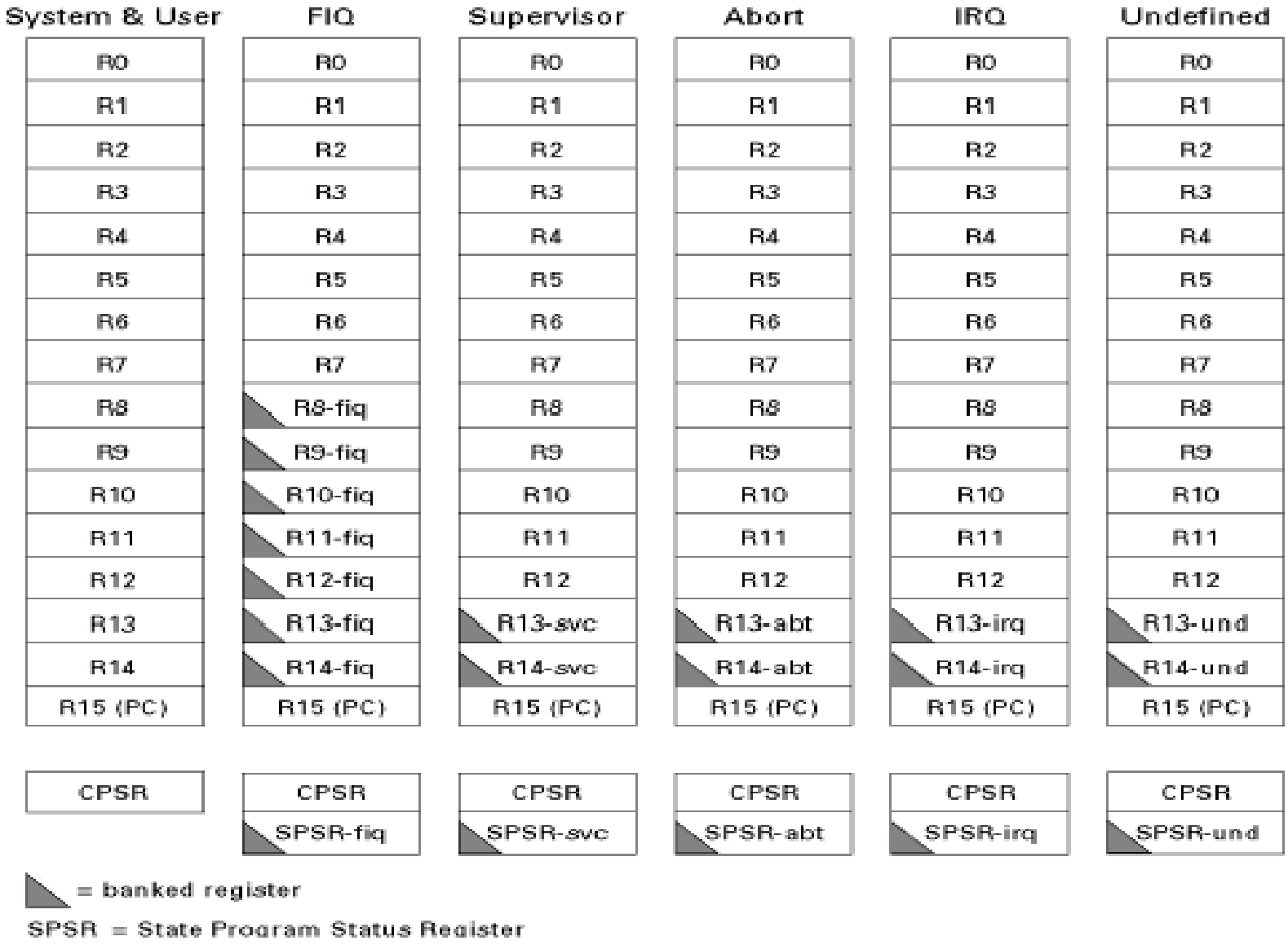
Banked Registers

- Banking of registers implies that the specific register depends not only on the number (r0, r1, r2 ... r15) but also on the processor mode
- The values stored in banked registers are preserved across mode changes.

