ARM Processors

Overview

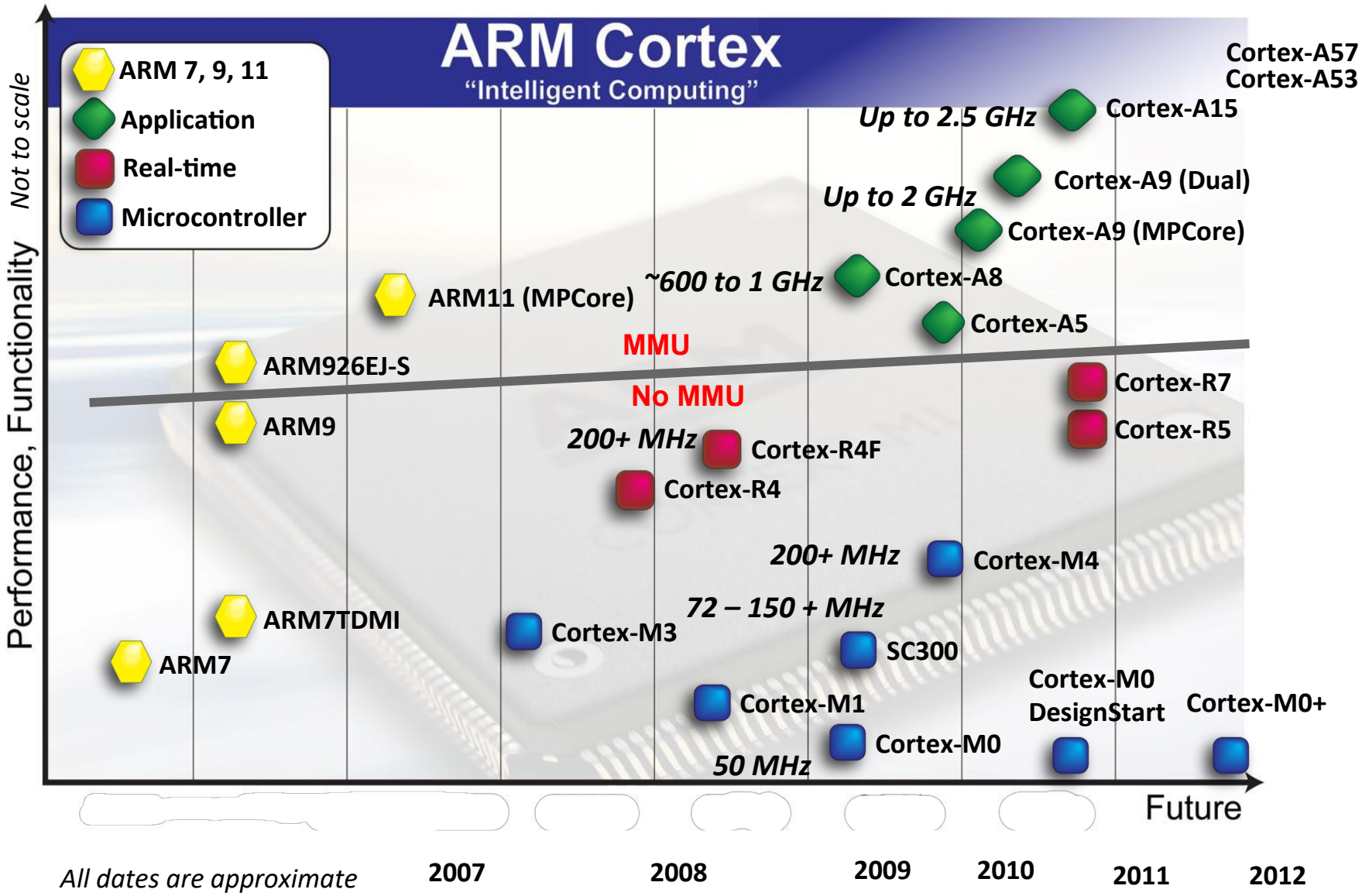
- ARM CPU Architectures
- ARM7TDMI Architectures
- ARM Cortex-M3 a small foot print Microcontroller
- ARM 11 MPCore
- TMS470 - Automotive Application

ARM CPU

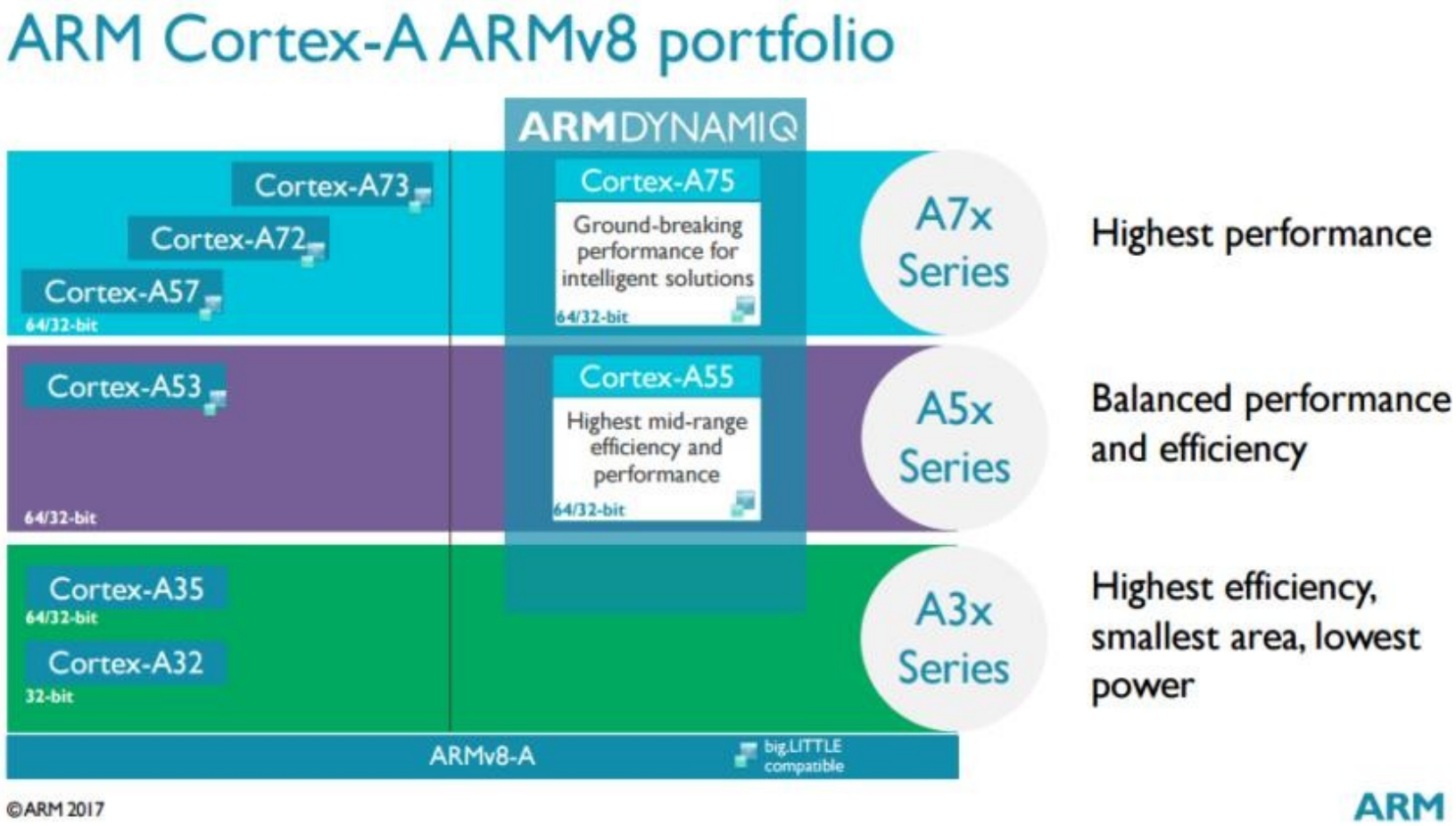
Versions, cores and architectures ?

- What is the difference between ARM7™ and ARMv7 ?
- ARM doesn't make chips….well maybe a few test chips.

Family	Architecture	Cores
ARM7TDMI	ARMv4T	ARM7TDMI(S)
ARM9 ARM9E	ARMv5TE(J)	ARM926EJ-S, ARM966E-S
ARM11	ARMv6 (T2)	ARM1136(F), 1156T2(F)-S, 1176JZ(F), ARM11 MPCore™
Cortex-A	ARMv7-A	Cortex-A5, A7, A8, A9, A15
Cortex-R	ARMv7-R	Cortex-R4(F)
Cortex-M	ARMv7-M	Cortex-M3, M4
	ARMv6-M	Cortex-M1, M0
NEW !	ARMv8-A	64 Bit

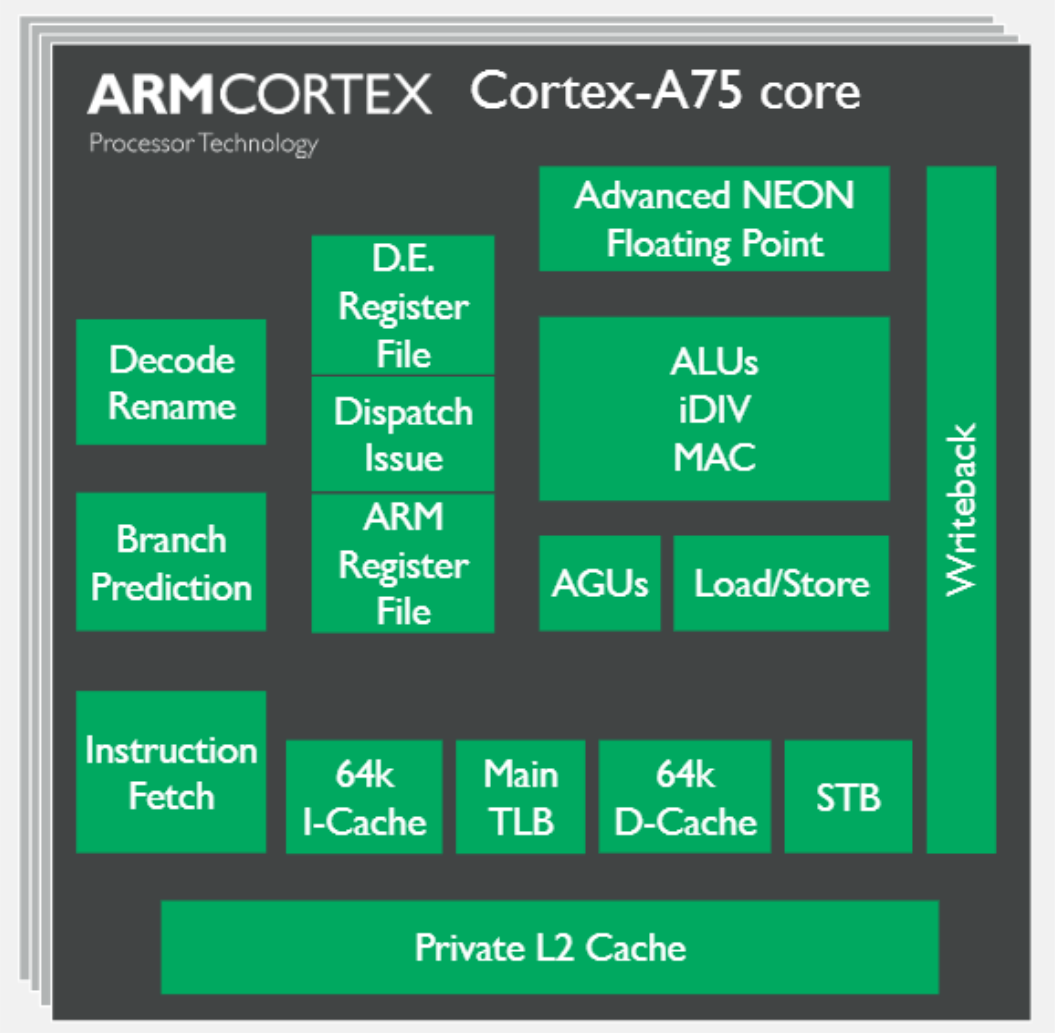

ARM Cortex
"Intelligent Computing"
ARM 7, 9, 11
Application
Real-time
Microcontroller
Performance, Functionality
Not to scale
Cortex-A57
Cortex-A53
Up to 2.5 GHz
Cortex-A15
Cortex-A9 (Dual)
Up to 2 GHz
Cortex-A9 (MPCore)
~600 to 1 GHz
Cortex-A8
ARM11 (MPCore)
Cortex-A5
MMU
ARM926EJ-S
No MMU
Cortex-R7
ARM9
Cortex-R5
200+ MHz
Cortex-R4F
Cortex-R4
200+ MHz
Cortex-M4
72 – 150 + MHz
ARM7TDMI
Cortex-M3
SC300
ARM7
Cortex-M0
DesignStart
Cortex-M0+
Cortex-M1
Cortex-M0
50 MHz
Future
All dates are approximate
2007
2008
2009
2010
2011
2012

ARMv8 64-bit Architecture



A72 has 3.5 times performance gain over A15. Pair with A53 to get big.Little Architecture. A73 introduced in 2016.
3GHz on a 10nm SoC & 2.8GHz on a 16nm SoC is achievable with A73.

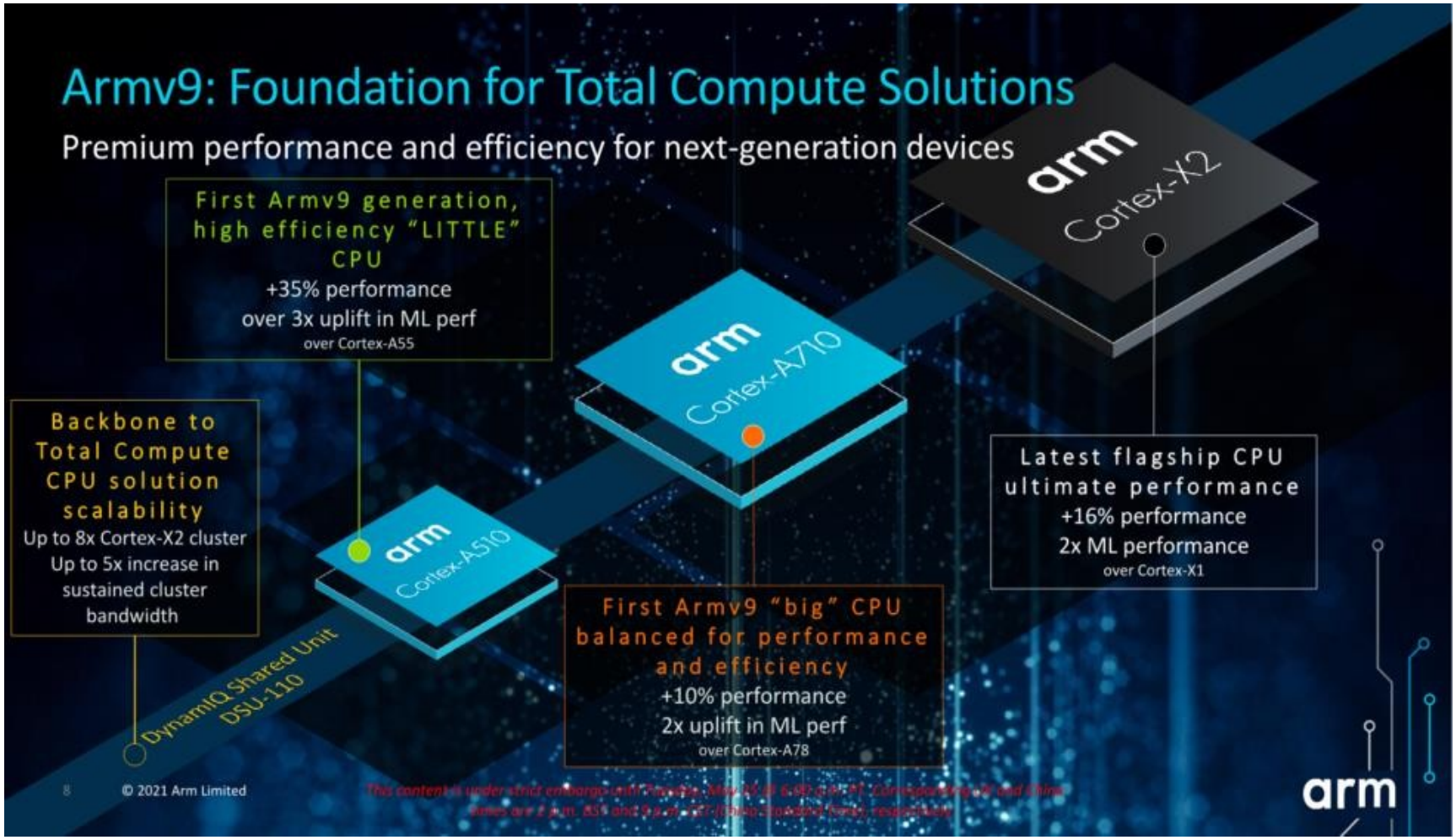
Cortex A75



Cortex-A75 execute up to 3 instructions per clock cycle.

A75 boasts 7 execution units, two load/stores, two NEON & FPU, a BPU and two integer cores.

