COMPUTER SYSTEMS ORGANIZATION

ARM ISA -- Spring 2012 -- IIIT-H -- Suresh Purini

ARM Processors – History

- First ARM Processor
  - Designed by: Acorn Computers Ltd.
  - Where: Cambridge, England.
  - Time: 1983-1985.
  - ARM stood for Acorn RISC Machine then.
- In 1990
  - Acorn Computers Ltd. became ARM Limited.
  - Acorn RISC Machine is renamed Advanced RISC Machine.

How ARM makes money?

- Not by selling processors like Intel.
- But by licensing its technology to a network of partners.
- A company that intends to use ARM core in its product
  - Have to pay an upfront licensee fee to gain access to the design.
  - Have to pay royalty for every chip that uses the licensed ARM design.

Homework: Read the web page at this link

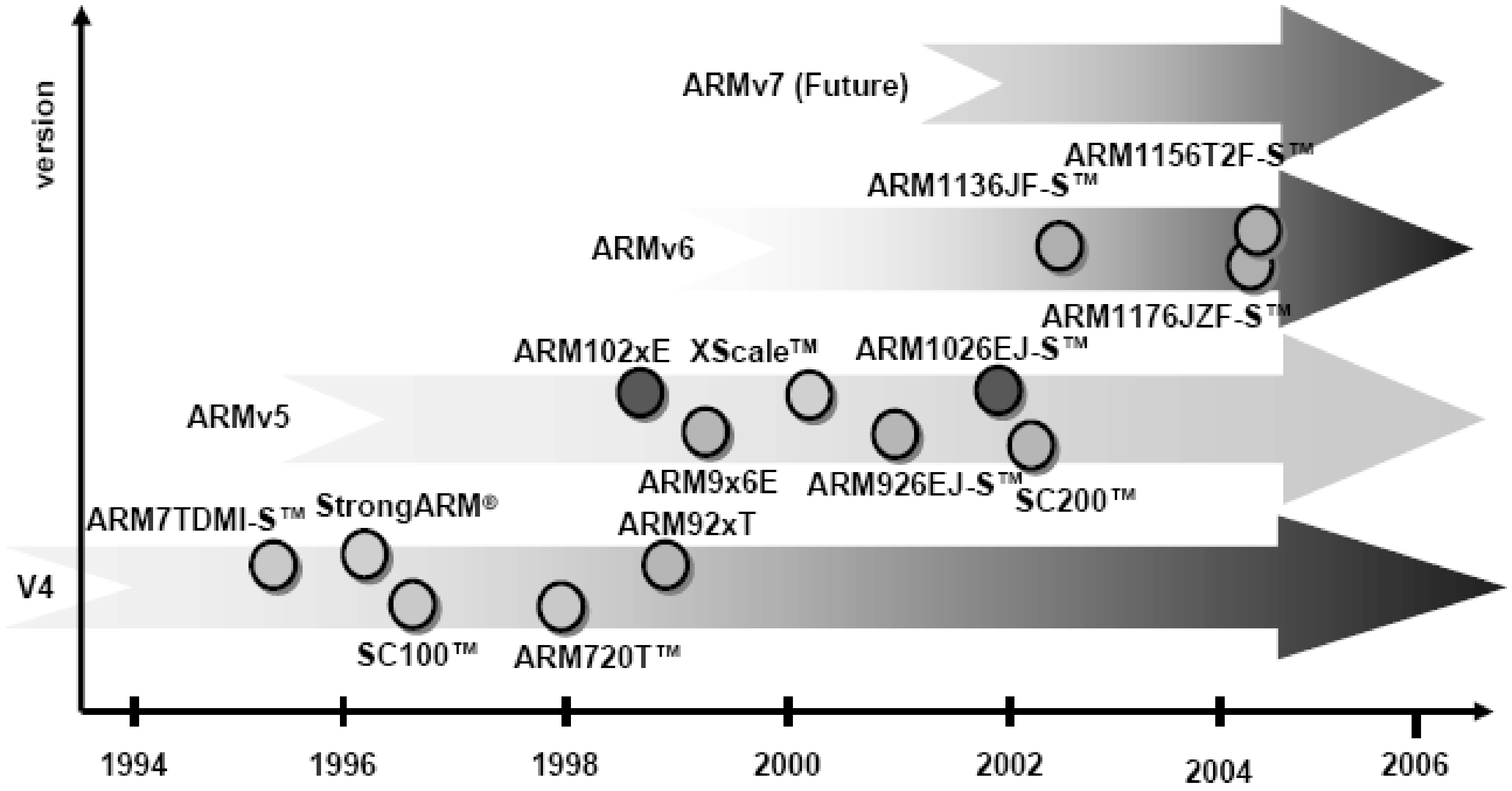
http://ir.arm.com/phoenix.zhtml?c=197211&p=irol-homeprofile

ARM Partnership Model

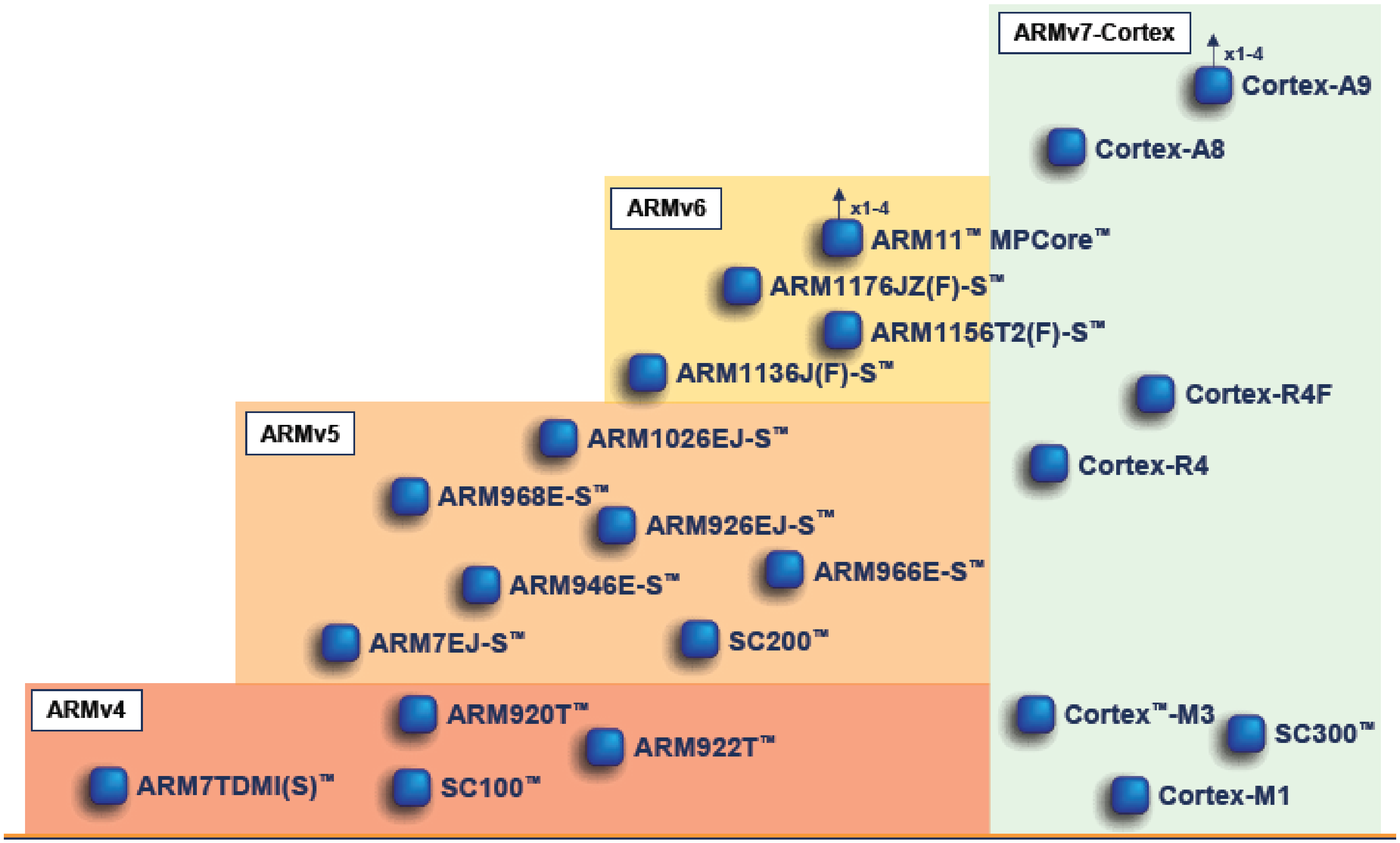
ATAP Partners
Tools Partners
RTOS Partners
Software Partners
ARM in Partnership