Example

- Assume that the processor is executing in user mode
- Assume that the processor writes 0 in r0 and 8 in r8.
- Processor changes to FIQ mode
  - In FIQ mode the value of r0 is ***0***
  - If processor overwrites both r0 and r8 with 1 in FIQ mode and changes back to user mode
    - The new value stored in r0 (in user mode) is ***1***
    - The new value stored in r8 (in user mode) is ***8***

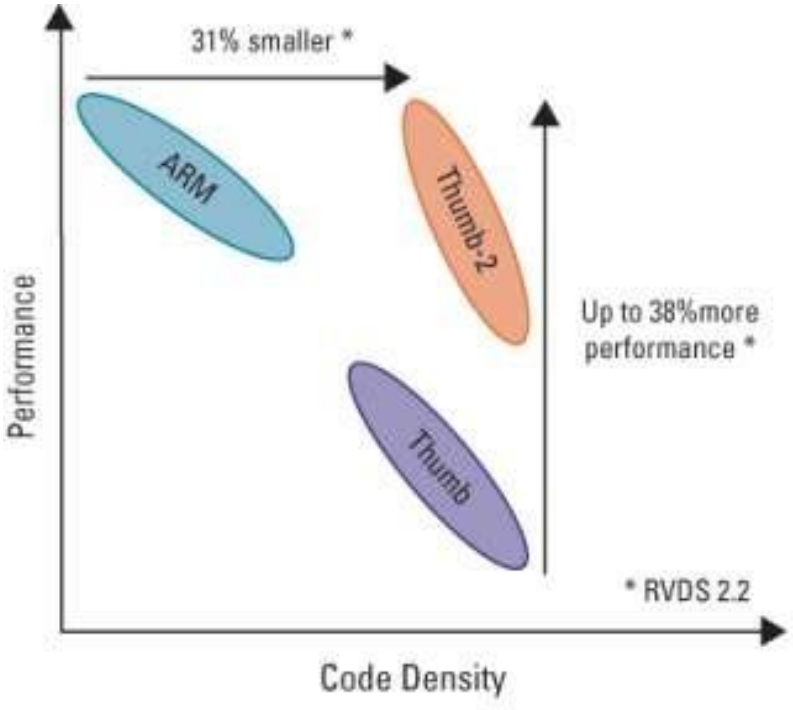
ARM Register Organization



Thumb instruction set

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 additional execution state – Thumb
- Switch between ARM and Thumb using BX instruction

