ARM

The ARM Architecture

Agenda

- **Introduction to ARM Ltd**
- Programmers Model
- Instruction Set
- System Design
- Development Tools

ARM Ltd

- **Founded in November 1990**
  - Spun out of Acorn Computers
- **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-in of the ARM architecture**
  - Software tools, boards, debug hardware, application software, bus architectures, peripherals etc

Why ARM?

- **One of the most licensed and thus widespread processor cores in the world**
  - Used in PDA, cell phones, multimedia players, handheld game console, digital TV and cameras
  - ARM7: GBA, iPod
  - ARM9: NDS, PSP, Sony Ericsson, BenQ
  - ARM11: Apple iPhone, Nokia N93, N800
  - 75% of 32-bit embedded processors
- **Used especially in portable devices due to its low power consumption and reasonable performance**

ARM Powered Products

ARM processors

- A simple but powerful design
- A whole family of designs sharing similar design principles and a common instruction set

Naming ARM

- **ARMxyzTDMIEJFS**
  - x: series
  - y: MMU
  - z: cache
  - T: Thumb
  - D: debugger
  - M: Multiplier
  - I: EmbeddedICE (built-in debugger hardware)
  - E: Enhanced instruction
  - J: Jazelle (JVM)
  - F: Floating-point
  - S: Synthesizible version (source code version for EDA tools)

Popular ARM architectures

- **ARM7TDMI**
  - 3 pipeline stages (fetch/decode/execute)
  - High code density/low power consumption
  - One of the most used ARM-version (for low-end systems)
  - All ARM cores after ARM7TDMI include TDMI even if they do not include TDMI in their labels
- **ARM9TDMI**
  - Compatible with ARM7
  - 5 stages (fetch/decode/execute/memory/write)
  - Separate instruction and data cache
- **ARM11**

ARM family comparison

ARM family attribute comparison.

year	1995	1997	1999	2003
	ARM7	ARM9	ARM10	ARM11
Pipeline depth	three-stage	five-stage	six-stage	eight-stage
Typical MHz	80	150	260	335
mW/MHz[a]	0.06 mW/MHz	0.19 mW/MHz (+ cache)	0.5 mW/MHz (+ cache)	0.4 mW/MHz (+ cache)
MIPS[b]/MHz	0.97	1.1	1.3	1.2
Architecture	Von Neumann	Harvard	Harvard	Harvard
Multiplier	8 × 32	8 × 32	16 × 32	16 × 32

[a] Watts/MHz on the same 0.13 micron process.

[b] MIPS are Dhrystone VAX MIPS.

ARM is a RISC

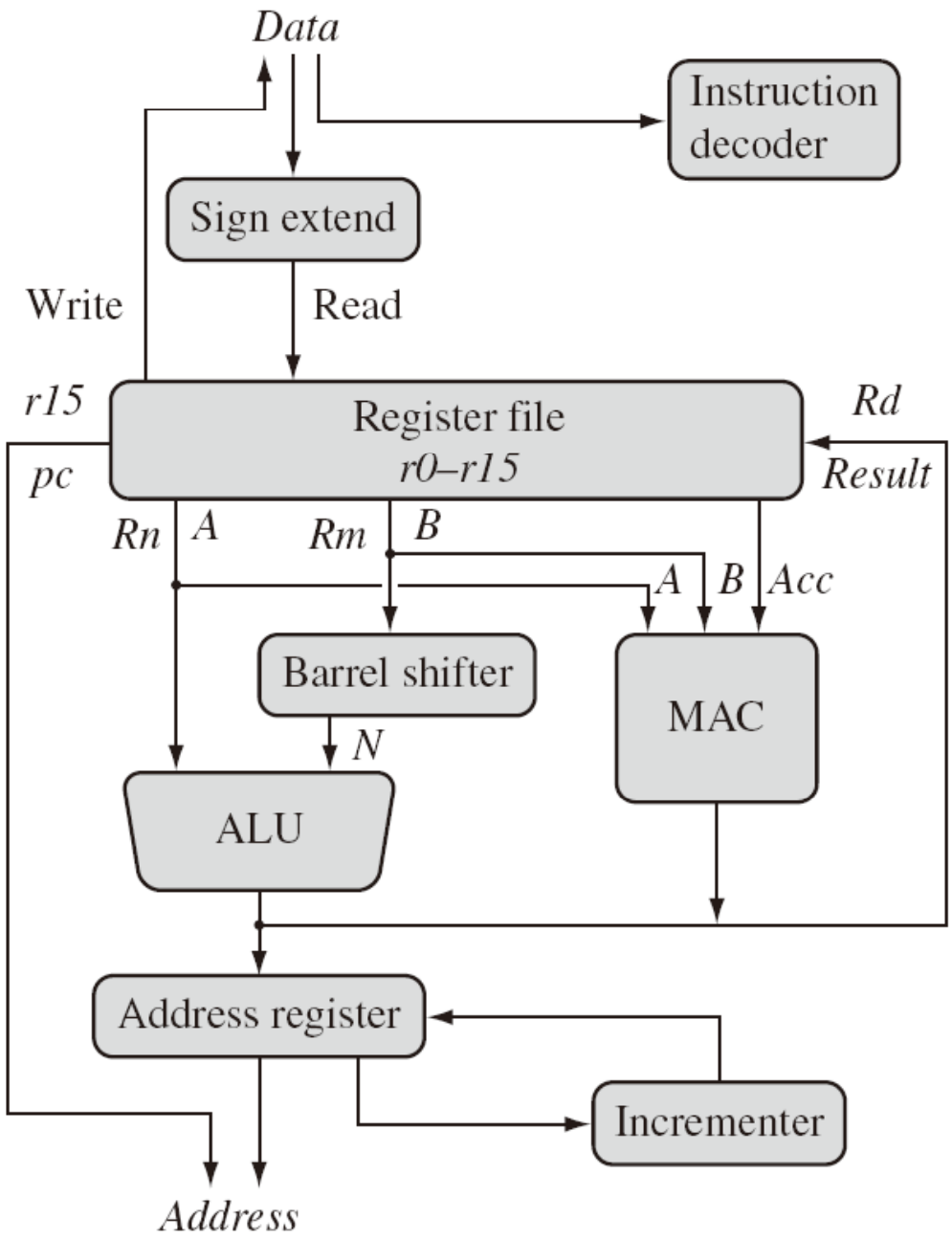
- **RISC: simple but powerful instructions that execute within a single cycle at high clock speed.**
- **Four major design rules:**
  - Instructions: reduced set/single cycle/fixed length
  - Pipeline: decode in one stage/no need for microcode
  - Registers: a large set of general-purpose registers
  - Load/store architecture: data processing instructions apply to registers only; load/store to transfer data from memory
- **Results in simple design and fast clock rate**
- **The distinction blurs because CISC implements RISC concepts**

ARM design philosophy

- Small processor for lower power consumption (for embedded system)
- High code density for limited memory and physical size restrictions
- The ability to use slow and low-cost memory
- Reduced die size for reducing manufacture cost and accommodating more peripherals

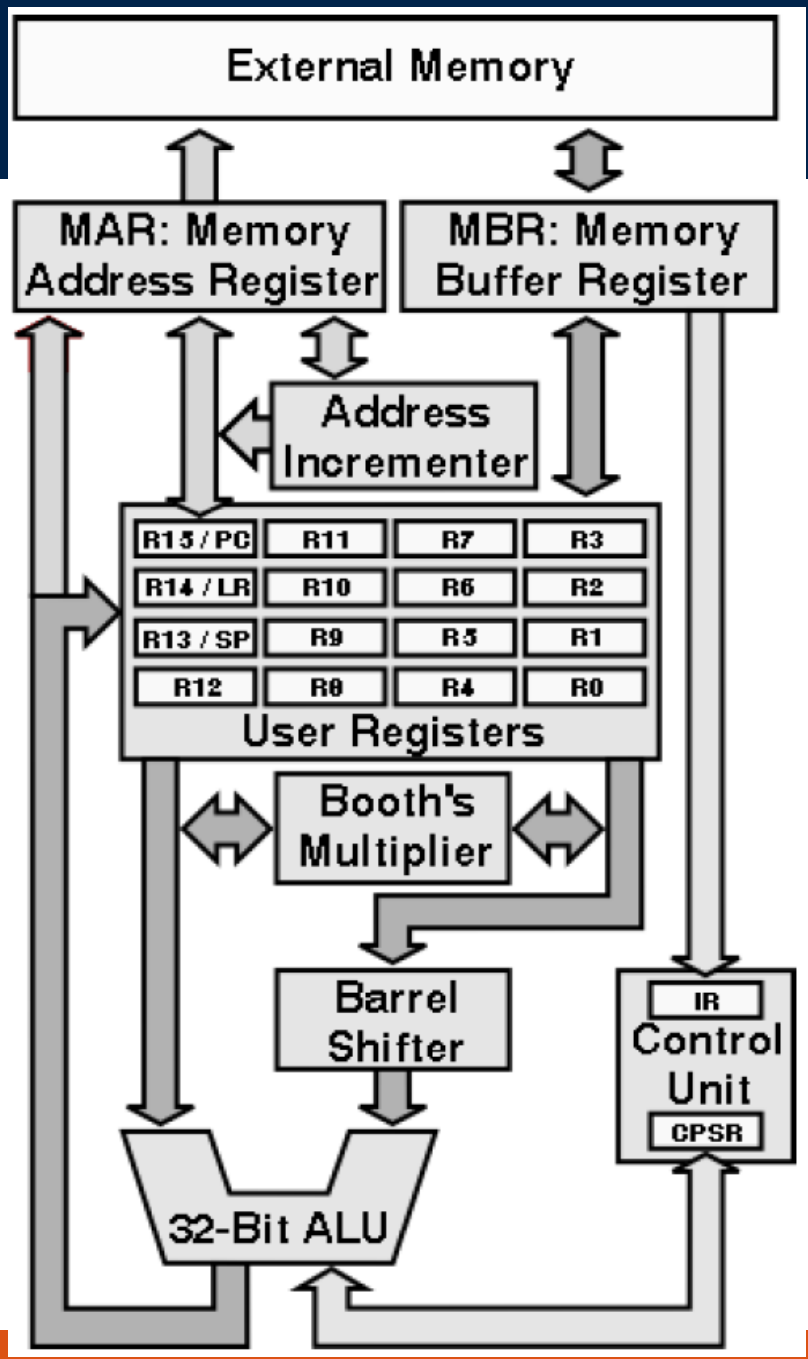
ARM features

- **Different from pure RISC in several ways:**
  - Variable cycle execution for certain instructions: multiple-register load/store (faster/higher code density)
  - Inline barrel shifter leading to more complex instructions: improves performance and code density
  - Thumb 16-bit instruction set: 30% code density improvement
  - Conditional execution: improve performance and code density by reducing branch
  - Enhanced instructions: DSP instructions

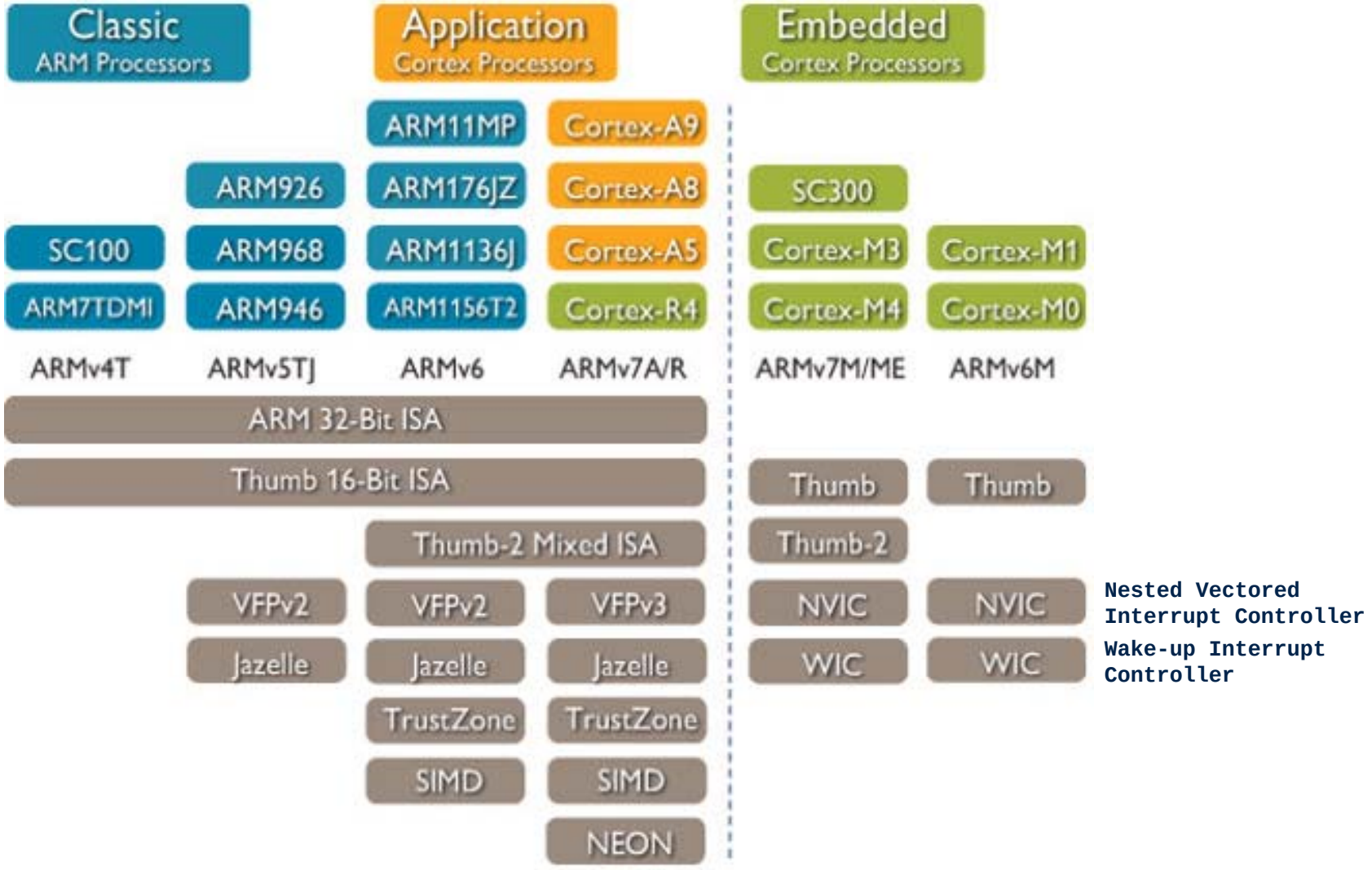

Data
Instruction decoder
Sign extend
Write
Read
r15
pc
Register file
r0–r15
Rd
Result
Rn
A
Rm
B
A
B
Acc
Barrel shifter
MAC
N
ALU
Address register
Incrementer
Address

ARM

- Load/store architecture
- A large array of uniform registers
- Fixed-length 32-bit instructions
- 3-address instructions

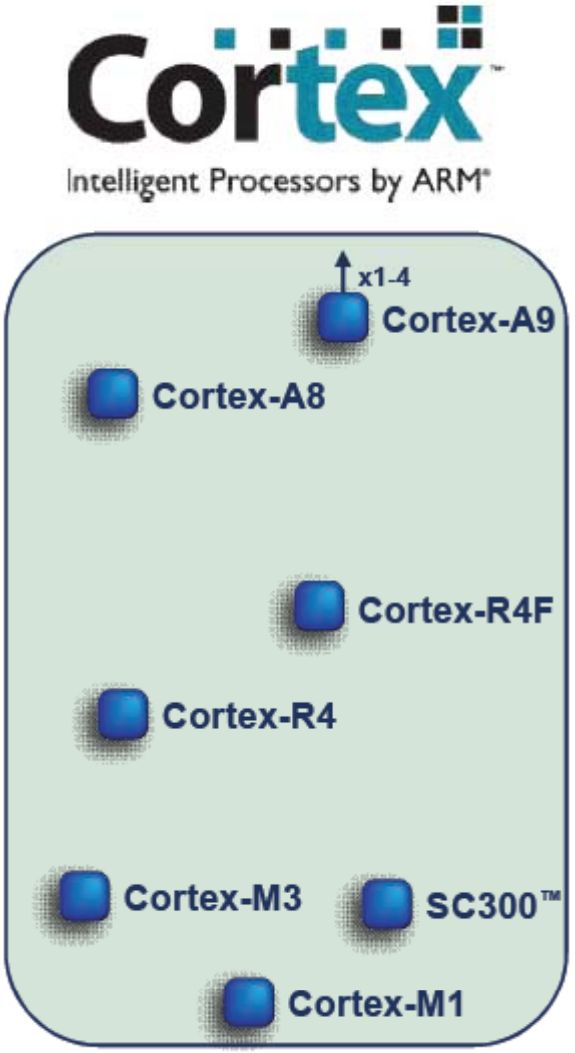


ARM family comparison

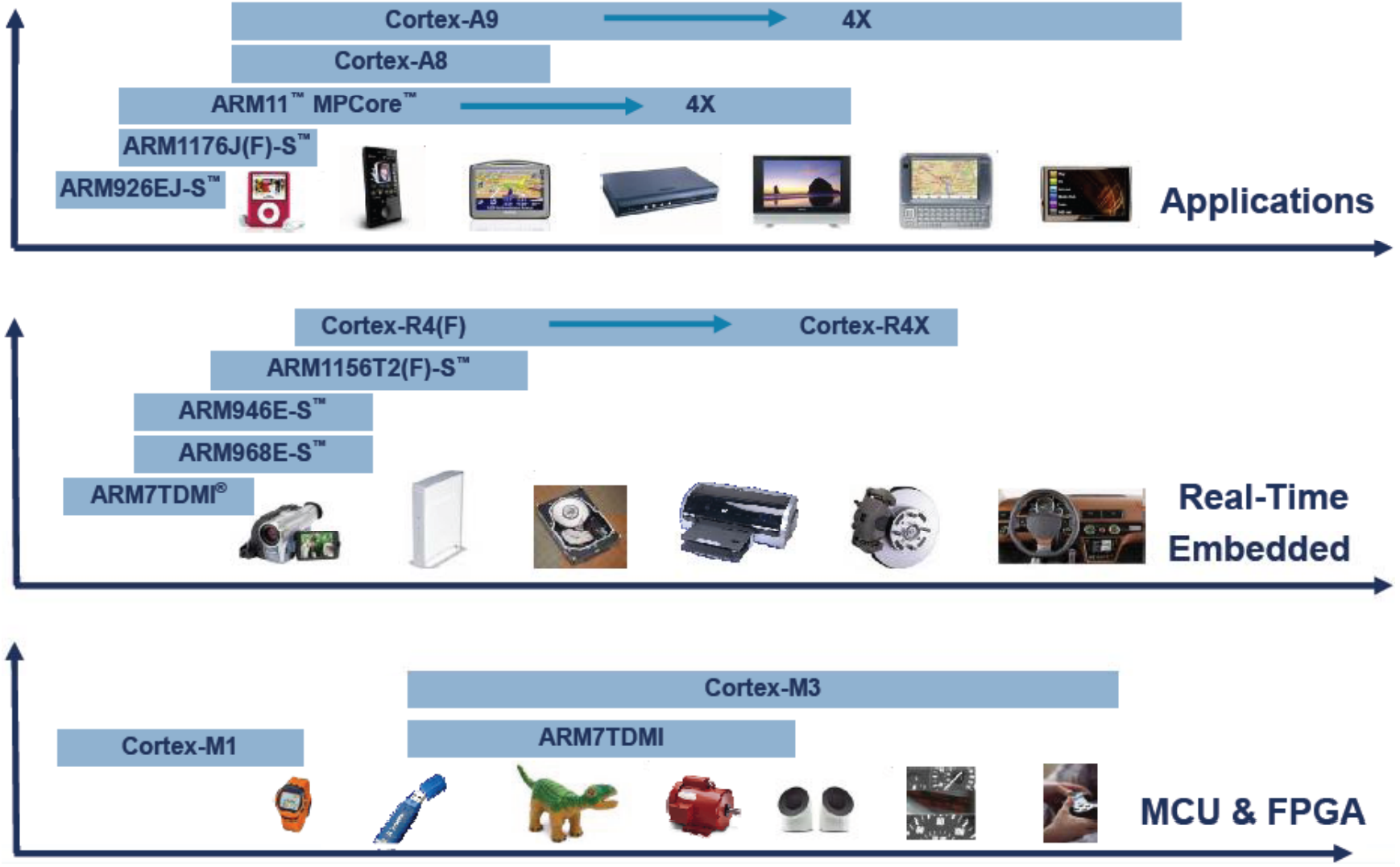


ARM Cortex

- **ARMv7 architecture**
  - Thumb-2 technology
    - optimized, mixed 16/32-bit instruction set
    - the performance advantages of the 32-bit ARM ISA with the code size advantages of the16-bit Thumb ISA
  - NEON™ technology
    - increase DSP and media processing throughput
    - offers improved floating point support to address the needs of next generation 3D graphics and games physics.
- **ARM Cortex-A Series:**
  - Applications processors for complex OS and user applications
- **ARM Cortex-R Series:**
  - Embedded processors for real-time signal processing and control applications
- **ARM Cortex-M Series:**
  - Deeply embedded processors optimized for microcontroller and low-power applications