Latest – Cortex A510, A710 and X2
Armv9 Generation

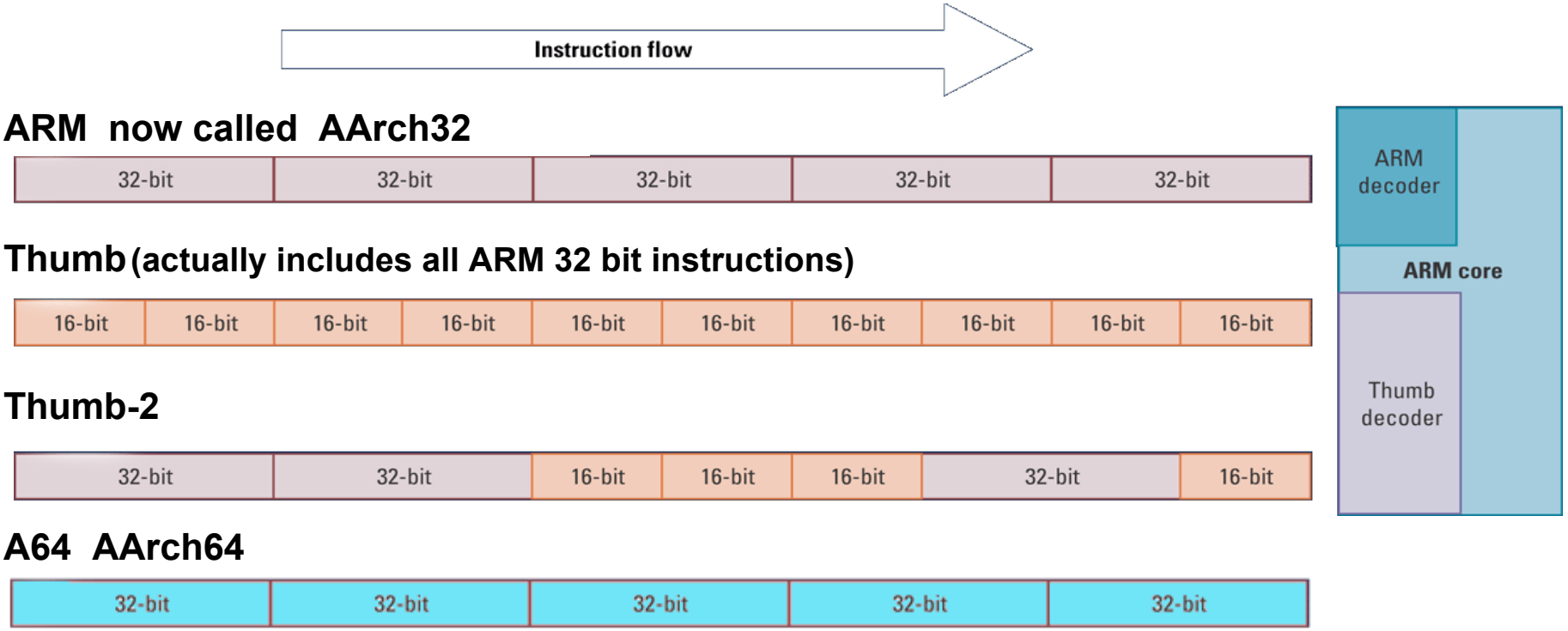


ARM Processor Licenses (the public ones)

- ARMv8-A ? NVIDIA, Applied Micro, Cavium, AMD, Broadcom, Calxeda, HiSilicon, Samsung and STMicroelectronics
- Cortex-A15 4 ST-Ericson, TI, Samsung, nVIDIA
- Cortex-A9 9 NEC, nVIDIA, STMicroelectronics, TI, Toshiba …
- Cortex-A8 9 Broadcom, Freescale, Matsushita, Samsung, STMicroelectronics, Texas Instruments, PMC-Sierra
- Cortex-A5 3 AMD ---
- Cortex-R4(F) 14 Broadcom, Texas Instruments, Toshiba, Inf
- Cortex-M4 5 Freescale, NXP, Atmel, ST
- Cortex-M3 29 Actel, Broadcom, Energy Micro, Luminary Micro, NXP, STMicroelectronics, TI, Toshiba, Zilog, …
- Cortex-M0 14 Austria-microsystems, Chungbuk Technopark, NXP, Triad Semiconductor, Melfas
- Cortex-M0+ Freescale, NXP
- ARM7 172, ARM9 271, ARM11 82

ARM Instruction Sets

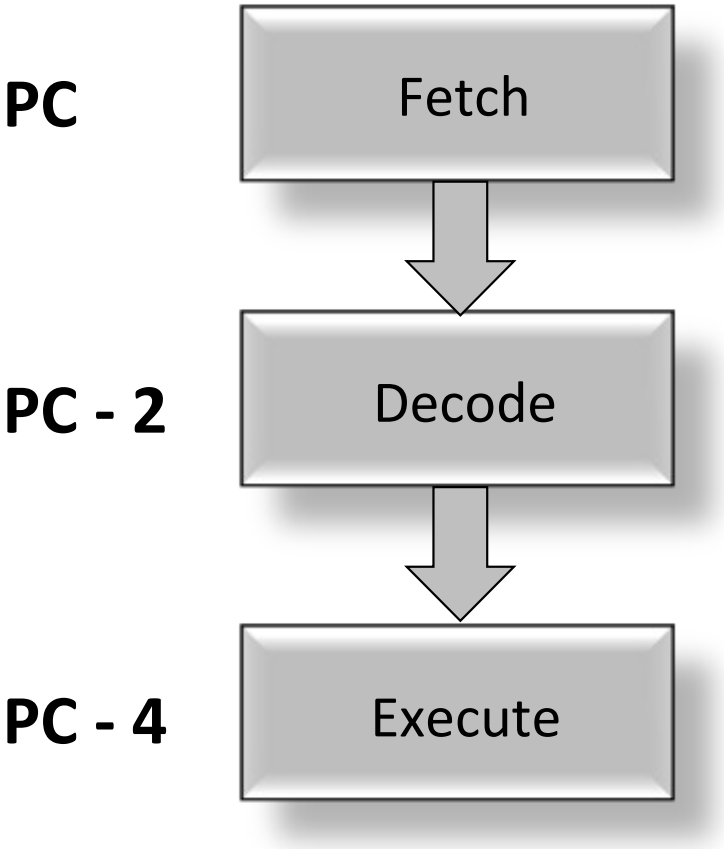
- ARM (32 bit) now referred as AArch32
- Thumb (16 bit)
- Thumb2: Cortex-Mx processors. Cortex-R, A have Thumb2 + ARM.
- A64 (64 bit) referred as AArch64



Tools: Keil MDK™ with µVision

- For Cortex-M and Cortex-R processors.
- Proprietary IDE µVision
- ARM compiler, assembler and linker.
- ULINK2, ULINK*pro*, CMSIS-DAP + more debug adapters.
- Many board support packages (BSP) and examples.
- MDK Professional: TCP/IP. CAN, USB & Flash middleware.
- Serial Wire Viewer and ETM, MTB & ETB Trace supported.
- Evaluation version is free from www.keil.com/arm.
- Is complete turn-key package: no add-ons needed to buy.
- Valuable technical support included for one year. Can be easily extended.
- Keil RTX RTOS included free with source code.

Pipelined Instruction Fetch, Decode & Execu



16-bit Instruction fetched from memory

Decompress thumb instruction, Decode ARM instruction
Select registers

Read register(s) from Register Bank,
Shift and ALU operation,
Write register(s) back to Register Bank

ARM7 Architecture

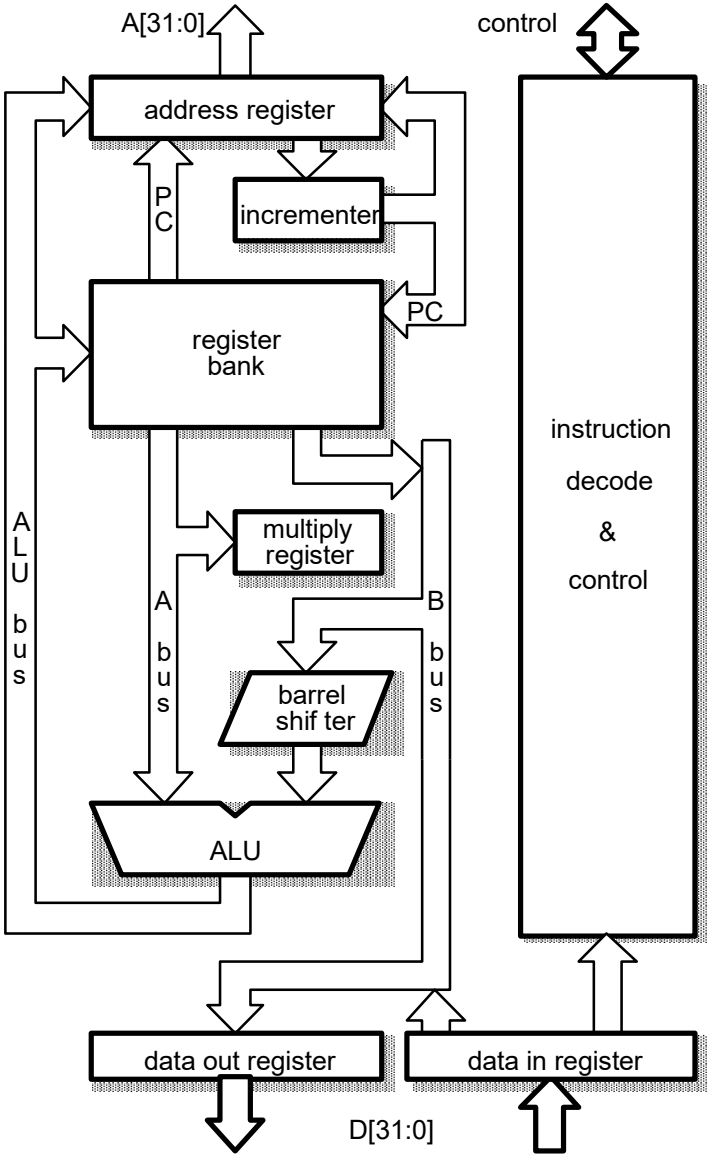
- Load/store architecture
- Most instructions are RISCy
  Some multi-register operations take multiple cycles
- All instructions can be executed conditionally

ARM7 is a small, low power, 32-bit microprocessor.
Three-stage pipeline, each stage takes one clock cycle

- Instruction fetch from memory
- Instruction decode
- Instruction execution.
  - Register read
  - A shift applied to one operand and the ALU operation
  - Register write

  This limits the CPU max clock speed to around 80 MHz on a 0.35 micron silicon process.

ARM CPU Core Organization



ARM7 Features

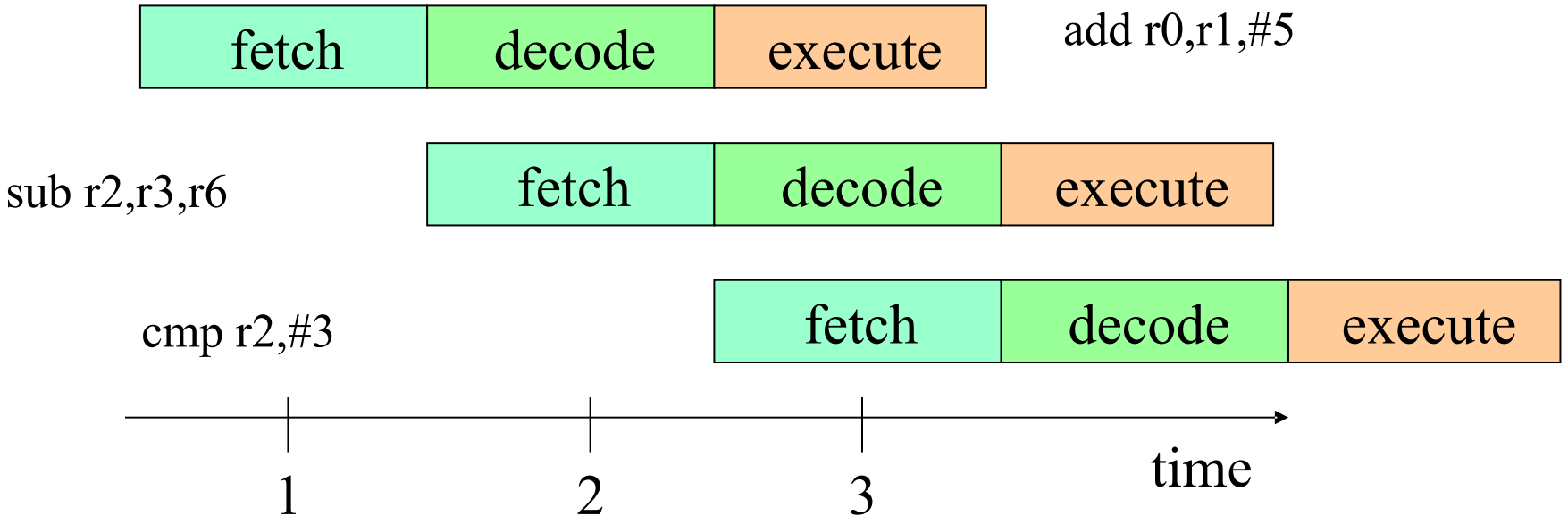
Combined Shift and ALU Execution Stage

- A single instruction can specify one of its two source operands fo shifting or rotation before it is passed to the ALU
- Allows very efficient bit manipulation and scaling code
- Eliminates virtually single shift instructions from ARM code. ARM7 CPU does not have explicit shift instructions.
- A move instruction can apply a shift to its operand

ARM7 uses von-Neumann memory architecture where instructions and data occupy single address space that can limit the performance

- Instruction fetching (and execution) must stop for instructions tha access memory
- The reduced cost of a single memory outweighs performance in many embedded applications.
- The pipeline stalls during load and store operations, ARM7 can continue useful work.

ARM7 Pipeline Execution



- **Latency**
  Time it takes for an instruction to get through the pipeline.
- **Throughput**
  Number of instructions executed per time period.

Pipelining increases throughput without reducing latency.

ARM CPU Features: Modified RISC

Multiple Load and Store Operation

Reduce the penalty of data accesses during a stall in the pipeline
Multiple load/store instructions can move any of the ARM registers to and from memory, and update the memory address register automatically after the transfer.

- This not only allows one instruction to transfer many words of data (in a single bus burst), it also reduces the amount of instructions needed to transfer data.
- Make the ARM code smaller than other 32-bit CPUs
- These instructions can specify an update of the base address register with a new address after (or even before) the transfer.

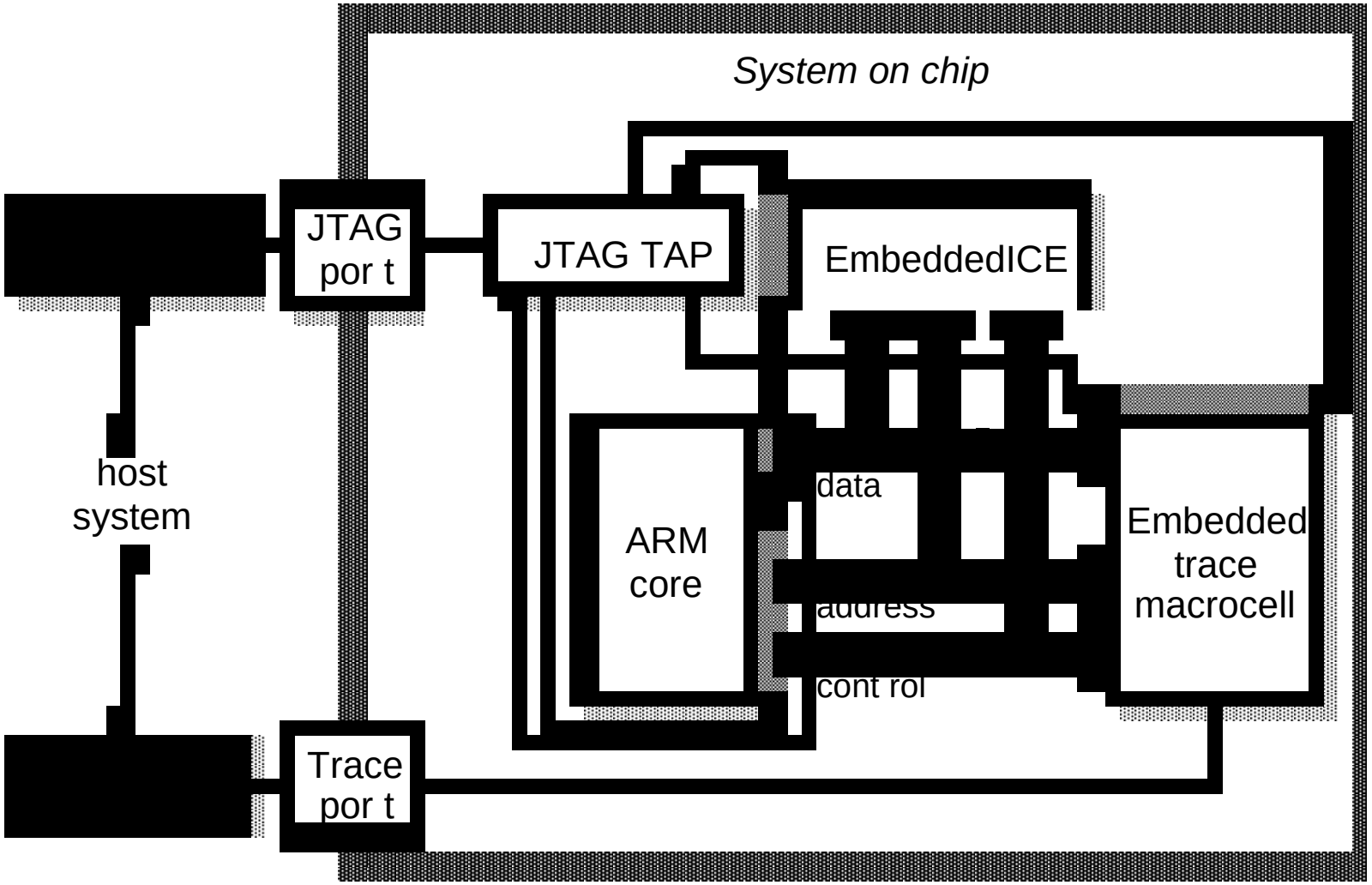
RISC CPU architectures would normally use a second instruction (add o subtract) to form the next address in a sequence.
ARM does it automatically with a single bit in the instruction, again a useful saving in code size.