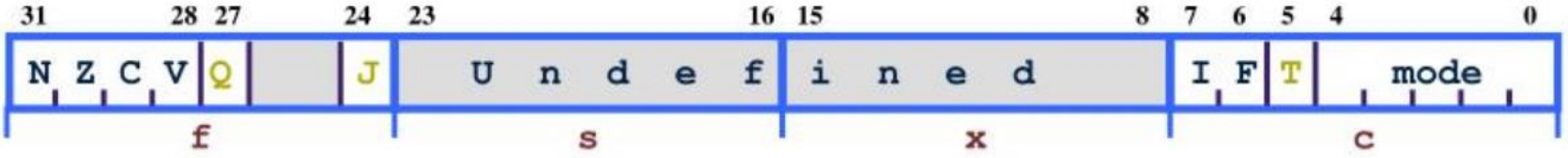


For most instructions generated by compiler

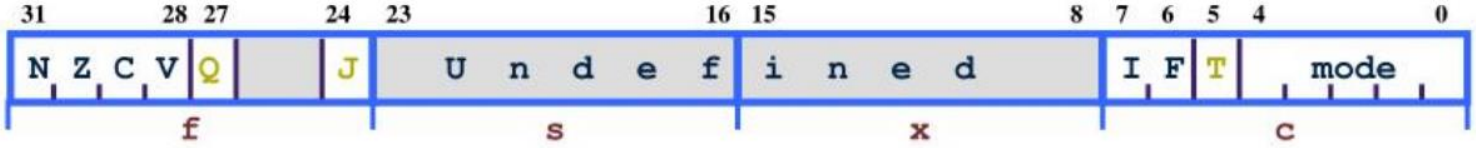
- Conditional execution is not used
- Source and destination registers identical
- Only Low registers used
- Constants are of limited size
- Inline barrel shifter not used

Current Program Status Register

- Current Program Status Register (cpsr) is a dedicated register
- Holds information about the most recently performed ALU operation
- Controls the enabling and disabling of interrupts (both IRQ and FIQ)
- Sets the processor operating mode
- Sets the processor state



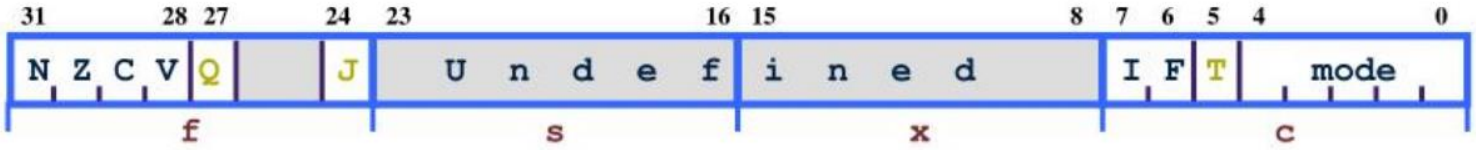
Current Program Status Register



- cpsr has two important pieces of information:
  - Flags: contains the condition flags
  - Control: contains the processor mode, state and interrupt mask bits
- All fields of the cpsr can be read/written in privileged modes
- Only the flag field of cpsr can be written in User mode, all fields can be read in User mode

M[4:0]	Mode
10000	User
10001	FIQ
10010	IRQ
10011	SVC
10111	Abort
11011	Undefined
11111	System

Current Program Status Register



- **Condition code flags**
  - N = **N**egative results from ALU
  - Z = **Z**ero result from ALU
  - C = ALU operation **C**arried out
  - V = ALU operation o**V**erflowed
- **Sticky Overflow flag – Q flag**
  - Architecture 5TE/J only
  - Indicates if saturation has occurred
- **J bit**
  - Architecture 5TE/J only
  - J = 1: Processor in Jazelle state
- **Interrupt Disable bits**
  - I = 1:Disables the IRQ
  - F = 1:Disables the FIQ
- **T bit**
  - Architecture xT only(Thumb)
  - T = 0: Processor in ARM state
  - T = 1: Processor in Thumb state
- **Mode bits**
  - Specify the processor mode

The ARM Assembly Language

- **ARM instructions can be broadly classified as:**
  - Data Processing Instructions: manipulate data within the registers
  - Branch Instructions: changes the flow of instructions or call a subroutine
  - Load-Store Instructions: transfer data between registers and memory
  - Software Interrupt Instructions: cause a software interrupt
  - Program Status Instructions: read / write the processor status registers

- **All instructions can access r0-r14 directly.**

- **Most instructions also allow use of the pc.**

- **Specific instructions to allow access to cpsr and spsr.**

Data Processing Instructions

- **Manipulate data within registers**
  - Move operations
  - Arithmetic operations
  - Logical operations
  - Comparison operations
  - Multiply operations

- **Appending the S suffix for an instruction, e.g. ADDS**
  - Signifies that the instruction's execution will update the flags in the *cpsr*

