ARM Architecture

Architecture ARMv7 profiles

- **Application profile (ARMv7-A)**
  - Memory management support (MMU)
  - Highest performance at low power
  - Influenced by multi-tasking OS system requirements
  - TrustZone and Jazelle-RCT for a safe, extensible system
  - e.g. Cortex-A5, Cortex-A9

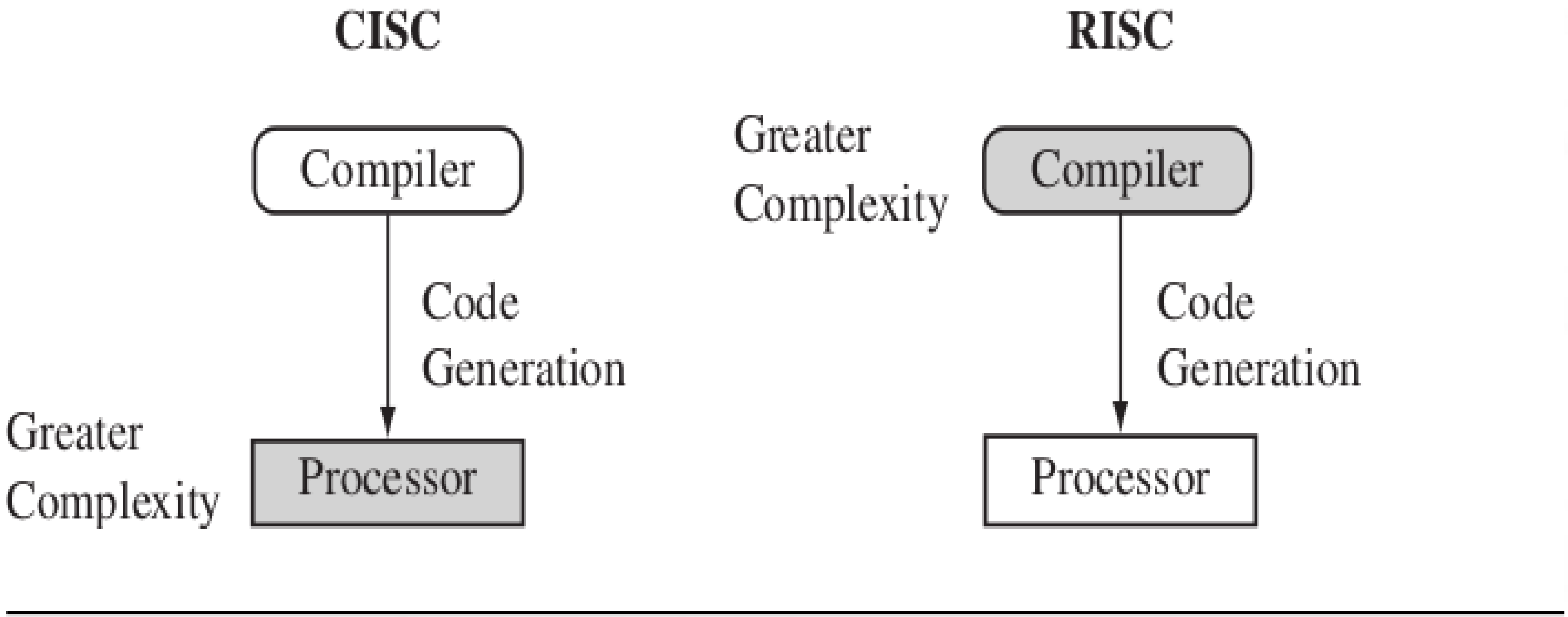
- **Real-time profile (ARMv7-R)**
  - Protected memory (MPU)
  - Low latency and predictability ‘real-time’ needs
  - Evolutionary path for traditional embedded business
  - e.g. Cortex-R4