Applications

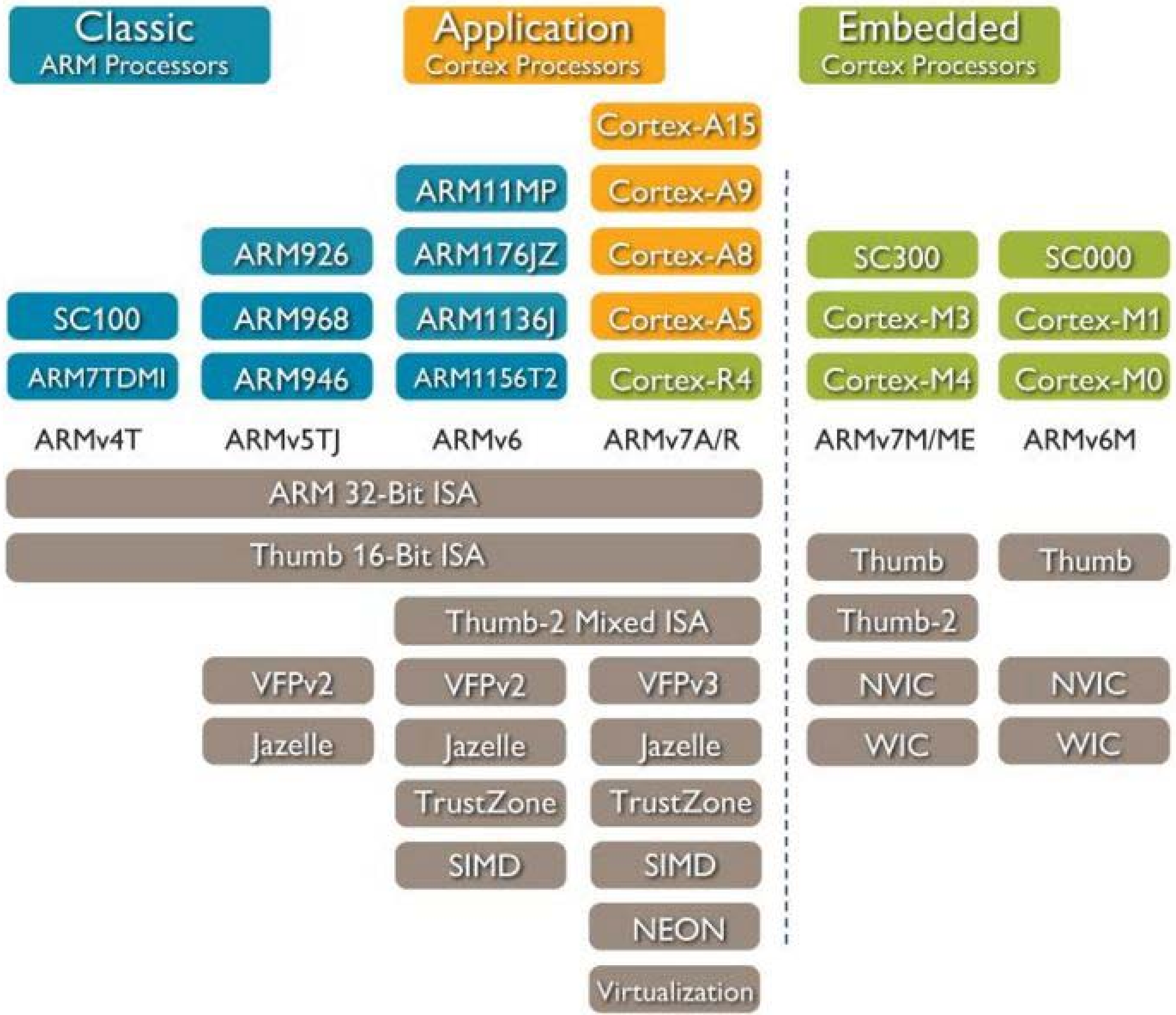
Architecture Revisions



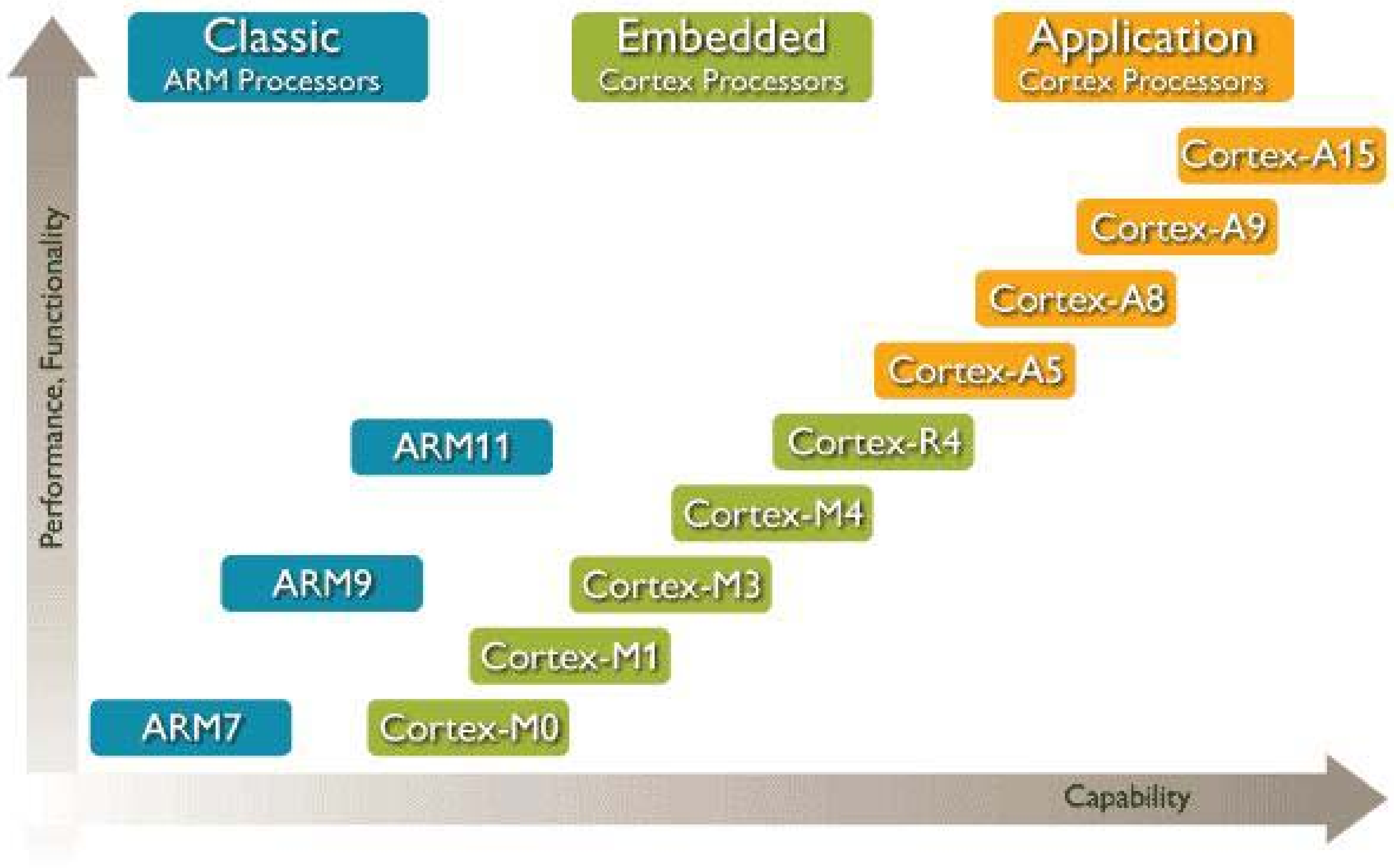
Architecture Versions



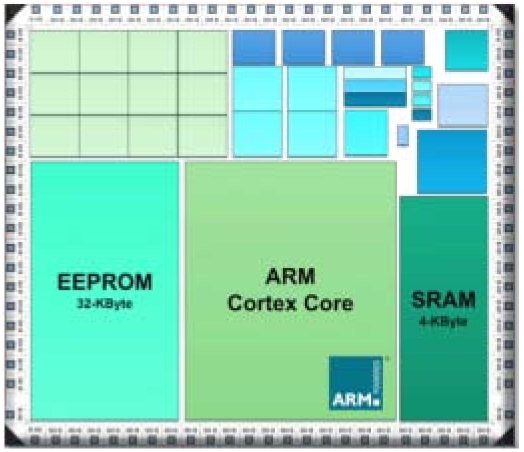
Architecture Versions and ISAs

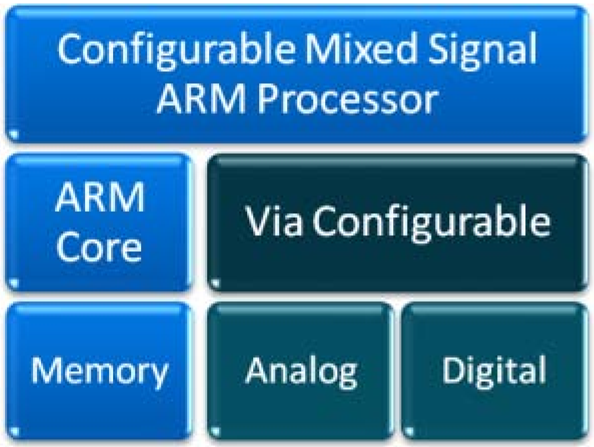


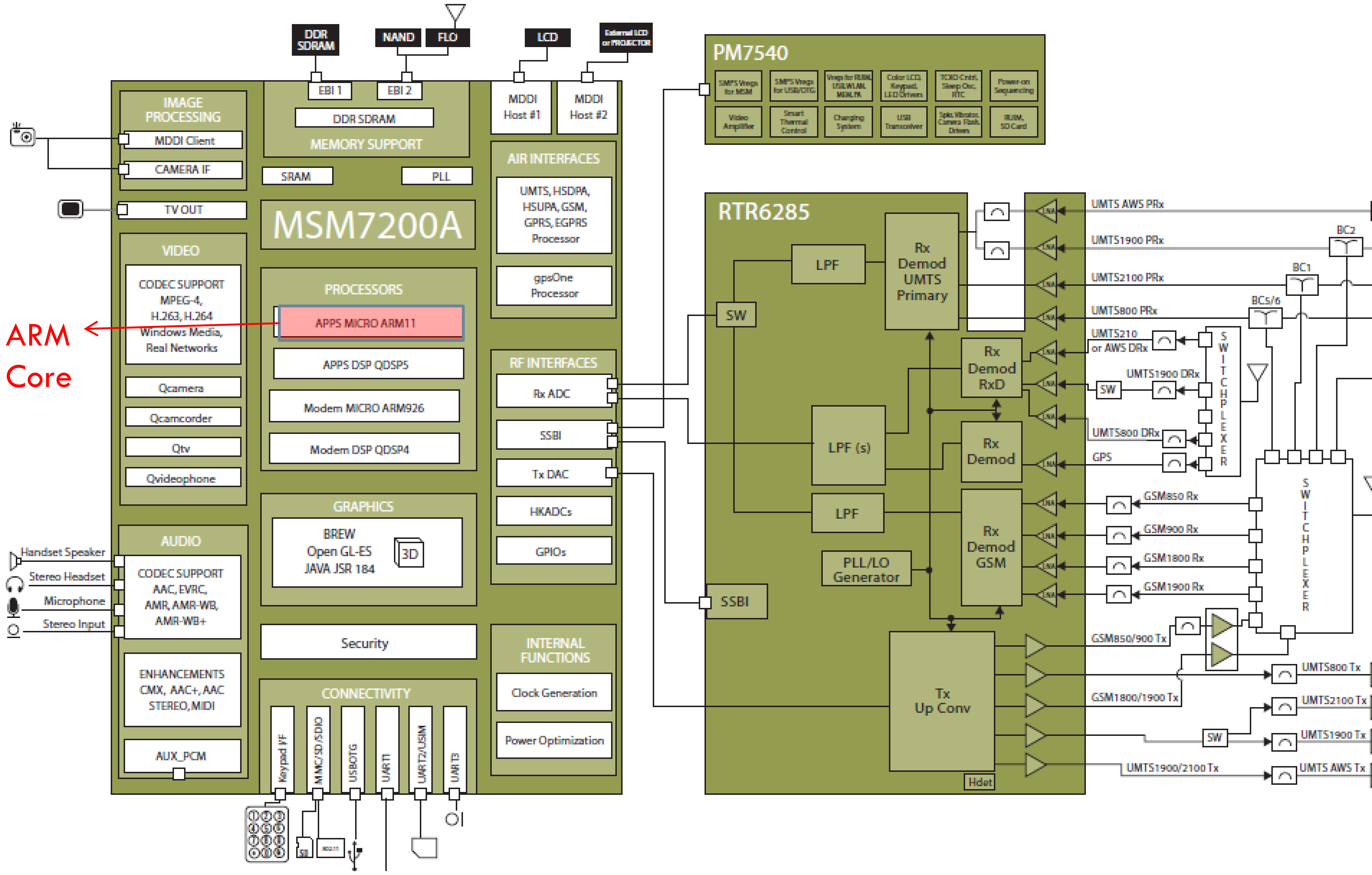
ARM Processors



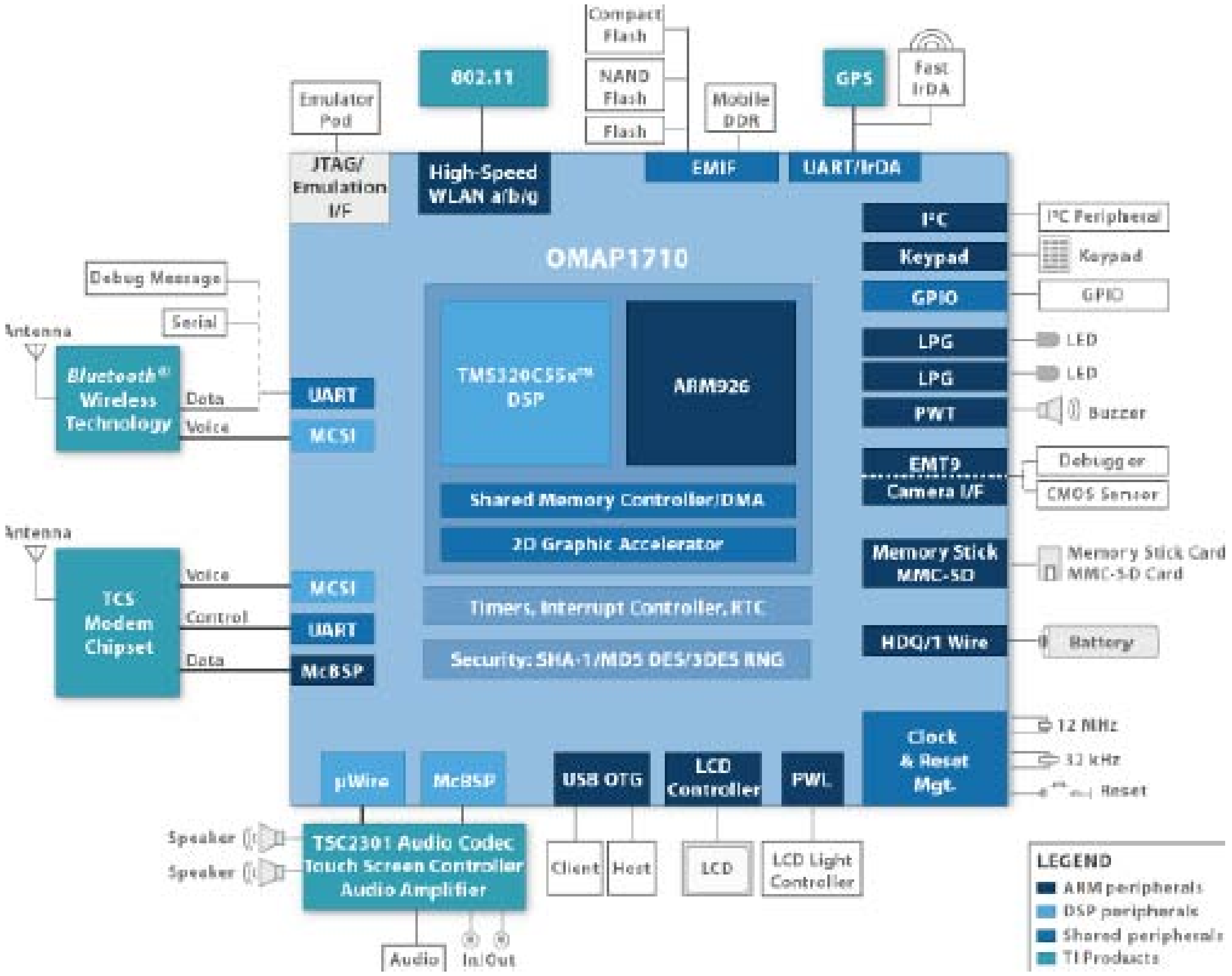
ARM Cores


EEPROM
32-KByte
ARM
Cortex Core
ARM
SRAM
4-KByte


Configurable Mixed Signal ARM Processor
ARM Core
Via Configurable
Memory
Analog
Digital