Typical ARM Data Processing Instructions

- **Operation:** Specifies the instruction to be performed Almost all ARM instructions can be conditionally executed
- **Cond:** Specifies the optional conditional flags which have to be set under which to execute the instruction
- **S bit:** Signifies that the instruction updates the conditional flags
- **Rd:** Specifies the destination register
- **Rn:** Specifies the first source operand register
- **ShifterOperand2:** Specifies the second source operand
  - Could be a register, immediate value, or a shifted register/immediate value

Some data processing instructions may not specify the destination register or the source register

Data Processing Instructions

- **Consist of**

Arithmetic:	**ADD**	**ADC**	**SUB**	**SBC**	**RSB**	**RSC**
Logical:	**AND**	**ORR**	**EOR**	**BIC**		
Comparisons:	**CMP**	**CMN**	**TST**	**TEQ**		
Data movement:	**MOV**	**MVN**				

These instruction only work on registers, **NOT** memory

Move Instructions

- *MOV* moves a 32-bit value into a register
- *MVN* moves the NOT of the 32-bit value into a register

Example

```
BEFORE    r5 = 5
          r7 = 8

          MOV r7,r5

AFTER     r5 =
          r7 =
```

Move Instructions

- *MOV* moves a 32-bit value into a register
- *MVN* moves the NOT of the 32-bit value into a register

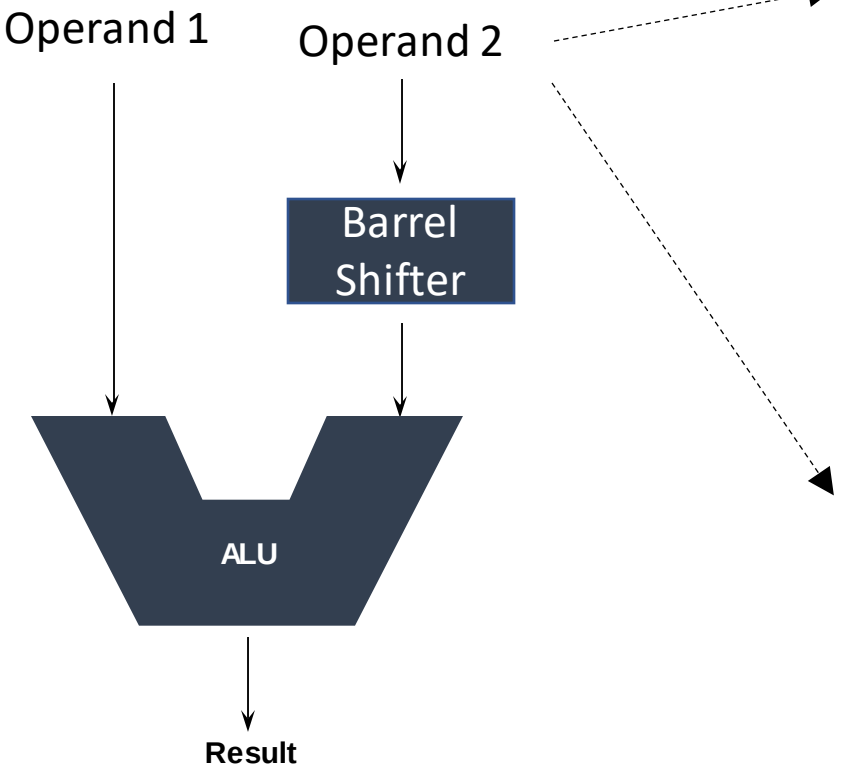
Example

```
BEFORE    r5 = 5
          r7 = 8

          MOV r7,r5

AFTER     r5 = 5
          r7 = 5
```