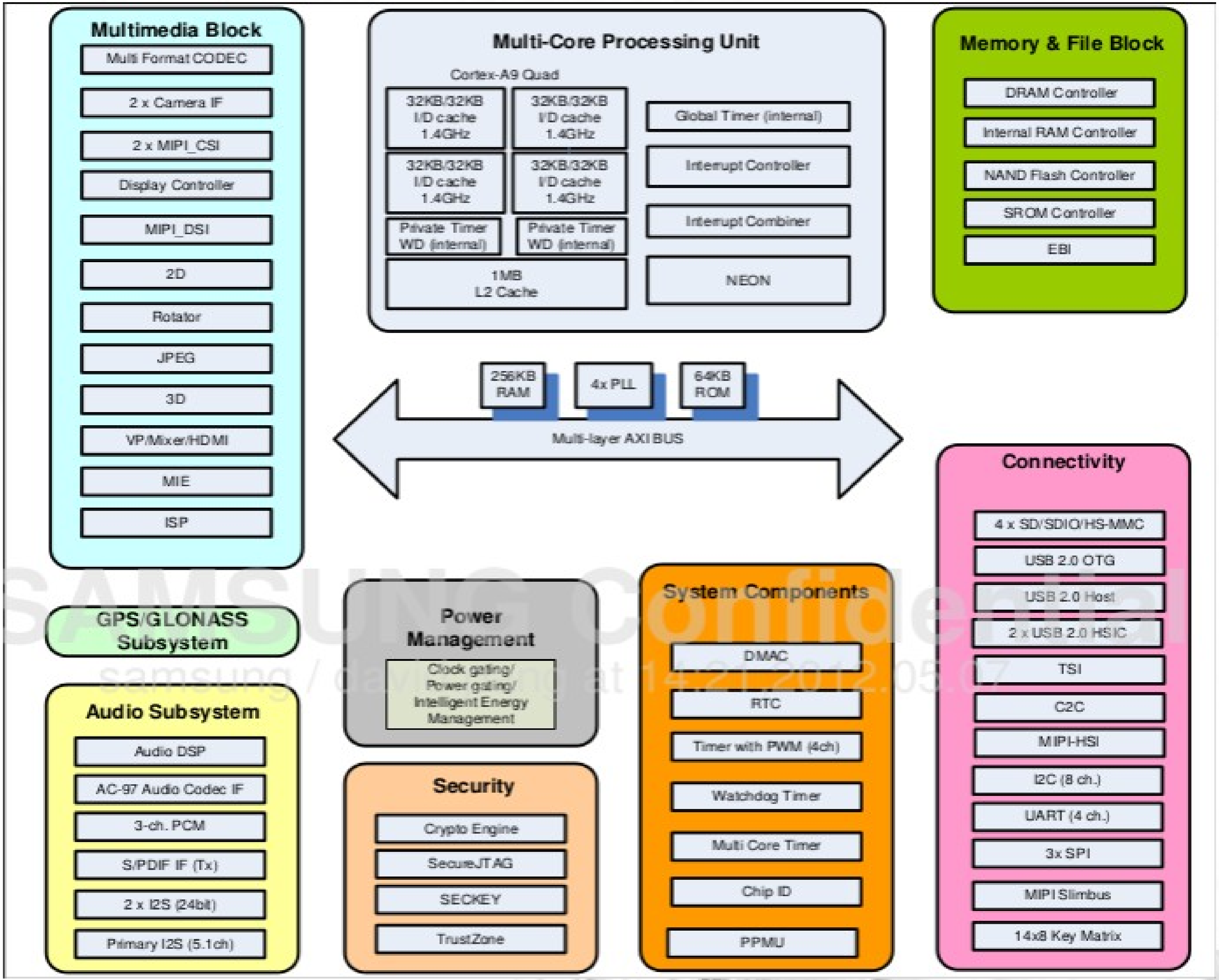
- **Microcontroller profile (ARMv7-M, ARMv7E-M, ARMv6-M)**
  - Lowest gate count entry point
  - Deterministic and predictable behavior a key priority
  - Deeply embedded use
  - e.g. Cortex-M3