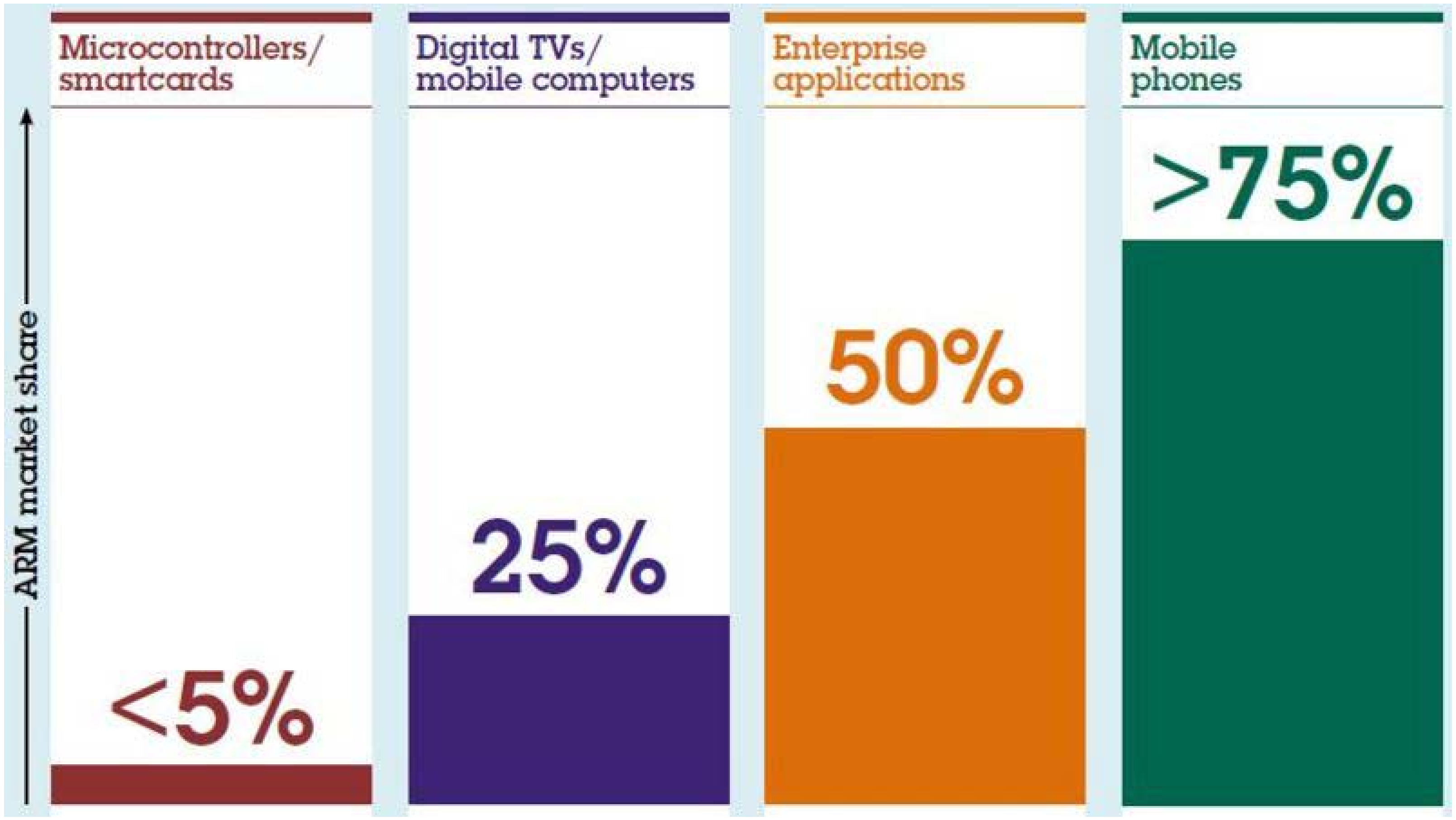

ARM Core
MSM7200A
DDR SDRAM
NAND
FLO
LCD
External LCD or PROJECTOR
EBI 1
EBI 2
DDR SDRAM
MEMORY SUPPORT
MDDI Host #1
MDDI Host #2
IMAGE PROCESSING
MDDI Client
CAMERA IF
TV OUT
SRAM
PLL
VIDEO
CODEC SUPPORT MPEG-4, H.263, H.264 Windows Media, Real Networks
Qcamera
Qcamcorder
Qtv
Qvideophone
PROCESSORS
APPS MICRO ARM11
APPS DSP QDSP5
Modem MICRO ARM926
Modem DSP QDSP4
AIR INTERFACES
UMTS, HSDPA, HSUPA, GSM, GPRS, EGPRS Processor
gpsOne Processor
RF INTERFACES
Rx ADC
SSBI
Tx DAC
HKADCs
GPIOs
GRAPHICS
BREW Open GL-ES JAVA JSR 184
3D
AUDIO
Handset Speaker
Stereo Headset
Microphone
Stereo Input
CODEC SUPPORT AAC, EVRC, AMR, AMR-WB, AMR-WB+
ENHANCEMENTS CMX, AAC+, AAC STEREO, MIDI
AUX_PCM
Security
INTERNAL FUNCTIONS
Clock Generation
Power Optimization
CONNECTIVITY
Keypad I/F
MMC/SD/SDIO
USBOTG
UART1
UART2/USIM
UART3
PM7540
SMPS Vregs for MSM
SMPS Vregs for USB/OTG
Vregs for RUIM, USB, WLAN, MDM, PA
Color LCD, Keypad, LCD Drivers
TCXO Cntrl, Sleep Osc, RTC
Power-on Sequencing
Video Amplifier
Smart Thermal Control
Charging System
USB Transceiver
Spkr, Vibrator, Camera Flash Drivers
RUIM, SD Card
RTR6285
LPF
Rx Demod UMTS Primary
SW
Rx Demod RxD
LPF (s)
Rx Demod
LPF
Rx Demod GSM
PLL/LO Generator
SSBI
Tx Up Conv
Hdet
UMTS AWS PRx
UMTS1900 PRx
UMTS2100 PRx
UMTS800 PRx
UMTS210 or AWS DRx
UMTS1900 DRx
UMTS800 DRx
GPS
BC2
BC1
BC5/6
SWITCHPLEXER
GSM850 Rx
GSM900 Rx
GSM1800 Rx
GSM1900 Rx
GSM850/900 Tx
GSM1800/1900 Tx
UMTS800 Tx
UMTS2100 Tx
UMTS1900 Tx
UMTS1900/2100 Tx
UMTS AWS Tx
SW

ARM Cores



Compact Flash
NAND Flash
Flash
Mobile DDR
802.11
GPS
Fast IrDA
Emulator Pod
JTAG/ Emulation I/F
High-Speed WLAN a/b/g
EMIF
UART/IrDA
I²C
I²C Peripheral
Keypad
Keypad
GPIO
GPIO
OMAP1710
Debug Message
Serial
Antenna
Bluetooth® Wireless Technology
Data
Voice
UART
MCSI
TMS320C55x™ DSP
ARM926
LPG
LED
LPG
LED
PWT
Buzzer
EMT9
Camera I/F
Debugger
CMOS Sensor
Shared Memory Controller/DMA
2D Graphic Accelerator
Antenna
TCS Modem Chipset
Voice
Control
Data
MCSI
UART
McBSP
Memory Stick MMC-SD
Memory Stick Card
MMC-SD Card
Timers, Interrupt Controller, RTC
Security: SHA-1/MD5 DES/3DES RNG
HDQ/1 Wire
Battery
12 MHz
32 kHz
Reset
Clock & Reset Mgt.
µWire
McBSP
USB OTG
LCD Controller
PWL
Speaker
Speaker
TSC2301 Audio Codec Touch Screen Controller Audio Amplifier
Client
Host
LCD
LCD Light Controller
Audio
In/Out
LEGEND
ARM peripherals
DSP peripherals
Shared peripherals
TI Products

ARM Market Share



ISA Design Philosophy

- Two ISA design philosophies
  - RISC – Reduced Instruction Set Computer
  - CISC – Complex Instruction Set Computer

RISC versus CISC

RISC	CISC
Each instruction does one simple task.	Each instruction can do multiple tasks.
Amount of work done in each instruction is roughly the same.	Amount of work done in each instruction could have huge variance.
Fixed Length instruction format.	Variable length instruction format.
Load-Store Architecture. Instruction operands should always reside in registers.	Instruction Operands can reside in memory also.
Large bank of general purpose registers.	Many special purpose registers.
Simple Addressing Modes.	Can have complex addressing modes.
Few Data Types (typically integer and float)	Could provide support for more data types like Strings.
Berkeley RISC , Stanford MIPS, ARM, HP's PA-RISC	Intel x86 line of processors

RISC or CISC – Which way should we go?

RISC	CISC
Simple, fast (pipelined) and power efficient hardware implementations.	Complex hardware, not so Power efficient, hard to come up with pipelined implementations.
Not so good for an Assemble Language Programmer when compared with CISC ISAs.	Good for an Assemble Language Programmer.
Good for compiler writer.	Compiler writer has to work hard to use the underlying CISC ISA features.
Less code density	Good code density

Principles of ISA Design

It is easy to see by formal-logical methods that there exist certain [instruction sets] that are in abstract adequate to control and cause the exception of any sequence of operations….The really decisive considerations from the present point of view, in selecting an [instruction set], are more of practical nature: simplicity of the equipment demanded by the [instruction set], and the clarity of its application to the actually important problems together with the speed of handling those problems.