The Barrel Shifter



Register, optionally with shift operation

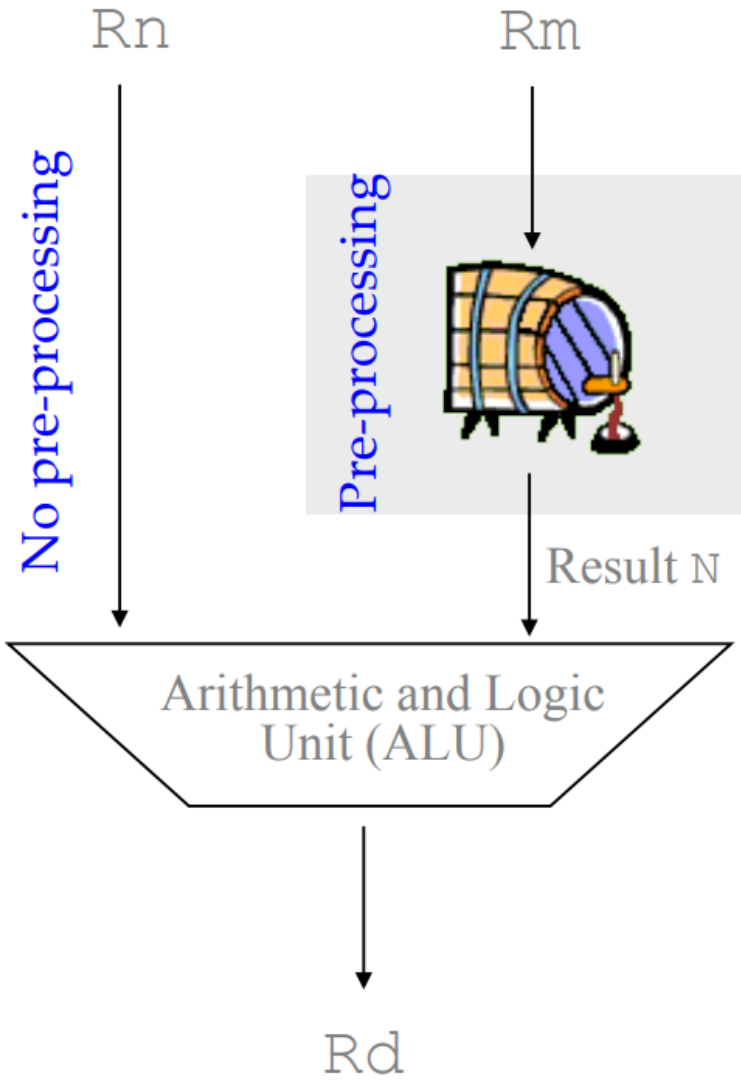
- Shift value can be either be:
  - 5-bit unsigned integer
  - Specified in bottom byte of another register
- Used for multiplication by constant

Immediate value, optionally with shift operation

- 8-bit number, with a range 0-255
- Allows increased range of 32-bit constants to be loaded directly into registers

The ARM Barrel Shifter

- Data processing instructions are processed within the ALU
- ARM can shift the 32-bit binary pattern in one of the source registers left or right by a specific number of positions before the value enters the ALU
- **Can achieve fast multiplies or division by a power of 2**
- Data-processing instructions that do not use the barrel shifter:
  - *MUL* (multiply)
  - CLZ (count leading zeros)



Using the Barrel Shifter

- *LSL* shifts bits to the left.

- Similar to the C–language operator <<

Example

```
BEFORE    r5 = 5
          r7 = 8

          MOV r7,r5, LSL #2

AFTER     r5 =
          r7 =
```

Using the Barrel Shifter

- *LSL* shifts bits to the left.

- Similar to the C–language operator <<

Example

```
BEFORE    r5 = 5
          r7 = 8

          MOV r7,r5, LSL #2

AFTER     r5 = 5
          r7 = 20
```

Arithmetic Instructions

- Addition and subtraction of 32-bit signed and unsigned values

Example - Subtraction

```
BEFORE   r0 = 0x00000000
         r1 = 0x00000002
         r2= 0x00000001

         SUB r0,r1,r2

AFTER    r0 =
         r1 =
         r2 =
```

Example - Addition

```
BEFORE   r0 = 0x00000000
         r1 = 0x00000005

         ADD r0,r1,r1, LSL #1

AFTER    r0 =
         r1 =
```