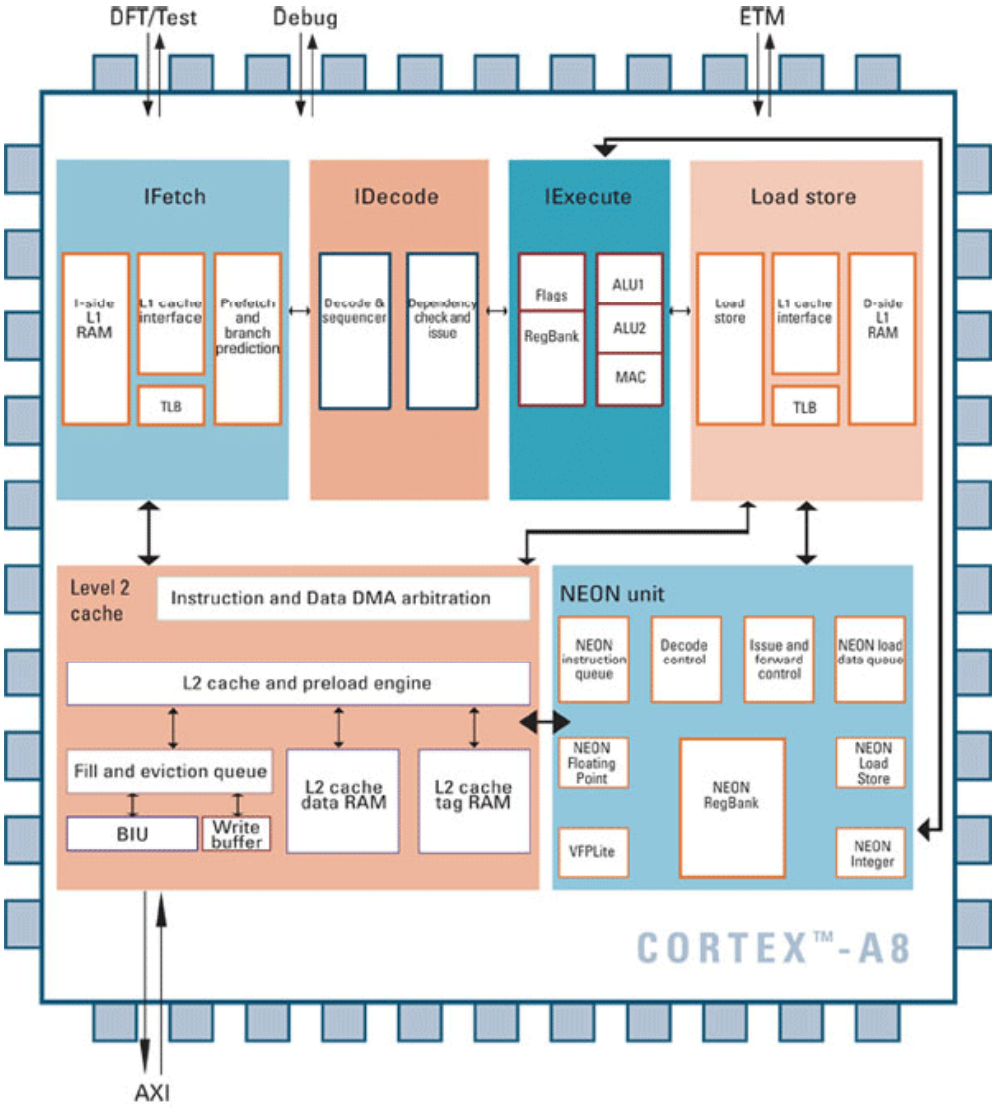


ARM Cortex


Cortex-A9
4X
Cortex-A8
ARM11™ MPCore™
4X
ARM1176J(F)-S™
ARM926EJ-S™
Applications
Cortex-R4(F)
Cortex-R4X
ARM1156T2(F)-S™
ARM946E-S™
ARM968E-S™
ARM7TDMI®
Real-Time
Embedded
Cortex-M3
Cortex-M1
ARM7TDMI
MCU & FPGA

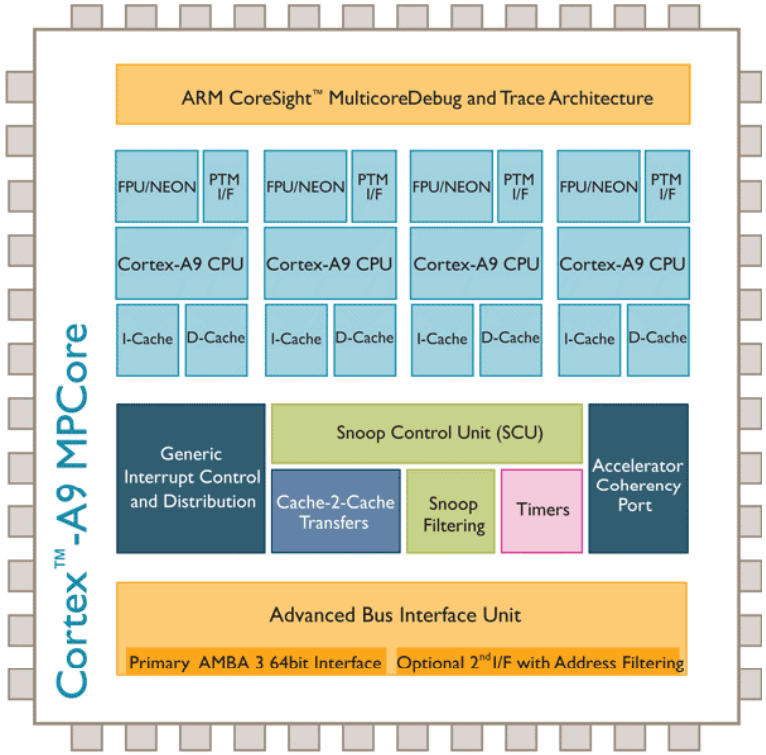
Cortex-A8

- **Highest performance Cortex-A processor**
  - Symmetric, superscalar pipeline for full dual-issue capability: 2 DMIPS/MHz
  - Thumb-2 ISA for performance and code density
  - TrustZone extensions for secure transactions and DRM
  - NEON multimedia and signal processing unit delivers over 2x performance of ARMv6 SIMD
  - Integrated L2 Cache with configurable size and ECC

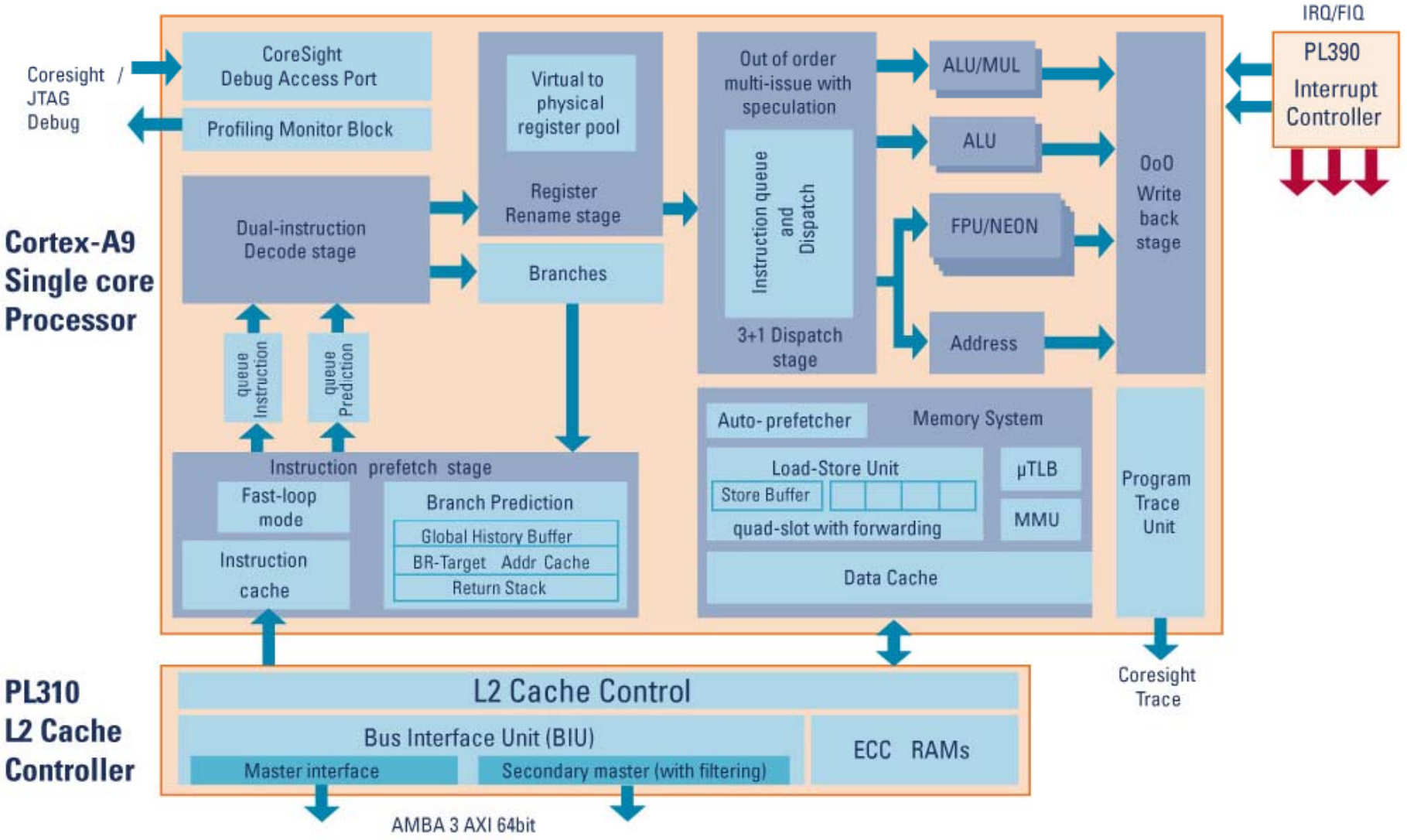


Cortex-A9

- **Optimized Cortex-A processor enabling breakthrough performance and power scalability**
  - 2nd generation 1-4X SMP technology
  - Delivered as Uniprocessor and 1-4X MP
  - Advanced pipeline delivering ~25% more DMIPS/MHz over Cortex-A8 (2.5)
  - Comparable F_{MAX} to fully synthesized Cortex-A8 in same configuration
  - Optimized floating-point unit; NEON engine
- **New system-level integration features for design optimization**
  - Accelerator coherence port
  - Advanced bus interface unit for maximum throughput
  - Generalized interrupt control and distribution system
- **Suitable for high-end enterprise networking through to wireless handsets**



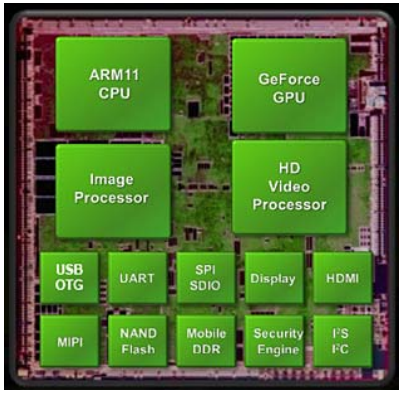
ARM Cortex-A9

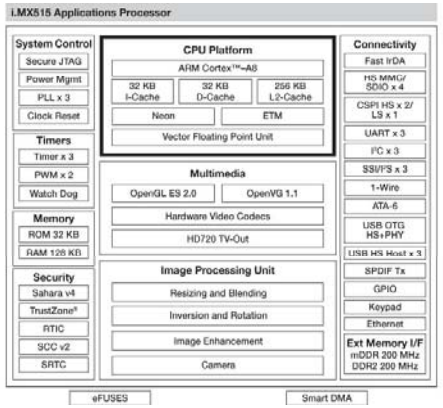


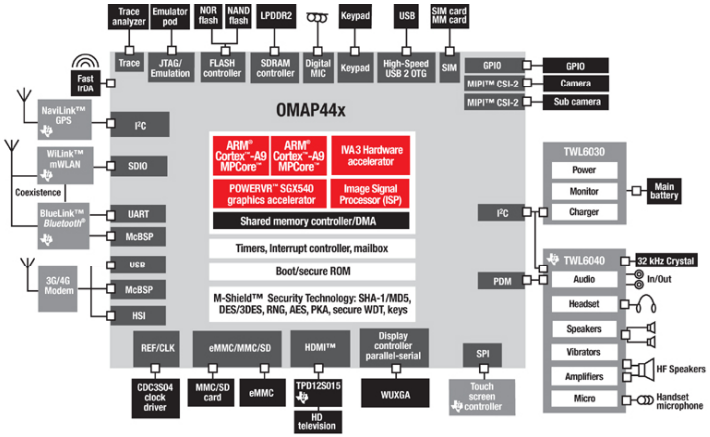
ARM-based Application Processors

Company	Product	Core	Speed	3D Graphics	HD Video	Radios	Linux	Window Mobile	Android	Window Embedded
NVIDIA	Tegra™ 610	ARM11 MPCore™	800 MHz	X	X			X		X
Samsung	6410	ARM11	667 MHz	X	SD	X	X	X		X
Samsung	S5PC100	Cortex-A8	800MHz	X	X		X	X		X
Qualcomm	SnapDragon™	V7 Architecture License	1 GHz	X	X	X	X	X		X
Freescale	iMX515	Cortex-A8	1 GHz	X	X		X	X		X
TI	OMAP™ 3	Cortex-A8	1 GHz	X	X	X	X	X		X
TI	OMAP™4	Cortex-A9	2 cores, 1GHz ea	X	X	X	X	X		X
Marvell	PXA3## (contact for all the products)	V5 Architecture License	803 MHz	X	SD	X	X	X		X

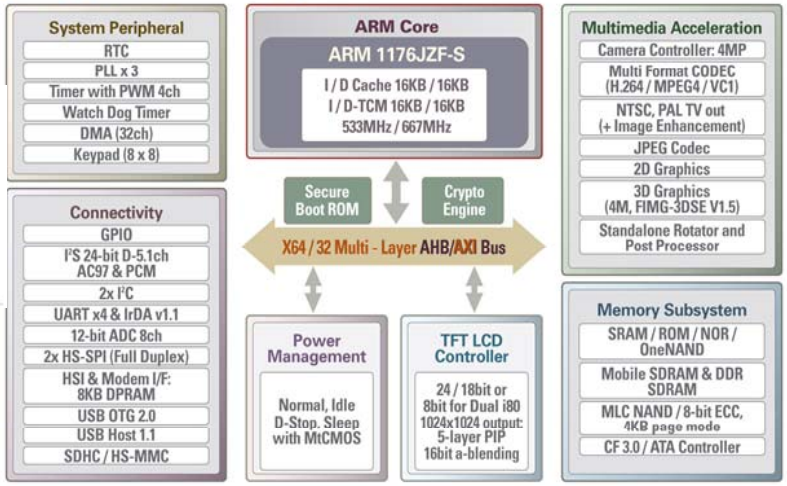
ARM-based Application Processors



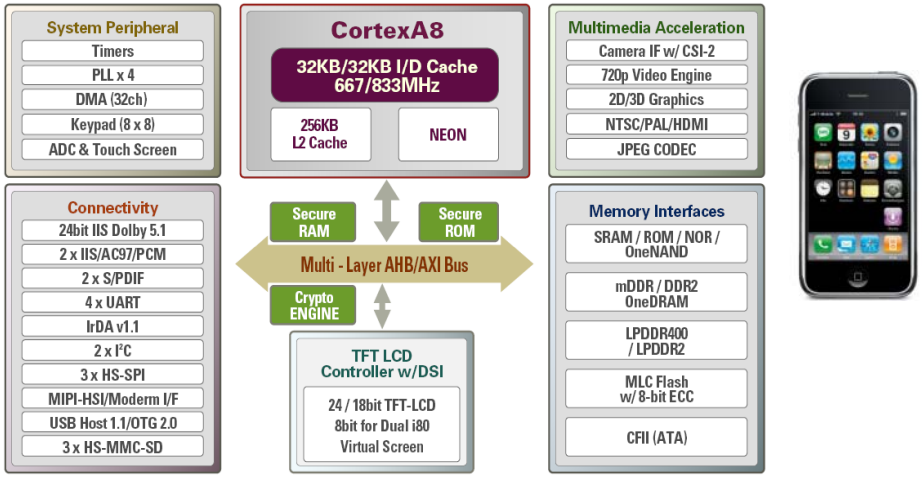




S3C6410 Block Diagram



S5PC100 Block Diagram

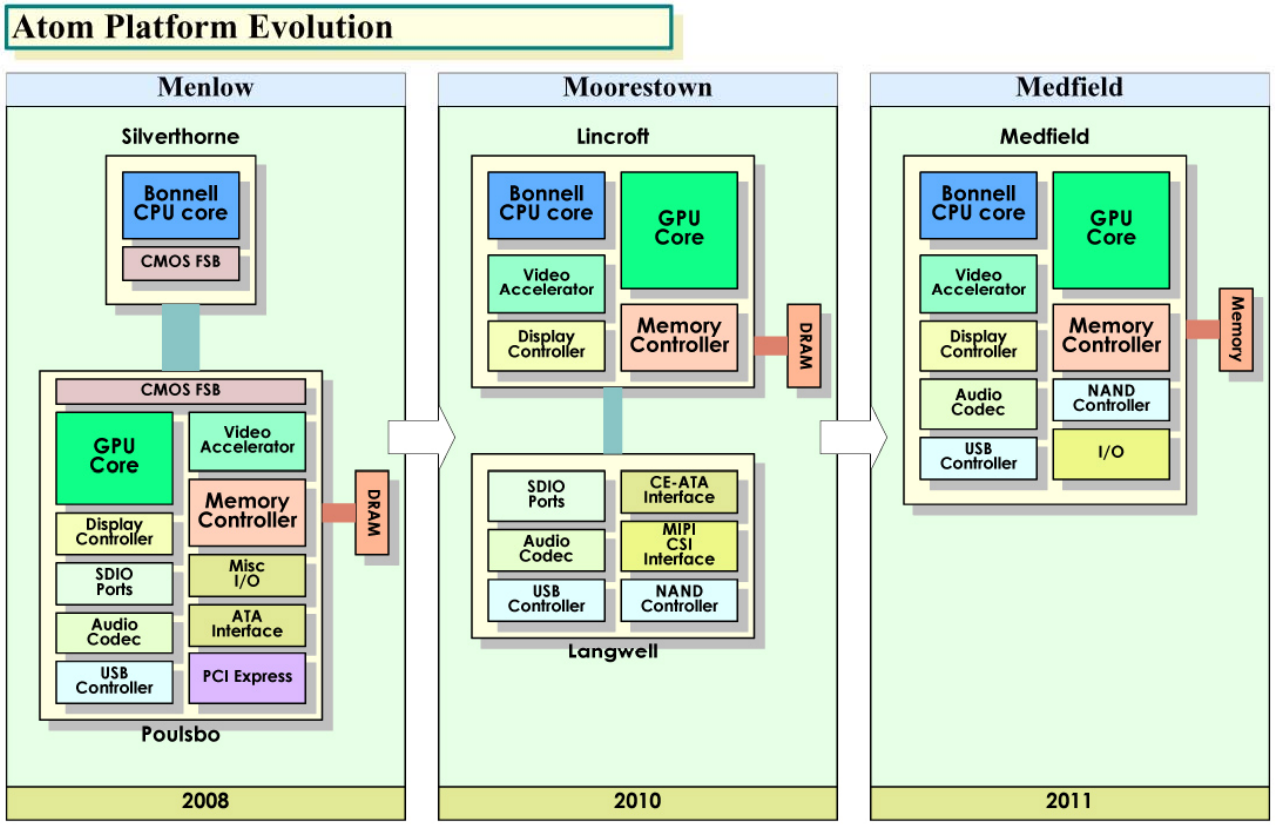


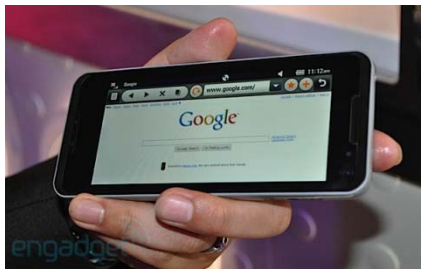
iPad's A4

A4 is a System-on-a-Chip, or SOC, that integrates the main processor [ARM Cortex-A9 MPCore i.e. Multi-Processing Core, identical to ones used in nVidia Tegra and Qualcomm Snapdragon] with graphics silicon [ARM Mali 50-Series GPU], and other functions like the memory controller on one piece of silicon – not unlike what Intel is trying to achieve with its future "Moorestown" Atom processor that debuted inside LG's Smartphone

iPad doesn't have ARM's next-generation Cortex A9 design which supports multicore processors. Instead, it is a single-core ARM Cortex A8 design which is along the same lines as the current iPhone 3GS, iPod touch as well as the Palm Pre, Droid, etc. The A4 is a 1GHz custom SoC with a single Cortex A8 core and a PowerVR SGX GPU (imagination).