- Burks, Goldstine and Von Neumann, 1947

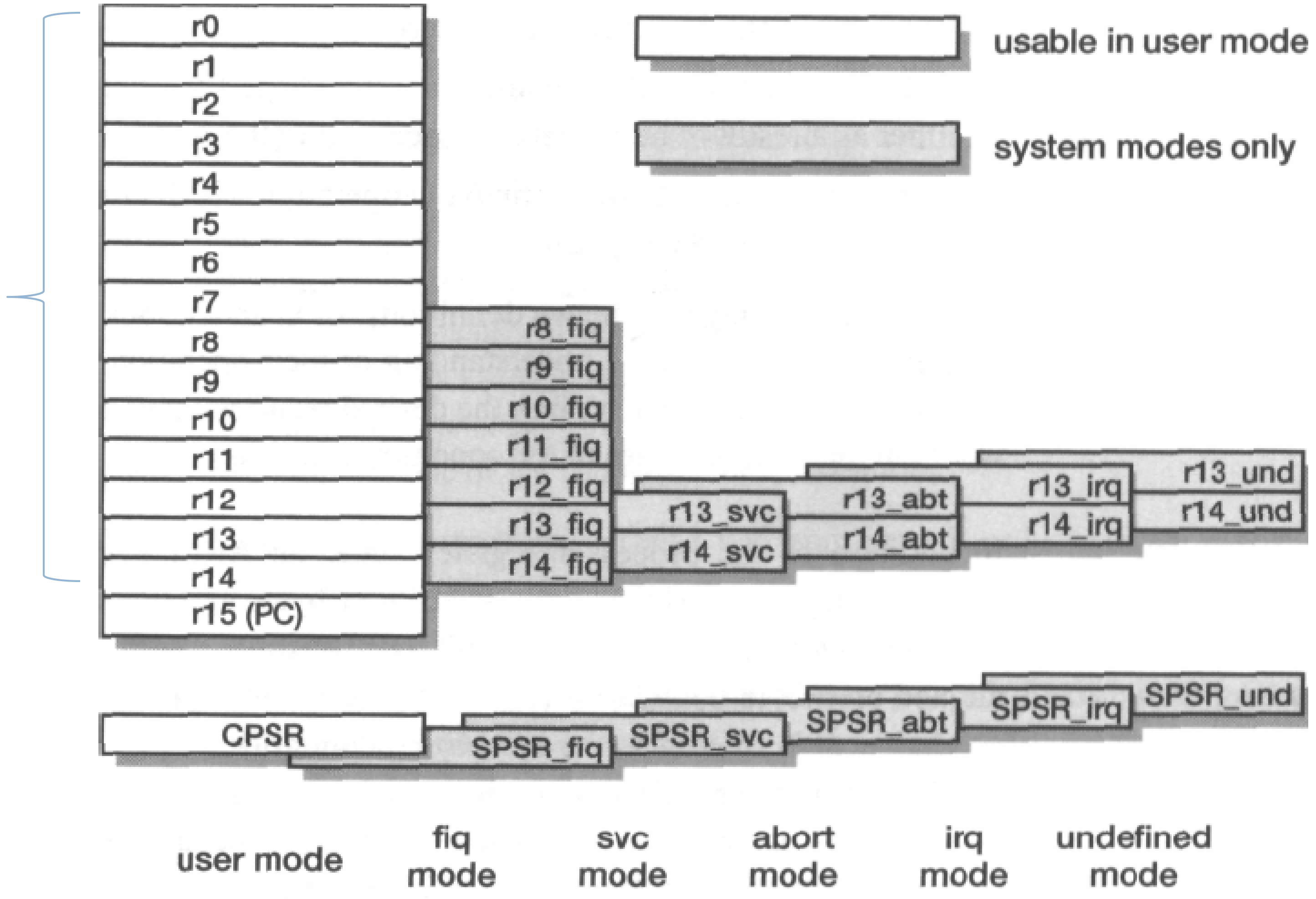
ARM ISA

- Follows RISC Philosophy but borrows some CISC ideas.
- ARM ISA goal
  - Efficient ISA implementation in hardware (Good Performance)
  - Good Code Density
  - Low Power Consumption
- 32-bit ISA – All instructions are encoded in 32-bits, 32-bit registers, Arithmetic on 32-bit values.
- 32-bit Address Space

ARM ISA – Register Set and Modes of Operation

ARM processor always runs either in user mode or one of the 5 System Modes.

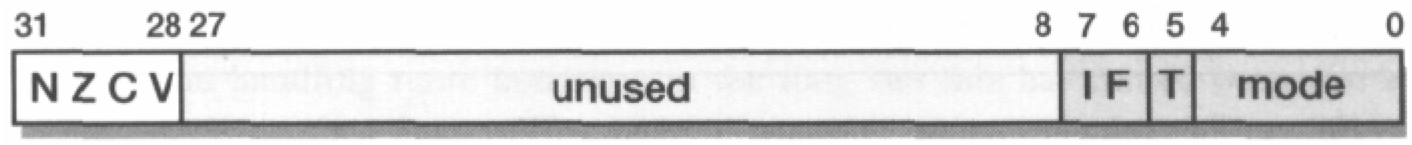
15 32-bit General Purpose Registers.



ARM ISA – Register Set

- All the registers are of length 32-bit.
- In the User Mode (Application Level Programs)
  - Registers r0 - r15 are available to the programmer.
  - r13 – sp (Stack Pointer)
  - r14 – lr (Link Register)
  - r15 – pc (Program Counter)
  - CPSR – Current Program Status Register is also accessible.
- Rest of the registers are used only in System-Level Programming and for handling Exceptions (like Interrupts)

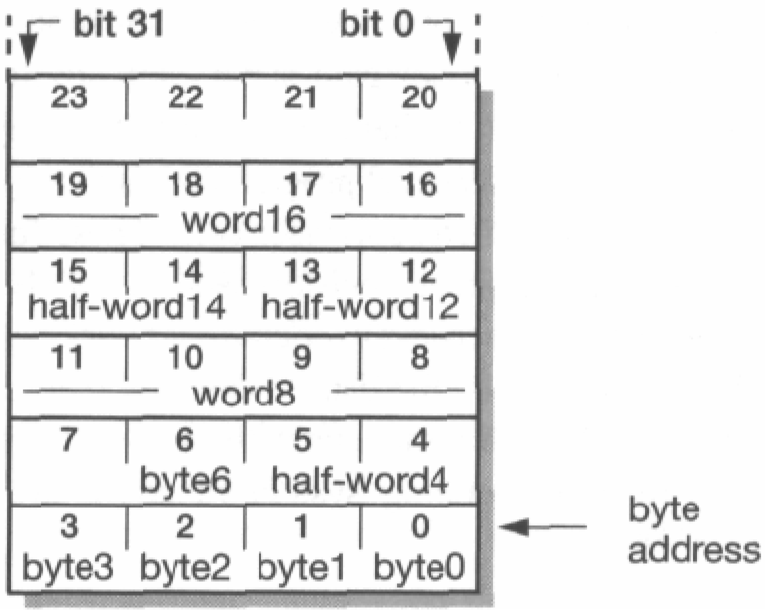
CPSR Register Format



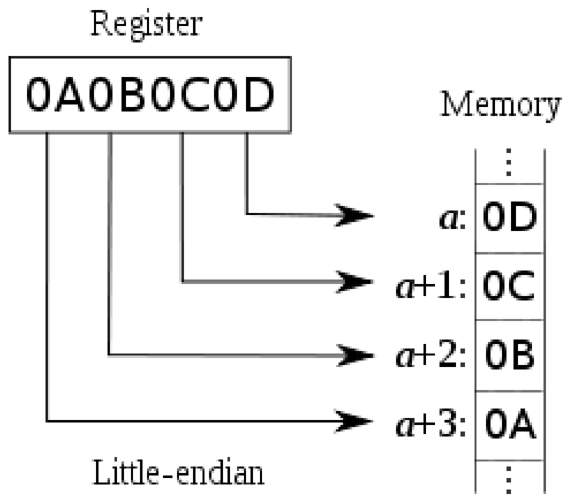
- N – Negative; the last ALU operation which changed the flags produced a negative result.
- Z – Zero
- C – Carry
- V – Overflow
- I and F – Interrupt enable flags (cannot be changed by programs running in User Mode)
- T – Thumb mode
- Mode – Mode bits indicates the Processor Mode.

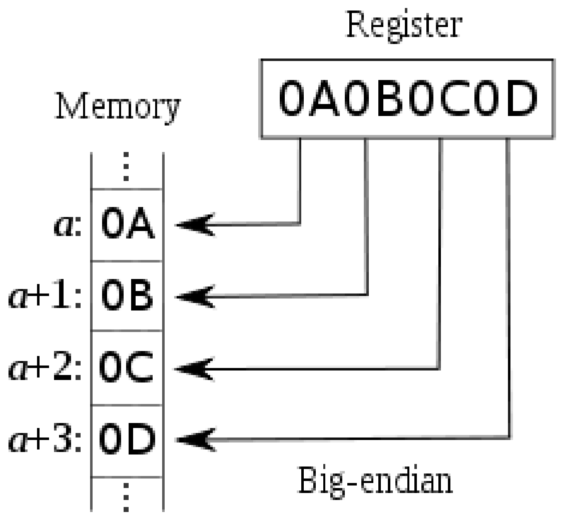
Memory System and Address Space

- Address Space Length: $2^{32} - 1$ bytes (4GB) .
- ARM Instruction can access
  - Byte sized data items
  - Half-word sized data items
  - Word-sized data items
- Alignment Restrictions: Instructions and words should be 4-byte aligned and half-words should be 2-byte aligned.
- Follows Little-Endian Convention
- Can be configured to be Big-Endian



Big-Endian Versus Little-Endian

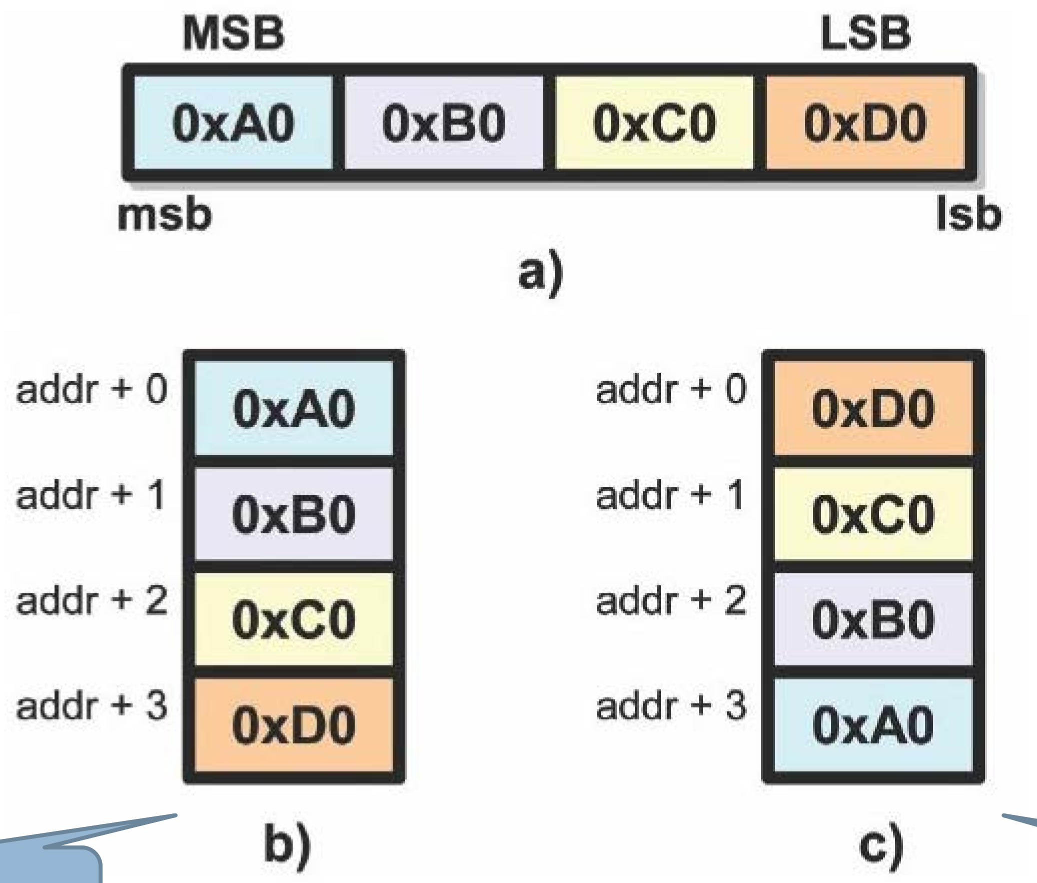




Little Endian: LSB goes to the smallest byte address.

Big Endian: MSB goes to the smallest byte address.

Big-Endian Versus Little-Endian



Big Endian

Little Endian

Big-Endian Versus Little-Endian

```
#include <stdio.h>
typedef unsigned char *byte_pointer;
void show_bytes(byte_pointer start, int len)
{
        int i;
        for (i = 0; i < len; i++)
                printf(" %.2x", start[i]);
        printf("\n");
}

main()
{
        int num = 0xF1F2F3F4;
        show_bytes((byte_pointer) &num, 4);
}
```

Is the output of this program same irrespective of whether it is run on a Little Endian machine or a Big Endian machine?

Load-Store Architecture

In Load-Store Architectures

- Instructions process (ADD, SUB, …) the data present only in the registers and result will also be placed in registers only.
- Only operations which apply to memory locations are ..
  - Load – Load from the memory location to a register.
  - Store – Store from the register to a memory location.

ARM Instruction Set

ARM Instructions can be classified into three categories.

1. Data Processing Instructions (like ADD, SUB, …)
2. Data Transfer Instructions (like LDR, STR, MOV, SWAP)
3. Control Flow Instructions

All Instructions are of 32-bit length.

Data Processing Instructions

- Arithmetic operations
- Bit-wise logical operations
- Register movement operations

Data Processing Instructions

- Let variables a and b are unsigned integers.
- Also let $r_0 \longleftarrow a$, $r1 \longleftarrow b$

C code

a = b + c

ARM Code

```
ADD r0, r1, r2   ; r0 = r1 + r2
```

r0 – Desitnation Operand (always a register, denoted as rd)

r1 – First Source Operand (always a register, denoted as rs)

r2 – Second Source Operand (Could be …. ?)