ARM CPU (More) Features

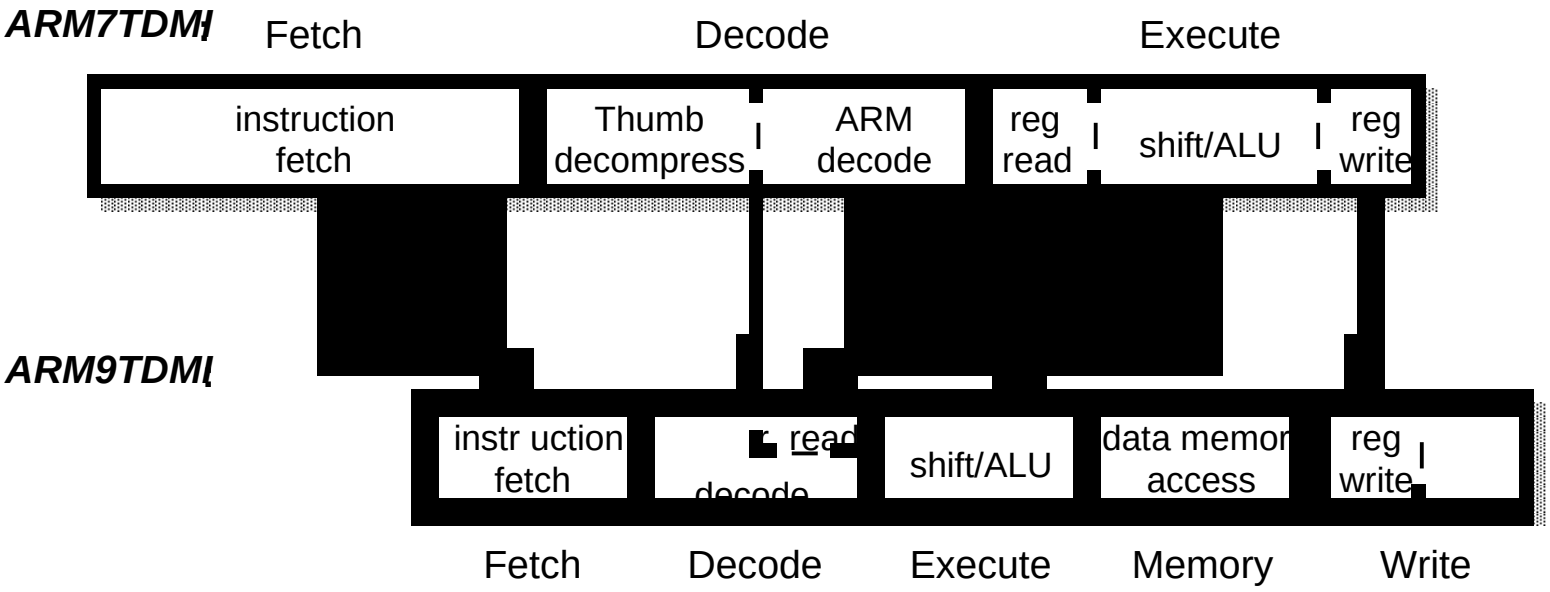
All instructions are conditionally executed:

- A very useful feature
- Loads, stores, procedure calls and returns, and all other operations can execute conditionally after some prior instruction to set the condition code flags
- Any ALU instruction may set the flags
- This eliminates short forward branches in ARM code
- It also improves code density and avoids flushing the pipeline for branches and increase execution performance
  - Most of the architectures have conditional branch instructions
  - These follow a test or compare instruction to control the flow o execution through the program
  - Some architectures also have a conditional move instruction, allowing data to be conditionally transferred between registers

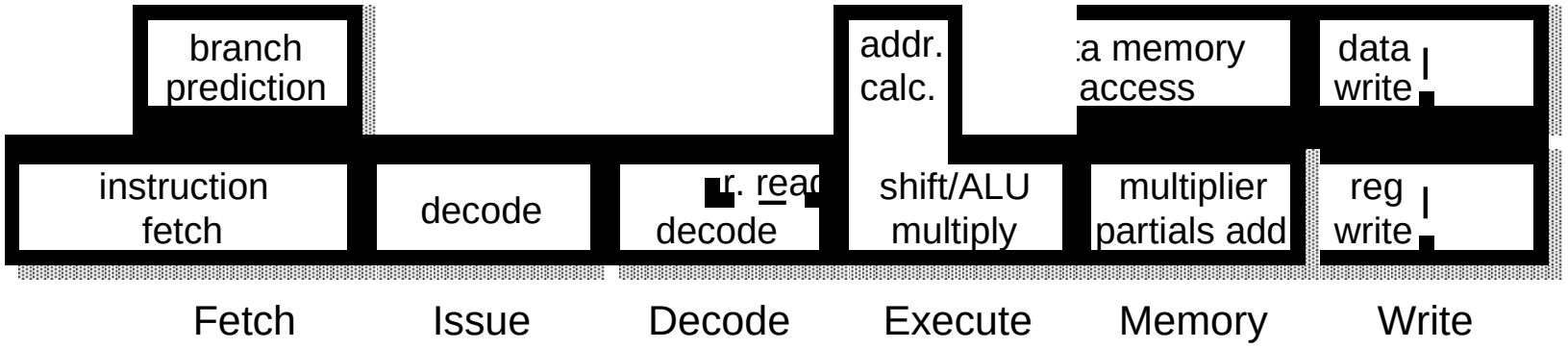
Real-timed Debug System Organization
(ARM7TDMI)



ARM7TDMI and ARM9 Pipeline



The ARM10TDMI pipeline

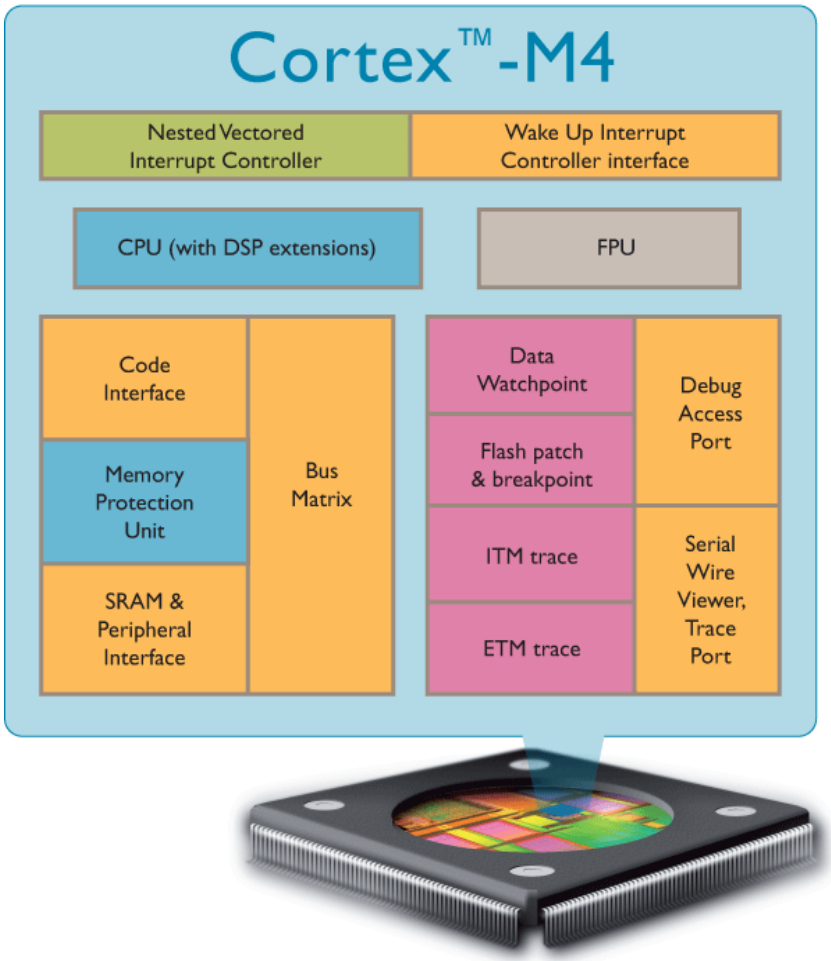


ARM Architectures

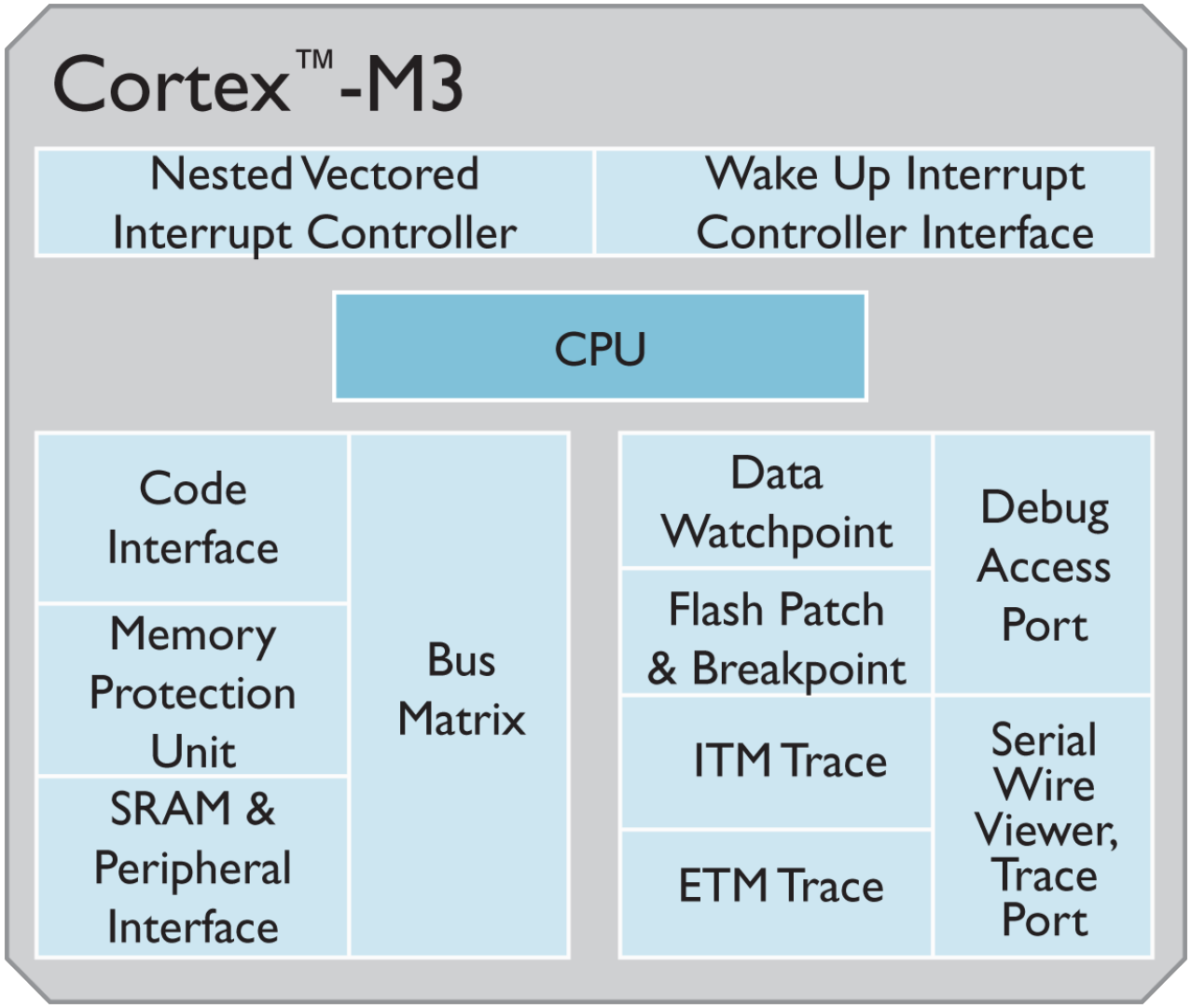
Core	Architecture
Classic ARM Processors	
ARM1	v1
ARM2, ARM2as, ARM3	v2, v2a
ARM6, ARM600, ARM610	v3
ARM7TDMI, ARM710T, ARM720T, ARM740T	v4T
ARM8, ARM810	v4
ARM9TDMI, ARM920T, ARM940T	v4T
ARM9ES	v5TE
ARM10TDMI, ARM1020E	v5TE
ARM11	v6
. . . . .	
ARM Cortex Processors	
ARM Cortex-M3	v7M
ARM Cortex-M4	v7ME
ARM Cortex-R4, R5, R7	v7R
ARM Cortex-A5, A8, A9, A15	v7A

ARM Cortex-M4

Latest Cortex-M series CPU that has a combination of efficient signal processing and low-power.

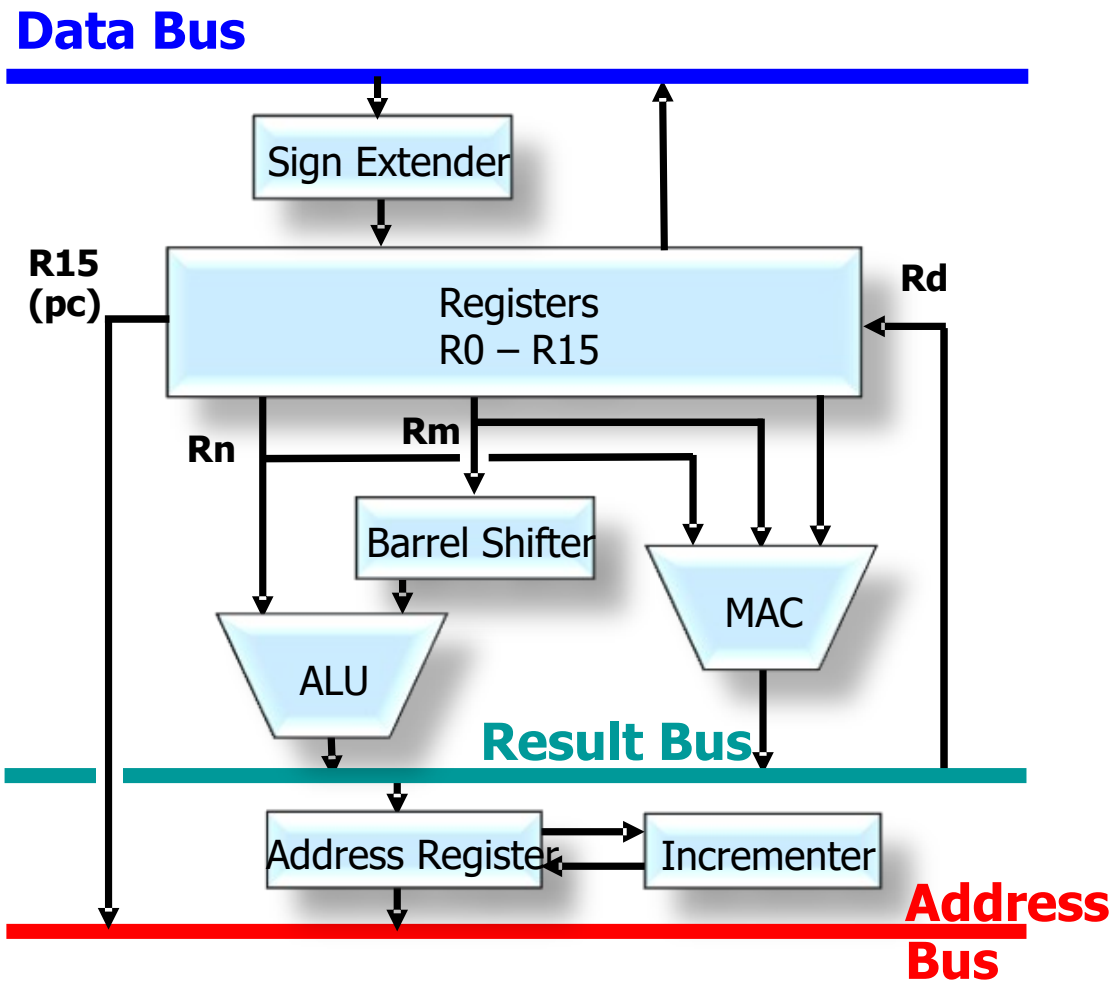


ARM Cortex-M3



ARM Cortex-M3

Introduced in 2004, the mainstream ARM processor developed specifically with microcontroller applications in mind.



ARM Cortex-M3

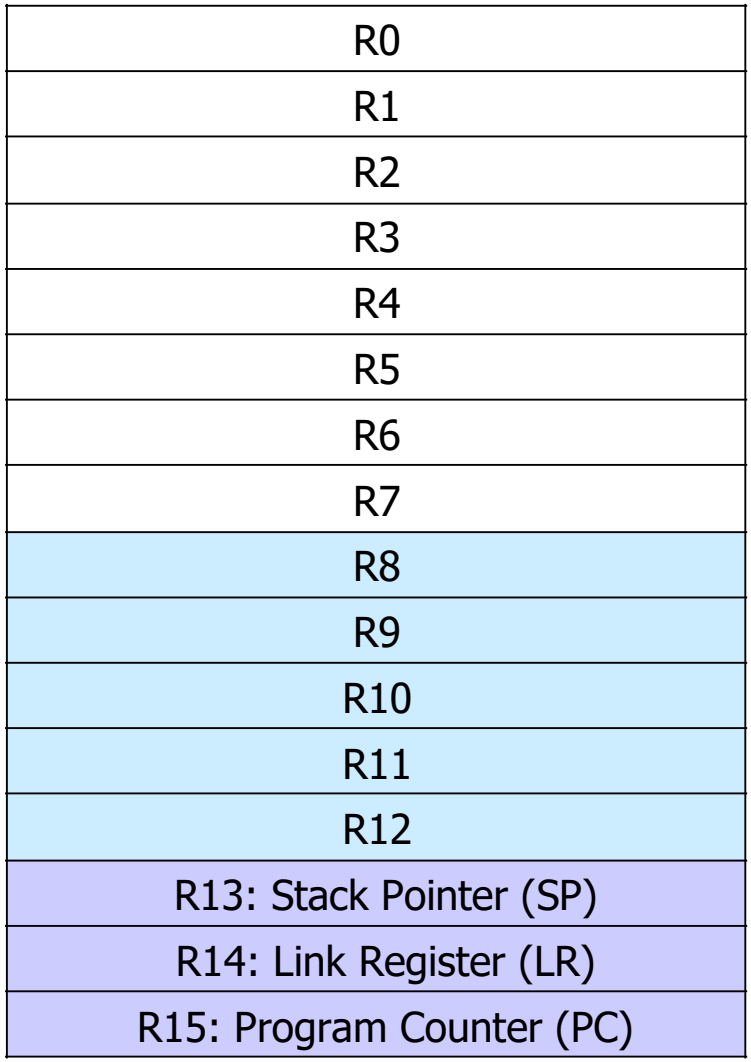
- Implement Thumb-2 instruction subset of ARM Instruction Set.
- Most Thumb-2 instructions are 16-bit wide that are expanded internally to a full 32-bit ARM instructions.
- ARM CPUs are capable of performing multiple low-level operations in parallel.
- A hardware sign extender convert 8-16 bit operands to 32-bit
- Load store architecture.
- Barrel shifter allows operand R_m to beshited first and then ALU can perform another operation (e.g. add, subtract, mul etc.)
- Barrel shifter can do $5X = X + 2^2X$; $-7X = X-2^3X$.

- MAC support Multiply Accumulate Instructions.
- R_0-R_{12} GPR, R_{13}-R_{15} special purpose registers i.e. SP, PC and LR (that holds the return address when a subroutine is called.