Intel's Atom Processor

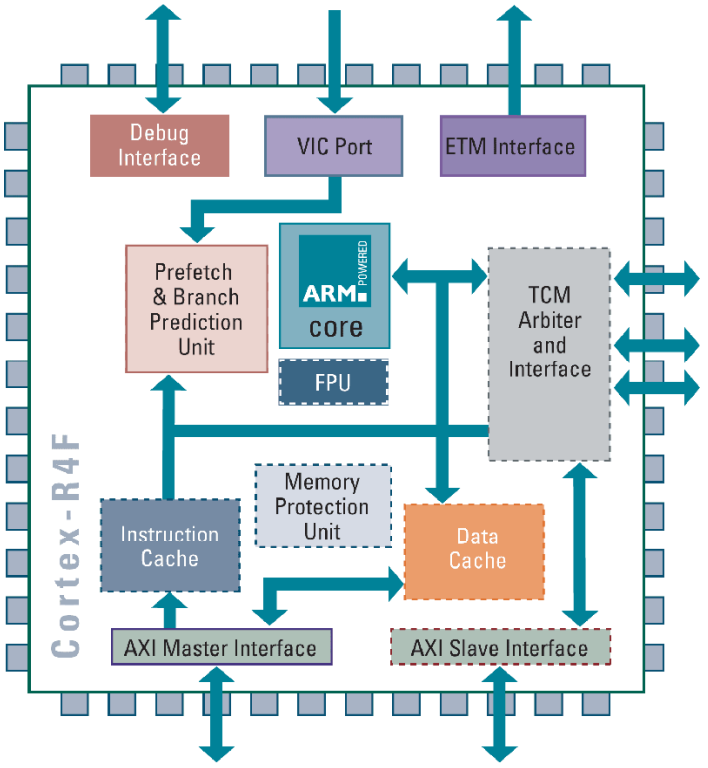




LG GW990

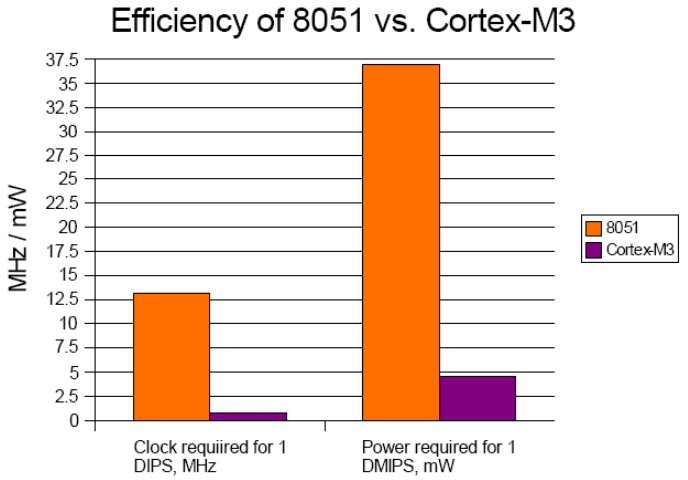
Cortex-R4(F)

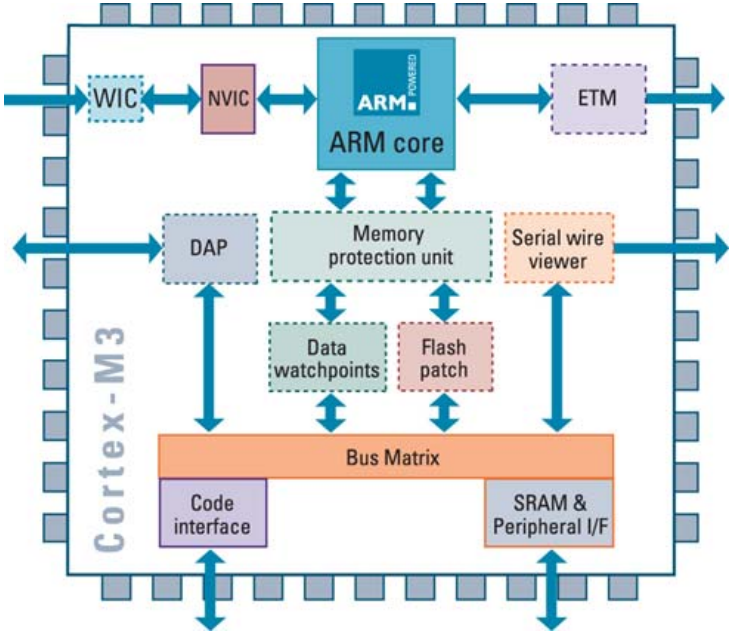
- **High-performance processor optimized for deeply embedded signal-processing and control applications**
  - 8-stage superscalar pipeline delivers up to 400MHz+ @ 1.6DMIPS/MHz on 90nm
  - Thumb-2 technology, hardware divider
  - State-of-the-art ECC support in all memories
  - AMBA 3 AXI slave port for DMA to TCMs
- **Fit-for-purpose configurability**
  - Separately configured L1 caches: 0kB, 4-64kB
  - 0 to 3 TCMs of up to 8MB each
  - 8 or 12 regions in MPU, or no MPU
  - 2 – 8 Breakpoints, 1 – 8 Watchpoints
  - Parity or ECC can be optionally included
  - Optional SP-optimized FPU (full IEEE754)
  - Optional slave port



Cortex-M3

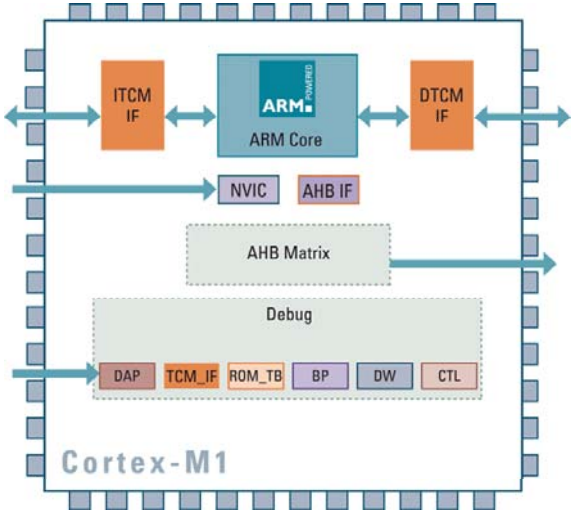
- **An ARM7TDMI-S for the 21st century**
  - For extreme cost and power-sensitive complex applications
  - Comparable or better F_{MAX} and gate count
  - 30% more DMIPS, 28% more geomean EEMBC
  - 85% more DMIPS per mW
- **State-of-the-art functionality**
  - Code *everything in C*
  - Thumb-2 ISA → 6X code density, 10X perf. vs. 8051
  - Integrated Nested Vectored Interrupt Controller (NVIC) with lowest interrupt latency of any ARM
  - Configurable/optional memory protection, debug, trace
  - uA device stand-by enabled with integrated sleep modes, ULL(Ultra-Low Leakage) libraries, state retention
- **Broad adoption within Microcontroller industry**





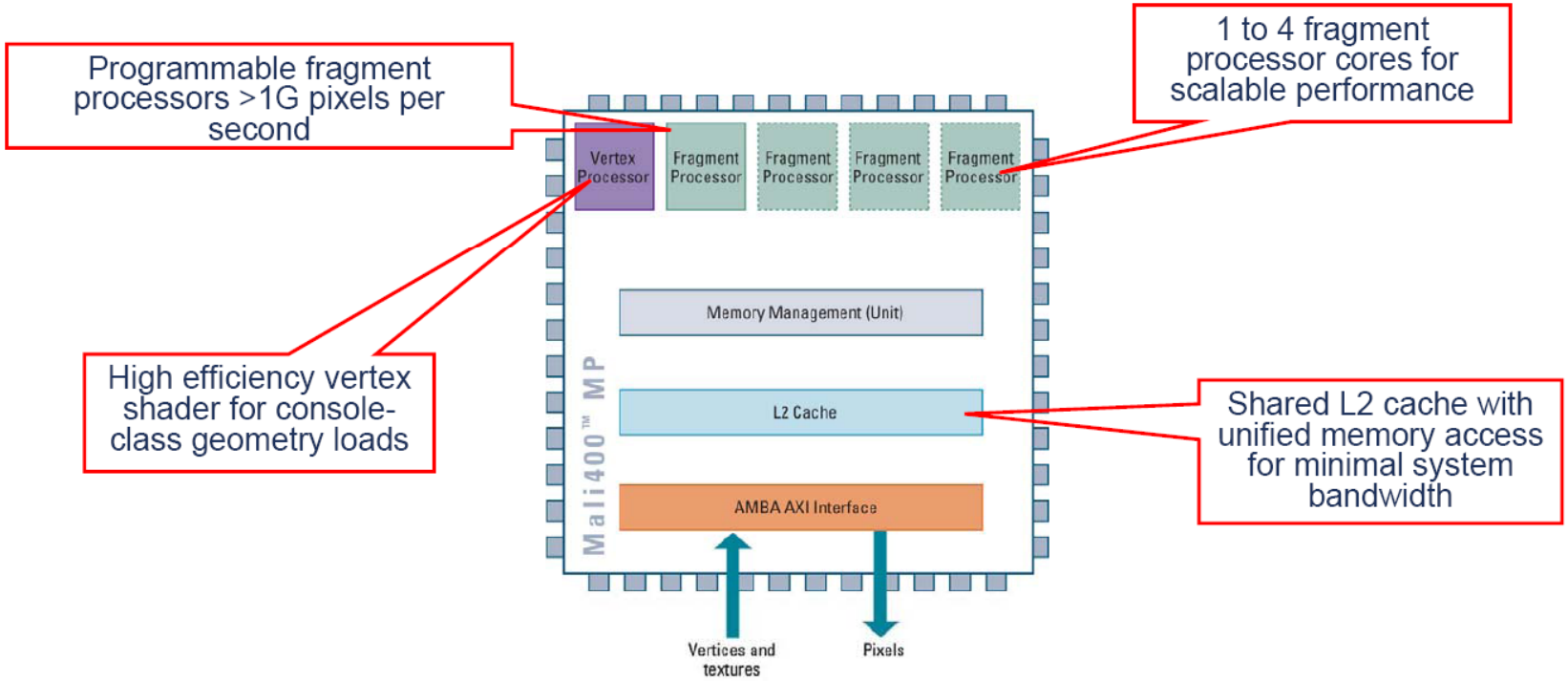
Cortex-M1

- **First ARM processor specifically optimized for FPGA**
  - High frequency, low-area soft processor for low-cost volume FPGA
  - Upwards compatible with Cortex-M3 onwards on ASIC/ASSP/MCU
  - Capable of up to 200MHz on fast FPGA device
  - Delivers up to 0.8 DMIPS/MHz efficiency from TCM
- **Designed for synthesis on multiple FPGA types**
  - Actel ProASIC3, Actel Igloo and Actel Fusion
  - Altera Cyclone-III, Altera Stratix-III
  - Xilinx Spartan-3, Xilinx Virtex-5



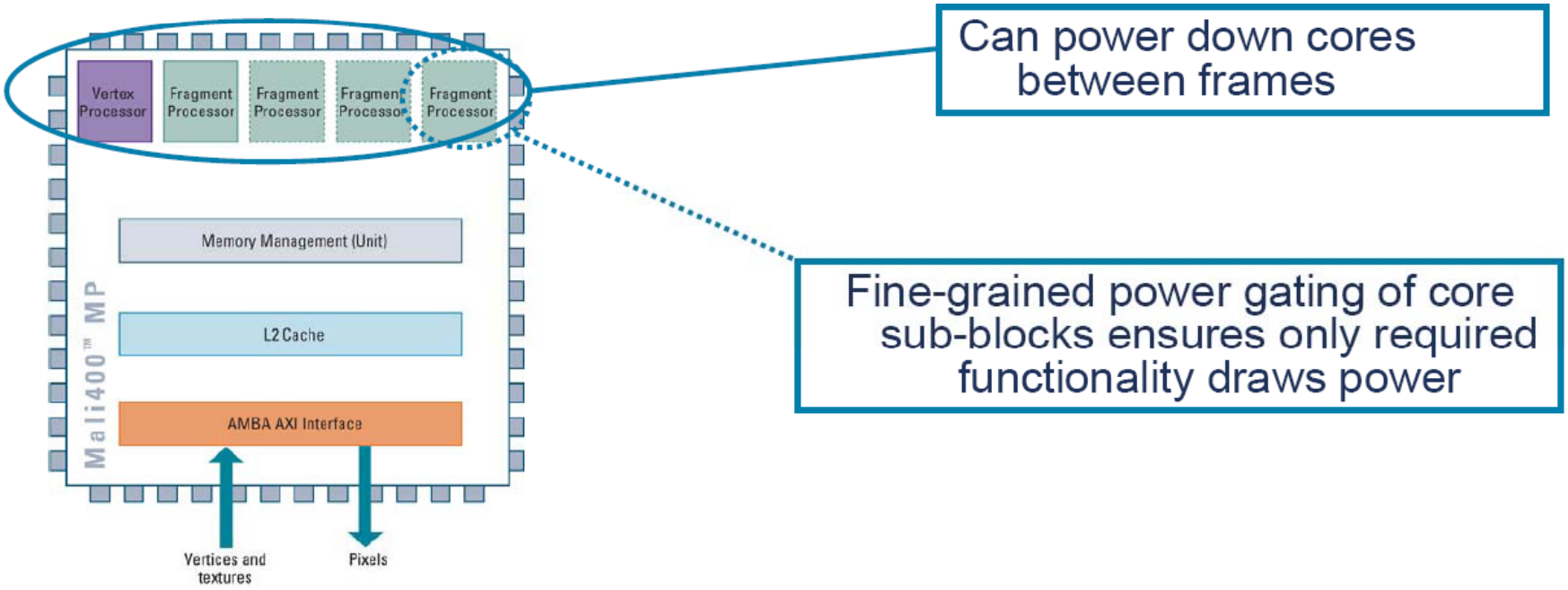
Mali-400 MP GPU

- **Pioneering Scalable Multicore Processor (MP) GPU**
  - Scalable architecture builds on ARM MPCore™ experience
    - Lowest memory bandwidth usage in the industry = lowest power
  - Performance scalability to satisfy future display requirements
    - …from WVGA feature phone to 1080p HDTV
  - Power and area scale to meet market needs

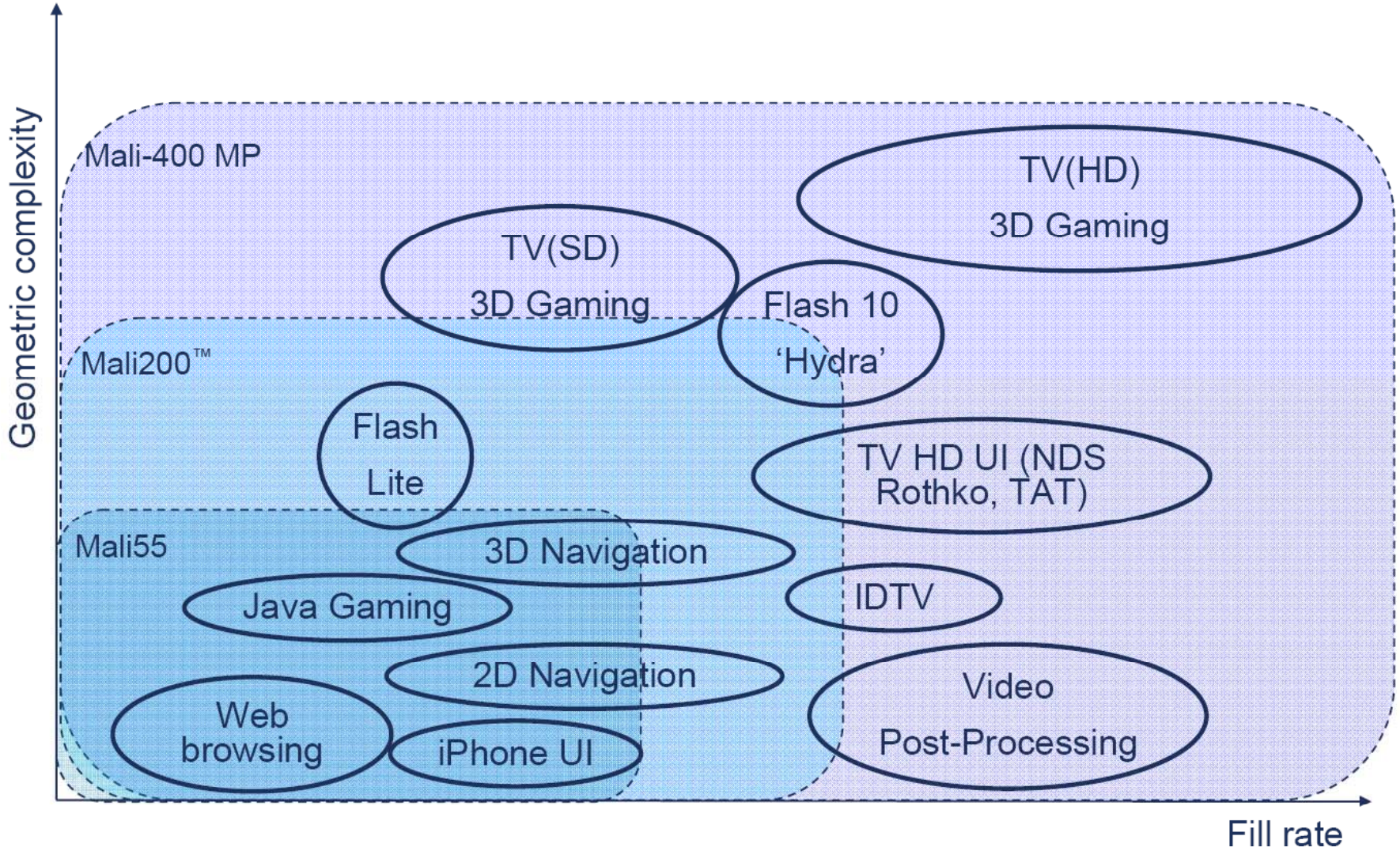


Mali-400 MP GPU

- **Power-efficient**
  - Memory bandwidth is the #1 power drain in graphics
  - Mali-400 MP GPU reduces memory bandwidth and lowers power
    - Combines best of immediate-mode and tile-based rendering
    - Shared L2 cache with unified memory access
    - Multiple levels of power gating supported