Shifted Register Operands

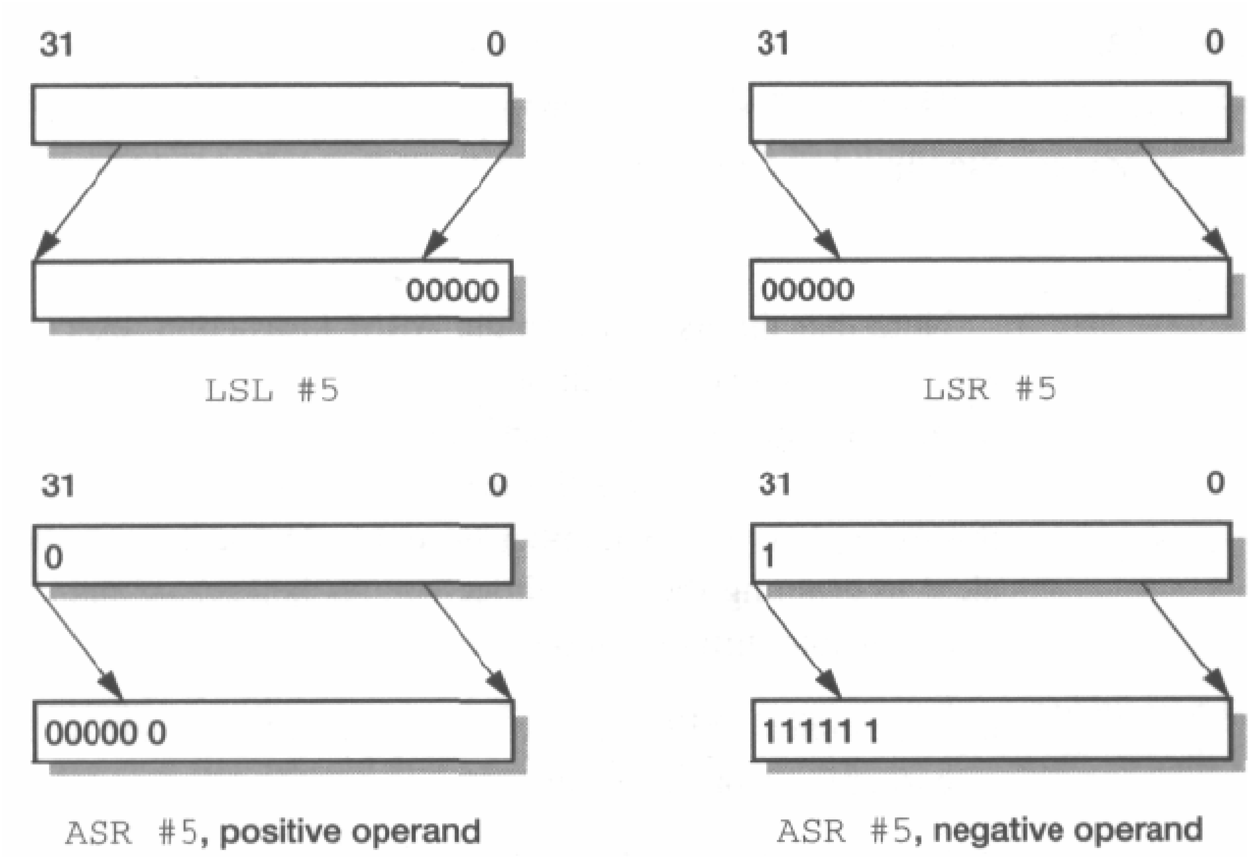
- The second source operand in an ARM instruction can be subjected shift operation.

```
ADD r3, r2, r1, LSL #3 ; r3 = r2 + 8 * r1
```

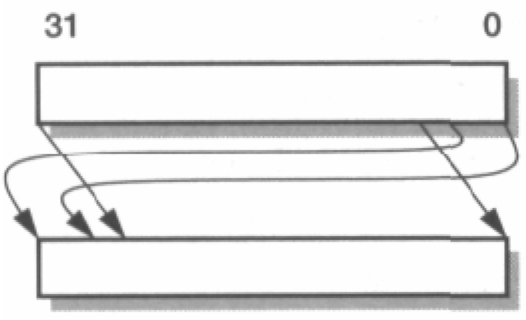
ADD r3, r2, r1, LSL r0 ; r3 = r2 + r1 * 2^{r0}

- For shift operations using immediate values, the operation can execute still within a clock cycle.
- For shift operations involving a register, the operation takes an extra clock cycle.

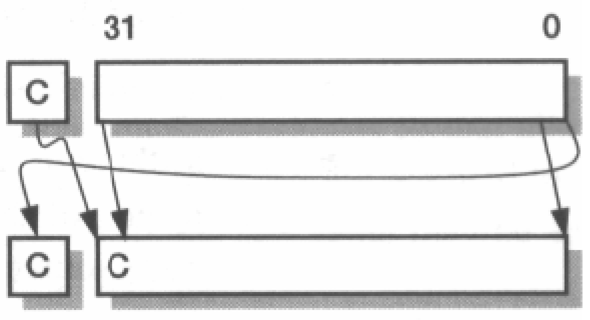
Shift Operations on the Second Source Operand



Shift Operations on the Second Source Operand



ROR #5



RRX

Data Processing Instructions – Immediate Operands

- Second Source Operand could be a constant

  ADD r0, r0, 1 ; r0 = r0 + 1

  ADD r2, r1, 8 ; r2 = r1 + 8

- Immediate operands should be of the form

  immediate = $(0 \rightarrow 255) * 2^{2n} \quad 0 \leq n \leq 12$

- But why?

Data Processing Instructions – Immediate Operands

- Immediate operands should be of the form

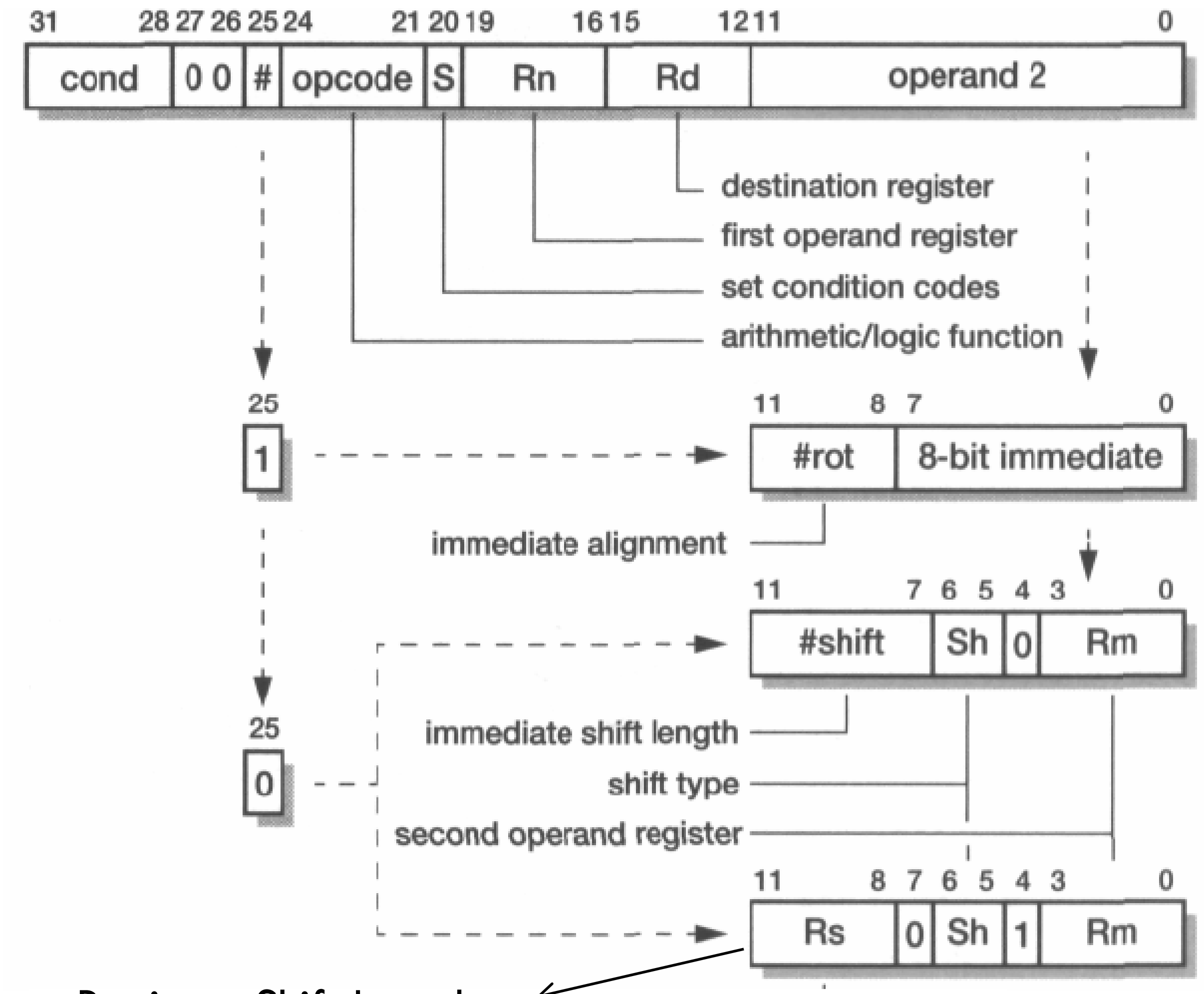
$$\text{immediate} = (0 \rightarrow 255) * 2^{2n} \quad 0 \leq n \leq 12$$

Valid Immediate Operands: 010000001, 1000000100, …

Invalid Immediate Operands: 100000010, 10000001000, …

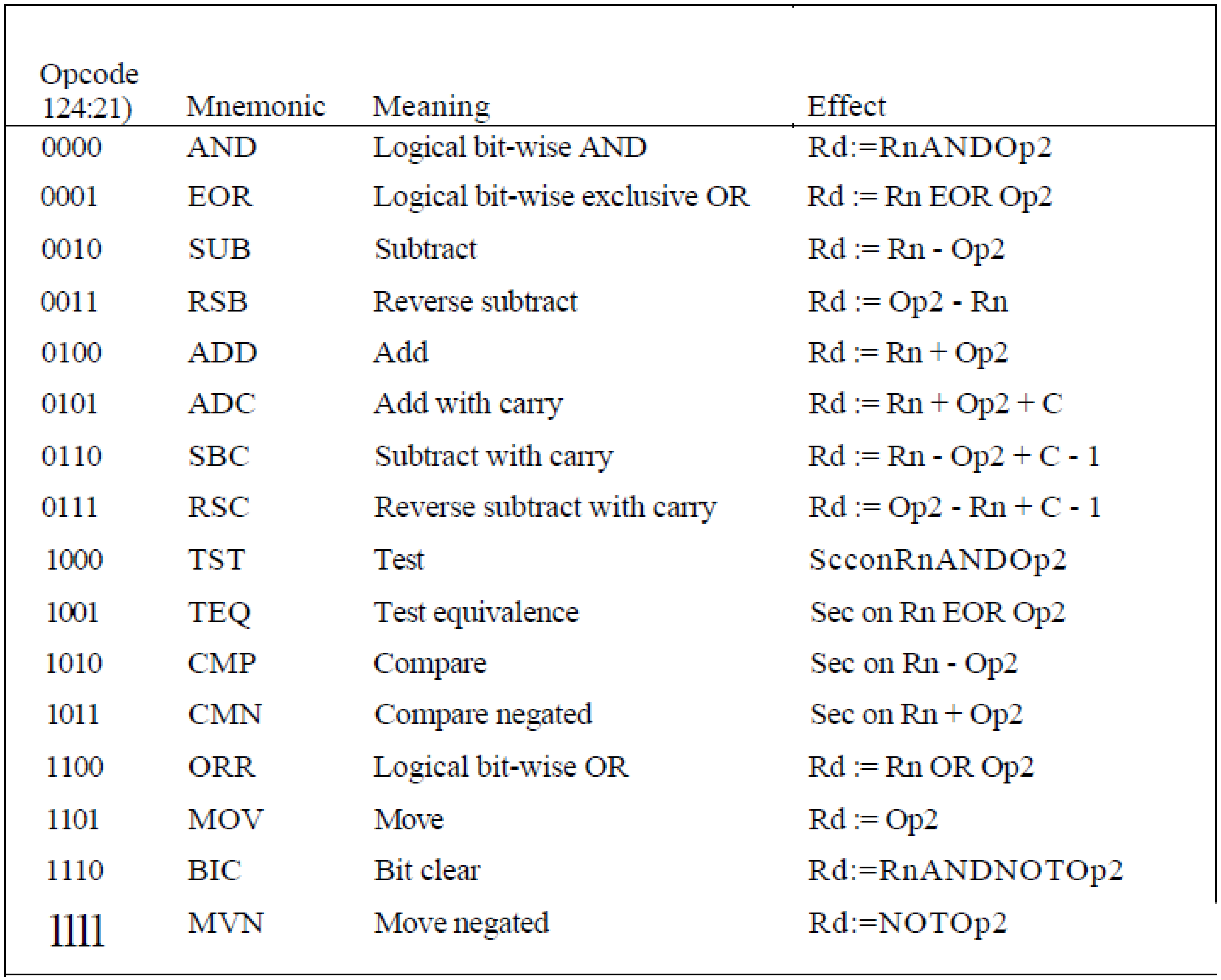
- What if want to use a constant in an instruction which is not a valid immediate operand?