ARM Registers

ARM Mode:

15 general purpose registers



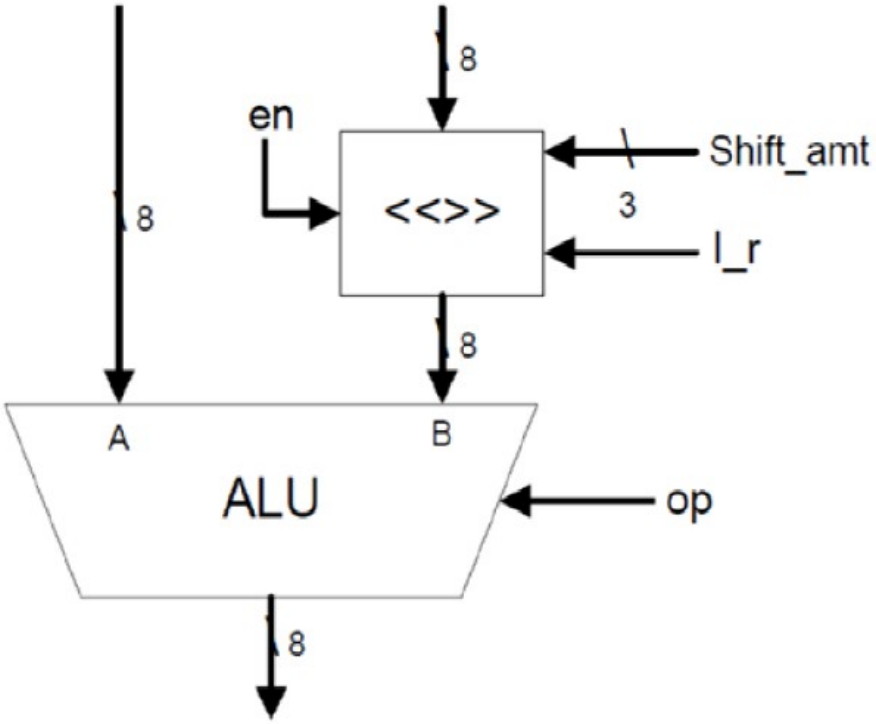
Thumb Mode:

8 general purpose registers

7 "high" registers

r8-R12 only accessible with MOV, ADD, or CMP

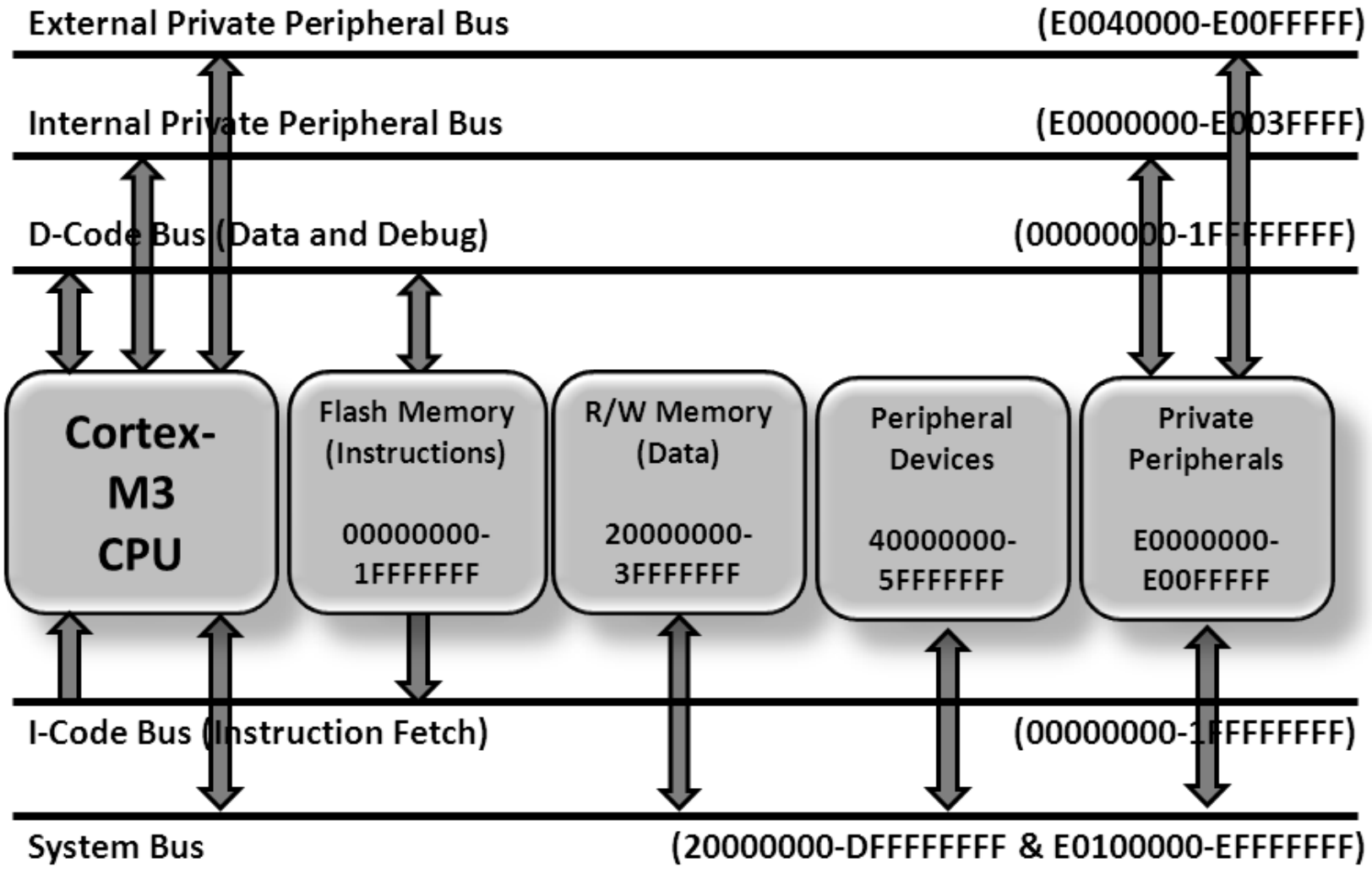
Barrel Shifting

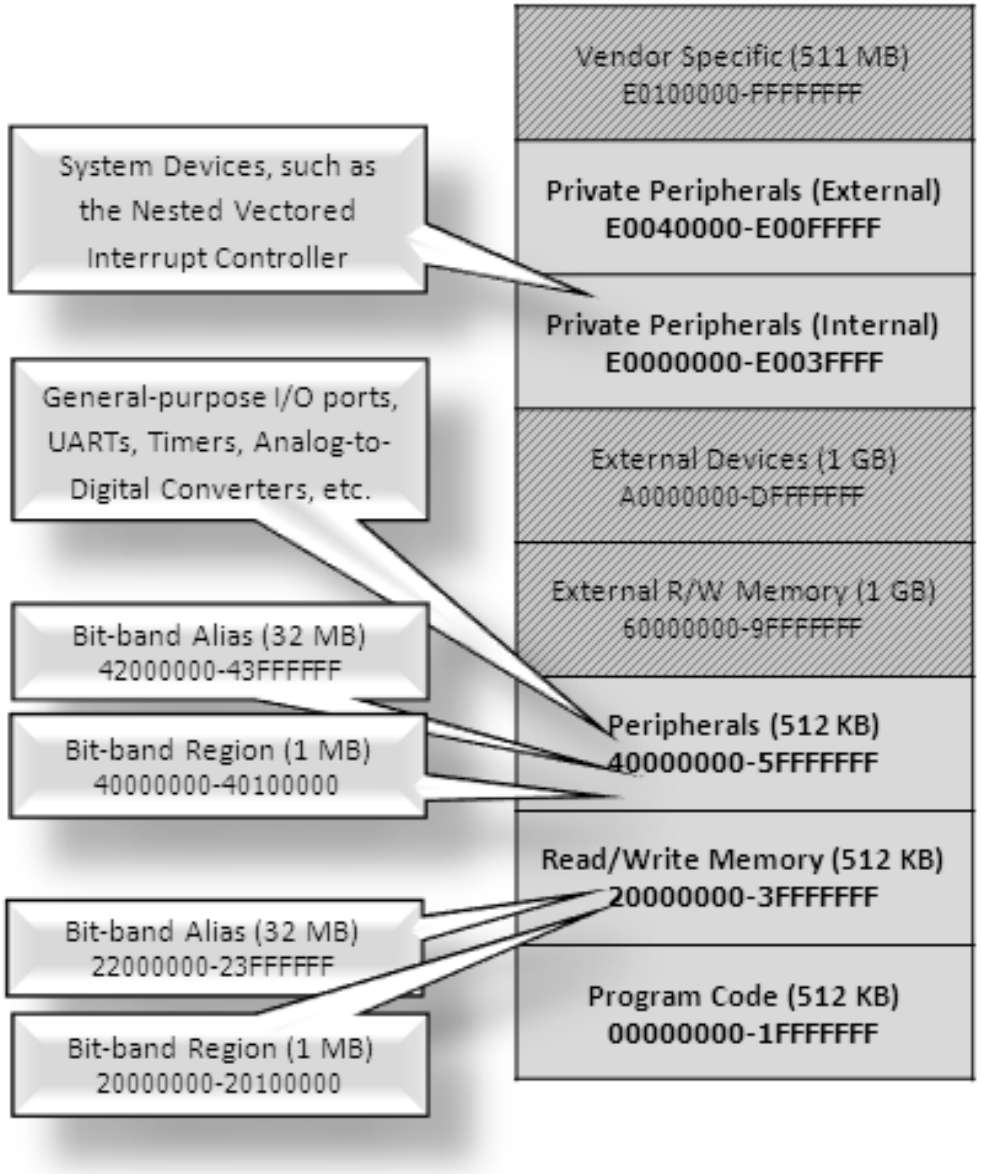


- Barrel shifter rotates/shift instruction operand prior to inputting the value into the ALU
- **Extensively used for signal processing application programs**

MUL R1 R2 #2 (LSL R1 R2 #2)
ADD R5, R1, R4 **translates to** ***ADD R5, R4, R2, LSL #2***

ARM Cortex-M3 Bus





ARM Cortex-M3 Memory

Bit Banding

- Memory mapped I/O, 4GB memory address space organized in bytes.
- 4GB is very large for small embedded applications.
- Bit-banding happens by taking advantage of this large memory space.
- Uses two different regions of the address space to refer the same physica data in the memory.
- In primary bit-band region each address corresponds to single data byte.
- In the bit-band alias each address corresponds to 1-bit of the same data.
- It allows the access of a bit of data (read or write) by a single instruction
- LDR can load a single bit and STR can write a single bit of data.
- Two bit band alias regions can be used to access individual status and control bit of I/O devices or to implement a set of 1-bit Boolean flags that can be used to implement a set of mutex objects.
- Bit-band hardware does not allow interruption of read-modify write.

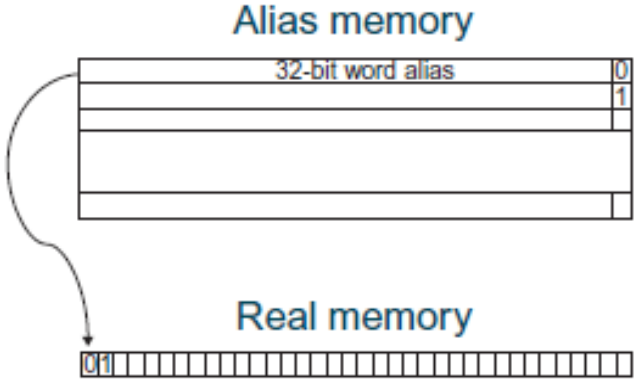
Bit_band alias address = Bit_band base + 128 x word_offset + 4 x bit #

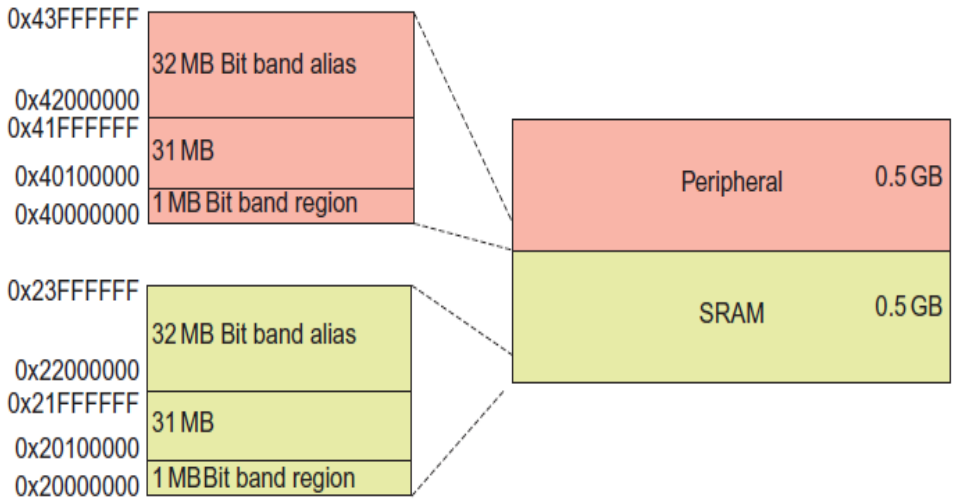
If bit-3 at address 20001000_{16} is to be modified the bit-band alias is

$22000000_{16} + 128_{10} \times 1000_{16} + 4 \times 3 = 2208000C_{16}$

Bit Banding

Address 0x20000000 = SRAM
0x40000000 = Peripheral = external RAM devices, memory vendor specific, etc.

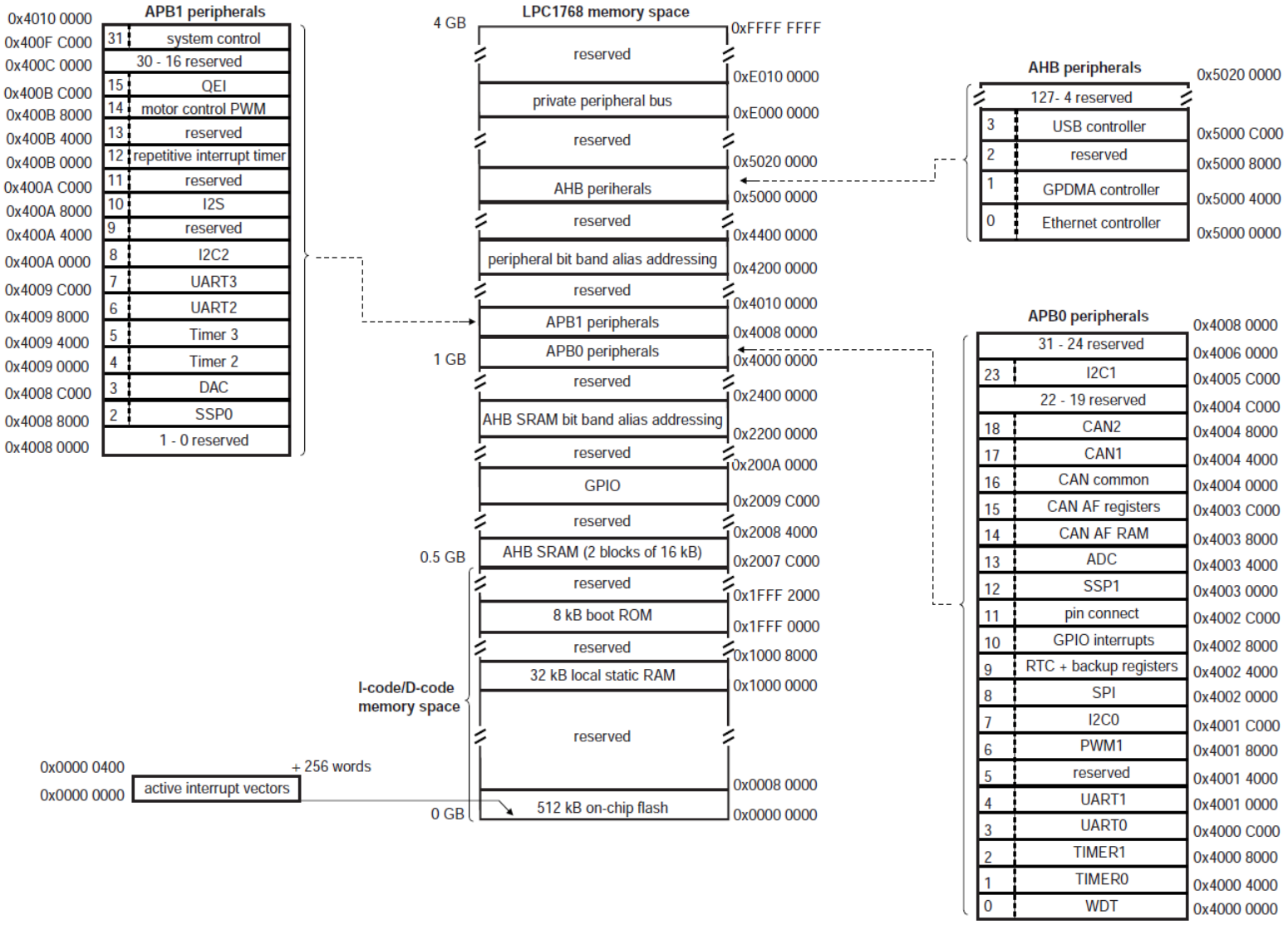




* One bit is addresses by its own 32-bit (word) in a separate part of memory (bit-band region)
* Bit-banding is for 2 predefined memory regions:
  - first 1MB of SRAM,
  - first 1MB of peripheral region
* To access each bit individually, we need to access a memory region referred to as the bit-band alias region.

BIT-BAND REGION: memory address region which supports bit-band operation
BIT-BAND ALIAS: Access to a bit-band alias will cause an access (bit-band operation) on the bit-band region.