The RISC design philosophy

- The ARM core uses a RISC architecture
- The RISC philosophy is implemented with four major design rules
    - Instructions
    - Pipelines
    - Registers
    - Load-store architecture

The RISC design philosophy





Multimedia Block

Multi Format CODEC

2 x Camera IF

2 x MIPI_CSI

Display Controller

MIPI_DSI

2D

Rotator

JPEG

3D

VP/Mixer/HDMI

MIE

ISP

Multi-Core Processing Unit

Cortex-A9 Quad

32KB/32KB I/D cache 1.4GHz

32KB/32KB I/D cache 1.4GHz

32KB/32KB I/D cache 1.4GHz

32KB/32KB I/D cache 1.4GHz

Private Timer WD (internal)

Private Timer WD (internal)

1MB L2 Cache

Global Timer (internal)

Interrupt Controller

Interrupt Combiner

NEON

Memory & File Block

DRAM Controller

Internal RAM Controller

NAND Flash Controller

SROM Controller

EBI

256KB RAM

4x PLL

64KB ROM

Multi-layer AXI BUS

Connectivity

4 x SD/SDIO/HS-MMC

USB 2.0 OTG

USB 2.0 Host

2 x USB 2.0 HSIC

TSI

C2C

MIPI-HSI

I2C (8 ch.)

UART (4 ch.)

3x SPI

MIPI Slimbus

14x8 Key Matrix

GPS/GLONASS Subsystem

Audio Subsystem

Audio DSP

AC-97 Audio Codec IF

3-ch. PCM

S/PDIF IF (Tx)

2 x I2S (24bit)

Primary I2S (5.1ch)

Power Management

Clock gating/ Power gating/ Intelligent Energy Management

Security

Crypto Engine

SecureJTAG

SECKEY

TrustZone

System Components

DMAC

RTC

Timer with PWM (4ch)

Watchdog Timer

Multi Core Timer

Chip ID

PPMU

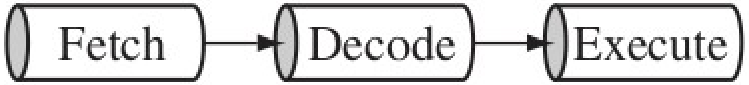
Pipeline

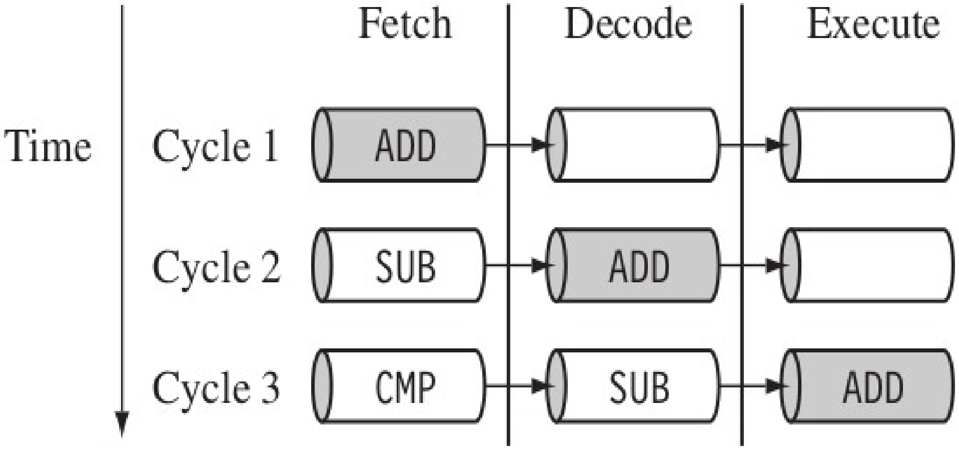
- A pipeline is the mechanism a RISC processor uses to execute instructions