Data Processing Instructions Format



Register Shift Length

Data Processing Instructions



Opcode 124:21)	Mnemonic	Meaning	Effect
0000	AND	Logical bit-wise AND	Rd:=RnANDOp2
0001	EOR	Logical bit-wise exclusive OR	Rd := Rn EOR Op2
0010	SUB	Subtract	Rd := Rn - Op2
0011	RSB	Reverse subtract	Rd := Op2 - Rn
0100	ADD	Add	Rd := Rn + Op2
0101	ADC	Add with carry	Rd := Rn + Op2 + C
0110	SBC	Subtract with carry	Rd := Rn - Op2 + C - 1
0111	RSC	Reverse subtract with carry	Rd := Op2 - Rn + C - 1
1000	TST	Test	SccontRnANDOp2
1001	TEQ	Test equivalence	Sec on Rn EOR Op2
1010	CMP	Compare	Sec on Rn - Op2
1011	CMN	Compare negated	Sec on Rn + Op2
1100	ORR	Logical bit-wise OR	Rd := Rn OR Op2
1101	MOV	Move	Rd := Op2
1110	BIC	Bit clear	Rd:=RnANDNOTOp2
1111	MVN	Move negated	Rd:=NOTOp2

Data Processing Instructions

- Let variables a, b, c, d, e be 32-bit signed or unsigned integers.
- Also let $r_0 \longleftarrow a$, $r1 \longleftarrow b$, $r2 \longleftarrow c$, $r3 \longleftarrow d$, $r4 \longleftarrow e$

C code

```
a = b + c

d = a - e
```

ARM Code

```
ADD r0, r1, r2   ; r0 = r1 + r2

SUB  r3, r0, r4   ; r3 = r0 – r4
```

ADD and SUB instructions here does not affect the flags in the CPSR register.

Data Processing Instructions

- Let variables f, g, h, i, j be 32-bit signed or unsigned integers.
- Also let $r_0 \longleftarrow f$, $r1 \longleftarrow g$, $r2 \longleftarrow h$, $r3 \longleftarrow i$, $r4 \longleftarrow j$

C code

```
f = (g + h) – (i + j)
```

ARM Code

```
ADD r5, r1, r2    ; r5 = g + h
ADD r6, r1, r2    ; r6 = i + j
SUB  r0, r5, r6   ; r5 and r6 hold the temporary intermediate values
```

Data Processing Instructions

- Let variables a, b, c be 64-bit signed or unsigned integers.
- Also let
  - $r_0 \longleftarrow$ lower half of a , r1 $\longleftarrow$ upper half of a
  - $r_2 \longleftarrow$ lower half of b , r3 $\longleftarrow$ upper half of b
  - $r_4 \longleftarrow$ lower half of c , r5 $\longleftarrow$ upper half of c

C code

```
c = a + b
```

ARM Code

```
ADDS r4, r0, r2   ; 'S' will set the carry flag if there is a carryout bit
ADC r5, r1, r3   ; r5 = r1 + r3 + C
```

Setting the Condition Code Flags

- Data processing instructions can set the condition codes (N, Z, C and V of CPSR) by adding the suffix ‘S’ to the instruction opcode.
  - ADDS, SUBS, ADCS, ….
- Comparison operations always set the condition codes even without the suffix ‘S’

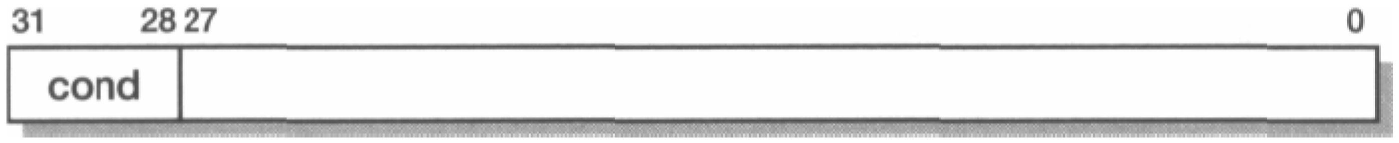
Setting the Condition Code Flags

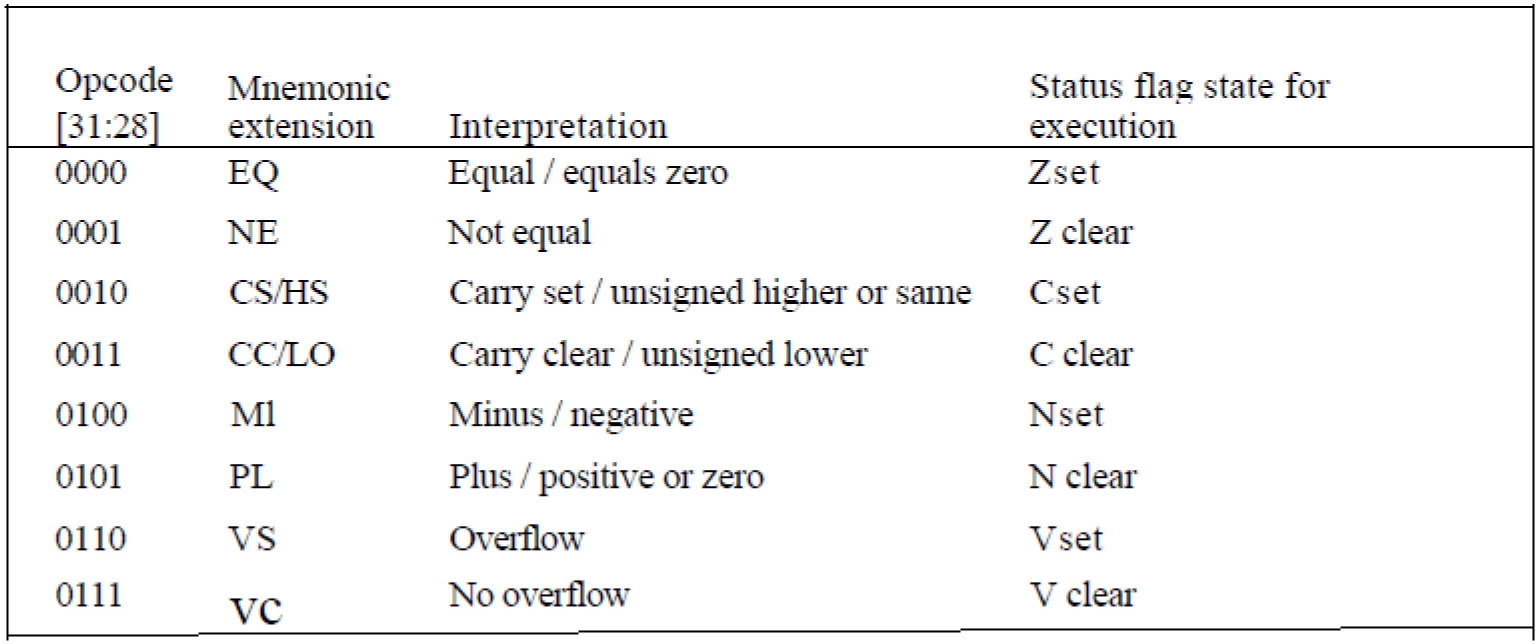
Always consult the ISA manual to check how the condition code flags are affected.

- The N flag is set if the result is negative (bit 31 of the result is set).
- Z flag is set if the result is zero, otherwise it is cleared.
- The C flag is set to carry-out from ALU when the operation is arithmetic (ADD, ADC, SUB, SBC, RSB, RSC, CMP, CMN)
  - Or to the carry-out from the shifter otherwise. If no shift is required, C is preserved.
- V flag is preserved in non-arithmetic operations. V flag is set if there is an overflow from bit 30 into bit 31 and cleared if no overflow occurs. V flag has significance only in signed arithmetic.

ARM Condition Codes – Predicated Execution of Instructions

- The most significant bits of any instruction specifies a condition under which the instruction will be executed.

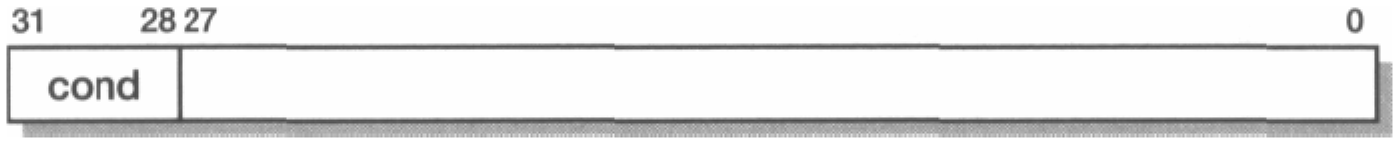


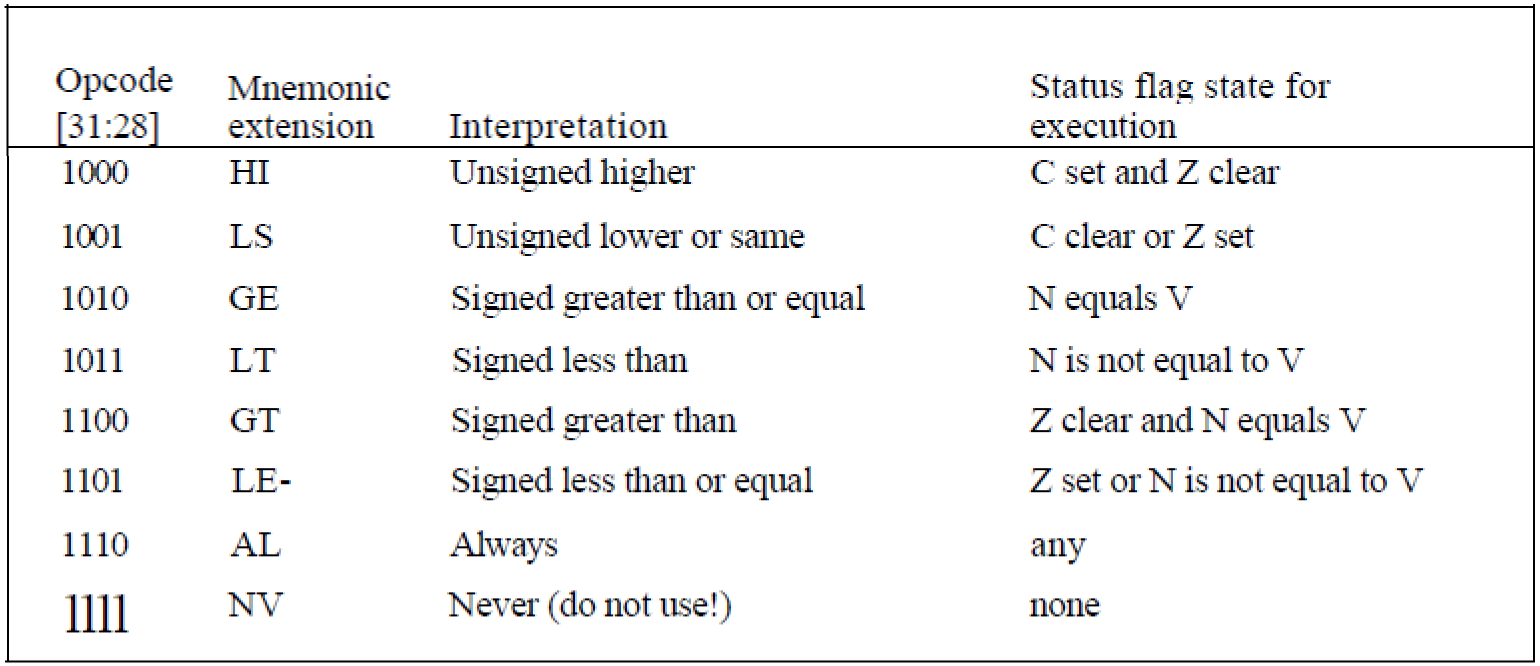


Opcode [31:28]	Mnemonic extension	Interpretation	Status flag state for execution
0000	EQ	Equal / equals zero	Zset
0001	NE	Not equal	Z clear
0010	CS/HS	Carry set / unsigned higher or same	Cset
0011	CC/LO	Carry clear / unsigned lower	C clear
0100	MI	Minus / negative	Nset
0101	PL	Plus / positive or zero	N clear
0110	VS	Overflow	Vset
0111	VC	No overflow	V clear

ARM Condition Codes – Predicated Execution of Instructions

- The most significant bits of any instruction specifies a condition under which the instruction will be executed.