Bit Banding

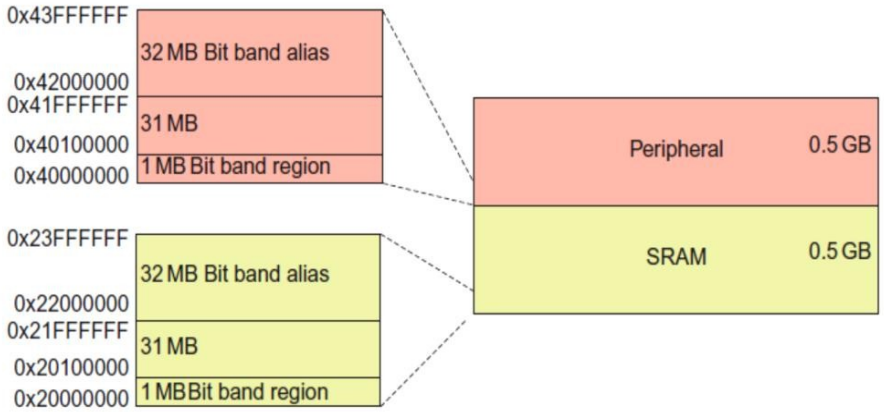

APB1 peripherals
0x4010 0000
0x400F C000
0x400C 0000
0x400B C000
0x400B 8000
0x400B 4000
0x400B 0000
0x400A C000
0x400A 8000
0x400A 4000
0x400A 0000
0x4009 C000
0x4009 8000
0x4009 4000
0x4009 0000
0x4008 C000
0x4008 8000
0x4008 0000
31 system control
30 - 16 reserved
15 QEI
14 motor control PWM
13 reserved
12 repetitive interrupt timer
11 reserved
10 I2S
9 reserved
8 I2C2
7 UART3
6 UART2
5 Timer 3
4 Timer 2
3 DAC
2 SSP0
1 - 0 reserved
LPC1768 memory space
4 GB
0xFFFF FFFF
reserved
0xE010 0000
private peripheral bus
0xE000 0000
reserved
0x5020 0000
AHB periherals
0x5000 0000
reserved
0x4400 0000
peripheral bit band alias addressing
0x4200 0000
reserved
0x4010 0000
APB1 peripherals
0x4008 0000
APB0 peripherals
1 GB
0x4000 0000
reserved
0x2400 0000
AHB SRAM bit band alias addressing
0x2200 0000
reserved
0x200A 0000
GPIO
0x2009 C000
reserved
0x2008 4000
AHB SRAM (2 blocks of 16 kB)
0.5 GB
0x2007 C000
reserved
0x1FFF 2000
8 kB boot ROM
0x1FFF 0000
reserved
0x1000 8000
32 kB local static RAM
0x1000 0000
I-code/D-code memory space
reserved
0x0008 0000
512 kB on-chip flash
0 GB
0x0000 0000
0x0000 0400
+ 256 words
0x0000 0000
active interrupt vectors
AHB peripherals
0x5020 0000
127- 4 reserved
3 USB controller
0x5000 C000
2 reserved
0x5000 8000
1 GPDMA controller
0x5000 4000
0 Ethernet controller
0x5000 0000
APB0 peripherals
0x4008 0000
31 - 24 reserved
0x4006 0000
23 I2C1
0x4005 C000
22 - 19 reserved
0x4004 C000
18 CAN2
0x4004 8000
17 CAN1
0x4004 4000
16 CAN common
0x4004 0000
15 CAN AF registers
0x4003 C000
14 CAN AF RAM
0x4003 8000
13 ADC
0x4003 4000
12 SSP1
0x4003 0000
11 pin connect
0x4002 C000
10 GPIO interrupts
0x4002 8000
9 RTC + backup registers
0x4002 4000
8 SPI
0x4002 0000
7 I2C0
0x4001 C000
6 PWM1
0x4001 8000
5 reserved
0x4001 4000
4 UART1
0x4001 0000
3 UART0
0x4000 C000
2 TIMER1
0x4000 8000
1 TIMER0
0x4000 4000
0 WDT
0x4000 0000

Bit Banding

Question: Find bit band word address for:
SRAM address 0x2008C000, bit 3.
Use equations (2) and (1):

Byte Offset = 0x2008C000 - 0x20000000
= 0x0008C000

Bit Band Word Address = 0x22000000 +
(0x0008C000 * 0x20) + (0x3 * 0x4)
= 0x22000000 + (0x1180000) + 0xC
= 0x2318000C



Bit Band Word Address =

Bit Band Alias Base Address + (Byte_Offset * 32) + (Bit Number * 4) (1)

Byte_Offset = Bit's Bit Band Base Address - Bit Band Base Address (2)

where: **Byte_Offset**

Bit's Bit Band Base Address - the base address for the targeted SRAM or peripheral regist
(The Effective Address of the Port) (= real address)

Bit Band Base Address: for SRAM = 0x20000000, for Peripherals = 0x40000000

Bit Band Alias Base Address:
for SRAM = 0x22000000, for Peripherals = 0x42000000

Bit Number: the bit position of the targeted register (i.e. pin of the port)

Bit Banding Example

Peripheral address 0x400ABC00, bit 8

Byte offset = 0x400ABC00 - 0x4000000 = 000ABC00

Bit band word address = 0x42000000 + (0x000ABC00 * 0x20) + (0x8 * 0x4)

= 0x42000000 + 0x01578000 + 20 = 0x43578020

Steps for bit banding:

1. Calculate the Word Address:

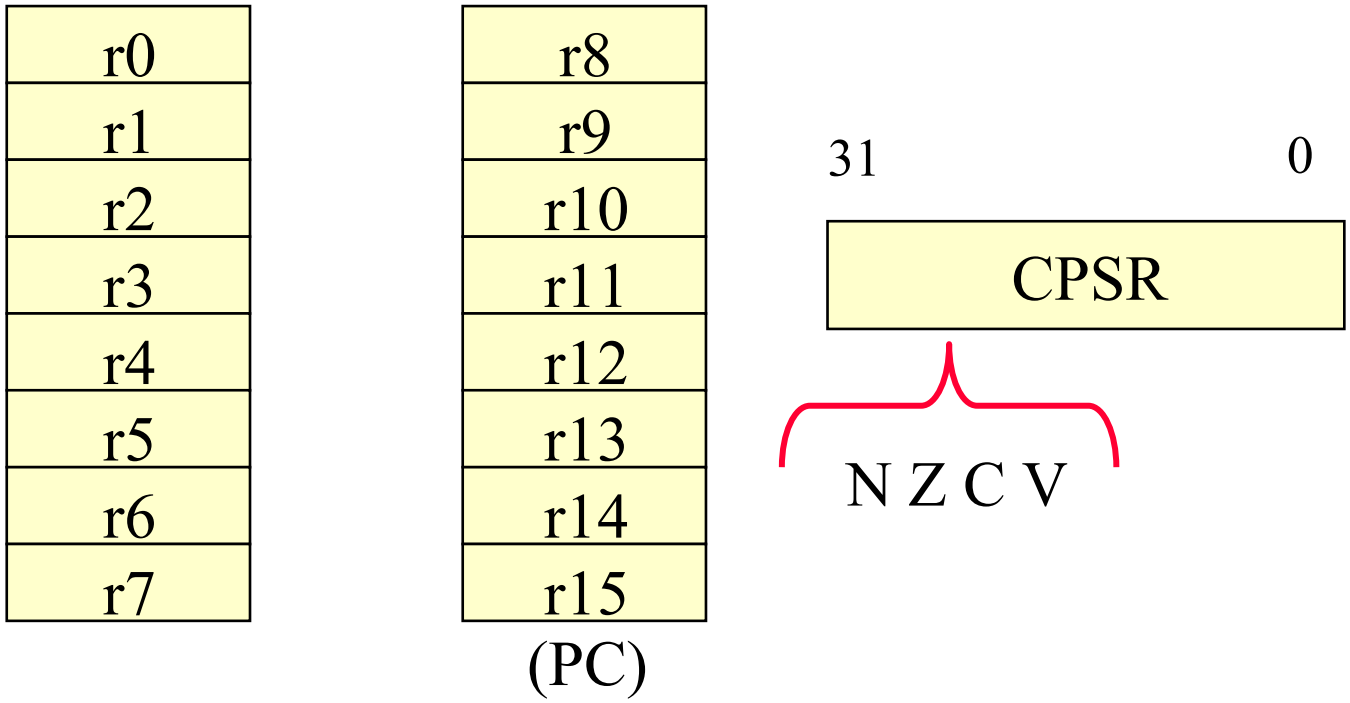
2. Define a Pointer to the Address:

#define BIT_ADDR= **(*(volatile unsigned long *)0x43578020)**

3. Assign a Value to the Port Bit:

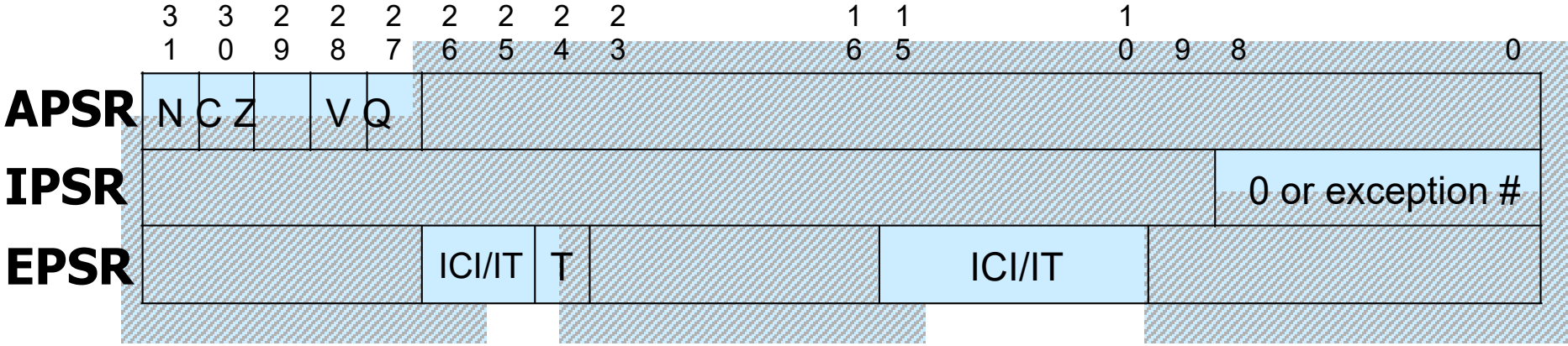
```
int main(void)  {
      BIT_ADDR = 1;
      ...

}
```

ARM7: Programming Model



- Word is 32 bits long.
- Word can be divided into four 8-bit bytes.
- ARM addresses can be 32 bits long.
- Address refers to byte.
  Address 4 starts at byte 4.

ARM Cortex Status Registers (xPSR)



Bits	Name	Description
31	**N**	**Negative (bit 31 of result is 1)**
30	**C**	**Unsigned Carry**
29	**Z**	**Zero or Equal**
28	**V**	**Signed Overflow**

Most important for application programming

PSR: Program Status Register

Divided into three bit fields

- Application Program Status Register (APSR)
- Interrupt Program Status Register (IPSR)
- Execution Program Status Register (EPSR)

Q-bit is the sticky saturarion bit and supports two rarely used instructions (SSAT and USAT)
SSAT{cond} Rd, #sat, Rm{, shift}

- EPSR holds the exception number is exception processing.
- ICI/IT bits holds the state information of for IT block instructions or instructions that are suspended during interrupt processing.
- T bit is always 1 to indicate Thumb instructions.

SSAT: Saturate Instruction

- Consider two numbers 0xFFFF FFFE and 0×0000 0002. A 32-bit mathematical addition would result in 0×1 0000 0001 which contain 9 hex digits or 33 binary bits. If the same arithmetic is done in a 32 bit processor ideally the carry flag will be set and the result in the register will be 0×0000 0001.
- If the operation was done by any comparison instruction this would not cause any harm but during any addition operation this may lead to unpredictable results if th code is not designed to handle such operations. Saturate arithmetic says that when the result crosses the extreme limit the value should be maintained at the respectiv maximum/minimum (in our case result will be maintained at 0xFFFF FFFF which is the largest 32-bit number).
- Saturate instructions are very useful in implementing certain DSP algorithms like audio processing where we have a cutoff high in the amplitude. For instance the highest amplitude is expressed by a 32-bit value and if my audio filter gives an output more than this I need not programatically monitor the result. Rather the value automatically saturates to the max limit.
- Also a new flag field called ‘Q’ has been added to the ARM processor to show us if there had been any such saturation taken place or the natural result itself was th maximum

SSAT or USAT Instructions

op{*cond*} *Rd*, #*n*, *Rm* {, *shift* #s}

op = SSAT Saturates a signed value to a signed range.
USAT Saturates a signed value to an unsigned range.

Cond condition code

Rd Specifies the destination register.

n Specifies the bit position to saturate to:
n ranges from 1 to 32 for SSAT
n ranges from 0 to 31 for USAT.

Rm Register containing the value to saturate.

shift #s optional shift applied to *Rm* before saturating.

These instructions saturate to a signed or unsigned *n*-bit value.

SSAT instruction applies the specified shift, then saturates to the signed range $-2^{n-1} \le x \le 2^{n-1}-1$.

The USAT instruction applies the specified shift, then saturates to the unsigned range $0 \le x \le 2^{n}-1$.

SSAT or USAT Instructions

If the returned result is different from the value to be saturated, it is called *saturation*.

If saturation occurs, the instruction sets the Q flag to 1 in the APSR. Otherwise, it leaves the Q flag unchanged.

Examples

```
SSAT    R7, #16, R7, LSL #4
                        ; Logical shift left value in R7 by 4, then
                        ; saturate it as a signed 16-bit value and
                        ; write it back to R7

USATNE  R0, #7, R5 ; Conditionally saturate value in R5 as an
                        ; unsigned 7 bit value and write it to R0.
```

ARM Operating Modes and Register Usage

CPSR[4:0]	Mode	Use	Registers
10000	User	Normal user code	user
10001	FIQ	Processing fast interrupts	_fiq
10010	IRQ	Processing standard interrupts	_irq
10011	SVC	Processing software interrupts (SWIs)	_svc
10111	Abort	Processing memory faults	_abt
11011	Undef	Handling undefined instruction traps	_und
11111	System	Running privileged operating system tasks	user

Exception vector addresses

Exception	Mode	Vector address
Reset	SVC	0x00000000
Undefined instruction	UND	0x00000004
Software interrupt (SWI)	SVC	0x00000008
Prefetch abort (instruction fetch memory fault)	Abort	0x0000000C
Data abort (data access memory fault)	Abort	0x00000010
IRQ (normal interrupt)	IRQ	0x00000018
FIQ (fast interrupt)	FIQ	0x0000001C

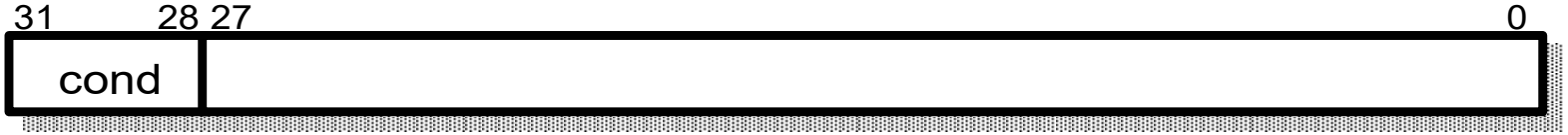
Load/Store Instructions

Load/Store Memory	*Operation*	*Notes*
LDR R_d,<mem>	$R_d \leftarrow mem_{32}$[address]	
LDRB R_d,<mem>	$R_d \leftarrow mem_{8}$[address]	Zero fills
LDRH R_d,<mem>	$R_d \leftarrow mem_{16}$[address]	Zero fills
LDRSB R_d,<mem>	$R_d \leftarrow mem_{8}$[address]	Sign extends
LDRSH R_d,<mem>	$R_d \leftarrow mem_{16}$[address]	Sign extends
LDRD R_t,R_{t2},<mem>	$R_{t2}.R_t \leftarrow mem_{64}$[address]	Addr. Offset must be imm.

Load/Store Memory	*Operation*	*Notes*
STR R_d,<mem>	$R_d \rightarrow mem_{32}$[address]	
STRB R_d,<mem>	$R_d \rightarrow mem_{8}$[address]	
STRH R_d,<mem>	$R_d \rightarrow mem_{16}$[address]	
STRD R_t,R_{t2},<mem>	$R_{t2}.R_t \rightarrow mem_{64}$[address]	Addr. Offset must be imm.

These instructions will not affect flags in CPSR!

The ARM Condition Code Field



ARM condition codes

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

Branch Instructions

Branch Instructions	***Operation***	***{S}***	***Notes***
B{c} label	PC ← PC + imm	n/a	"c" is an *optional* condition code
BL label	PC ← PC + imm; LR ← rtn adr	n/a	Subroutine call
BX reg	PC ← reg	n/a	**"BX LR" often used as function return**
CBZ R_n,label	If R_n=0, PC ← PC + imm	n/a	Cannot append condition code to CBZ
CBNZ R_n,label	If R_n≠0, PC ← PC + imm	n/a	Cannot append condition code to CBNZ
IT$c_1c_2c_3$ cond	Each c_i is one of T, E, or *empty*	n/a	Controls 1-4 instructions in "IT block"

Branch Conditions

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 occurred
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 Data Processing Instructions

Opcode [24:21]	Mnemonic	Meaning	Effect
0000	AND	Logical bit-wise AND	Rd := Rn AND Op2
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	Scc on Rn AND Op2
1001	TEQ	Test equivalence	Scc on Rn EOR Op2
1010	CMP	Compare	Scc on Rn - Op2
1011	CMN	Compare negated	Scc on Rn + Op2
1100	ORR	Logical bit-wise OR	Rd := Rn OR Op2
1101	MOV	Move	Rd := Op2
1110	BIC	Bit clear	Rd := Rn AND NOT Op2
1111	MVN	Move negated	Rd := NOT Op2

Bitwise Instructions

Bitwise Instructions	*Operation*	*{S}*	*<op>*	*Notes*
AND R_d, R_n,<op>	$R_d \leftarrow R_n$ & <op>	NZC	imm. const. -or- reg{,<shift>}	
ORR R_d, R_n,<op>	$R_d \leftarrow R_n$ | <op>	NZC		
EOR R_d, R_n,<op>	$R_d \leftarrow R_n$ ^ <op>	NZC		
BIC R_d, R_n,<op>	$R_d \leftarrow R_n$ & ~<op>	NZC		
ORN R_d, R_n,<op>	$R_d \leftarrow R_n$ | ~<op>	NZC		
MVN R_d, R_n	$R_d \leftarrow \sim R_n$	NZC		

Shift Instructions

<shift>	*Meaning*	*Notes*
LSL #n	Logical shift left by n bits	Zero fills; $0 \leq n \leq 31$
LSR #n	Logical shift right by n bits	Zero fills; $1 \leq n \leq 32$
ASR #n	Arithmetic shift right by n bits	Sign extends; $1 \leq n \leq 32$
ROR #n	Rotate right by n bits	$1 \leq n \leq 32$
RRX	Rotate right w/C by 1 bit	

Conditional Execution

ADD instruction with the EQ condition appended.
This instruction will only be executed when the zero flag in the *cpsr* is set;

```
ADDEQ    r0, r1, r2    ; r0 = r1 + r2 if zero flag is set
```

```
while (a!=b) {                ; Greatest Common Divisor Algorithm
      if (a > b) a-= b; else b-= a;
}
```

Register *r1* represent *a* and register *r2* represent *b*.

```
gcd       CMP r1, r2
          BEQ complete
          BLT lessthan
          SUB r1, r1, r2
          B gcd
lessthan  SUB r2, r2, r1
          B gcd
complete
...
```

This dramatically reduces the number of instructions

```
gcd  CMP  r1, r2
     SUBGT  r1, r1, r2
     SUBLT  r2, r2, r1
     BNE gcd
complete
...
```

```
gcd  CMP  r1, r2

     SUBGT  r1, r1, r2
     SUBLT  r2, r2, r1
     BNE gcd
complete
...
```

Thumb 2 Instructions

IT (If-Then)

IT (If-Then) instruction makes up to four following instructions (the *IT block*) conditional. The conditions can be all the same, or some of them can be the logical inverse of the others.