Fetch loads an instruction from memory

Decode identifies the instruction to be executed

Execute processes the instruction and writes the result back to a register





Time	Fetch	Decode	Execute
Cycle 1	ADD		
Cycle 2	SUB	ADD	
Cycle 3	CMP	SUB	ADD

AMBA Bus Protocol

- **A**dvanced **M**icrocontroller **B**us **A**rchitecture
- ASB - ARM System Bus
- APB - ARM Peripheral Bus
- AHB - ARM High Performance Bus
- AXI - Advanced eXtensible Interface
- ACE - AXI Coherency Extensions

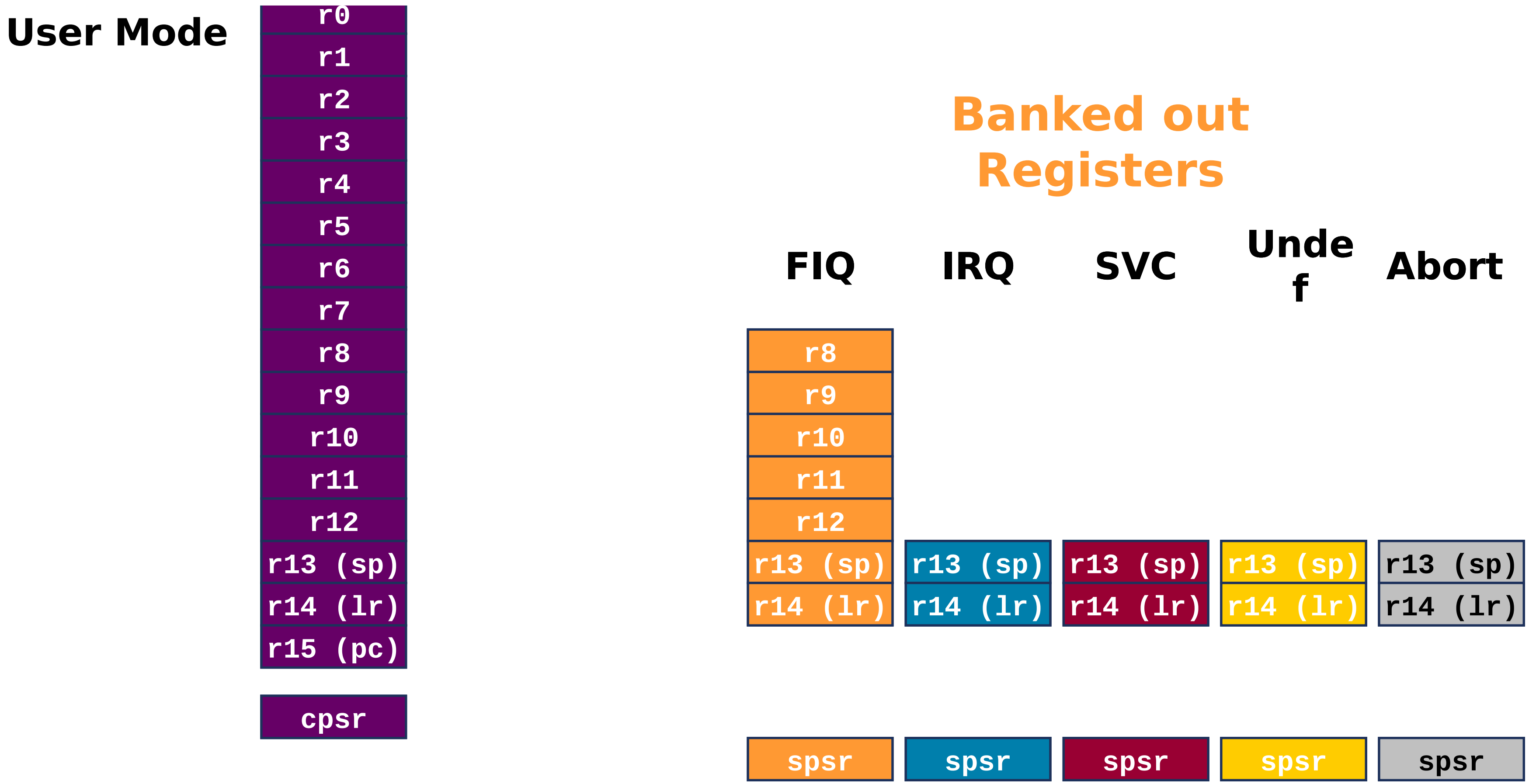
Processor Modes

- **ARM has seven basic operating modes**
  - Each mode has access to its own stack space and a different subset of registers
  - Some operations can only be carried out in a privileged mode