Arithmetic Instructions

- Addition and subtraction of 32-bit signed and unsigned values

Example - Subtraction

```
BEFORE   r0 = 0x00000000
         r1 = 0x00000002
         r2= 0x00000001

         SUB r0,r1,r2

AFTER    r0 = 1
         r1 = 2
         r2 = 1
```

Example - Addition

```
BEFORE   r0 = 0x00000000
         r1 = 0x00000005

         ADD r0,r1,r1, LSL #1

AFTER    r0 = 15
         r1 = 5
```

Logical Instructions

- Bit-wise logical operations on two source registers
- *AND, ORR, EOR, BIC*

Example - Logical OR

```
BEFORE   r0 = 0x00000000
         r1 = 0x02040608
         r2 = 0x10305070

         ORR r0,r1,r2

AFTER    r0 =
         r1 =
         r2 =
```

Example - Logical bit clear (BIC)

```
BEFORE   r1 = 0b1111
         r2 = 0b0101

         BIC r0,r1,r2

AFTER    r0 =
         r1 =
         r2 =
```

Logical Instructions

- Bit-wise logical operations on two source registers
- *AND, ORR, EOR, BIC*

Example - Logical OR

```
BEFORE   r0 = 0x00000000
         r1 = 0x02040608
         r2= 0x10305070

         ORR r0,r1,r2

AFTER    r0 = 0x12345678
         r1 = 0x02040608
         r2 = 0x10305070
```

Example - Logical bit clear (BIC)

```
BEFORE   r1 = 0b1111
         r2 = 0b0101

         BIC r0,r1,r2

AFTER    r0 = 0b1010
         r1 = 0b1111
         r2 = 0b0101
```

Comparison Instructions

- Compare or test a register with a 32-bit value
- CMP, CMN, TEQ, TST
- Registers under comparison are not affected; cpsr updated
- Do not need the S suffix

Example

```
BEFORE   cpsr = nzcvqiFt_USER
         r0 = 4
         r9 = 4

         CMP r0, r9

AFTER    cpsr=
         r0 =
         r9 =
```

Comparison Instructions

- Compare or test a register with a 32-bit value
- CMP, CMN, TEQ, TST
- Registers under comparison are not affected; cpsr updated
- Do not need the S suffix

Example

```
BEFORE   cpsr = nzcvqiFt_USER
         r0 = 4
         r9 = 4

         CMP r0, r9

AFTER    cpsr=8-010000
         r0 = 4
         r9 = 4
```

Multiply Instructions

- Multiple a pair of registers and optionally add (accumulate) the value stored in another register
- Special instructions called long multiplies accumulate onto a pair of registers representing a 64-bit value
- SMLAL, SMULL, UMLAL, UMUL

Example

```
BEFORE   r0 = 0
         r1 = 2
         r2 = 2

         MUL r0,r1,r2

AFTER    r0 =
         r1 =
         r2 =
```

Multiply Instructions

- Multiple a pair of registers and optionally add (accumulate) the value stored in another register
- Special instructions called long multiplies accumulate onto a pair of registers representing a 64-bit value
- SMLAL, SMULL, UMLAL, UMUL

Example

```
BEFORE   r0 = 0
         r1 = 2
         r2 = 2

         MUL r0,r1,r2

AFTER    r0 =4
         r1 =2
         r2 =2
```

Branch Instructions

- To change the flow of execution or to call a routine
- Supports subroutine calls, if-then-else structures, loops
- Change of execution forces the pc to point to a new address

- Four different branch instructions on the ARM
  - B<cond> label
  - BL<cond> label
  - BX<cond> Rm
  - BLX<cond> label | Rm