IT {*x* {*y* {*z*} } } {*cond*}

where: ***x****:* specifies the condition switch for the second instruction i the IT block.

y*:* specifies condition switch for the third instruction in the IT block

z*:* specifies condition switch for the fourth instruction in the IT bloc

cond: specifies the condition for first instruction in the IT block

Condition switch for 2nd, 3rd & 4th instruction in the IT block either:

- T Then. Applies the condition *cond* to the instruction.
- E Else. Applies the inverse condition of *cond* to the instruction.

The instructions (including branches) in the IT block, except the BKPT instruction, must specify the condition in the {*cond*} part of their syntax.

IT (If-Then) instruction

- You do not need to write IT instructions in your code.
- The assembler generates them automatically according to the conditions specified on the following instructions.
- Writing the IT instructions ensures that you consider the placing of conditional instructions, and the choice of conditions.
- When assembling to ARM code, the assembler performs the same checks but does not generate any IT instructions.
- With the exception of CMP, CMN, and TST, the 16-bit instructions that normally affect the condition code flags, do not affect them in IT block.
- A BKPT instruction in an IT block is always executed, so it does not need a condition in the {*cond*} part of its syntax. The IT block continues from the next instruction.
- You can use an IT block for unconditional instructions by using the AL.
- Conditional branches inside an IT block have a longer branch range than those outside the IT block.

IT (If-Then) instruction

The following instructions are not permitted in an IT block:

- IT
- CBZ and CBNZ
- TBB and TBH
- CPS, CPSID and CPSIE
- SETEND.

Other restrictions when using an IT block are:

- A branch or any instruction that modifies the PC is only permitted in an IT block if it is the last instruction in the block.
- You cannot branch to any instruction in an IT block, unless when returning from an exception handler.

Architectures

- This 16-bit Thumb instruction is available in ARMv6T2 and above.
- In ARM code, IT is a pseudo-instruction that does not generate any code.

IT Examples

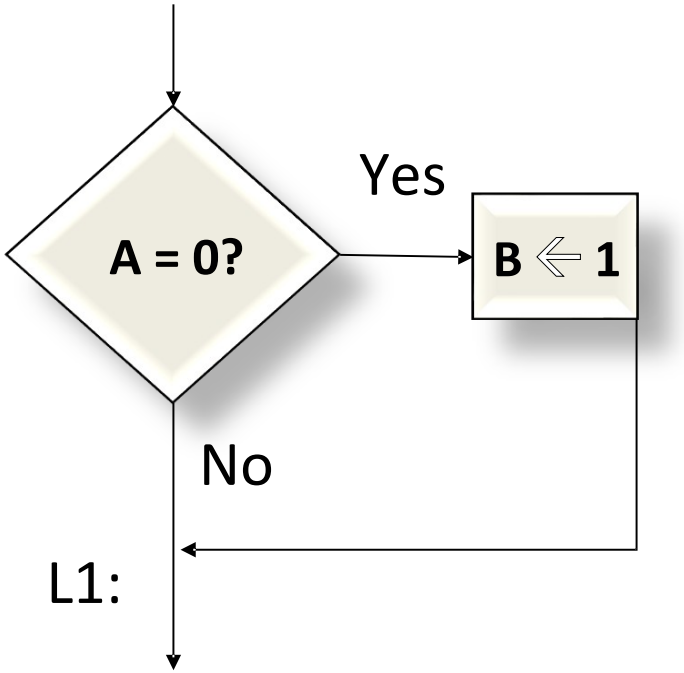
```
ITTE   NE         ; IT can be omitted
ANDNE  r0,r0,r1   ; 16-bit AND, not ANDS
ADDSNE r2,r2,#1   ; 32-bit ADDS (16-bit ADDS dos'nt set flags in IT)
MOVEQ  r2,r3      ; 16-bit MOV

ITT    AL         ; emit 2 non-flag setting 16-bit instructions
ADDAL  r0,r0,r1   ; 16-bit ADD, not ADDS
SUBAL  r2,r2,#1   ; 16-bit SUB, not SUB
ADD    r0,r0,r1   ; expands into 32-bit ADD, and is not in IT block

ITT    EQ
MOVEQ  r0,r1
BEQ    dloop      ; branch at end of IT block is permitted

ITT    EQ
MOVEQ  r0,r1
BKPT   #1         ; BKPT always executes
ADDEQ  r0,r0,#1
```

Incorrect example

```
IT  NE
ADD r0,r0,r1; syntax error: no condition code used in IT
```

if-then statement

if (a == 0) b = 1 ;

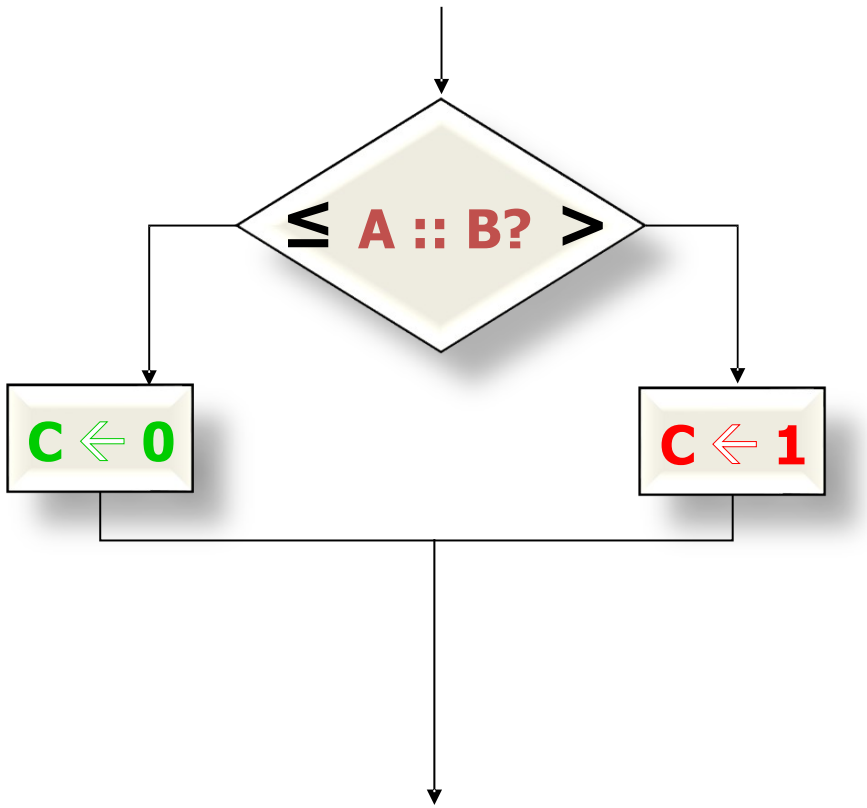



```
        LDR     R0,A
        CMP     R0,#0
        BNE     L1
        LDR     R0,=1
        STR     R0,B
L1:     …
```

- or –

```
        LDR     R0,A
        CMP     R0,#0
        ITT     EQ
        LDREQ   R0,=1
        STREQ   R0,B
```

if-then-else statement




```
     LDR     R0,A
     LDR     R1,B
     CMP     R0,R1
     BLE     L1
     LDR     R0,=1
     B       L2
L1:  LDR     R0,=0
L2:  STR     R0,C
     ...
```

- or –

```
     LDR     R0,A
     LDR     R1,B
     CMP     R0,R1
     ITE     GT
     LDRGT   R0,=1
     LDRLE   R0,=0
     STR     R0,C
```

An ITTE Block

```
if (R1<R2) then
    R2=R2-R1
    R2=R2/2
else
    R1=R1-R2
    R1=R1/2
```

```
CMP     R1, R2 ; If R1 < R2 (less then)
ITTEE   LT     ; then execute instruction 1 and 2
               ; (indicated by T)
               ; else execute instruction 3 and 4
               ; (indicated by E)
SUBLT.W R2,R1  ; 1st instruction
LSRLT.W R2,#1  ; 2nd instruction
SUBGE.W R1,R2  ; 3rd instruction (notice the GE is
               ; opposite of LT)
LSRGE.W R1,#1  ; 4th instruction
```

Conditional Execution

- ARM allows non-control flow based instructions to be appended with conditional codes.
- It allows for more efficient coding and processor performance.

Conditional Instruction Method

```
        CMP     r2, #5      //if (a <= 5)
        MOVLE   r2, #10     //a = 10;
        MOVGT   r2, #1      //else a = 1;
```

Non-Conditional Method

```
        CMP     r2, #5
        BGT     t_else
        MOV     r2, #10
t_else: MOV     r2, #1
```

Loops: Variable #Iterations

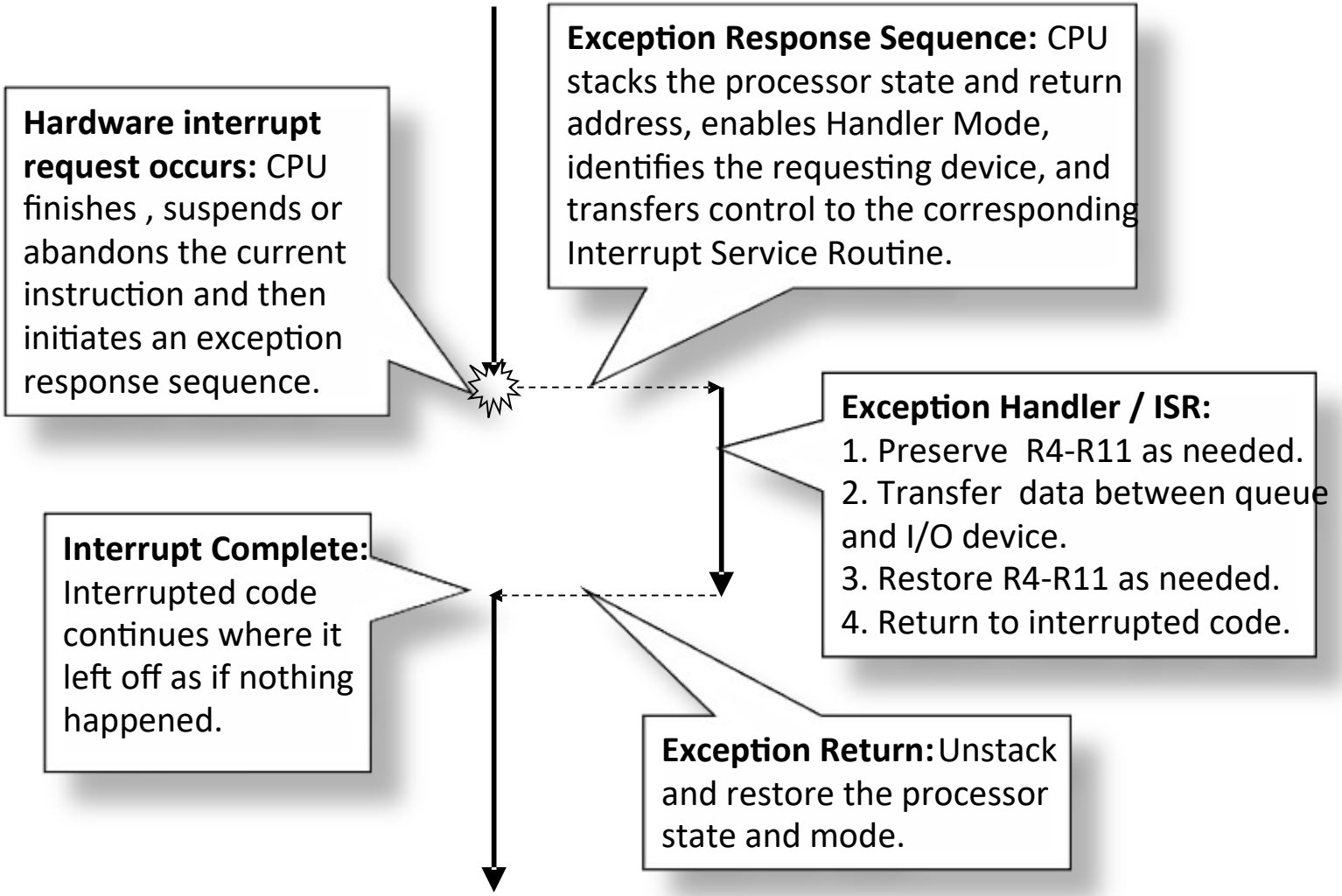
GCD (a, b) – Greatest Common Divisor

```
while (a != b)  {                    LDR     R0,a
    if (a > b) a = a – b ;           LDR     R1,b
    else b = b – a ;          top:   CMP     R0,R1
}                                    BEQ     done
                                     ITE     GT
                                     SUBGT    R0,R0,R1
                                     SUBLE    R1,R1,R0
                                     B       top
                              done:
                                     ; R0 = R1 = GCD(a,b)
```

ARM Procedure Call Standard

Register Number	APCS Name	APCS Role
0	a1	argument 1 / integer result / scratch register
1	a2	argument 2 / scratch register
2	a3	argument 3 / scratch register
3	a4	argument 4 / scratch register
4	v1	register variable
5	v2	register variable
6	v3	register variable
7	v4	register variable
8	v5	register variable
9	sb/v6	static base / register variable
10	sl/v7	stack limit / stack chunk handle / reg. variable
11	fp	frame pointer
12	ip	scratch register / new-sb in inter-link-unit calls
13	sp	lower end of current stack frame
14	lr	link address / scratch register
15	pc	program counter

Interrupt Processing



Exception/Interrupt Handler

Exception: a condition that needs to halt the normal sequential flow o instruction execution.

Exception Categories: Reset, SVC Supervisor Call (Software Interrupt), Fault (e.g., undefined op-code) and Interrupts

Each exception has:

- An exception number
- A priority level
- An exception handler routine (such as ISR)
- An entry in the vector table (address of associated ISR)

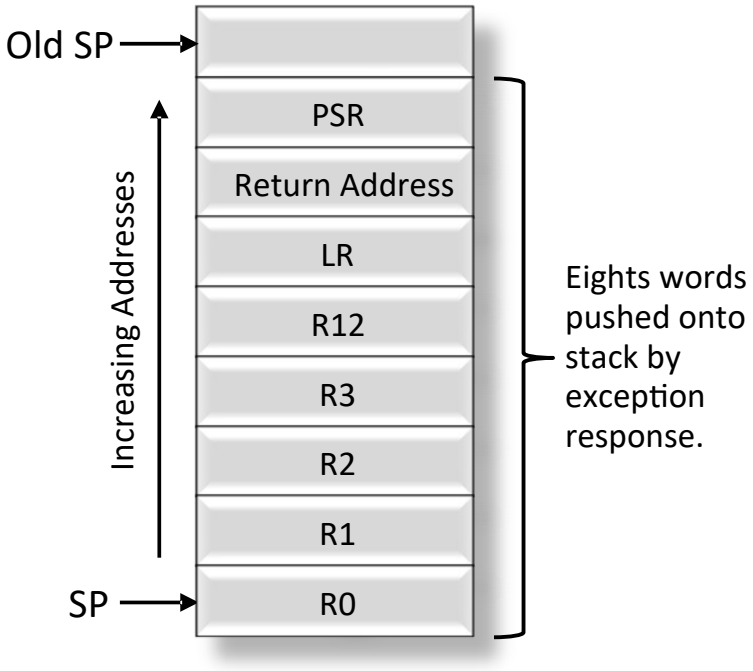
Exception Response

- Processor state (8 words) stored on stack: CPSR, Return Address, LR, R12, R3 - R0. *Allows a regular C function to be an ISR!*
- Processor switched (from Thread Mode) to Handler Mode (recorded in xPSR or CPSR).
- PC ← vector table [exception #]

Exception Handlers and Return

An exception handler (ISR) is a software routine that is executed when a specific exception condition occurs.
Most, but not all, exception handlers return to the previous code.