Opcode [31:28]	Mnemonic extension	Interpretation	Status flag state for execution
1000	HI	Unsigned higher	C set and Z clear
1001	LS	Unsigned lower or same	C clear or Z set
1010	GE	Signed greater than or equal	N equals V
1011	LT	Signed less than	N is not equal to V
1100	GT	Signed greater than	Z clear and N equals V
1101	LE-	Signed less than or equal	Z set or N is not equal to V
1110	AL	Always	any
1111	NV	Never (do not use!)	none

ARM Code Snippets

C code: if (i==j) f = g+h ; else f = g-h

Assume r0 ⟵ f, r1 ⟵ g, r2 ⟵ h, r3 ⟵ i, r4 ⟵ j.

ARM code 1:

```
cmp r3, r4
addeq r0, r1, r2
subne r0, r1, r2
```

ARM code 2:

```
      cmp r3, r4
      bne else
      add r0, r1, r2
      b exit
else: sub r0, r1, r2
exit:
```

ARM code 3:

```
      cmp r3, r4
      beq if
      sub r0, r1, r2
      b exit
if:   add r0, r1, r2
exit:
```

Which of the 3 code sequences are good?

ARM Code Snippets

C code: if (i<=j) f = g+h ; else f = g-h **(i, j are unsigned numbers)**

Assume r0 ⟵ f, r1 ⟵ g, r2 ⟵ h, r3 ⟵ i, r4 ⟵ j.

ARM code 1:

```
cmp r3, r4
addls r0, r1, r2
subhi r0, r1, r2
```

ARM code 2:

```
        cmp r3, r4
        bhi else
        add r0, r1, r2
        b exit
else:   sub r0, r1, r2
exit:
```

ARM code 3:

```
        cmp r3, r4
        bls if
        sub r0, r1, r2
        b exit
if:     add r0, r1, r2
exit:
```

Which of the 3 code sequences are good?

ARM Code Snippets

C code: if (i<=j) f = g+h ; else f = g-h **(i, j are signed numbers)**

Assume r0 ⟵ f, r1 ⟵ g, r2 ⟵ h, r3 ⟵ i, r4 ⟵ j.

ARM code 1:

```
cmp r3, r4
addle r0, r1, r2
subgt r0, r1, r2
```

ARM code 2:

```
      cmp r3, r4
      bgt else
      add r0, r1, r2
      b exit
else: sub r0, r1, r2
exit:
```

ARM code 3:

```
         cmp r3, r4
         ble if
         sub r0, r1, r2
         b exit
if:      add r0, r1, r2
exit:
```

Which of the 3 code sequences are good?

ARM Code Snippets

C code: while (save[i] == k) i+=1 ;

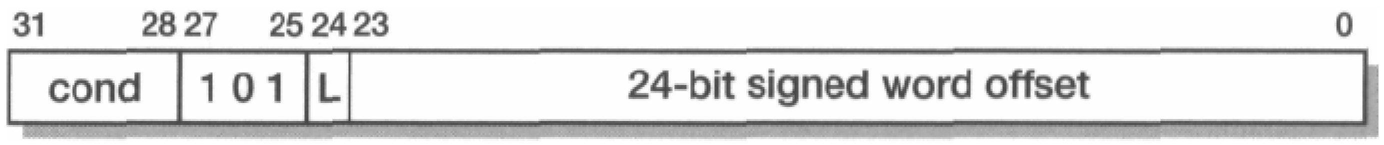
r3 ⟵ i r5 ⟵ k r6 ⟵ save

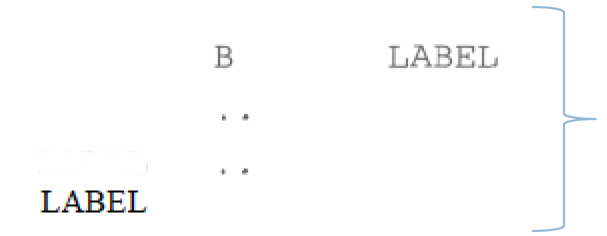
```
loop:   add r12, r6, r3, LSL #2      ; r12 = &save[i]
        ldr r0, [r12, #0]            ;  r0 = save[i]
        cmp r0, r5
        bne  exit                    ; branch if save[i] != k
        add r3, r3, #1
        b loop
exit:
```

Control Flow Instructions and PC-Relative Addressing

ARM provides two branch instructions

B (branch) and BL (branch and link)





Example usage

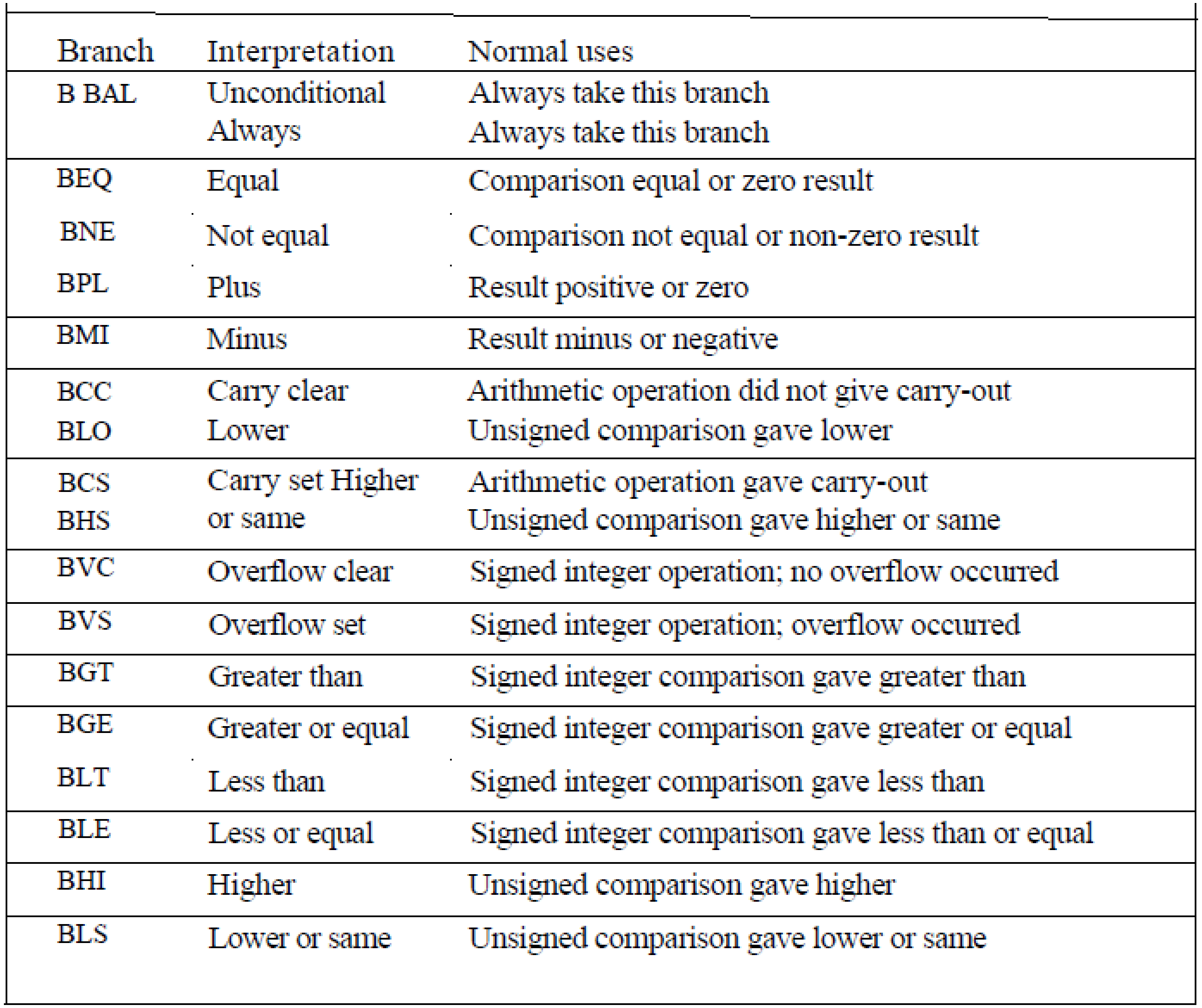
$$PC = PC + (SignExtend\_30(signed\_immed\_24) << 2)$$

Hey, but what is the contents of the register r15 (PC) now ?

- PC = Address of the Branch Instruction + 8

Branch range: Approximately (-32MB to +32MB)

Conditional Branch Instruction Variants



Branch	Interpretation	Normal uses
B	Unconditional	Always take this branch
BAL	Always	Always take this branch
BEQ	Equal	Comparison equal or zero result
BNE	Not equal	Comparison not equal or non-zero result
BPL	Plus	Result positive or zero
BMI	Minus	Result minus or negative
BCC	Carry clear	Arithmetic operation did not give carry-out
BLO	Lower	Unsigned comparison gave lower
BCS	Carry set	Arithmetic operation gave carry-out
BHS	Higher or same	Unsigned comparison gave higher or same
BVC	Overflow clear	Signed integer operation; no overflow occured
BVS	Overflow set	Signed integer operation; overflow occurred
BGT	Greater than	Signed integer comparison gave greater than
BGE	Greater or equal	Signed integer comparison gave greater or equal
BLT	Less than	Signed integer comparison gave less than
BLE	Less or equal	Signed integer comparison gave less than or equal
BHI	Higher	Unsigned comparison gave higher
BLS	Lower or same	Unsigned comparison gave lower or same

ARM Code Snippets

```
        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
```

ARM Code Snippets

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

Equivalent ARM Code Sequence

```
CMP      r0, #5                ; if (r0 != 5) {
ADDNE    r1, r1, r0            ;    r1 := r1 + r0 - r2
SUBNE    r1, r1, r2            ; }
```

ARM Code Snippets

if (r0 == r1) { r2 = r2 + 1; r3 = r3 + 1; r4 = r4 + 1; r5 = r5 + 1 }
else { r6 = r6 + 1; r7 = r7 + 1; r8 = r8 + 1; r9 = r9 + 1 }

Code Sequence 1

```
cmp r0, r1
addeq r2, #1
addeq r3, #1
addeq r4, #1
addeq r5, #1
addne r6, #1
addne r7, #1
addne r8, #1
addne r9, #1
```

Code Sequence 2

```
        cmp r0, r1
        bne else
        add r2, #1
        add r3, #1
        add r4, #1
        add r5, #1
        b exit
else:   add r6, #1
        add r7, #1
        add r8, #1
        add r9, #1
exit:
```

Code Sequence 3

```
        cmp r0, r1
        beq if
        add r6, #1
        add r7, #1
        add r8, #1
        add r8, #1
        b exit
if:     add r2, #1
        add r3, #1
        add r4, #1
        add r5, #1
exit:
```

ARM Code Snippets

C code: if ((a==b) && (c==d)) e++;