Condition Mnemonics

Suffix/Mnemonic	Description	Flags tested
EQ	Equal	**Z=1**
NE	Not equal	**Z=0**
CS/HS	Unsigned higher or same	**C=1**
CC/LO	Unsigned lower	**C=0**
MI	Minus	**N=1**
PL	Positive or Zero	**N=0**
VS	Overflow	**V=1**
VC	No overflow	**V=0**
HI	Unsigned higher	**C=1 & Z=0**
LS	Unsigned lower or same	**C=0 or Z=1**
GE	Greater or equal	**N=V**
LT	Less than	**N!=V**
GT	Greater than	**Z=0 & N=V**
LE	Less than or equal	**Z=1 or N=!V**
AL	Always	

Conditional Execution

- Most ARM instructions are conditionally executed
  - Instruction executes only if the condition-code flags satisfy a given state
- Increases performance
  - Reduces the number of branches, which reduces the number of pipeline flushes
- Improves code density
- Two-letter mnemonic appended to the instruction mnemonic

Usage Example
Naïve add operation ADD r0,r1,r2 Add operation if Zero flag is set

EQ	Equal	Z=1

ADDEQ r0,r1,r2

Value of Conditional Execution

- This improves code density and performance by reducing the number of forward branch instructions

Example		
C code	**Assembly**	**Conditional Assembly**

`if (x != 0)`
`  a=b+c;`
`else`
`  a=b-c;`
	`  CMP r3, #0`
	`  BEQ skip`
	`  ADD r0, r1, r2`
	`  B afterskip`
	`skip`
	`  SUB r0,r1,r2`
	`afterskip`
		`CMP r3,#0`
		`ADDNE r0,r1,r2`
		`SUBEQ r0,r1,r2`

Single Register Data Transfer

- ARM is based on a "load/store" architecture
- All operands should be in registers
- Load instructions are used to move data from memory into registers
- Store instructions are used to move data from registers to memory
- Flexible – allow transfer of a word or a half-word or a byte to and from memory

LDR/STR	Word
LDRB/STRB	Byte
LRH/STRH	Half-word Signed
LDRSB	byte load
LDRSH	Signed half-word load

- Syntax:

LDR<cond><size> Rd, <address>
STR<cond><size> Rd, <address>

LDR and STR

- LDR and STR instructions can load and store data on a boundary alignment that is the same as the datatype size being loaded or stored.

- LDR can only load 32-bit words on a memory address that is a multiple of 4 bytes – 0, 4, 8, and so on

- LDR r0, [r1]
  - Loads register r0 with the contents of the memory address pointed to by r1

- STR r0, [r1]
  - Stores the contents of register r0 to the memory address pointed to by r1

- Register r1 is called **base address register**

Addressing modes

- ARM provides three addressing modes:
  - Preindex with writeback
  - Preindex
  - Postindex

- Preindex mode useful for accessing a single element in a data structure

- Postindex and preindex with writeback useful for traversing an array

Addressing modes

- **Preindex with writeback**
  - Calculates address from a base register plus address offset
  - Updates the address in the base register with the new address
  - This is the address used to access memory
  - Example: LDR r0, [r1, #4]!

Example

```
BEFORE    r0 = 0x00000000
          r1 = 0x00009000
          mem32[0x00009000] = 0x01010101
          mem32[0x00000004] = 0x02020202

          LDR r0, [r1, #4]!

AFTER     r0 =
          r1 =
```

Addressing modes

- **Preindex with writeback**
  - Calculates address from a base register plus address offset
  - Updates the address in the base register with the new address
  - This is the address used to access memory
  - Example: LDR r0, [r1, #4]!