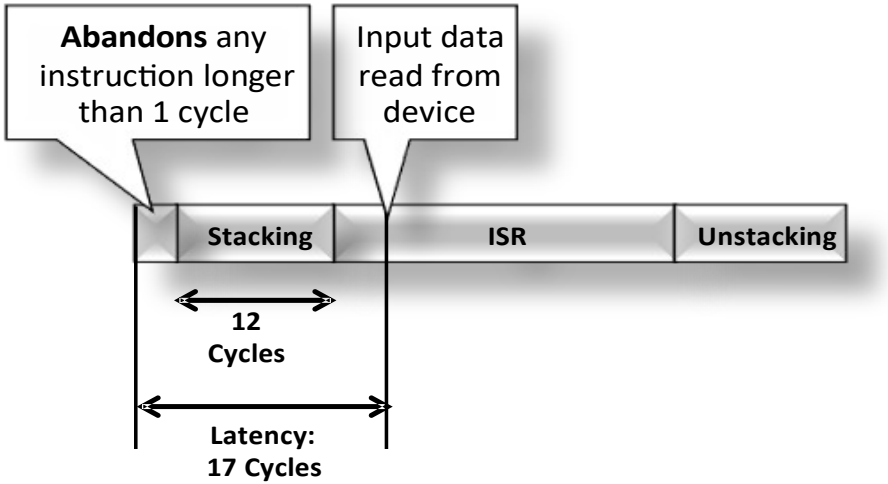
Interrupt Stacking



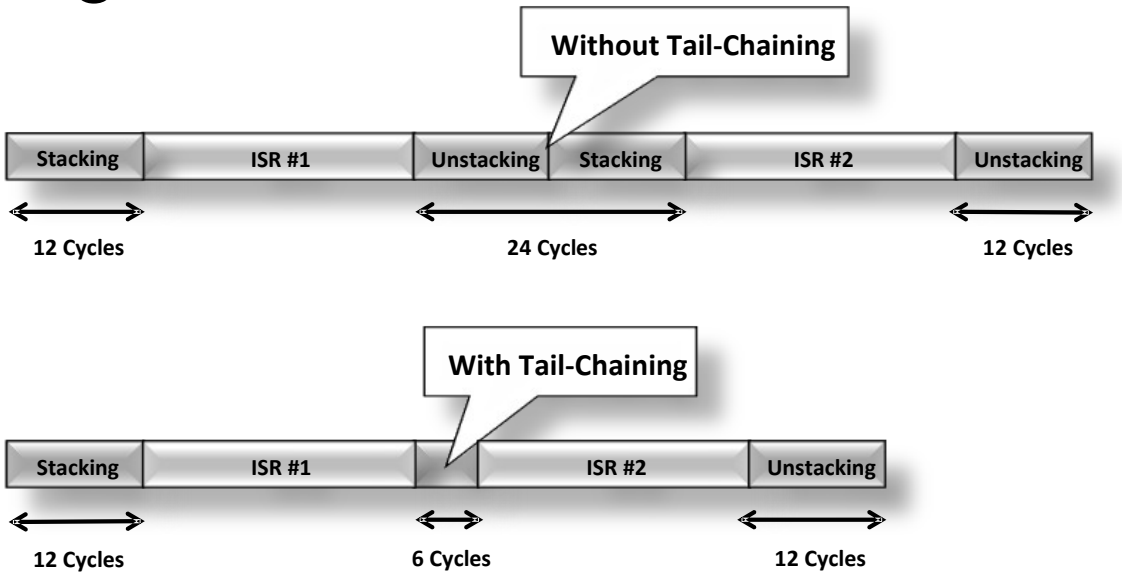
Exception return occurs *when in Handler Mode* and one of the following instructions is executed:

- POP/LDM includes the PC, or
- LDR with PC as the destination, or
- BX with any register as the source

Interrupt Latency



Tail Chaining



Interrupt Latency Reduction

Time from interrupt request to the corresponding interrupt handler begins to execute.

1. Suspend or Abandon Instruction Execution:
No need to suspend single cycle instruction but multiple cycle ones such as LDM, STM, PUSH and POP that transfer multiple words to/from memory.

2. Late Arrival Processing:
CPU has begun an interrupt response sequence and another high priority interrupt arrive during the stacking operation. The CPU will redirect the remainder of the interrupt response so that it can handle the late arriving (higher priority) interrupt.

3. Tail Chaining:
In most CPUs when two ISRs execute back to back, the state information (8 word o CPU state) is popped off the stack at the end of 1st interrupt only to be pushed back at the beginning of the 2nd (next) interrupt.
M3 completely eliminates this useless pop-push sequence with a technique called tail-chaining, lowering the ISR transition time from 24 down to 6 clock cycles.

```
CPSIE  i  ; Enable External Interrupts
CPSID  i  ; Disable External Interrupts
```

Cortex-M3 (Interrupt/Excep.) Vector Table

Exception Type	Position	Priority	Comment
	0	-	Initial SP value (loaded on reset)
Reset	1	-3	Power up and warm reset
NMI	2	-2	Non-Maskable Interrupt
Hard Fault	3	-1	
Memory Mgmt	4	Settable	
Bus Fault	5		Address/Memory-related faults
Usage Fault	6		Undefined instruction
	7-10		*Reserved*
SVCall	11		Software Interrupt (SVC instruction)
Debug Monitor	12		
	13		*Reserved*
PendSV	14		
SysTick	15		System Timer Tick
Interrupts	≥16		External; fed through NVIC

Nested Vectored Interrupt Controller

Mapped to addresses $E000E100_{16}$-$E000ECFF_{16}$

It provides ability to:

- Individually Enable/Disable interrupts from specific devices.
- Establishes relative priorities among the various interrupts.

NVIC INTERRUPTS

0-4	GPIO Ports A-E	18	Watchdog Timer
5,6	UART 0 & 1	19-24	Timer 0a-2b
7	SSI	25	Analog Comparator
8	I^2C	26-27	Reserved
9	PWM Fault	28	System Control
10-12	PWM Generator 0-2	29	Flash Control
13	Reserved	30-31	Reserved
14-17	ADC Sequence 0-3		

Bit in the interrupt registers

ARM9 CPU Applications

Nokia 500 Navigation

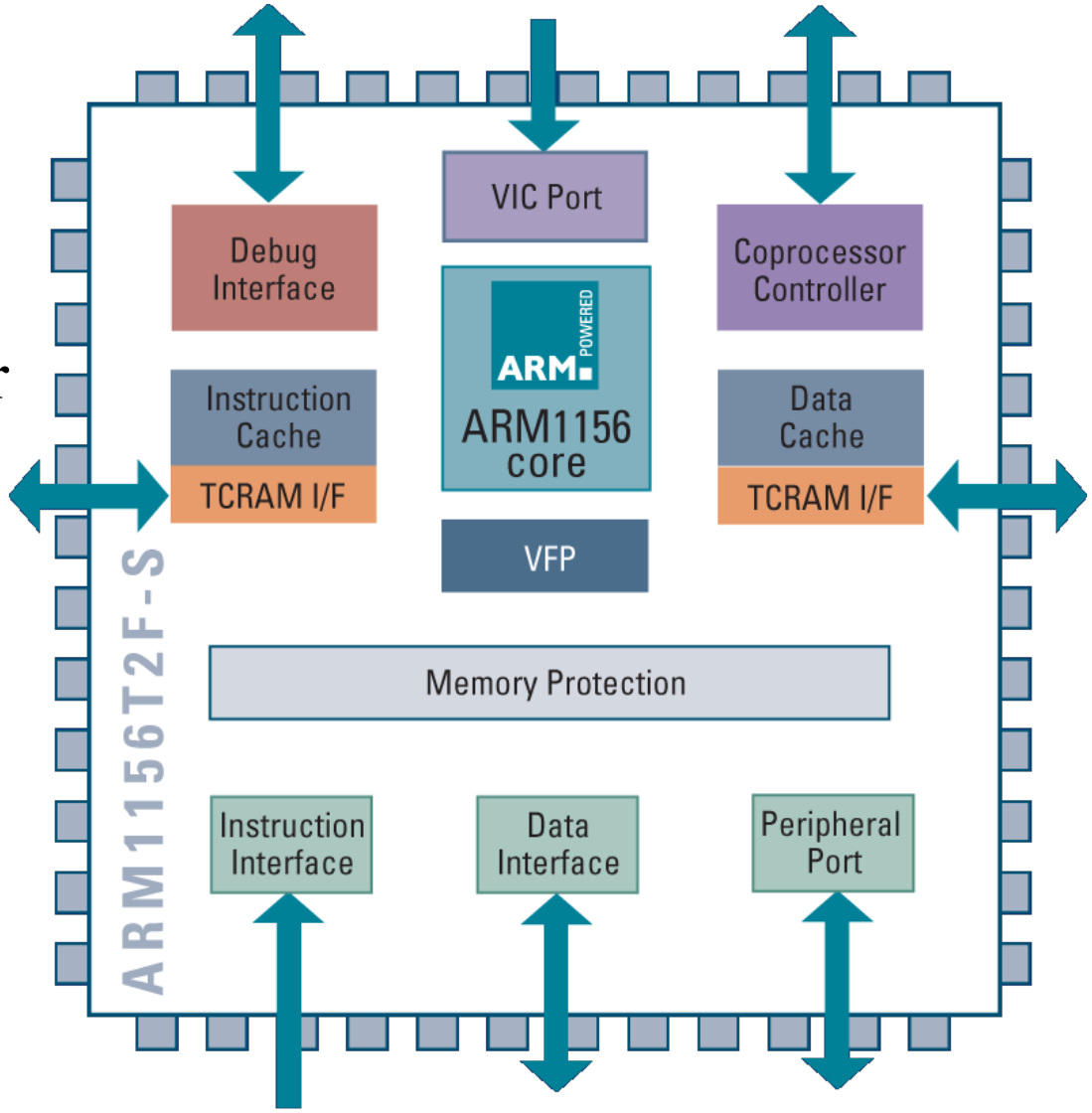
- Runs Windows CE 5.0
- Features a 400MHz ARM9 CPU
- 4.3-inch touch-screen
- GPS
- Bluetooth 2.0, music and video playback capabilities
- An integrated FM transmitter
- The Nokia 500 Auto Navigation system can do voice dialing.
- Built-in FM transmitter can broadcast phone calls or turn-by-turn driving directions to a car radio.

Sony Playstation 3 (60GB)

- ARM9E Processor
- CPU: Cell Processor
- GPU: RSX
- Memory: 256MB XDR Main RAM, 256MB GDDR3 VRAM
- HDD: 2.5" Serial ATA (60GB)
- I/O: USB 2.0 (x4) -
- Communication: Ethernet, IEEE 802.11 b/g - Bluetooth 2.0 (EDR)
- AV Output: Screen size 480i, 480p, 720p, 1080i, 1080p
- HDMI OUT - (x1 / HDMI NextGen)
- DIGITAL OUT (OPTICAL) (x1)
- BD/DVD/CD Drive

ARM 11 (v6) CPU Core

- 8-stage pipeline with branch prediction.
- 16k to 64k instruction and data L1 cache
- 32-bit RISC processor with ARM/Thumb ISA, SIMD and Java support
- Optional VFP coprocessor for high performance 3D graphics and DSP.
- Low-power design



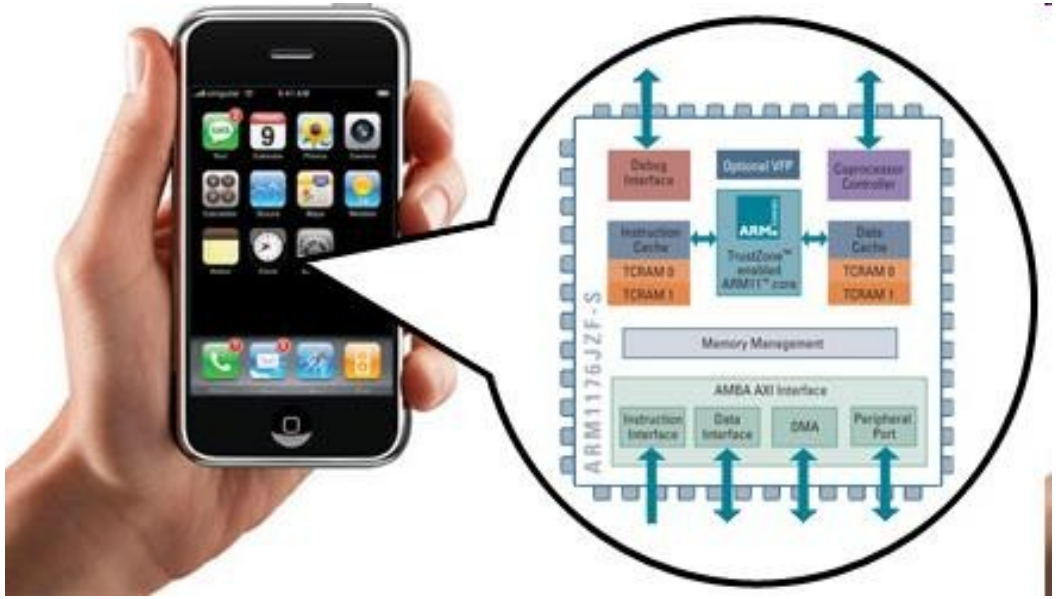
Why ARM 11 (ARMv6)

- The function, performance, speed, power, area and cost parameters must be balanced to meet the requirements of each application.
- ARMv6 offers better ways of optimizing these constraints across a number of vertical market segments. Delivering leading performance/power (MIPS/Watt) has been the mai goal of ARMv6 architecture.
- ARMv6 will benefit developers targeting wireless, networking, automotive and consumer entertainment markets.
- ARM has worked with architecture licensees and partners such as Intel, Microsoft, Symbian and TI in specifying the requirements for ARMv6.

ARM 11 Applications

Navigation System

iPhone based on ARM1176JZ



Target Applications

Next Generation Consumer Devices

From rich single-CPU handheld devices through to embedded general-purpose computing platforms

Rich Multi-function Networking/enterprise Appliances

High throughput devices
Computer graphics
Imaging

Auto-Infotainment

Navigation

Routing
Recognition
Media-Player, etc

Wireless Handsets

Open multi-media platforms

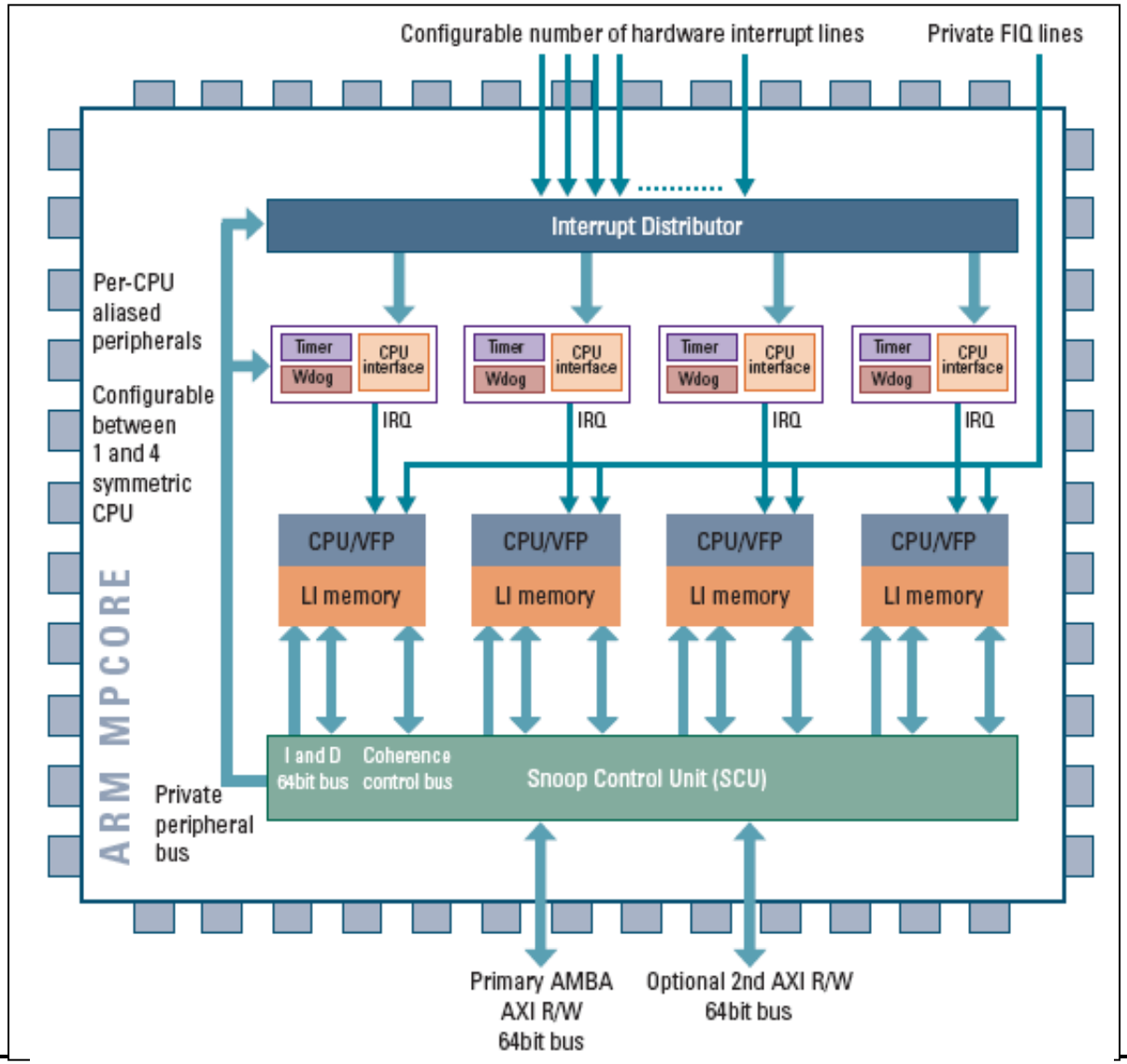
Current devices using ARM (non-MPCore)