	Mode	Description	
Exception modes	**Supervisor (SVC)**	Entered on reset and when a Supervisor call instruction (SVC) is executed	**Privileged modes**
	FIQ	Entered when a high priority (fast) interrupt is raised	
	IRQ	Entered when a normal priority interrupt is raised	
	Abort	Used to handle memory access violations	
	Undef	Used to handle undefined instructions	
	System	Privileged mode using the same registers as User mode	
	User	Mode under which most Applications / OS tasks run	**Unprivileged mode**

The ARM Register Set

Current Visible Registers



User Mode

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

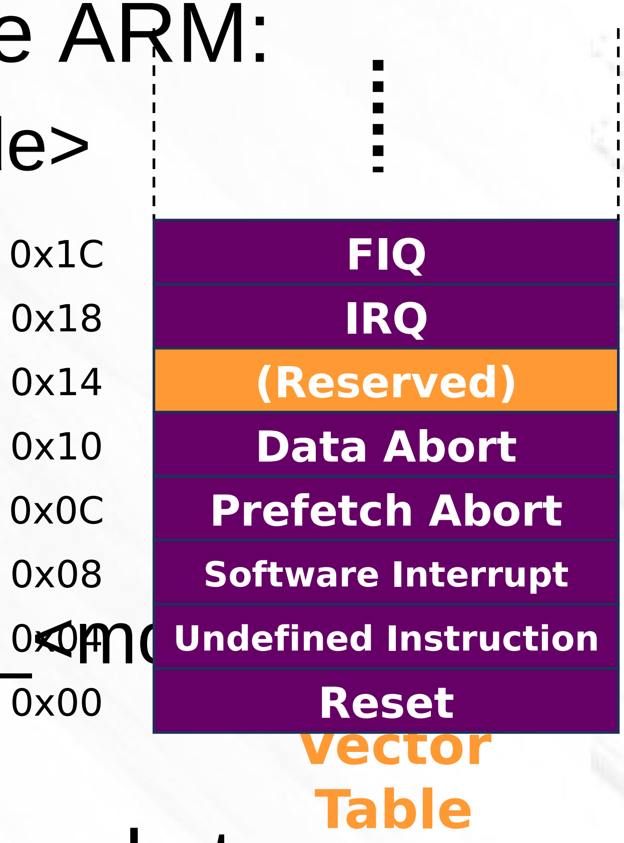
cpsr

Banked out Registers

FIQ	IRQ	SVC	Undef	Abort
r8				
r9				
r10				
r11				
r12				
r13 (sp)	r13 (sp)	r13 (sp)	r13 (sp)	r13 (sp)
r14 (lr)	r14 (lr)	r14 (lr)	r14 (lr)	r14 (lr)
spsr	spsr	spsr	spsr	spsr

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_<mo
  - Sets PC to vector address