Mali-400 MP GPU



Agenda

- Introduction to ARM Ltd
- **Programmers Model**
- Instruction Sets
- System Design
- Development Tools

Data Sizes and Instruction Sets

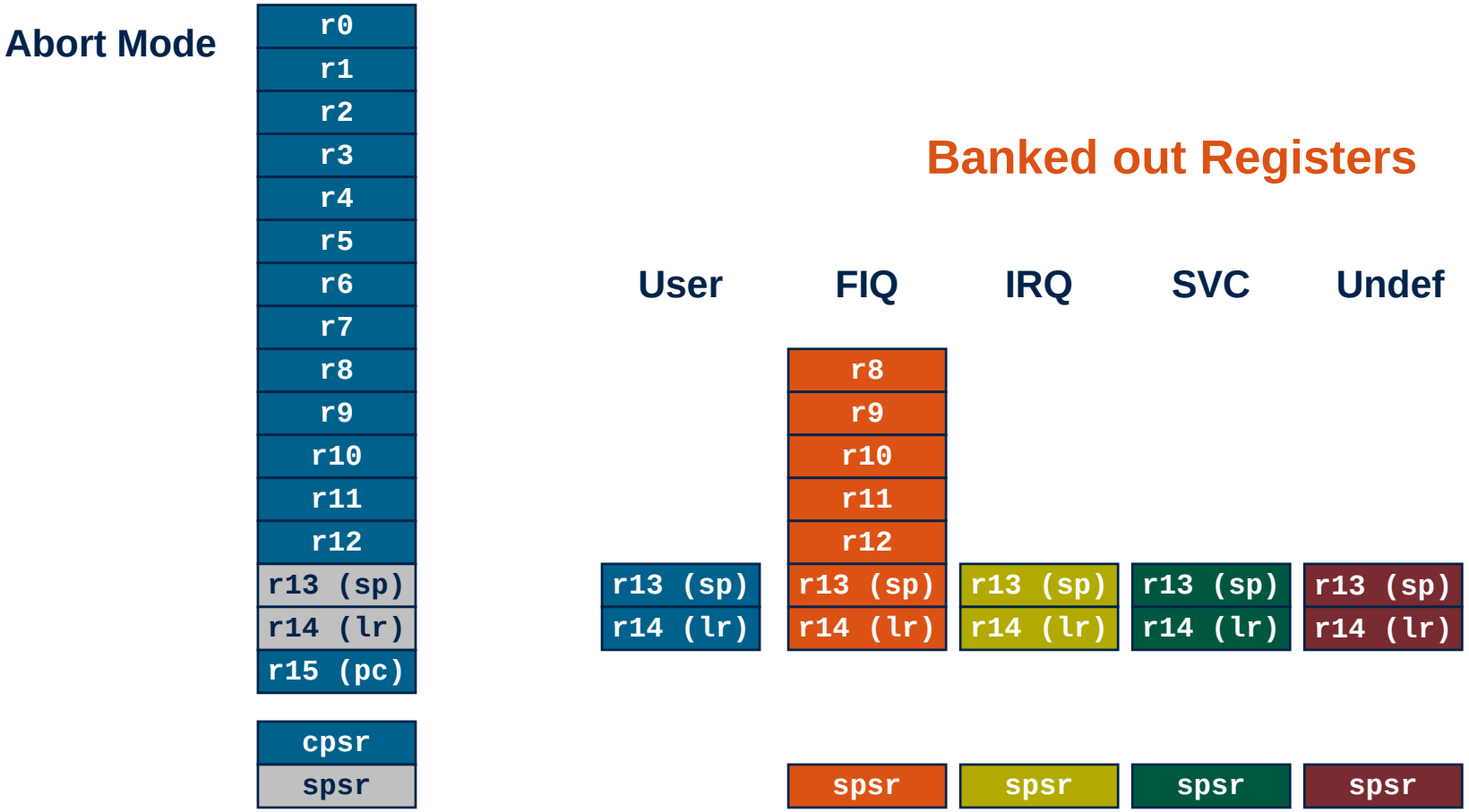
- **The ARM is a 32-bit architecture.**
- **When used in relation to the ARM:**
  - **Byte** means 8 bits
  - **Halfword** means 16 bits (two bytes)
  - **Word** means 32 bits (four bytes)
- **Most ARM's implement two instruction sets**
  - 32-bit ARM Instruction Set
  - 16-bit Thumb Instruction Set
- **Jazelle cores can also execute Java bytecode**

Processor Modes

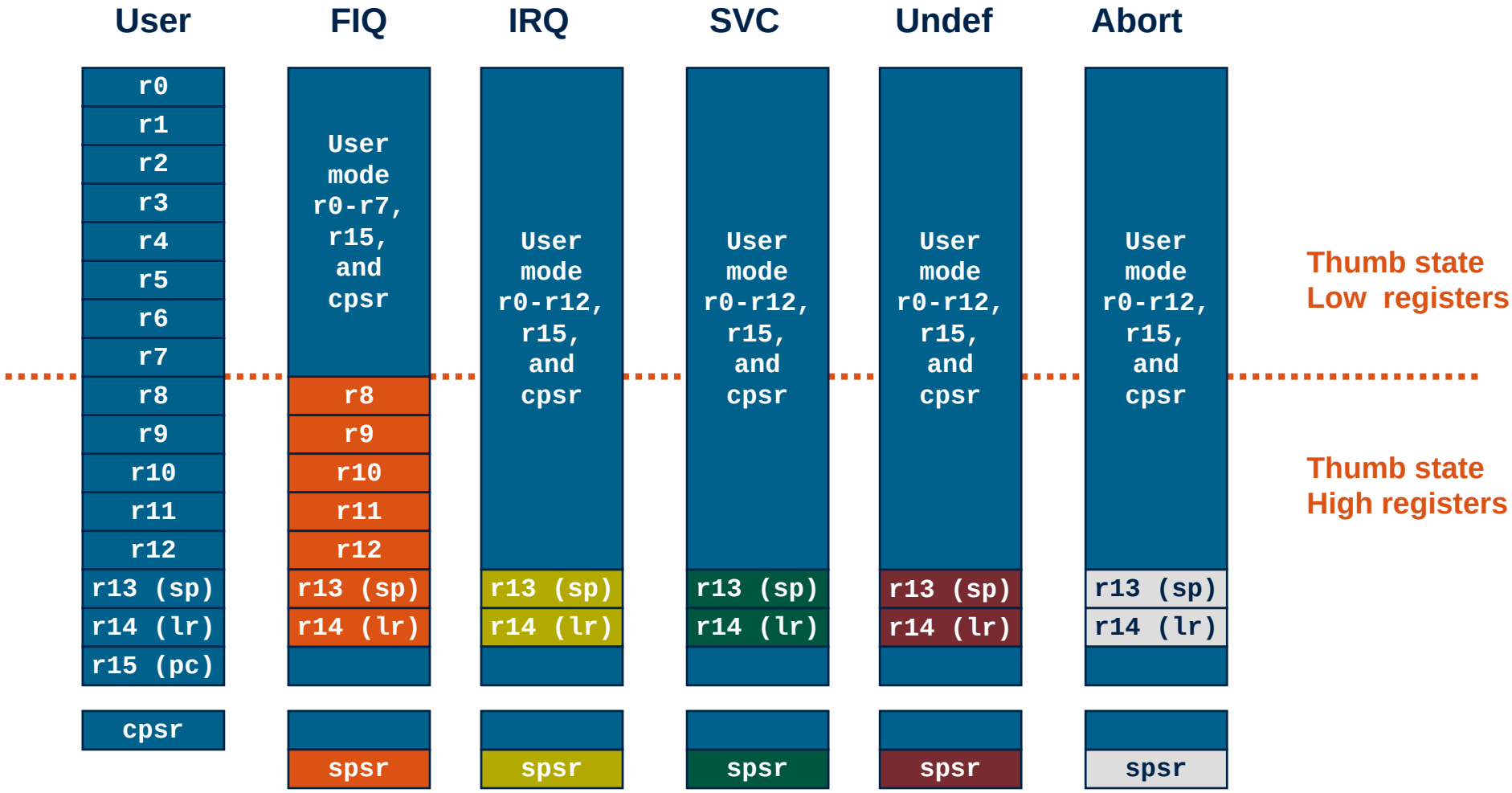
- **The ARM has seven basic operating modes:**
  - **User** : unprivileged mode under which most tasks run
  - **FIQ** : entered when a high priority (fast) interrupt is raised
  - **IRQ** : entered when a low priority (normal) interrupt is raised
  - **Supervisor** : entered on reset and when a Software Interrupt instruction is executed
  - **Abort** : used to handle memory access violations
  - **Undef** : used to handle undefined instructions
  - **System** : privileged mode using the same registers as user mode

The ARM Register Set

Current Visible Registers



Register Organization Summary



Note: System mode uses the User mode register set

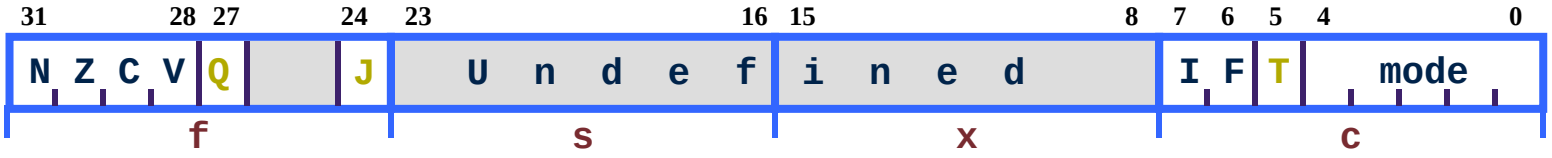
The Registers

- **ARM has 37 registers all of which are 32-bits long.**
  - 1 dedicated program counter
  - 1 dedicated current program status register
  - 5 dedicated saved program status registers
  - 30 general purpose registers

- **The current processor mode governs which of several banks is accessible. 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 spsr (saved program status register)

Program Status Registers



- **Condition code flags**
  - N = Negative result from ALU
  - Z = Zero result from ALU
  - C = ALU operation Carried out
  - V = ALU operation oVerflowed
- **Sticky Overflow flag - Q flag**
  - Architecture 5TE/J only
  - Indicates if saturation has occurred
- **J bit**
  - Architecture 5TEJ 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
  - T = 0: Processor in ARM state
  - T = 1: Processor in Thumb state
- **Mode bits**
  - Specify the processor mode

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

Program Counter (r15)

- **When the processor is executing in ARM state:**
  - All instructions are 32 bits wide
  - All instructions must be word aligned
  - Therefore the **pc** value is stored in bits [31:2] with bits [1:0] undefined (as instruction cannot be halfword or byte aligned).

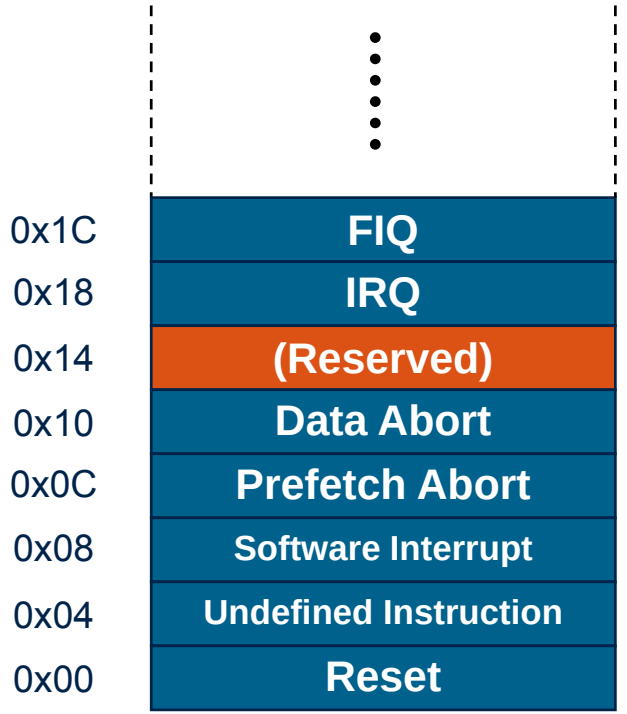
- **When the processor is executing in Thumb state:**
  - All instructions are 16 bits wide
  - All instructions must be halfword aligned
  - Therefore the **pc** value is stored in bits [31:1] with bit [0] undefined (as instruction cannot be byte aligned).

- **When the processor is executing in Jazelle state:**
  - All instructions are 8 bits wide
  - Processor performs a word access to read 4 instructions at once

Exception Handling

- **When an exception occurs, the ARM:**
  - Copies CPSR into SPSR_<mode>
  - Sets appropriate CPSR bits
    - Change to ARM state
    - Change to exception mode
    - Disable interrupts (if appropriate)
  - Stores the return address in LR_<mode>
  - Sets PC to vector address
- **To return, exception handler needs to:**
  - Restore CPSR from SPSR_<mode>
  - Restore PC from LR_<mode>

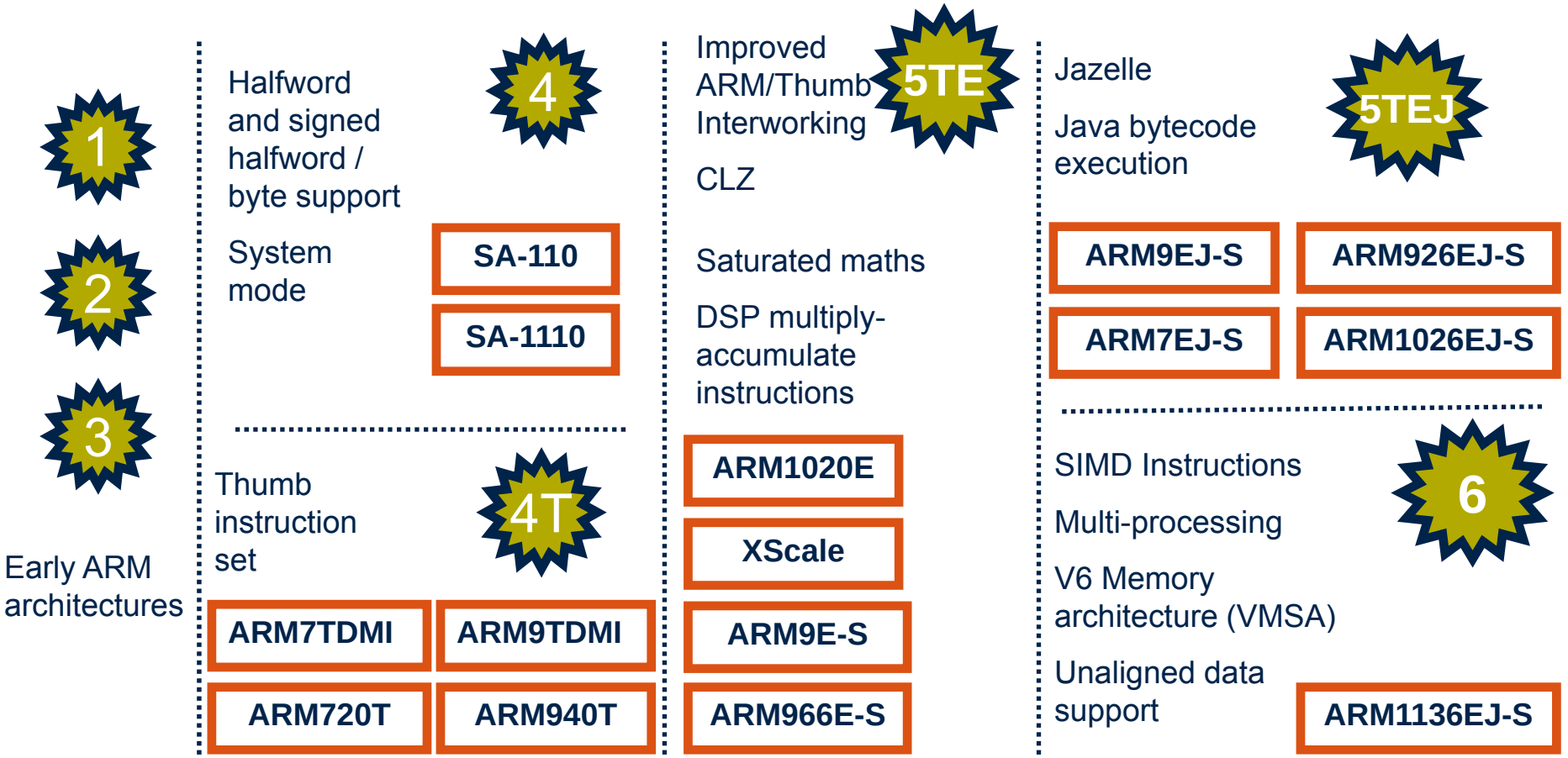
This can only be done in ARM state.



Vector Table

Vector table can be at **`0xFFFF0000`** on ARM720T and on ARM9/10 family devices

Development of the ARM Architecture



Agenda

- Introduction to ARM Ltd
- Programmers Model
- **Instruction Sets**
- System Design
- Development Tools

Conditional Execution and Flags

- **ARM instructions can be made to execute conditionally by postfixing them with the appropriate condition code field.**
  - This improves code density *and* performance by reducing the number of forward branch instructions.

```
  CMP   r3,#0
   BEQ   skip
   ADD   r0,r1,r2
skip
```

```
CMP   r3,#0
 ADDNE r0,r1,r2
```

- **By default, data processing instructions do not affect the condition code flags but the flags can be optionally set by using "S". CMP does not need "S".**

```
loop
  …
  SUBS r1,r1,#1
  BNE loop
```

decrement r1 and set flags

if Z flag clear then branch

Setting the condition codes

- Any data processing instruction can set the condition codes if the programmers wish it to

64-bit addition

```
ADDS   R2, R2, R0
ADC    R3, R3, R1
```

Condition Codes

- **The possible condition codes are listed below:**
  - Note AL is the default and does not need to be specified

Suffix	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	

Examples of conditional execution

- Use a sequence of several conditional instructions

```
if (a==0) func(1);
    CMP     r0,#0
    MOVEQ   r0,#1
    BLEQ    func
```

- Set the flags, then use various condition codes

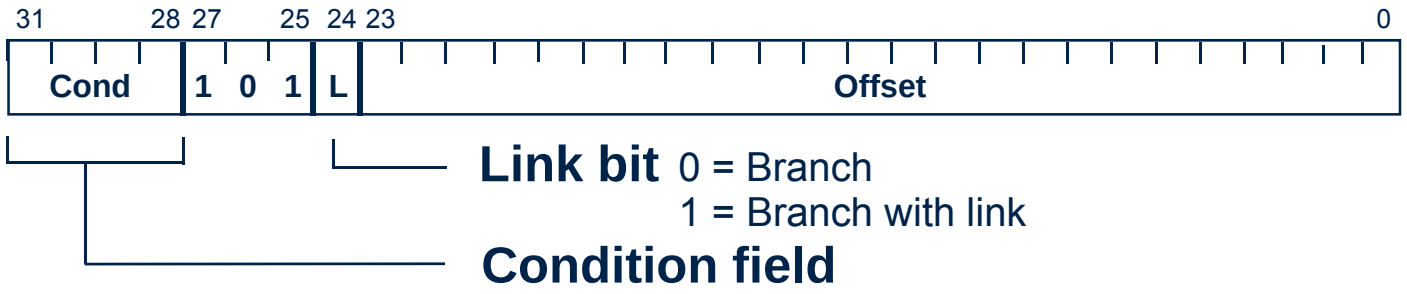
```
if (a==0) x=0;
if (a>0)  x=1;
    CMP     r0,#0
    MOVEQ   r1,#0
    MOVGT   r1,#1
```

- Use conditional compare instructions

```
if (a==4 || a==10) x=0;
    CMP     r0,#4
    CMPNE   r0,#10
    MOVEQ   r1,#0
```

Branch instructions

- **Branch :** `B{<cond>} label`
- **Branch with Link :** `BL{<cond>} subroutine_label`



- **The processor core shifts the offset field left by 2 positions, sign-extends it and adds it to the PC**
  - ± 32 Mbyte range
  - How to perform longer branches?