Example

```
BEFORE    r0 = 0x00000000
          r1 = 0x00009000
          mem32[0x00009000] = 0x01010101
          mem32[0x00009004] = 0x02020202

          LDR r0, [r1, #4]!

AFTER     r0 = 0x02020202
          r1 = 0x00009004
```

Addressing modes

- **Preindex**
  - Same as preindex with writeback, but does not update the base register
  - Example: LDR r0, [r1, #4]

Example

```
BEFORE    r0 = 0x00000000
          r1 = 0x00009000
          mem32[0x00009000] = 0x01010101
          mem32[0x00009004] = 0x02020202

          LDR r0, [r1, #4]

AFTER     r0 =
          r1 =
```

Addressing modes

- **Preindex**
  - Same as preindex with writeback, but does not update the base register
  - Example: LDR r0, [r1, #4]

Example

BEFORE r0 = 0x00000000
r1 = 0x00009000
mem32[0x00009000] = 0x01010101
mem32[0x00009004] = 0x02020202

LDR r0, [r1, #4]

AFTER r0 = ***0x02020202***
r1 = ***0x00009000***

Addressing modes

- **Postindex**
  - Only updates the base register *after* the address is used
  - Example: LDR r0, [r1], #4

Example

```
BEFORE    r0 = 0x00000000
          r1 = 0x00009000
          mem32[0x00009000] = 0x01010101
          mem32[0x00009004] = 0x02020202

          LDR r0, [r1], #4

AFTER     r0 =
          r1 =
```

Addressing modes

- **Postindex**
  - Only updates the base register *after* the address is used
  - Example: LDR r0, [r1], #4

Example

```
BEFORE    r0 = 0x00000000
          r1 = 0x00009000
          mem32[0x00009000] = 0x01010101
          mem32[0x00009004] = 0x02020202

          LDR r0, [r1], #4

AFTER     r0 = 0x01010101
          r1 = 0x00009004
```

More on addressing modes

- Address <address> accessed by LDR/STR is specified by a base register plus an offset.

Offset takes one of the three formats:

- **Immediate**
  - Offset is a number that can be added to or subtracted from the base register
  - LDR r0, [r1,#8]; r0=mem[r1+8]
  - LDR r0, [r1,#-8]; r0=mem[r1-8]
- **Register**
  - Offset is a general-purpose register that can be added to or subtracted from the base register.
  - LDR r0, [r1,r2]; r0=mem[r1+r2]
  - LDR r0, [r1,-r2]; r0=mem[r1-r2]
- **Scaled Register**
  - Offset is a general-purpose register shifted by an immediate value and then added to or subtracted from the base register.
  - LDR r0, [r1, r2, LSL #2]; r0=mem[r1+4*r2]
  - LDR r0, [r1, r2, RRX]; r0=mem[r1+RRX(r2)]

Why assembly?

- Break the conventions of your usual compiler, which might allow some optimizations
  - Example: Temporarily breaking rules about memory allocation, threading, calling conventions etc.
- Build interfaces between code fragments using incompatible conventions
- Example: Code produced by different compilers
- Gain access to unusual programming modes of your processor
- Example: 16 bit mode to interface startup
- Produce reasonably fast code for tight loops to cope with a bad non-optimizing compiler
- Produce hand-optimized code perfectly tuned for your particular hardware setup, though not to someone else's

Why assembly sucks?

- Long and tedious to write initially
- Bug-prone and bugs can be very difficult to chase
- Code can be fairly difficult to understand, modify and maintain.
- Code is non-portable to other architectures.
- Code is optimized only for a certain implementation of a specific architecture.
- You spend more time on a few details and don't focus on algorithmic design, which is where the most optimization opportunities often lie.
- Small changes in algorithmic design can completely invalidate all your existing code.
- Commercial optimizing compilers can outperform hand-coded assembly.
- "Compilers make it a lot easier to use complex data structures, compilers don't get bored halfway through, and generate reliably pretty good code." John Levine

Middle Ground

From Charles Fiterman on comp.compilers about Human vs. Computer-generated assembly code:

- The human should always win and here is why.
  - **First** the human writes the whole thing in a high level language.
  - **Second** he profiles it to find the hot spots where it spends its time.
  - **Third** he has the compiler produce assembly for those small sections of code.
  - **Fourth** he hand-tunes them looking for tiny improvements over the machine generated code.
- The human wins because he can use the machine.

Questions?