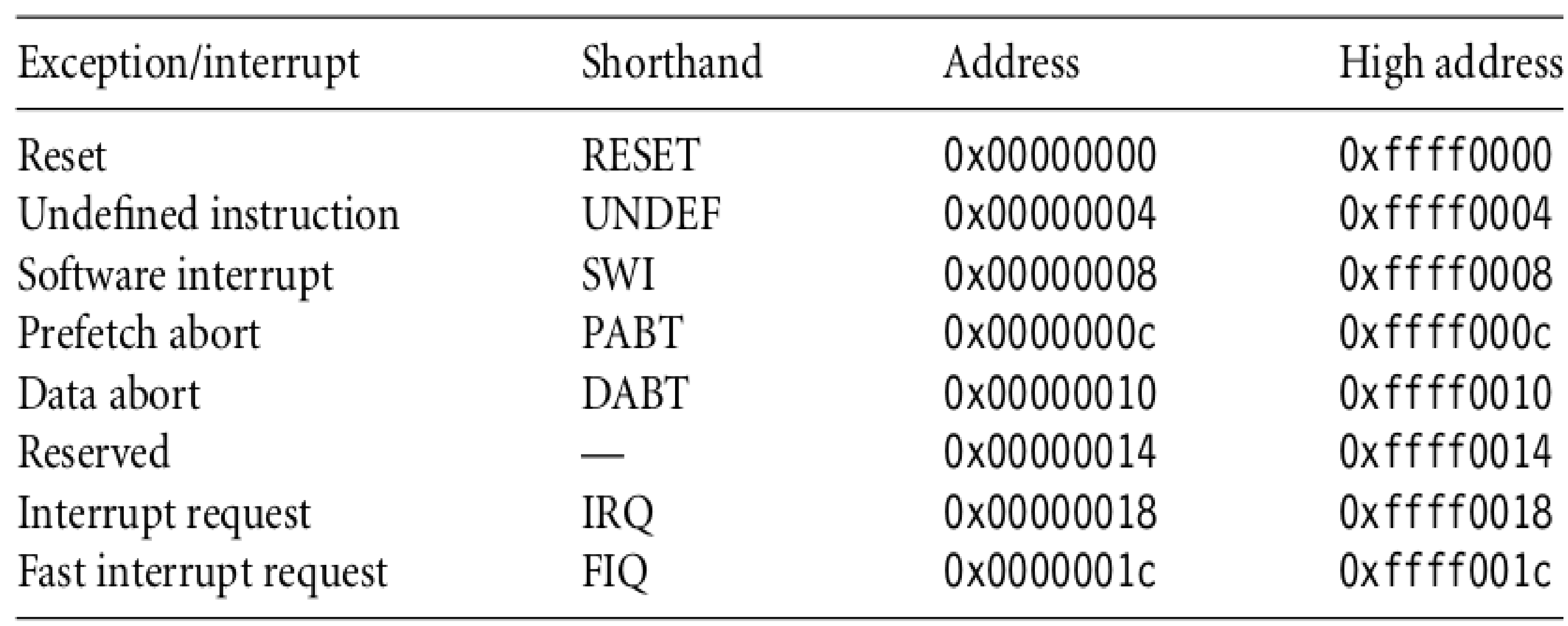


0x1C	FIQ
0x18	IRQ
0x14	(Reserved)
0x10	Data Abort
0x0C	Prefetch Abort
0x08	Software Interrupt
0x04	Undefined Instruction
0x00	Reset