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 instructions only work on registers, NOT memory.**

- **Syntax:**

```
<Operation>{<cond>}{S} Rd, Rn, Operand2
```

  - Comparisons set flags only - they do not specify Rd
  - Data movement does not specify Rn

- **Second operand is sent to the ALU via barrel shifter.**

Data processing Instructions

Arithmetic Operations

ADD r0, r1, r2	r0 := r1 + r2
ADC r0, r1, r2	r0 := r1 + r2 + C
SUB r0, r1, r2	r0 := r1 - r2
SBC r0, r1, r2	r0 := r1 - r2 + C - 1
RSB r0, r1, r2	r0 := r2 – r1
RSC r0, r1, r2	r0 := r2 – r1 + C - 1

Bit-wise Logical Operations

AND r0, r1, r2	r0 := r1 and r2
ORR r0, r1, r2	r0 := r1 or r2
EOR r0, r1, r2	r0 := r1 xor r2
BIC r0, r1, r2	r0 := r1 and (not) r2

Register Movement

MOV r0, r2	r0 := r2
MVN r0, r2	r0 := not r2

Comparison Operations

CMP r1, r2	set cc on r1 - r2
CMN r1, r2	set cc on r1 + r2
TST r1, r2	set cc on r1 and r2
TEQ r1, r2	set cc on r1 xor r2

The Barrel Shifter

LSL : Logical Left Shift

Multiplication by a power of 2

ASR: Arithmetic Right Shift

Division by a power of 2,
preserving the sign bit

LSR : Logical Shift Right

Division by a power of 2

ROR: Rotate Right

Bit rotate with wrap around
from LSB to MSB

RRX: Rotate Right Extended

Single bit rotate with wrap around
from CF to MSB

Shifted register operands

Mnemonic	Description	Shift	Result
LSL	logical shift left	*x*LSL *y*	$x \ll y$
LSR	logical shift right	*x*LSR *y*	$(\text{unsigned})x \gg y$
ASR	arithmetic right shift	*x*ASR *y*	$(\text{signed})x \gg y$
ROR	rotate right	*x*ROR *y*	$((\text{unsigned})x \gg y) \mid (x \ll (32 - y))$
RRX	rotate right extended	*x*RRX	$(c \text{ flag} \ll 31) \mid ((\text{unsigned})x \gg 1)$

Logical shift left


```
MOV   R0, R2, LSL #2 @ R0:=R2<<2
                     @ R2 unchanged
Example: 0…0 0011 0000
Before R2=0x00000030
After  R0=0x000000C0
       R2=0x00000030
```

Logical shift right


```
MOV   R0, R2, LSR #2 @ R0:=R2>>2
                     @ R2 unchanged
Example: 0…0 0011 0000
Before R2=0x00000030
After  R0=0x0000000C
       R2=0x00000030
```

Arithmetic shift right


```
MOV   R0, R2, ASR #2 @ R0:=R2>>2
                     @ R2 unchanged
Example: 1010 0…0 0011 0000
Before R2=0xA0000030
After  R0=0xE800000C
       R2=0xA0000030
```

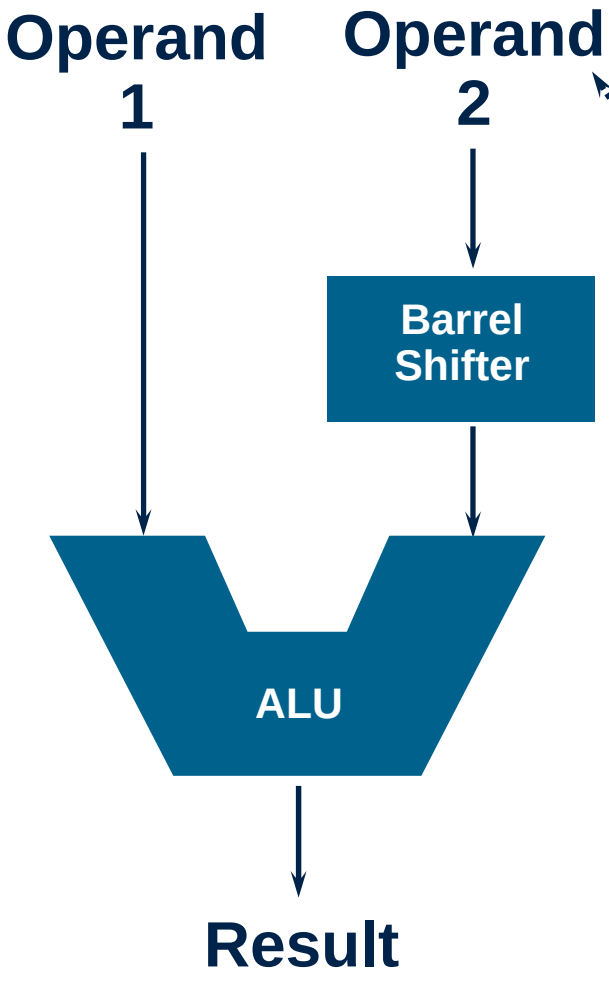
Rotate right


```
MOV  R0, R2, ROR #2 @ R0:=R2 rotate
                    @ R2 unchanged
Example: 0…0 0011 0001
Before R2=0x00000031
After  R0=0x4000000C
       R2=0x00000031
```

Rotate right extended


```
MOV  R0, R2, RRX      @ R0:=R2 rotate
                      @ R2 unchanged
Example: 0…0 0011 0001
Before R2=0x00000031, C=1
After  R0=0x80000018, C=1
       R2=0x00000031
```

Using the Barrel Shifter: The Second Operand



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

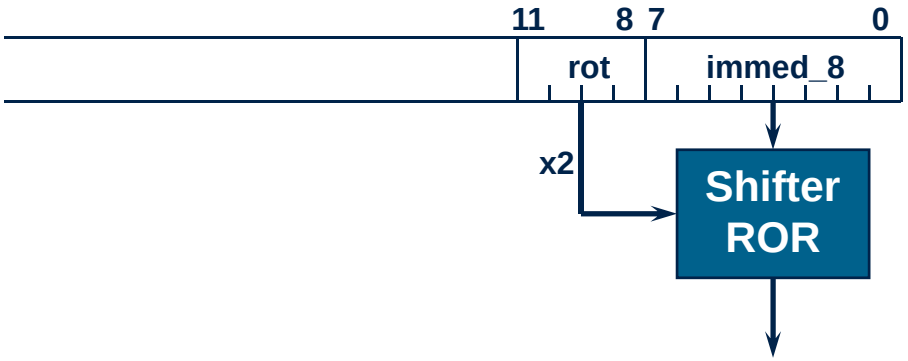
Immediate value

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

```
ADD     r0, r1, r2
ADD     r0, r1, r2, LSL#7
ADD     r0, r1, r2, LSL r3
ADD     r0, r1, #0x4E
```

Immediate constants (1)

- **No ARM instruction can contain a 32 bit immediate constant**
  - All ARM instructions are fixed as 32 bits long
- **The data processing instruction format has 12 bits available for operand2**

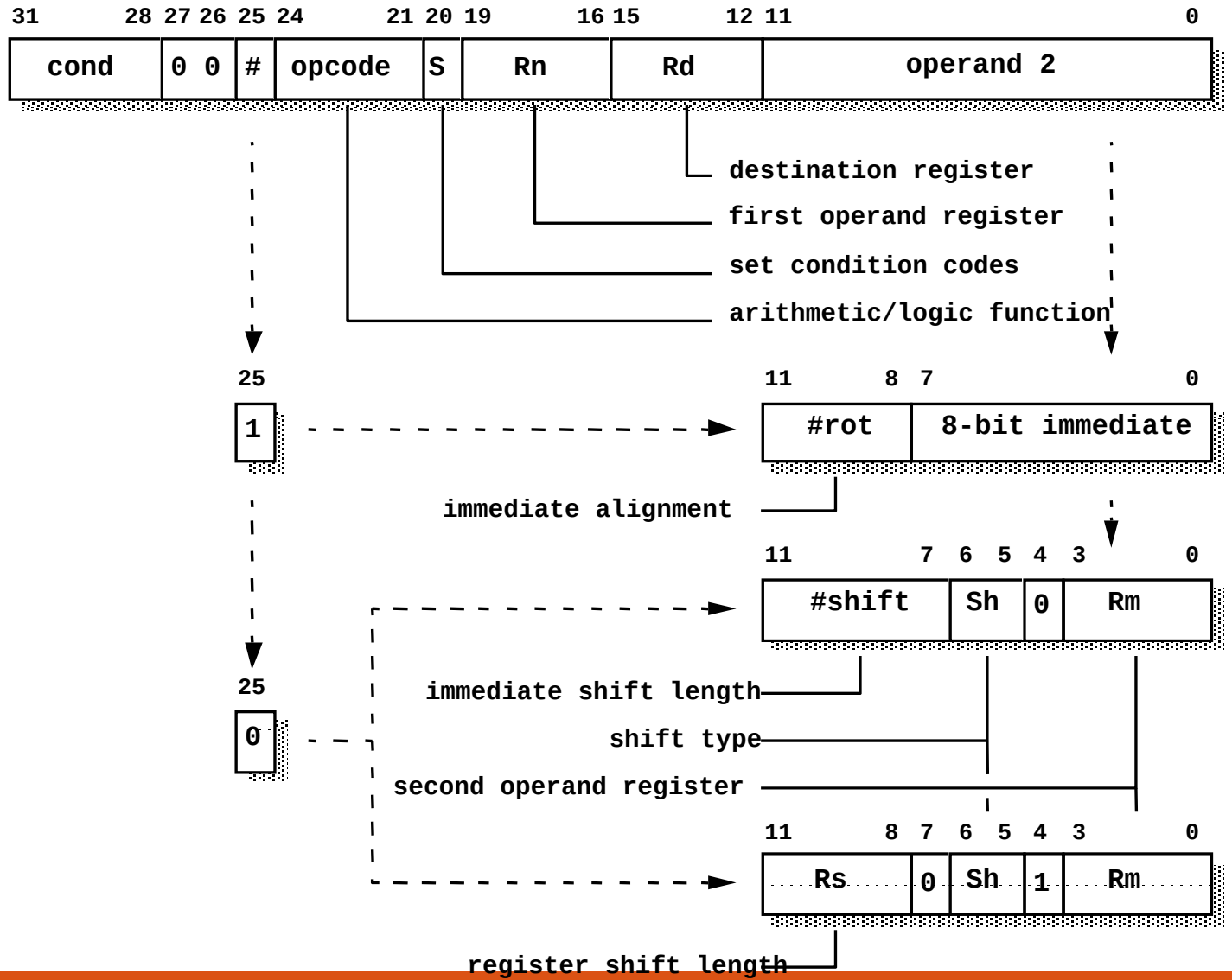


Quick Quiz:

```
0xe3a004ff
MOV r0, #???
```

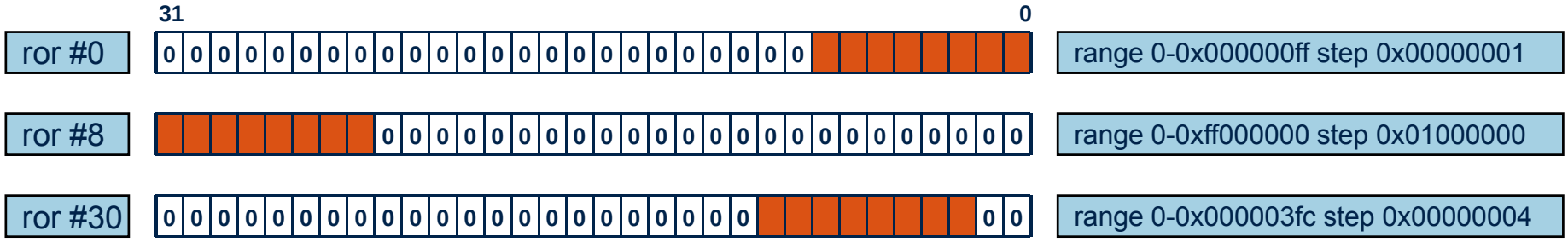
- **4 bit rotate value (0-15) is multiplied by two to give range 0-30 in steps of 2**
- **Rule to remember is "8-bits shifted by an even number of bit positions".**

Encoding data processing instructions



Immediate constants (2)

- **Examples:**



- **The assembler converts immediate values to the rotate form:**
  - `MOV r0,#4096 ; uses 0x40 ror 26`
  - `ADD r1,r2,#0xFF0000 ; uses 0xFF ror 16`

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

- **Values that cannot be generated in this way will cause an error.**

Loading 32 bit constants

- **To allow larger constants to be loaded, the assembler offers a pseudo-instruction:**
  - `LDR rd, =const`
- **This will either:**
  - Produce a **`MOV`** or **`MVN`** instruction to generate the value (if possible).

  or
  - Generate a **LDR** instruction with a PC-relative address to read the constant from a *literal pool* (Constant data area embedded in the code).
- **For example**
  - `LDR r0,=0xFF` => `MOV r0,#0xFF`
  - `LDR r0,=0x55555555` => `LDR r0,[PC,#Imm12]`

```
...
...
DCD 0x55555555
```

- **This is the recommended way of loading constants into a register**

Multiply

- **Syntax:**
  - MUL{<cond>}{S} Rd, Rm, Rs — Rd = Rm * Rs
  - MLA{<cond>}{S} Rd,Rm,Rs,Rn — Rd = (Rm * Rs) + Rn
  - [U|S]MULL{<cond>}{S} RdLo, RdHi, Rm, Rs — RdHi,RdLo := Rm*Rs
  - [U|S]MLAL{<cond>}{S} RdLo, RdHi, Rm, Rs — RdHi,RdLo := (Rm*Rs)+RdHi,RdLo

- **Cycle time**
  - Basic MUL instruction
    - 2-5 cycles on ARM7TDMI
    - 1-3 cycles on StrongARM/XScale
    - 2 cycles on ARM9E/ARM102xE
  - +1 cycle for ARM9TDMI (over ARM7TDMI)
  - +1 cycle for accumulate (not on 9E though result delay is one cycle longer)
  - +1 cycle for “long”

- **Above are “general rules” - refer to the TRM for the core you are using for the exact details**

Multiplication

- Multiply-accumulate (2D array indexing)

```
MLA   R4, R3, R2, R1   @ R4 = R3xR2+R1
```

- Multiply with a constant can often be more efficiently implemented using shifted register operand

```
MOV   R1, #35
MUL   R2, R0, R1
```

or

```
ADD   R0, R0, R0, LSL #2   @ R0'=5xR0
RSB   R2, R0, R0, LSL #3   @ R2 =7xR0'
```

Single register data transfer

LDR	**STR**	Word
LDRB	**STRB**	Byte
LDRH	**STRH**	Halfword
LDRSB		Signed byte load
LDRSH		Signed halfword load

- **Memory system must support all access sizes**

- **Syntax:**
  - **LDR**{<cond>}{<size>} Rd, <address>
  - **STR**{<cond>}{<size>} Rd, <address>

  e.g. **LDREQB**

Address accessed

- **Address accessed by LDR/STR is specified by a base register plus an offset**
- **For word and unsigned byte accesses, offset can be**
  - An unsigned 12-bit immediate value (ie 0 - 4095 bytes).

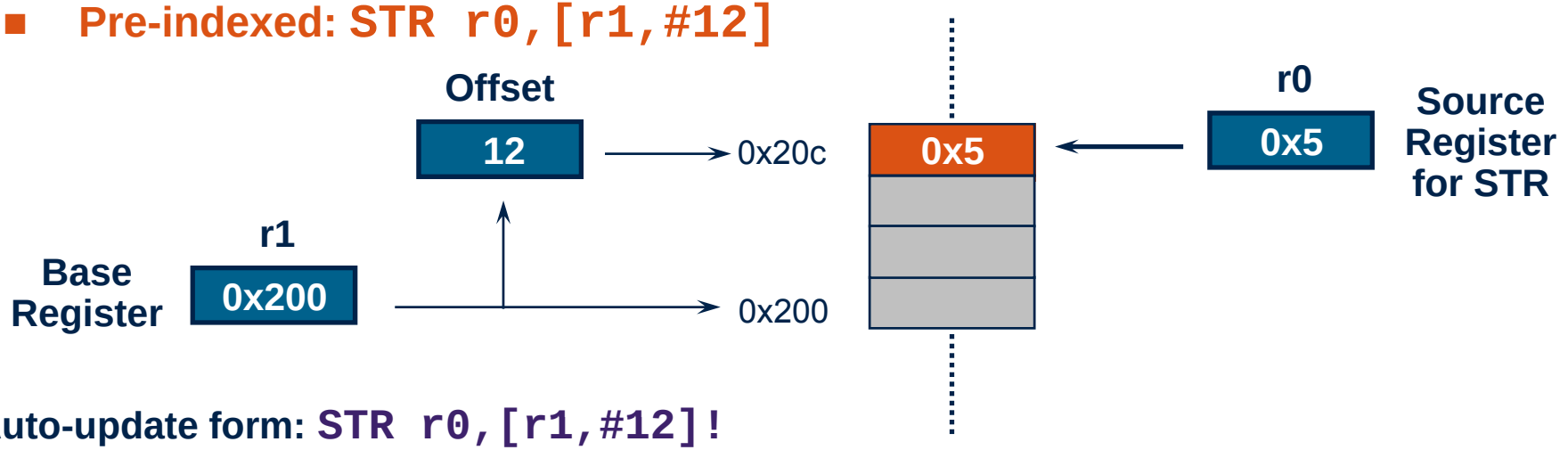
    ```
    LDR r0,[r1,#8]
    ```
  - A register, optionally shifted by an immediate value

    ```
    LDR r0,[r1,r2]
    LDR r0,[r1,r2,LSL#2]
    ```
- **This can be either added or subtracted from the base register:**

  ```
  LDR r0,[r1,#-8]
  LDR r0,[r1,-r2]
  LDR r0,[r1,-r2,LSL#2]
  ```
- **For halfword and signed halfword / byte, offset can be:**
  - An unsigned 8 bit immediate value (ie 0-255 bytes).
  - A register (unshifted).
- **Choice of *pre-indexed* or *post-indexed* addressing**

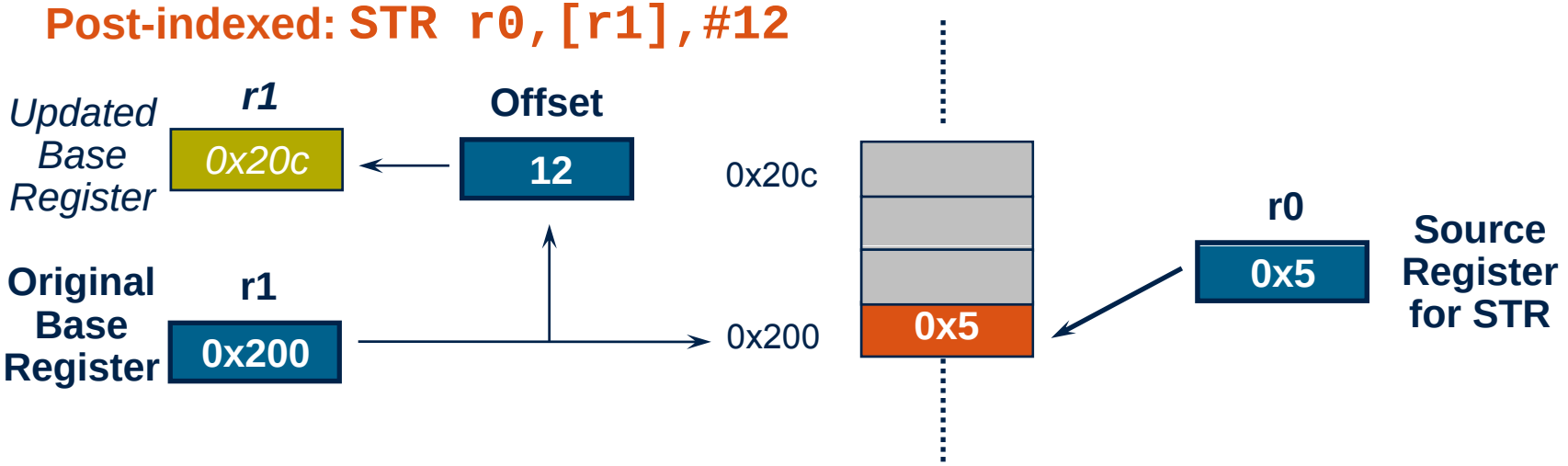
Pre or Post Indexed Addressing?