ARM11 MPCore

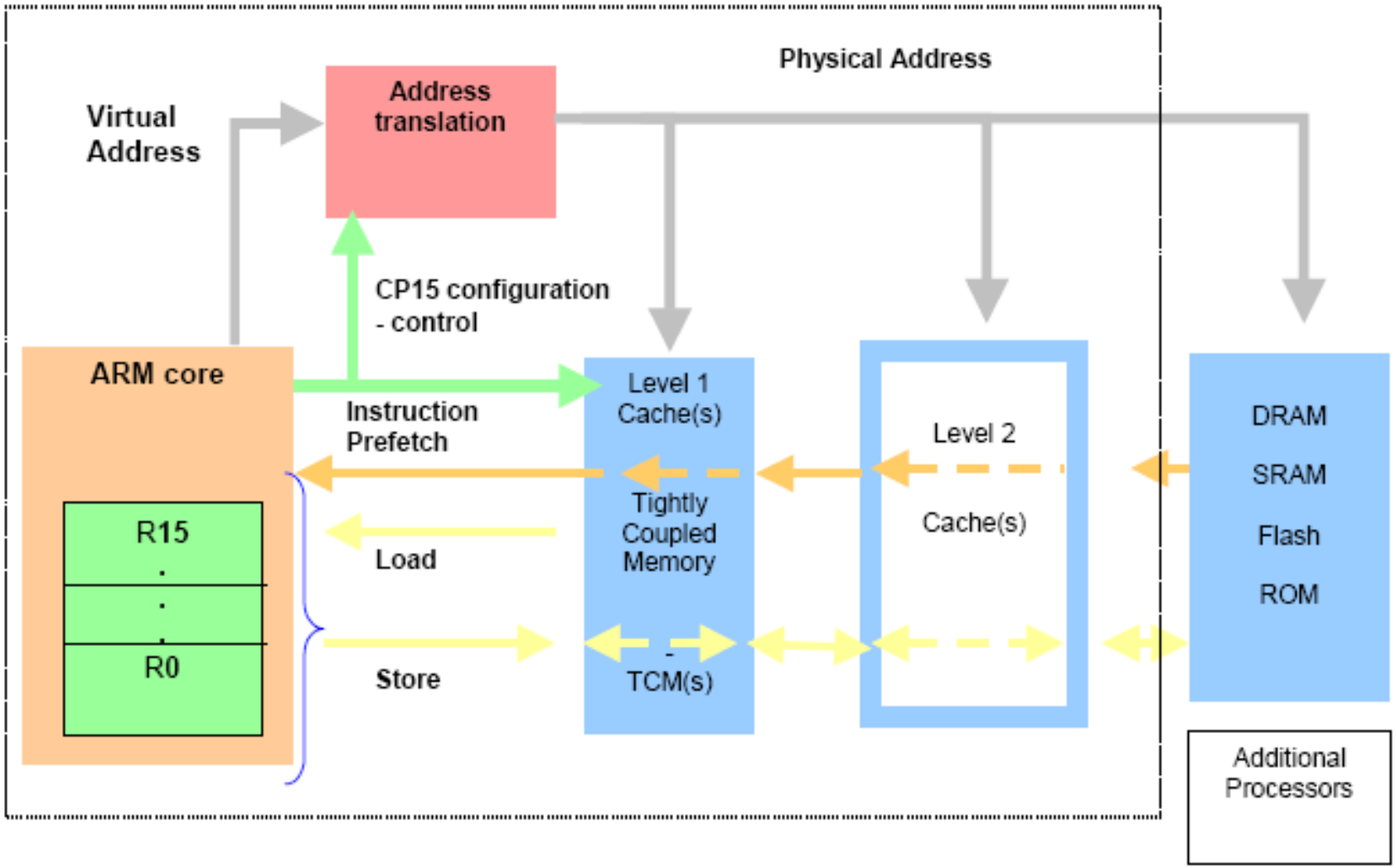
- Includes one to four ARM11™ micro-architecture processors.
- 8-stage integer pipeline with branch prediction and folding.
- 16k to 64k instruction and data L1 cache preprocessor.
- Low-power design

32K INSTRUCTION 32K DATA CACHE 32 INTERRUPTS including VFP	POWER TSMC 0.13 µm GENERIC, 1.2v	DIE AREA TSMC 0.13 µm GENERIC	PERFORMANCE 335-550* MHZ, W/C ON VARIOUS TSMC 0.13 µm
1 CPU	0.8 mW/MHz	7.0 mm^2	*680 DMIPS
2 CPU	1.8 mW/MHz	15.2 mm^2	*1360 DMIPS
4 CPU	3.3 mW/MHz	30.5 mm^2	*2720 DMIPS

ARM11 MPCore



ARMv6 Memory Model



Power Management Modes - ARM11

- **Run Mode**
  This mode is the normal mode of operation in which all of the functionality of the ARM11 processor is available. If an ARM1176 or other IEM-aware core is used, the Energy Management capabilities of the IEM module are used in Run Mode.
- **Standby Mode**
  This mode disables most of the clocks of the device, while keeping the device powered up. This reduces the power drawn to the static leakage current, plus a tiny clock power overhead required to enable the device to wake up from Standby Mode.
- **Shutdown Mode**
  This mode allows the entire device to power down. All processor state, including cache and TCM state, must be saved externally.
- **Dormant Mode**
  This mode enables the ARM11 processor core to power down while leaving the caches and the TCM powered up and maintaining their state

TMS470 Family MCUs

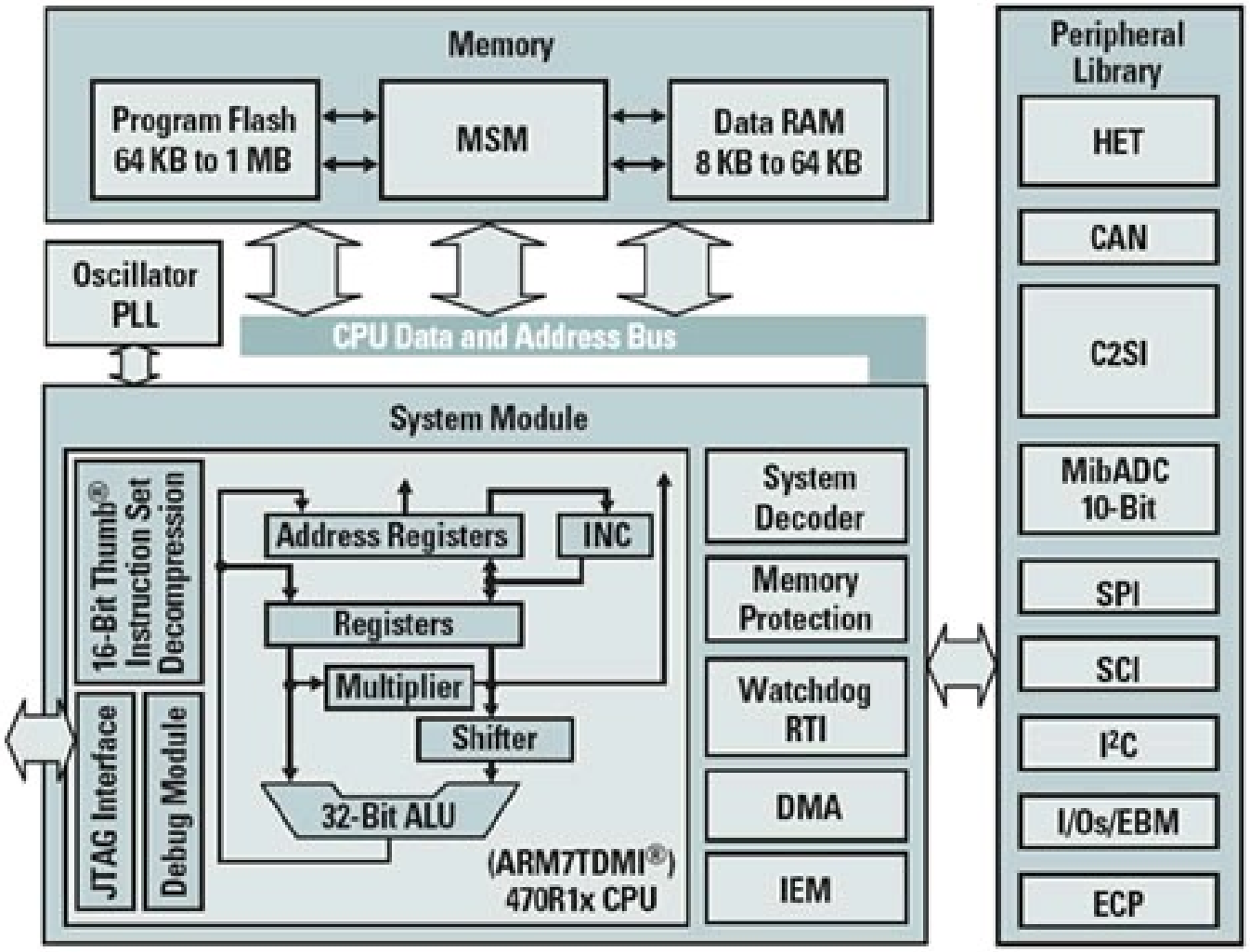
TMS470 family of automotive micro-controllers:

- Texas Instruments (TI) offers the TMS470 micro-controllers
- Derived from the 16/32-bit ARM7TDMI and other ARM cores
- Licensed by Texas Instruments (TI) from ARM Ltd.
- Launched in 1995

Typical applications include:

- Industrial systems
- Medical instrumentation
- Consumer electronics
- Data processing and many other general purpose embedded applications.

TMS470 ARM based TI Micro-controller

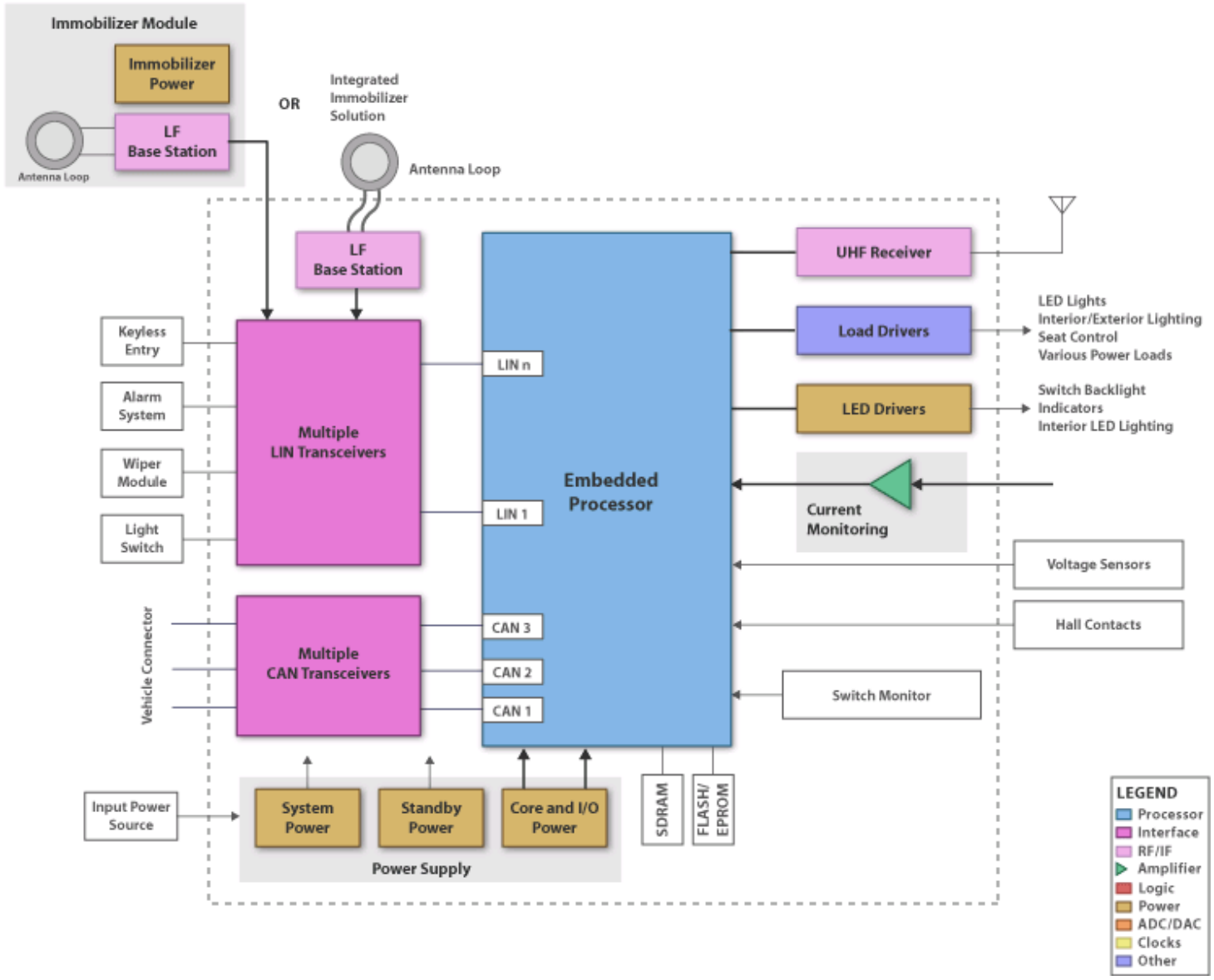


TMS470: Automotive Application

AutomotiveμC address automotive application needs including:

- Anti-lock Braking System (ABS)
- Electro Mechanical Braking
- Electronic Stability Control (ESP)
- Automotive Central Body Controller
  - Supervises and controls functions related to the car body such as: lights, windows, door lock and works as a gateway for CAN and LIN (Local Interconnect Networks) networks.
  - Load control can either be directly from the DBM or via CAN/LIN communication with remote ECUs.
  - The central body controller often incorporates RFID functions like remote keyless entry and immobilizer.

Automotive Central Body Controller



Automotive Central Body Controller

Micro-Controller

The μC works as gateway for the bus and network interfaces and controls the various load drivers.

Communication Interfaces

- Allow data exchange between independent electronic modules in the car, as well as remote sub modules.
- High Speed (up to 1Mbps) CAN (Control Area Network) is a 2-wire, fault tolerant differential bus.
- It serves as the main vehicle bus type for connecting the various electronic modules in the car with each other.
- LIN supports low speed (up to 20 kbps) single bus wire networks, used to communicate with remote sub functions of the infotainment system.

Automotive Central Body Controller

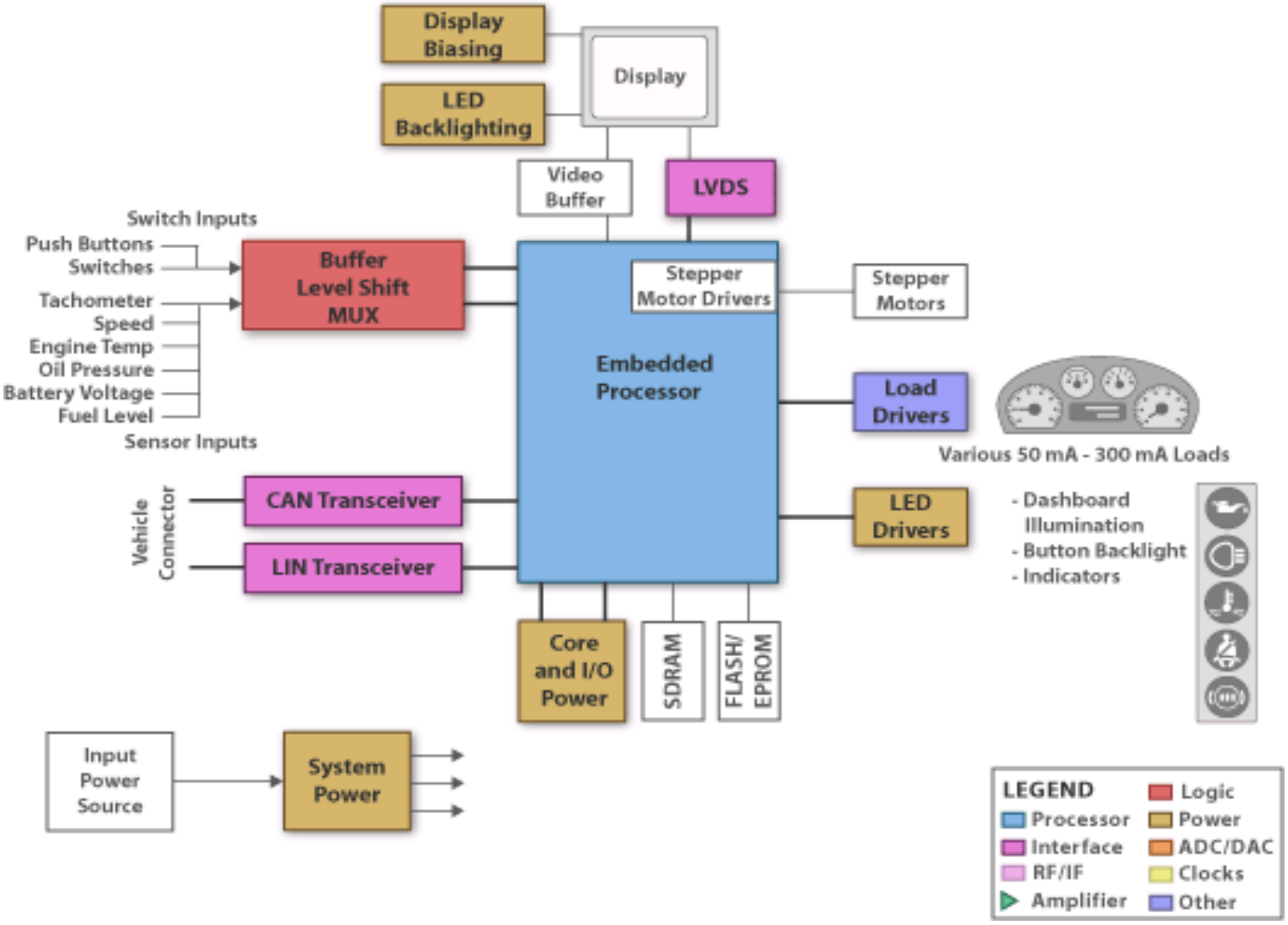
Load Drivers:

- Main load driver types in a central body controller are lights and relay drivers.
- The switches and drivers for the exterior lighting are placed on the controller directly.
- Relays are used to power other electronic modules or loads.
- Current monitoring supervises demand from the distributed loads, other ECUs and used for charge & load management of the car battery.

RFID Functions -Most common automotive RFID functions are:

- Immobilizer and the remote keyless entry system.
- LF base station IC for encrypted communication with the ignition key (immobilizer)
- Ultra Low power sub 1-GHz UHF transceiver for communication with remote control for locking/ unlocking the doors and the alarm system.

Automotive Instrument Cluster



Display information/status of vehicle systems and drive conditions

Instrument Cluster

The information/status include gauges for various parameters, indicators, and status-lights as well as acoustical effects.

- Displays range from small dot matrix up to large color, high resolution LCD displays

In addition to CAN/LIN interface there are LVDS interfaces.

- LVDS interfaces are used to transfer large amounts of data via a high-speed serial connection to an external location like a video screen.

The main load types in a Cluster are the stepper motors that operate the gauges and the various indicator and back light sources.

- The Stepper motor drivers are typically integrated in the µC.
- LED drivers are typically multi-channel devices with serial interfaces to the µC or Darlington arrays.

Instrument Cluster

Depending on the display type, a power supply solution for the display biasing is required on top of the LED or CCVF drivers for backlighting. The video information is either sent directly or via a LVDS interface depending on the size of the display.

Micro-controllers aimed at driver information and cluster system needs to drive multiple stepper motors and displays.
These devices need to integrate:

- High performance CPU cores
- Multi-channel DMA controllers
- TFT controllers
- Fast external memory interfaces with adequate system performance to implement graphic functions such as anti-aliasing, texturing, animation, chroma-coding, etc.
- The MCU also needs to high enough performance speed to service the stepper motors in real time.