ARM Code:

```
cmp r0, r1
cmpeq r2, r3
addeq r4, r4, #1
```

Base Plus Offset Addressing Modes

- Indexed Addressing Mode with no write back

  ldr r0, [r1, #4]   ; r0 = mem_{32}[r1+4]
- Pre-Indexed Addressing Mode

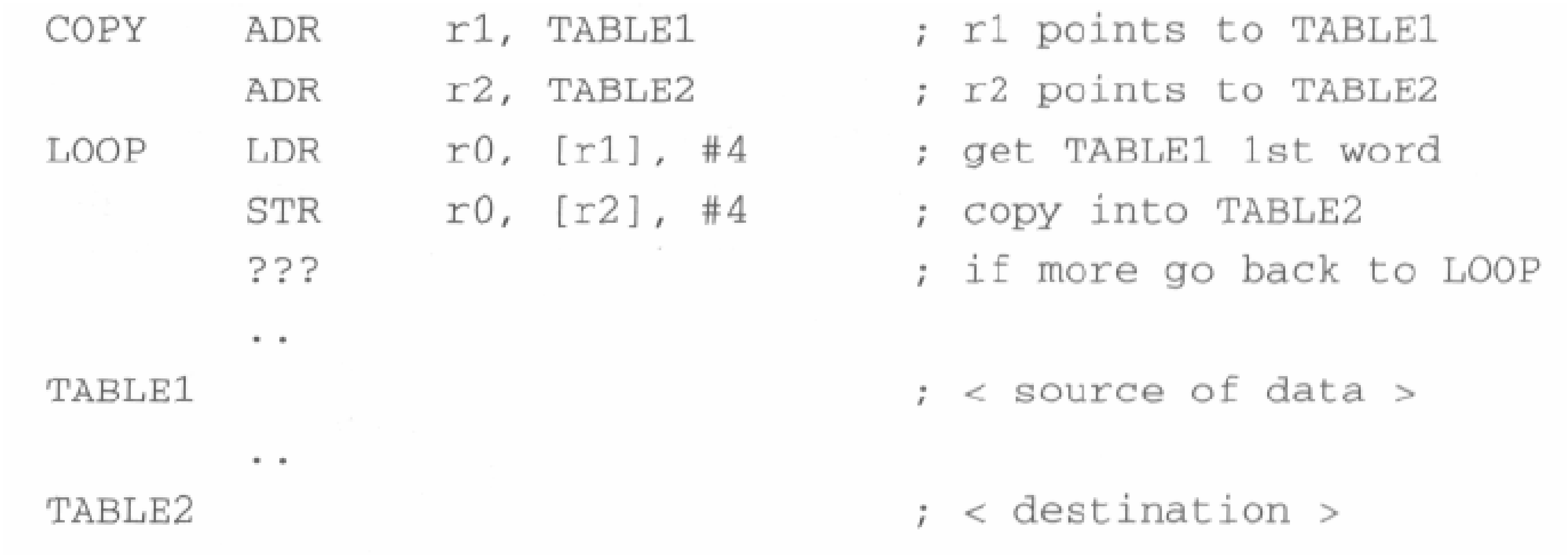
  ldr r0, [r1, #4]! ; r0 = mem_{32}[r1+4]

  ; r1 = r1 + 4
- Post-indexed Addressing Mode

  ldr r0, [r1], #4   ; r0 := mem_{32}[r1]

  ; r1 := r1 + 4

Indexed Addressing Mode



```
COPY    ADR    r1, TABLE1          ; r1 points to TABLE1
        ADR    r2, TABLE2          ; r2 points to TABLE2
LOOP    LDR    r0, [r1], #4        ; get TABLE1 1st word
        STR    r0, [r2], #4        ; copy into TABLE2
        ???                        ; if more go back to LOOP
        ..
TABLE1                             ; < source of data >
        ..
TABLE2                             ; < destination >
```

ADR is not an ARM instruction. It is an Assembler Pseudo-op.

Summary of ARM Addressing Modes

- Register-indirect addressing
  - LDR r0, [r1]
- Pre-indexed addressing
  - LDR r0, [r1 , # offset]
- Pre-indexed, auto-indexing
  - LDR r0, [r1 , # offset]!
- Post-indexed, auto-indexing
  - LDR r0, [r1], # offset
- PC relative addressing
  - ADR r0, address_label

More Data Transfer Instructions

- ldrb – Load an unsigned byte extended by 0
- ldrsb – Load a sign extended byte
- ldrh – Load an unsigned half word extended by 0
- ldrsh – Load a sign extended half word.
- ldrd – Load two consecutive words into a even register pair (like r12-r13 but not r11-r12)

Block Data Transfer Instructions

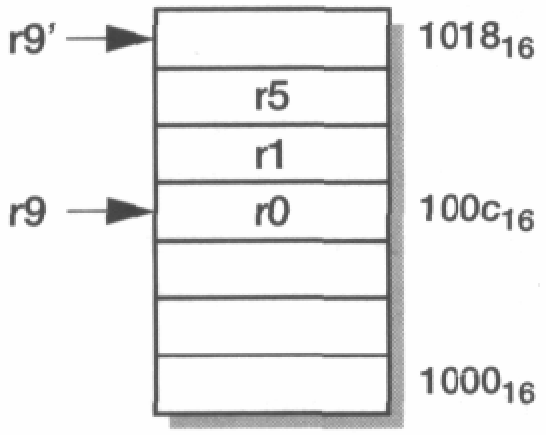
- LDMIA r1, {r0, r2, r5}
  - r0 = mem_{32}[r1]
  - r2 = mem_{32}[r1+4]
  - r5 = mem_{32}[r1+8]
- LDMIB r1, {r0, r2, r5}
  - r0 = mem_{32}[r1+4]
  - r2 = mem_{32}[r1+8]
  - r5 = mem_{32}[r1+12]
- LDMIA r1!, {r0, r2, r5}
  - r0 = mem_{32}[r1]
  - r2 = mem_{32}[r1+4]
  - r5 = mem_{32}[r1+8]
  - r1 = r1 + 12
- LDMIB r1!, {r0, r2, r5}
  - r0 = mem_{32}[r1+4]
  - r2 = mem_{32}[r1+8]
  - r5 = mem_{32}[r1+12]
  - r1 = r1+12

Block Data Transfer Instructions

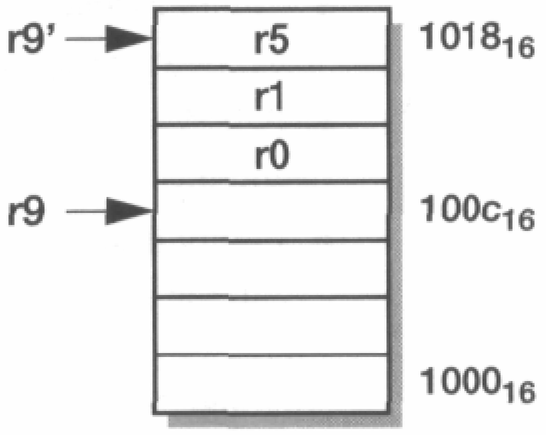
- LDMDA r1, {r0, r2, r5}
  - r5 = mem_{32}[r1]
  - r2 = mem_{32}[r1- 4]
  - r0 = mem_{32}[r1- 8]
- LDMDB r1, {r0, r2, r5}
  - r5 = mem_{32}[r1- 4]
  - r2 = mem_{32}[r1- 8]
  - r0 = mem_{32}[r1 - 12]
- LDMDA r1!, {r0, r2, r5}
  - r5 = mem_{32}[r1]
  - r2 = mem_{32}[r1- 4]
  - r0 = mem_{32}[r1- 8]
  - r1 = r1 - 12
- LDMDB r1!, {r0, r2, r5}
  - r5 = mem_{32}[r1- 4]
  - r2 = mem_{32}[r1- 8]
  - r0 = mem_{32}[r1 - 12]
  - r1 = r1 - 12

Block Data Transfer Instructions

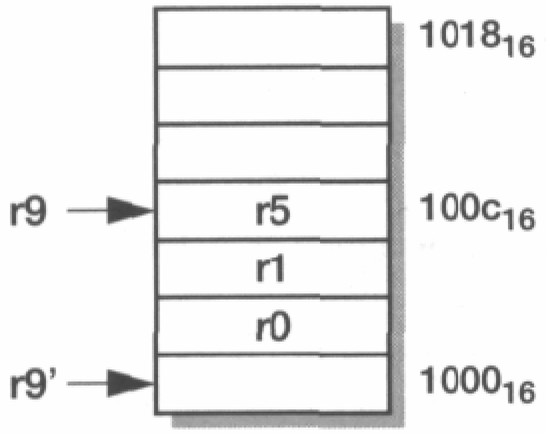
1. There are analogous load instructions also.
2. These instructions take more than 3 cycles causing pipeline imbalance.
3. This breaks the RISC architecture principle of single cycle per instruction execution model.



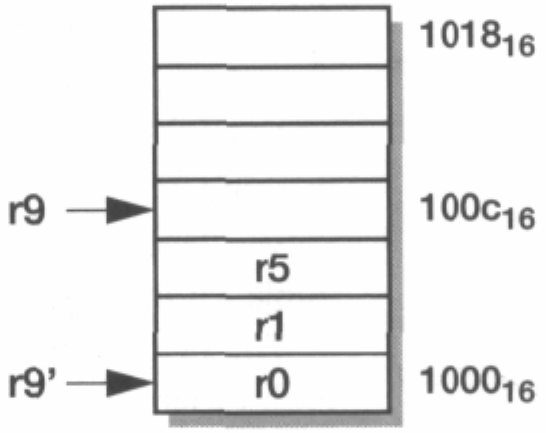
`STMIA r9!, {r0,r1,r5}`



`STMIB r9!, {r0,r1,r5}`

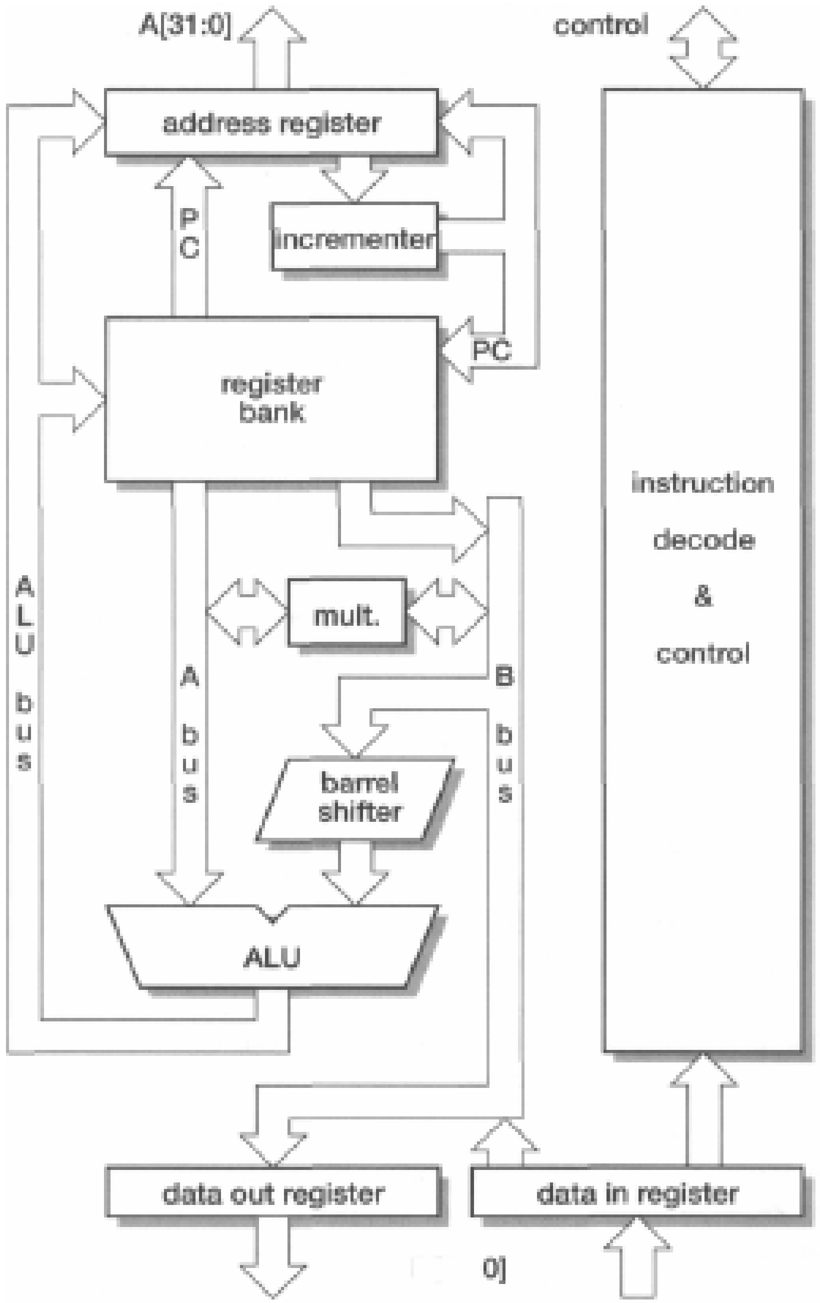


`STMDA r9!, {r0,r1,r5}`



`STMDB r9!, {r0,r1,r5}`

3-Stage Pipelined implementation of ARM Processor


A[31:0]
control
address register
P C
incrementer
PC
register bank
instruction decode & control
A L U bus
mult.
A bus
B bus
barrel shifter
ALU
data out register
data in register
0]