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



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

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

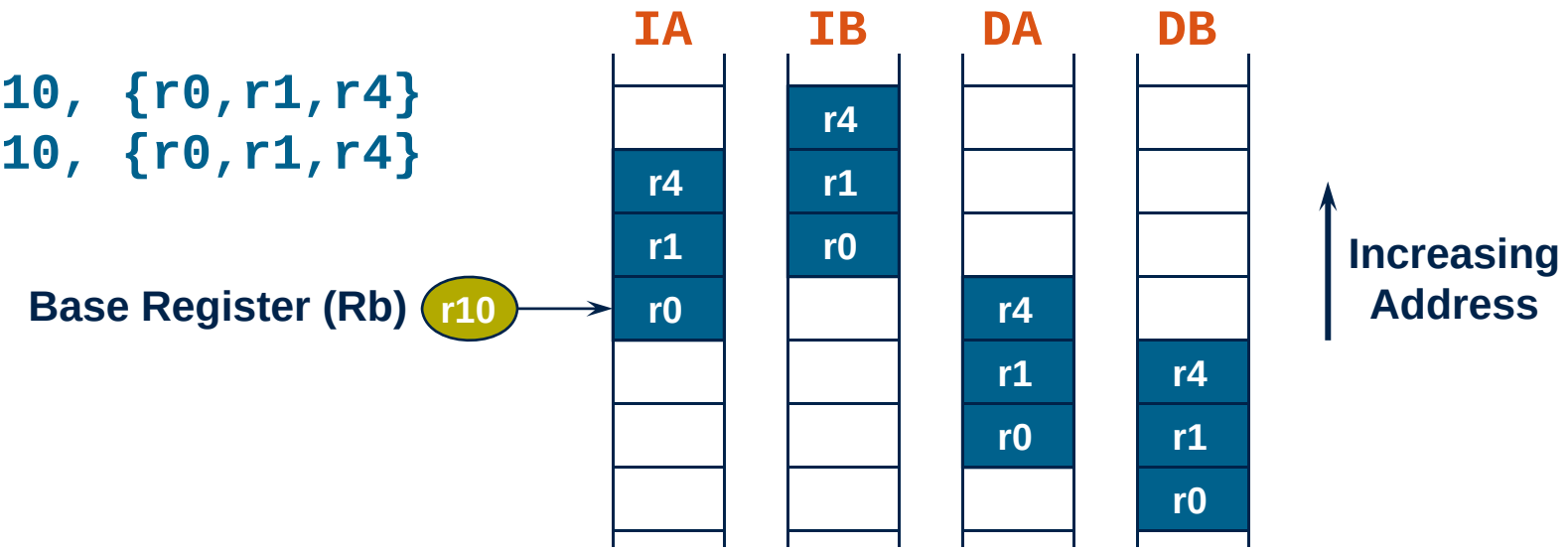


LDM / STM operation

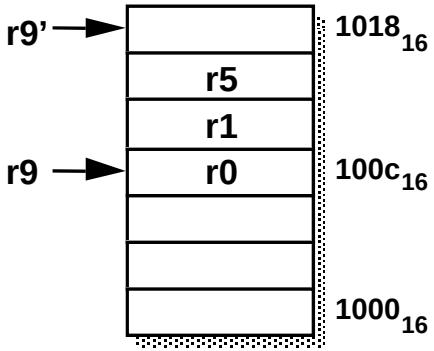
- **Syntax:**
  `<LDM|STM>`{<cond>}<addressing_mode> Rb{!}, <register list>
- **4 addressing modes:**

`LDMIA / STMIA`	increment after
`LDMIB / STMIB`	increment before
`LDMDA / STMDA`	decrement after
`LDMDB / STMDB`	decrement before

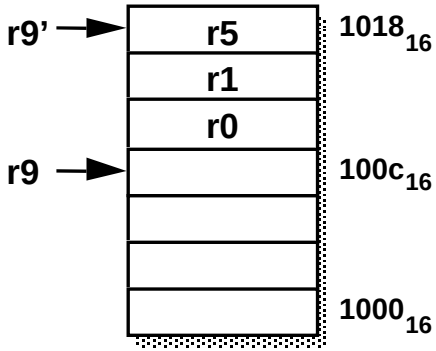
```
LDMxx r10, {r0,r1,r4}
STMxx r10, {r0,r1,r4}
```



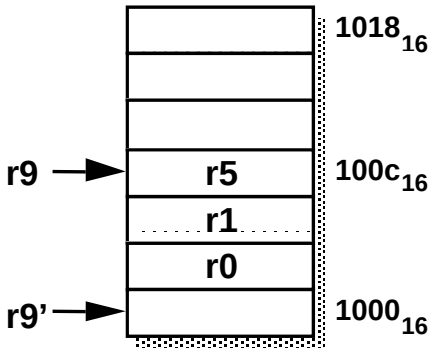
LDM / STM operation



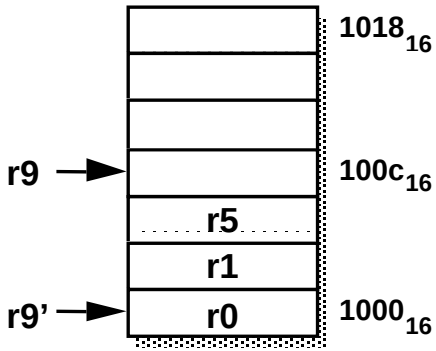
`STMIA r9!, {r0,r1,r5}`



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



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

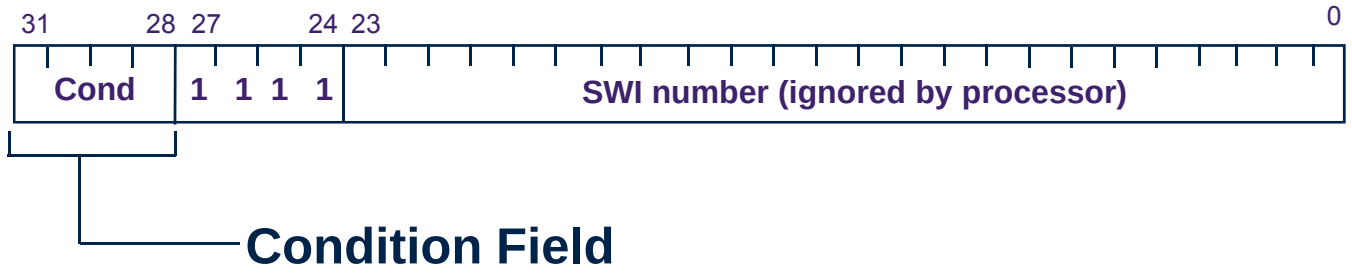


`STMDB r9!, {r0,r1,r5}`

The mapping between the stack and block copy views

		Ascending		Descending	
		Full	Empty	Full	Empty
Increment	Before	STMIB STMFA			LDMIB LDMED
	After		STMIA STMEA	LDMIA LDMFD	
Decrement	Before		LDMDB LDMEA	STMDB STMFD	
	After	LDMDA LDMFA			STMDA STMED

Software Interrupt (SWI)



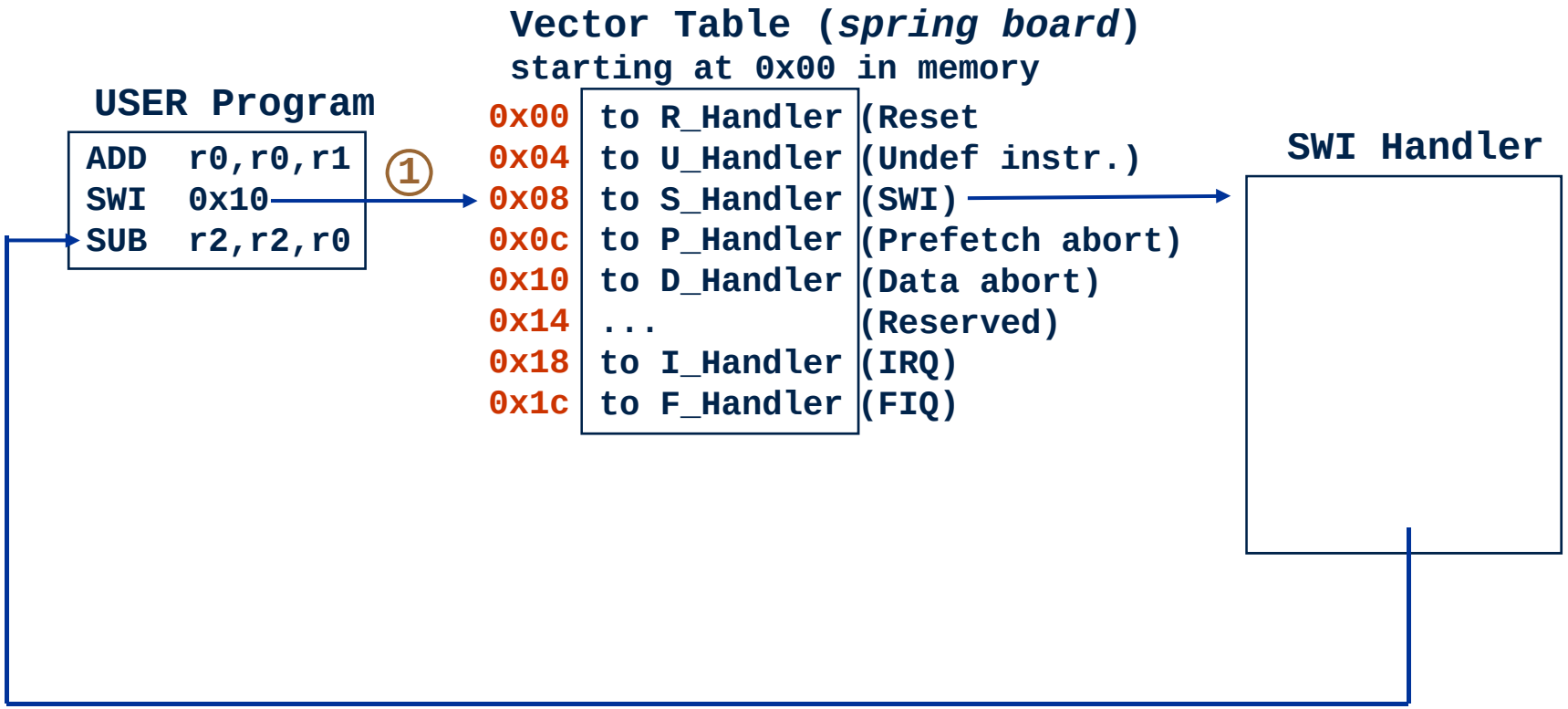
- Causes an exception trap to the SWI hardware vector
- The SWI handler can examine the SWI number to decide what operation has been requested.
- By using the SWI mechanism, an operating system can implement a set of privileged operations which applications running in user mode can request.
- Syntax:
  - `SWI{<cond>} <SWI number>`

What do SWIs do?

- **SWIs (often called software traps) allow a user program to "call" the OS -- that is, SWIs are how system calls are implemented.**
- **When SWIs execute, the processor changes modes (from User to Supervisor mode on the ARM) and disables interrupts.**
- **Types of SWIs in ARM Angel (`axd or armsd`)**
  - `SWI_WriteC(SWI 0)` Write a byte to the debug channel
  - `SWI_Write0(SWI 2)` Write the null-terminated string to debug channel
  - `SWI_ReadC(SWI 4)` Read a byte from the debug channel
  - `SWI_Exit(SWI 0x11)` Halt emulation - this is how a program exits
  - `SWI_EnterOS(SWI 0x16)` Put the processor in supervisor mode
  - `SWI_Clock(SWI 0x61)` Return the number of centi-seconds
  - `SWI_Time(SWI 0x63)` Return the number of secs since Jan. 1, 1970

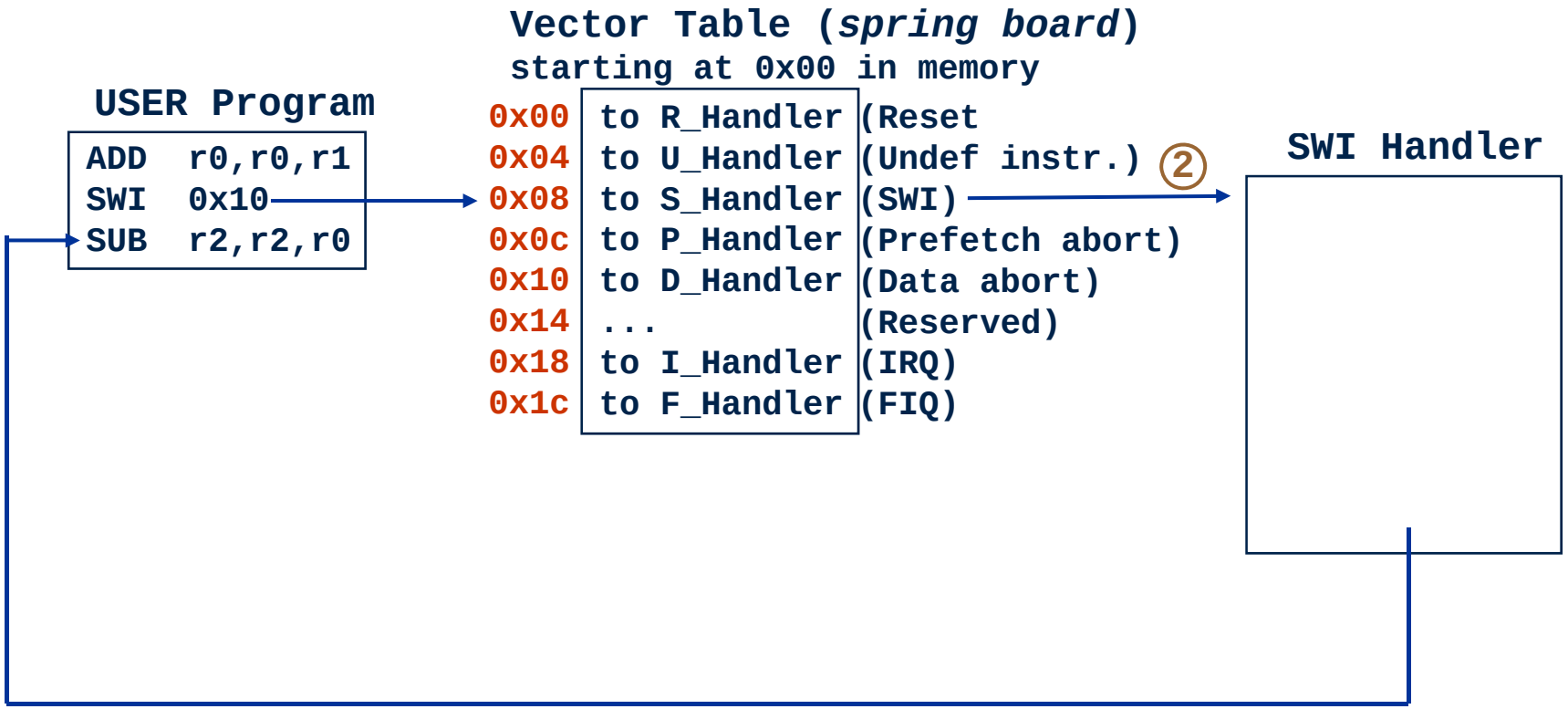
What Happens on an SWI? (1)

- The ARM architecture defines a Vector Table indexed by exception type
- One SWI, CPU does the following: PC ①<-- 0x08
- Also, sets LR_svc, SPSR_svc, CPSR (supervisor mode, no IRQ)



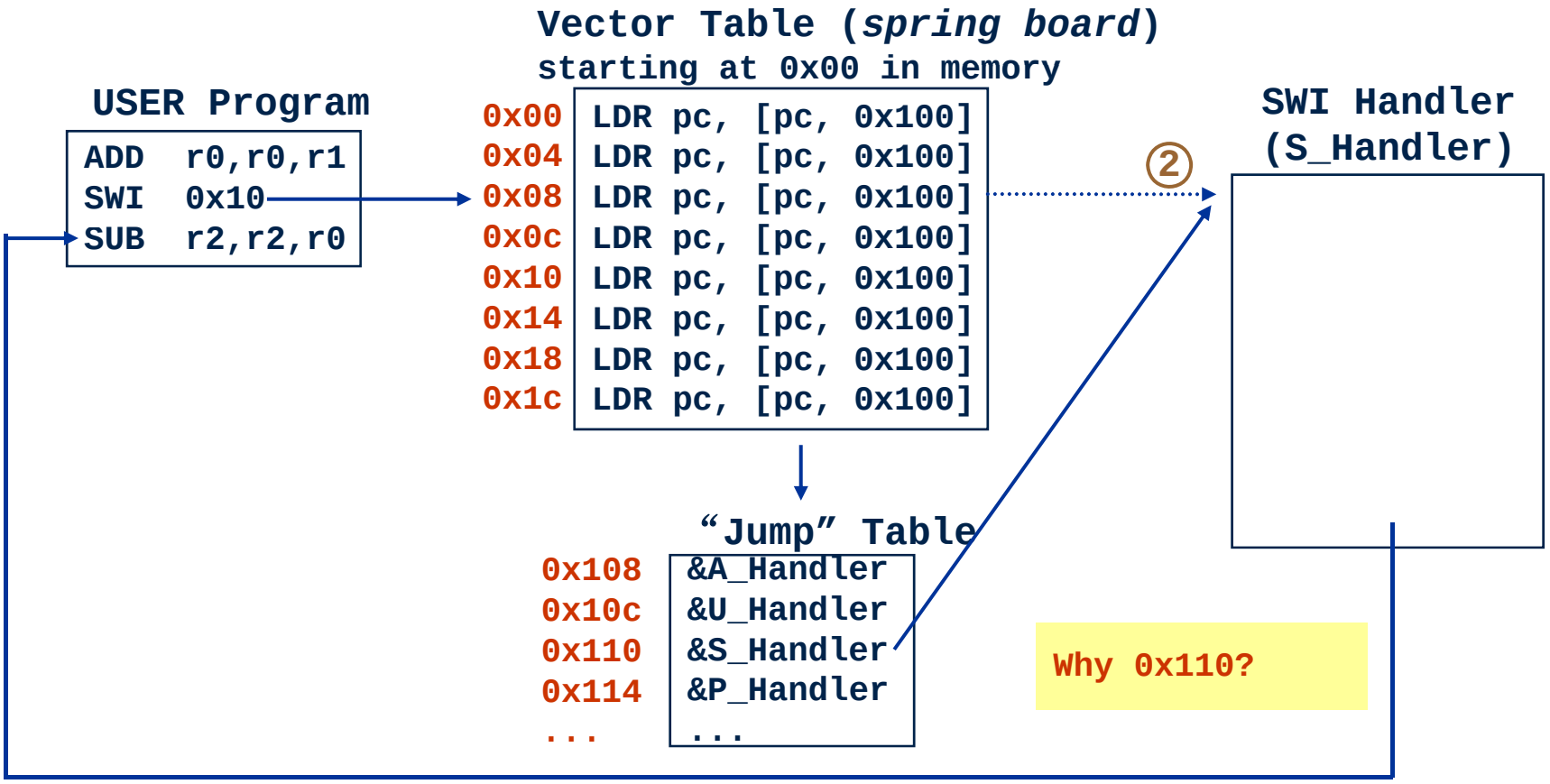
What Happens on an SWI? (2)

- Not enough space in the table (only one instruction per entry) to hold all of the code for the SWI handler function
- This *one* instruction must transfer control to appropriate SWI Handler ②
- Several options are presented in the next slide



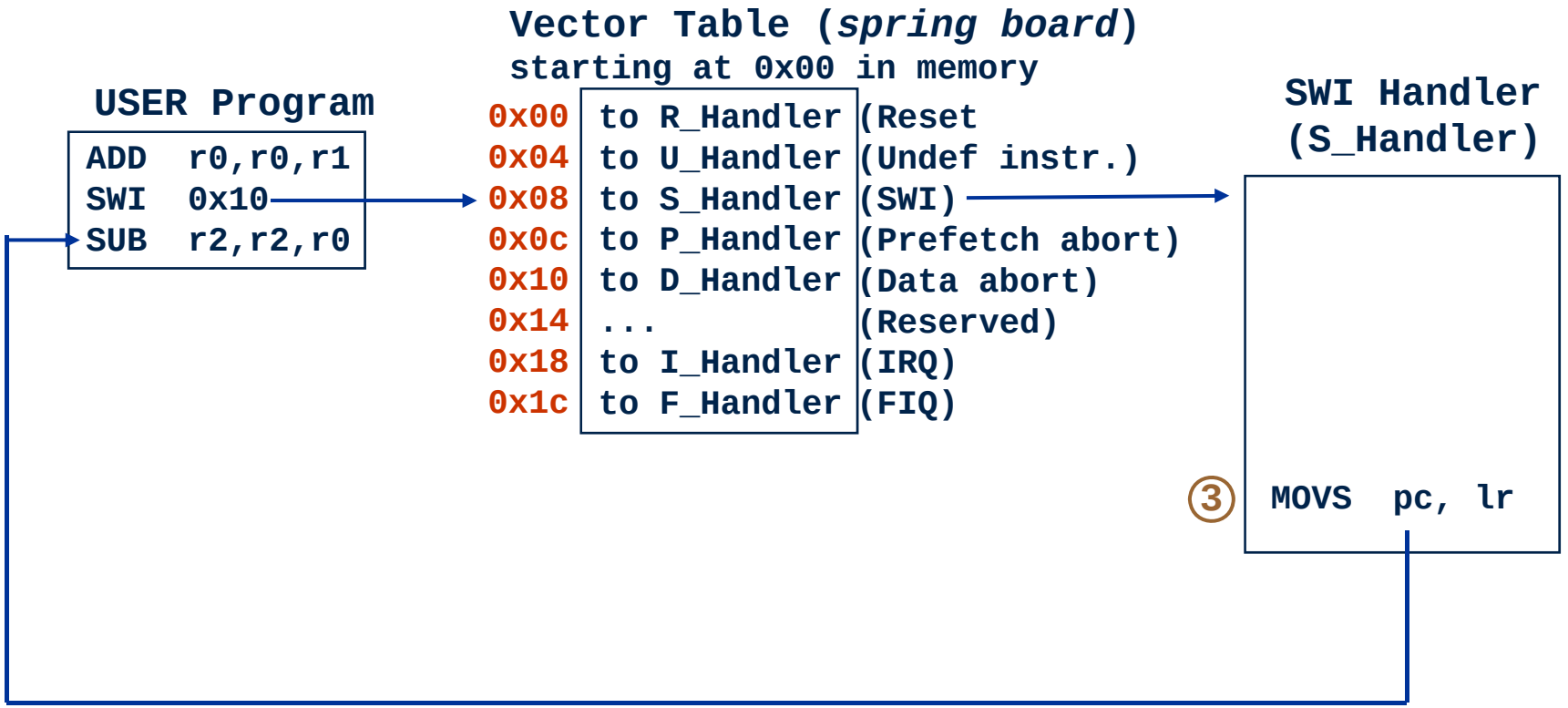
“Vectoring” Exceptions to Handlers

- Option of choice: Load PC from jump table (shown below)
- Another option: Direct branch (limited range)



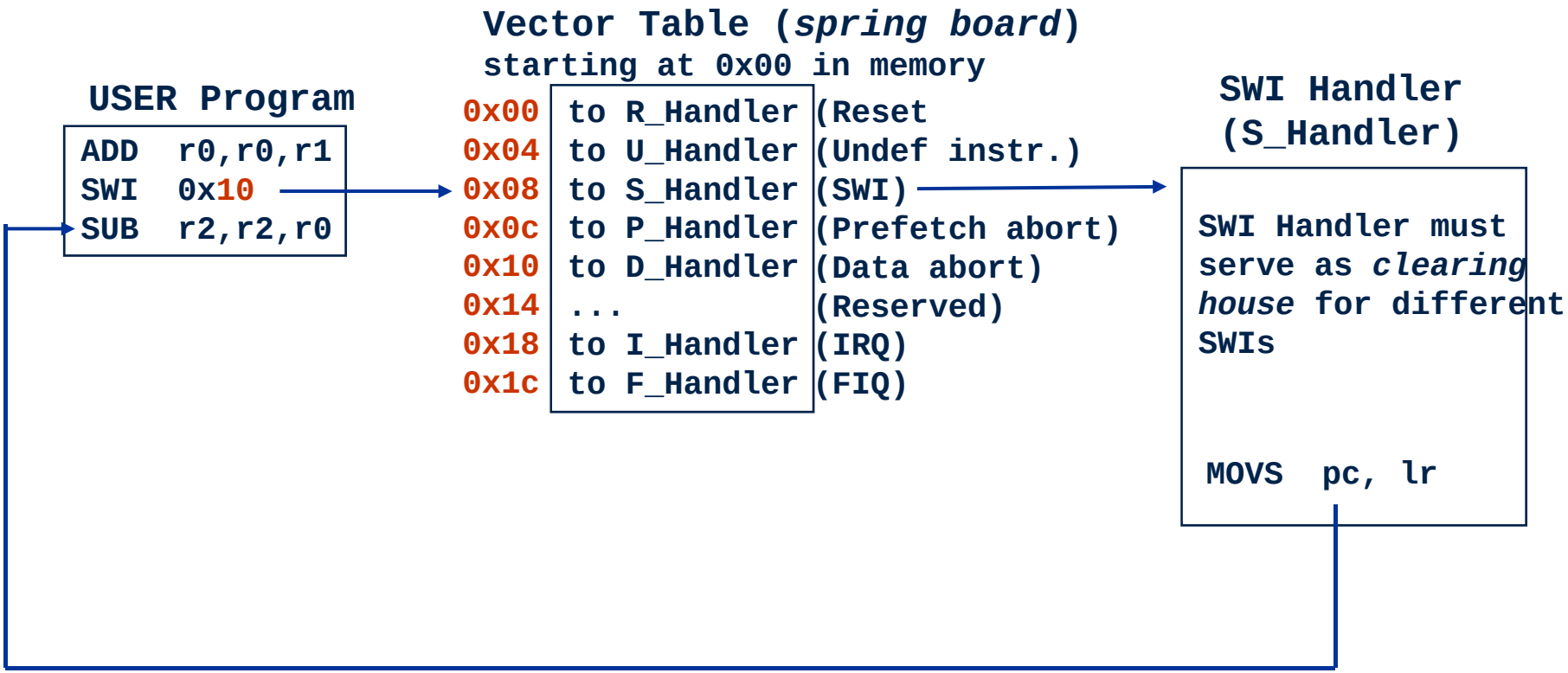
What Happens on SWI Completion?

- Vectoring to the S_Handler starts executing the SWI handler
- When the handler is done, it returns to the program -- at the instruction following the SWI
- MOVS restores the original CPSR as well as changing pc ③



How Do We Determine the SWI number?

- **All SWIs go to 0x08**



SWI Handler Uses the "Comment" Field

On SWI, the processor

(1) copies CPSR to SPSR_SVC
(2) set the CPSR mode bits to supervisor mode
(3) sets the CPSR IRQ to disable
(4) stores the value (PC + 4) into LR_SVC
(5) forces PC to 0x08

cond	1 1 1 1	24-bit "comment" field (ignored by processor)

USER Program

```
ADD  r0,r0,r1
SWI  0x10
SUB  r2,r2,r0
```

Vector Table (*spring board*) starting at 0x00 in memory

```
0x00  to R_Handler  (Reset
0x04  to U_Handler  (Undef instr.)
0x08  to S_Handler  (SWI)
0x0c  to P_Handler  (Prefetch abort)
0x10  to D_Handler  (Data abort)
0x14  ...           (Reserved)
0x18  to I_Handler  (IRQ)
0x1c  to F_Handler  (FIQ)
```

SWI Handler (S_Handler)

```
LDR r0,[lr,#-4]
BIC r0,r0,#0xff000000

R0 holds SWI number


MOVS  pc, lr
```

Use The SWI # to Jump to "Service Routine"

On SWI, the processor

(1) copies CPSR to SPSR_SVC
(2) set the CPSR mode bits to supervisor mode
(3) sets the CPSR IRQ to disable
(4) stores the value (PC + 4) into LR_SVC
(5) forces PC to 0x08

cond	1111	24-bit "comment" field (ignored by processor)

USER Program

```
ADD  r0,r0,r1
SWI  0x10
SUB  r2,r2,r0
```

Vector Table (*spring board*) starting at 0x00 in memory

```
0x00  to R_Handler  (Reset
0x04  to U_Handler  (Undef instr.)
0x08  to S_Handler  (SWI)
0x0c  to P_Handler  (Prefetch abort)
0x10  to D_Handler  (Data abort)
0x14  ...           (Reserved)
0x18  to I_Handler  (IRQ)
0x1c  to F_Handler  (FIQ)
```

SWI Handler (S_Handler)

```
LDR r0,[lr,#-4]
BIC r0,r0,#0xff000000

switch (r0){
case 0x00: service_SWI1();
case 0x01: service_SWI2();
case 0x02: service_SWI3();
…
}
MOVS  pc, lr
```

Problem with The Current Handler

On SWI, the processor

(1) copies CPSR to SPSR_SVC
(2) set the CPSR mode bits to supervisor mode
(3) sets the CPSR IRQ to disable
(4) stores the value (PC + 4) into LR_SVC
(5) forces PC to 0x08

What was in R0? User program may have been using this register. Therefore, cannot just use it - must first save it

USER Program

```
ADD   r0,r0,r1
SWI   0x10
SUB   r2,r2,r0
```

Vector Table (*spring board*) starting at 0x00 in memory

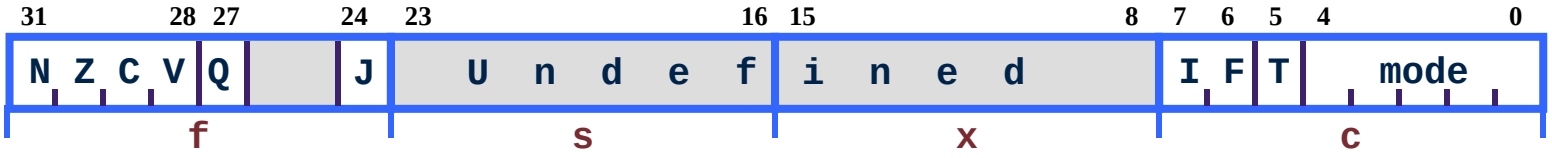
```
0x00  to R_Handler  (Reset
0x04  to U_Handler  (Undef instr.)
0x08  to S_Handler  (SWI)
0x0c  to P_Handler  (Prefetch abort)
0x10  to D_Handler  (Data abort)
0x14  ...           (Reserved)
0x18  to I_Handler  (IRQ)
0x1c  to F_Handler  (FIQ)
```

SWI Handler (S_Handler)

```
LDR r0,[lr,#-4]
BIC r0,r0,#0xff000000

switch (r0){
case 0x00: service_SWI1();
case 0x01: service_SWI2();
case 0x02: service_SWI3();
…
}
MOVS  pc, lr
```

PSR Transfer Instructions

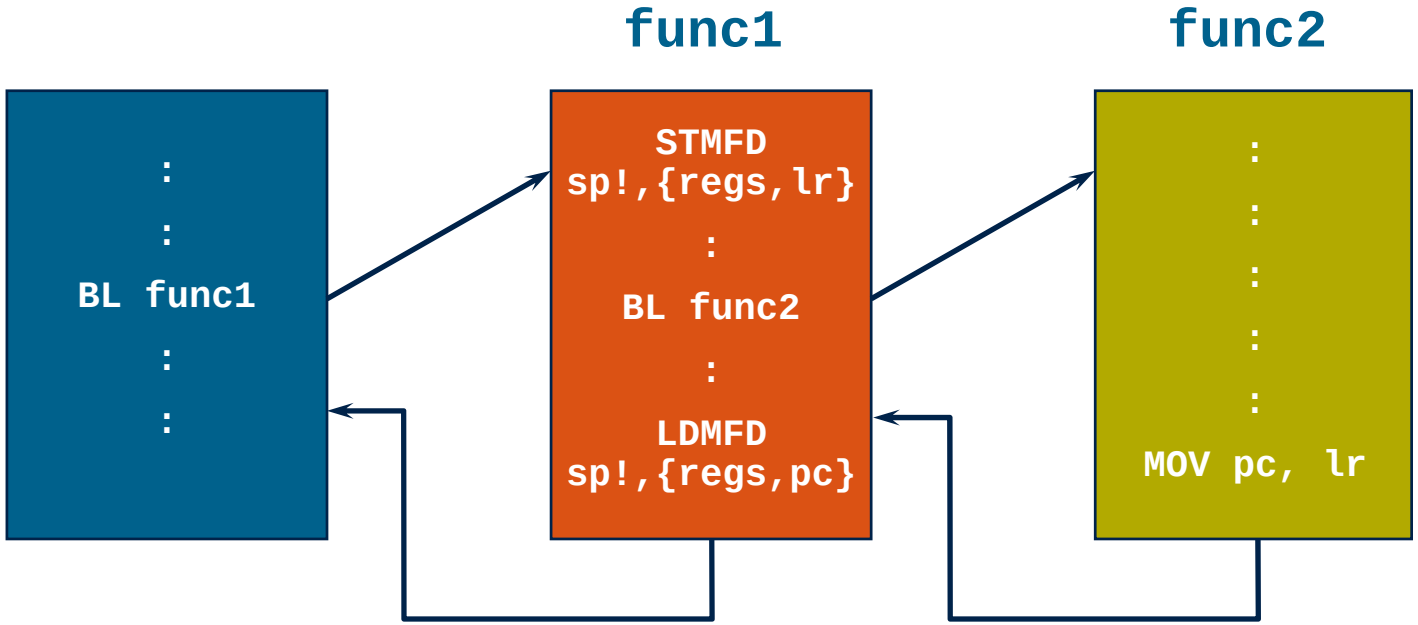


- MRS and MSR allow contents of CPSR / SPSR to be transferred to / from a general purpose register.
- Syntax:
  - `MRS{<cond>} Rd,<psr>  ; Rd = <psr>`
  - `MSR{<cond>} <psr[_fields]>,Rm  ; <psr[_fields]> = Rm`

  where
  - `<psr> = CPSR or SPSR`
  - `[_fields] = any combination of ‘fsxc’`
- Also an immediate form
  - `MSR{<cond>} <psr_fields>,#Immediate`
- In User Mode, all bits can be read but only the condition flags (_f) can be written.

ARM Branches and Subroutines

- **B <label>**
  - PC relative. ±32 Mbyte range.
- **BL <subroutine>**
  - Stores return address in LR
  - Returning implemented by restoring the PC from LR
  - For non-leaf functions, LR will have to be stacked



Flow control instructions

```
Syntax: B{<cond>} label
        BL{<cond>} label
        BX{<cond>} Rm
        BLX{<cond>} label | Rm
```

B	branch	*pc* = *label* **pc-relative offset within 32MB**
BL	branch with link	*pc* = *label* *lr* = address of the next instruction after the BL
BX	branch exchange	*pc* = *Rm* & 0xfffffffe, *T* = *Rm* & 1
BLX	branch exchange with link	*pc* = *label*, *T* = 1 *pc* = *Rm* & 0xfffffffe, *T* = *Rm* & 1 *lr* = address of the next instruction after the BLX

Flow control instructions

- Branch instruction

```
            B   label
            …
label:      …
```

- Conditional branches

```
            MOV   R0, #0
loop:       …
            ADD   R0, R0, #1
            CMP   R0, #10
            BNE   loop
```

Branch conditions

Mnemonic	Name	Condition flags
EQ	equal	*Z*
NE	not equal	*z*
CS HS	carry set/unsigned higher or same	*C*
CC LO	carry clear/unsigned lower	*c*
MI	minus/negative	*N*
PL	plus/positive or zero	*n*
VS	overflow	*V*
VC	no overflow	*v*
HI	unsigned higher	*zC*
LS	unsigned lower or same	*Z* or *c*
GE	signed greater than or equal	*NV* or *nv*
LT	signed less than	*Nv* or *nV*
GT	signed greater than	*NzV* or *nzv*
LE	signed less than or equal	*Z* or *Nv* or *nV*
AL	always (unconditional)	ignored

Branches

Branch	Interpretation	Normal uses
B BAL	Unconditional Always	Always take this branch 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 BLO	Carry clear Lower	Arithmetic operation did not give carry-out Unsigned comparison gave lower
BCS BHS	Carry set Higher or same	Arithmetic operation gave carry-out 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

Branch and link

- BL instruction save the return address to R14 (lr)

```
        BL    sub       @ call sub
        CMP   R1, #5    @ return to here
        MOVEQ R1, #0
        …
sub:    …               @ sub entry point
        …
        MOV   PC, LR    @ return
```

Branch and link

```
        BL    sub1      @ call sub1
        …
```

use stack to save/restore the return address and registers

```
sub1:   STMFD R13!, {R0-R2,R14}
        BL    sub2
        …
        LDMFD R13!, {R0-R2,PC}
```

```
sub2:   …
        …
        MOV   PC, LR
```

Conditional execution

```
        CMP   R0, #5
        BEQ   bypass      @ if (R0!=5) {
        ADD   R1, R1, R0  @  R1=R1+R0-R2
        SUB   R1, R1, R2  @ }
bypass: …
```

smaller and faster

```
        CMP   R0, #5
        ADDNE R1, R1, R0
        SUBNE R1, R1, R2
```

Rule of thumb: if the conditional sequence is three instructions or less, it is better to use conditional execution than a branch.

Conditional execution

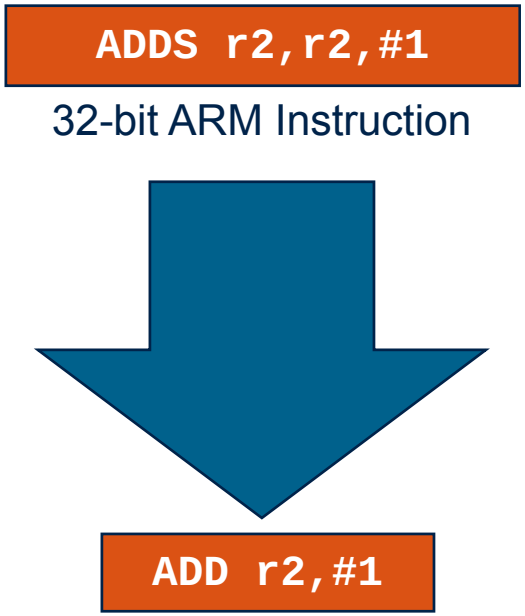
```
  if ((R0==R1) && (R2==R3)) R4++
```

```
      CMP   R0, R1
      BNE   skip
      CMP   R2, R3
      BNE   skip
      ADD   R4, R4, #1
skip: …
```

```
      CMP   R0, R1
      CMPEQ R2, R3
      ADDEQ R4, R4, #1
```

Thumb

- **Thumb is a 16-bit instruction set**
  - Optimised for code density from C code (~65% of ARM code size)
  - 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

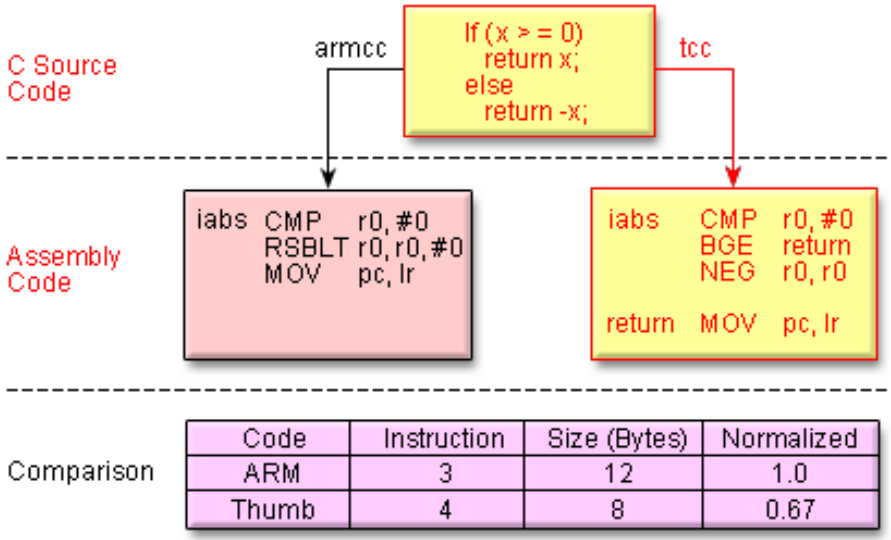


For most instructions generated by compiler:

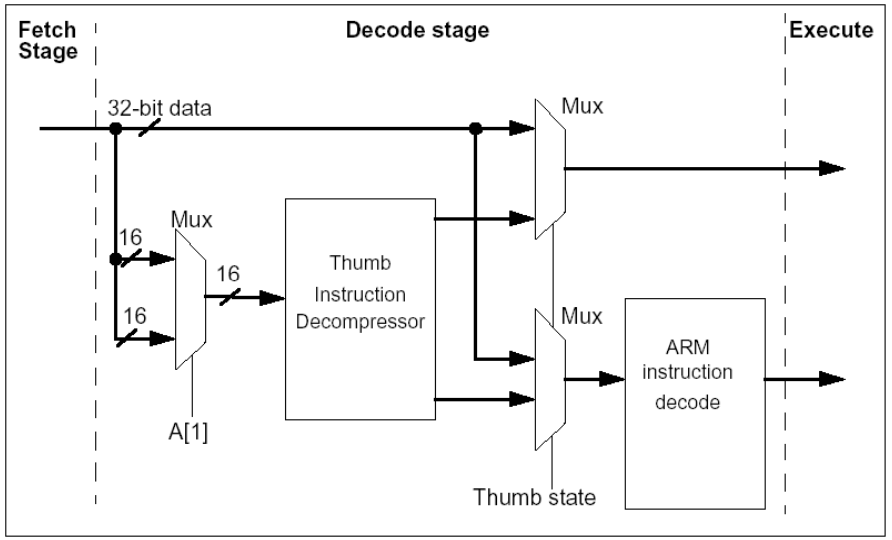
- Conditional execution is not used
- Source and destination registers identical
- Only Low registers used
- Constants are of limited size
- Inline barrel shifter not used

Thumb Benefits

- **Thumb programs typically are:**
  - ~30% smaller than ARM programs
  - ~30% faster when accessing 16-bit memory
- **Thumb reduces 32-bit system to 16-bit cost:**
  - Consumes less power ~30%
  - Requires less external memory
- **But, can be slower than ARM**
  - ~40% more instructions
  - 32-bit memory: ARM code is 40% faster than Thumb code.
  - 16-bit memory: Thumb code is 45% faster than ARM code.



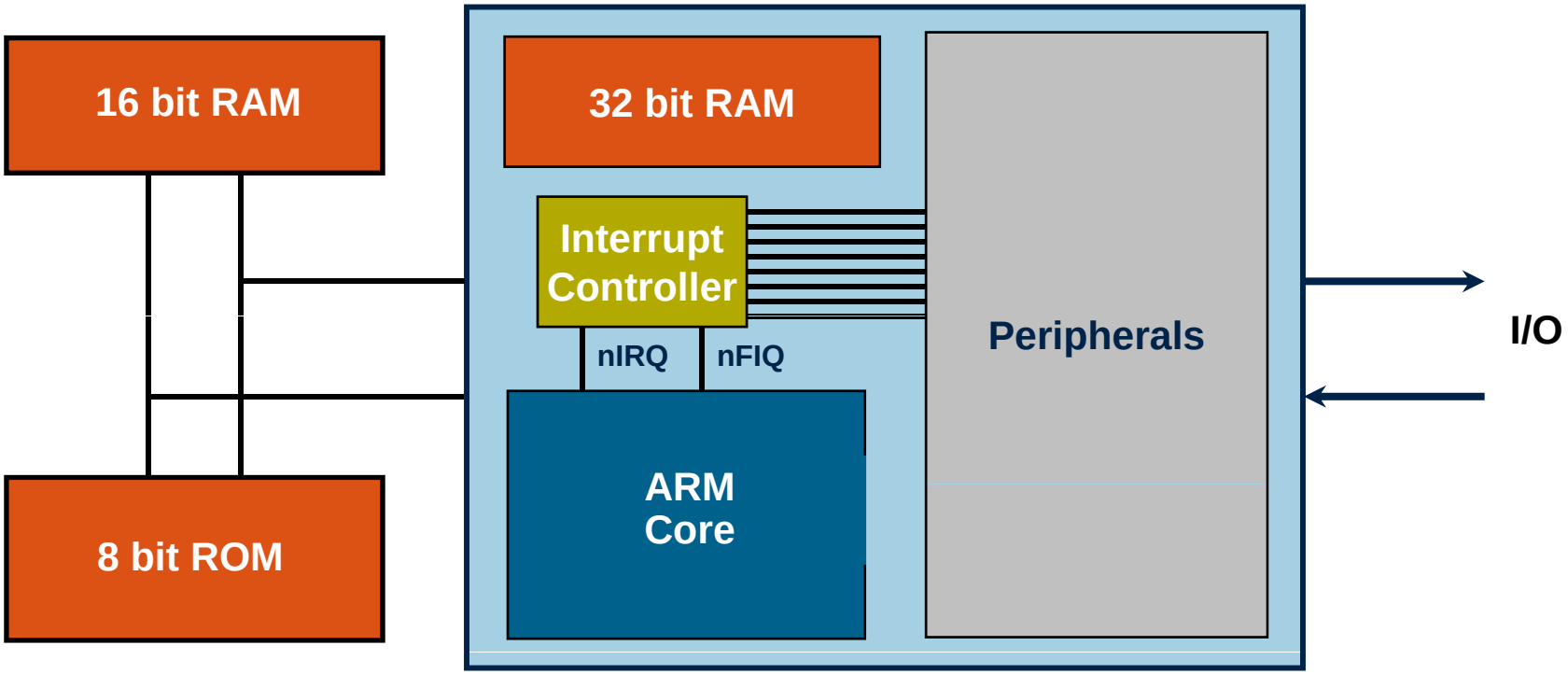
Code	Instruction	Size (Bytes)	Normalized
ARM	3	12	1.0
Thumb	4	8	0.67



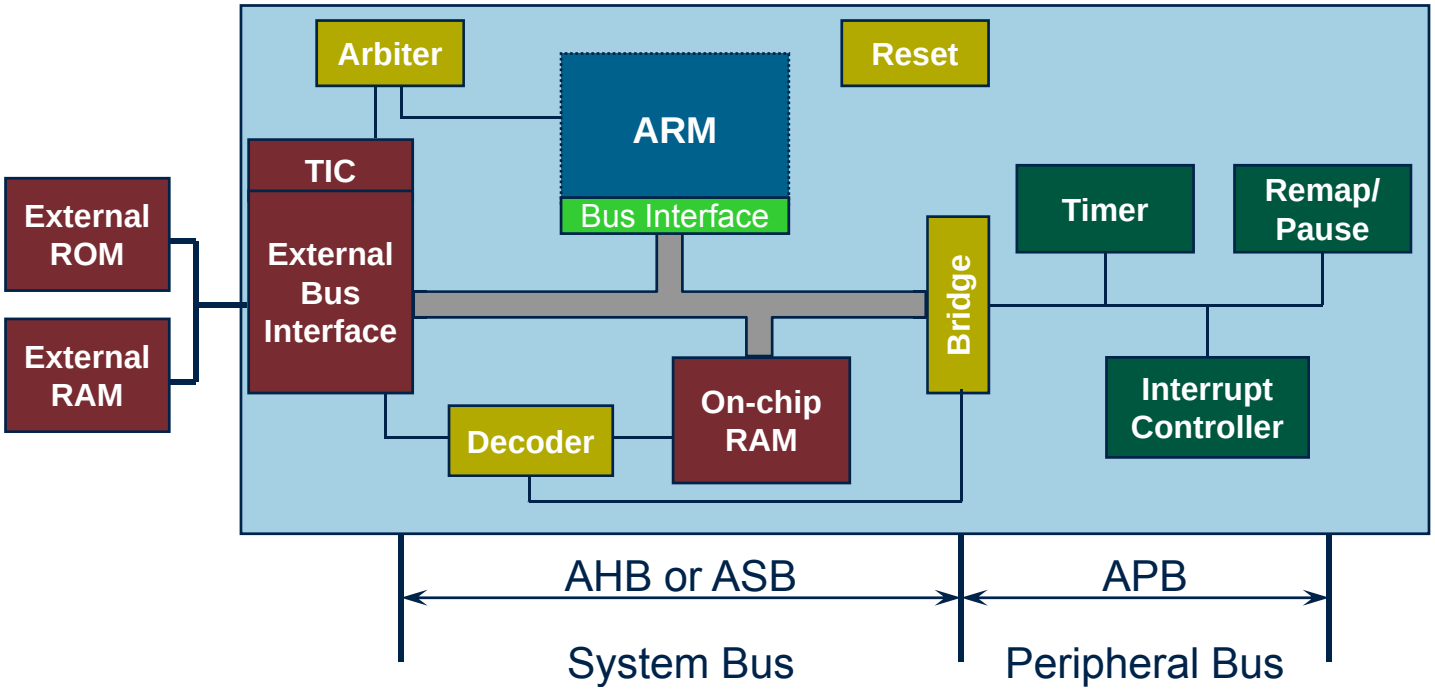
Agenda

- Introduction
- Programmers Model
- Instruction Sets
- **System Design**
- Development Tools

Example ARM-based System



AMBA



- **AMBA**
  - Advanced Microcontroller Bus Architecture
- **ADK**
  - Complete AMBA Design Kit
- **ACT**
  - AMBA Compliance Testbench
- **PrimeCell**
  - ARM's AMBA compliant peripherals

Agenda

- Introduction
- Programmers Model
- Instruction Sets
- System Design
- **Development Tools**

The RealView Product Families

Compilation Tools

ARM Developer Suite (ADS) – Compilers (C/C++ ARM & Thumb), Linker & Utilities

RealView Compilation Tools (RVCT)

Debug Tools

AXD (part of ADS)

Trace Debug Tools

Multi-ICE

Multi-Trace

RealView Debugger (RVD)

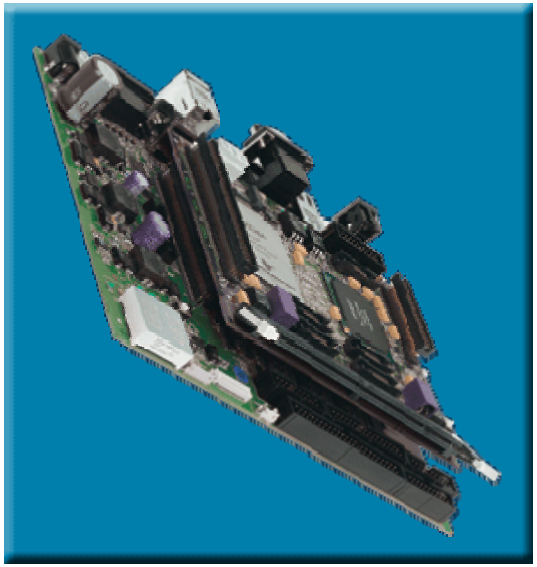
RealView ICE (RVI)

RealView Trace (RVT)

Platforms

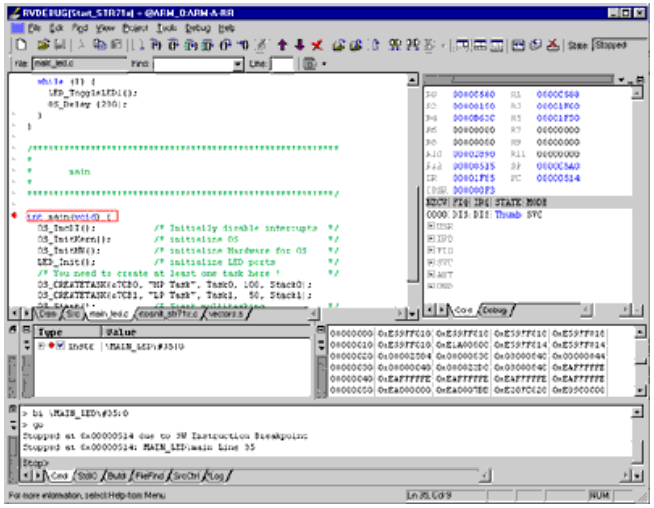
ARMulator (part of ADS)

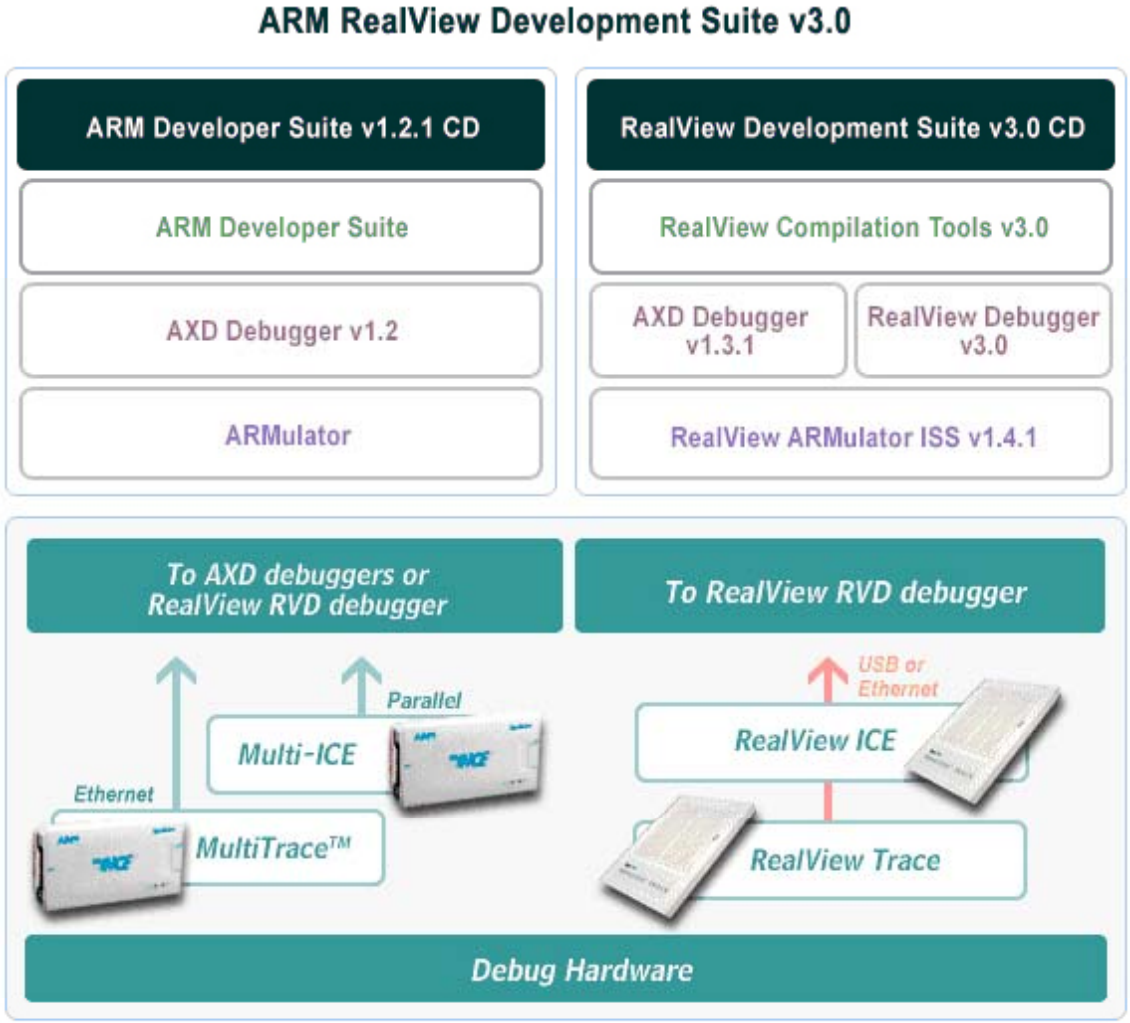
Integrator™ Family



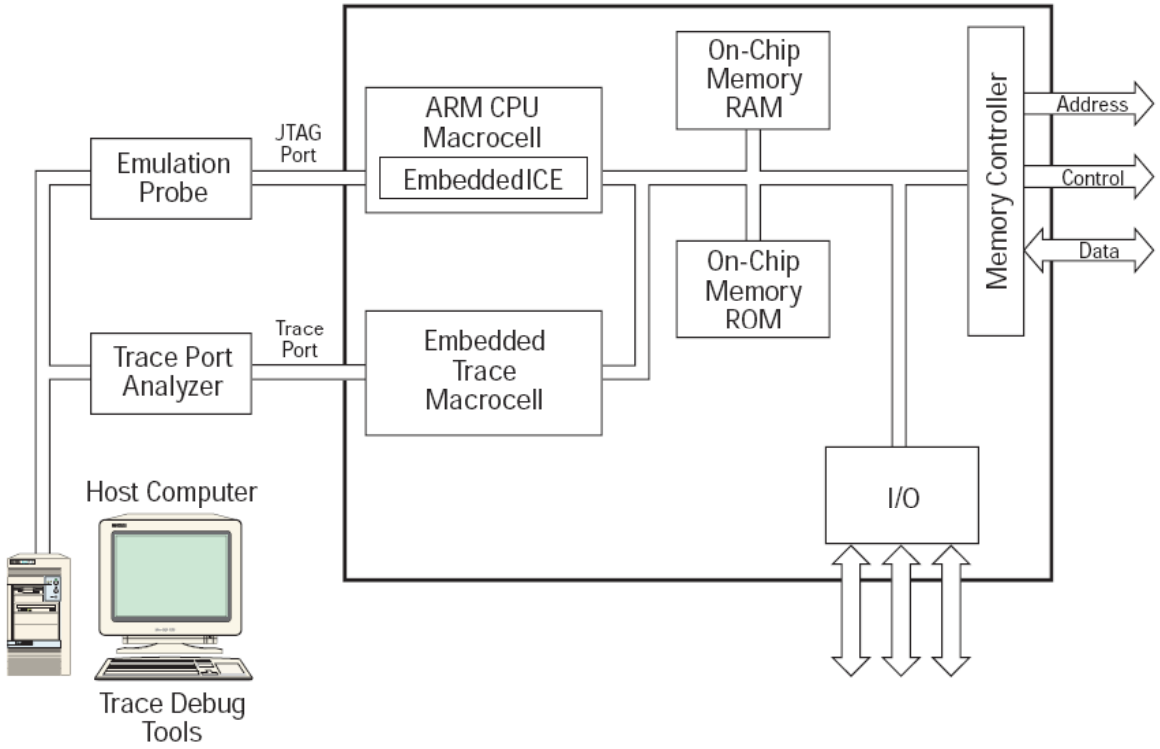
RealView ARMulator ISS (RVISS)

ADS & RealView (ARM)





ARM Debug Architecture



- **EmbeddedICE Logic**
  - Provides breakpoints and processor/system access
- **JTAG interface (ICE)**
  - Converts debugger commands to JTAG signals
- **Embedded trace Macrocell (ETM)**
  - Compresses real-time instruction and data access trace
  - Contains ICE features (trigger & filter logic)
- **Trace port analyzer (TPA)**
  - Captures trace in a deep buffer