Vector Table

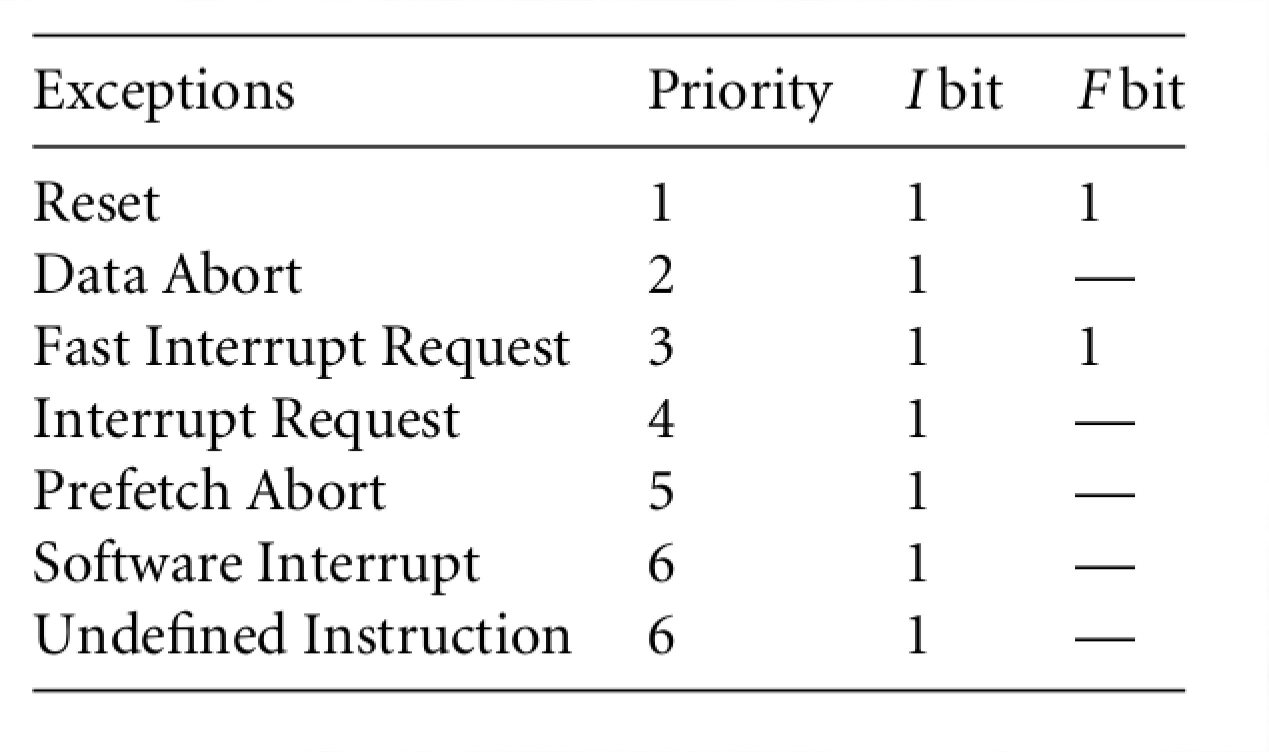
- To return, exception handler needs to:
  - Restore CPSR from SPSR_<mode>
  - Restore PC from LR_<mode>

The vector table



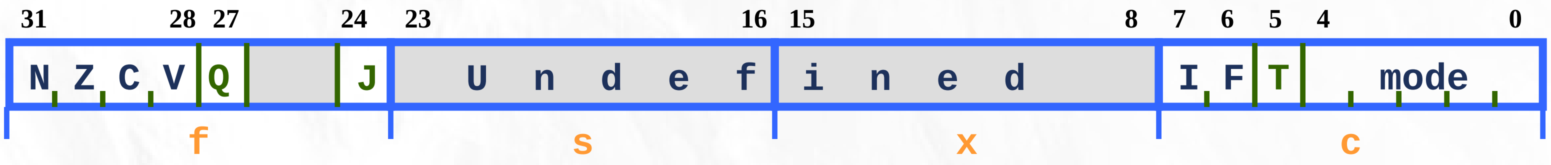
Exception/interrupt	Shorthand	Address	High address
Reset	RESET	0x00000000	0xffff0000
Undefined instruction	UNDEF	0x00000004	0xffff0004
Software interrupt	SWI	0x00000008	0xffff0008
Prefetch abort	PABT	0x0000000c	0xffff000c
Data abort	DABT	0x00000010	0xffff0010
Reserved	—	0x00000014	0xffff0014
Interrupt request	IRQ	0x00000018	0xffff0018
Fast interrupt request	FIQ	0x0000001c	0xffff001c

Exception priority levels



Exceptions	Priority	*I* bit	*F* bit
Reset	1	1	1
Data Abort	2	1	—
Fast Interrupt Request	3	1	1
Interrupt Request	4	1	—
Prefetch Abort	5	1	—
Software Interrupt	6	1	—
Undefined Instruction	6	1	—

Program Status Registers



- Condition code flags
  - N = **N**egative result from ALU
  - Z = **Z**ero result from ALU
  - C = ALU operation **C**arried out
  - V = ALU operation o**V**erflowed
- Sticky Overflow flag - Q flag
  - Architecture 5TE/J only
  - Indicates if saturation has occurred
- J bit
  - Architecture 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

Register Usage

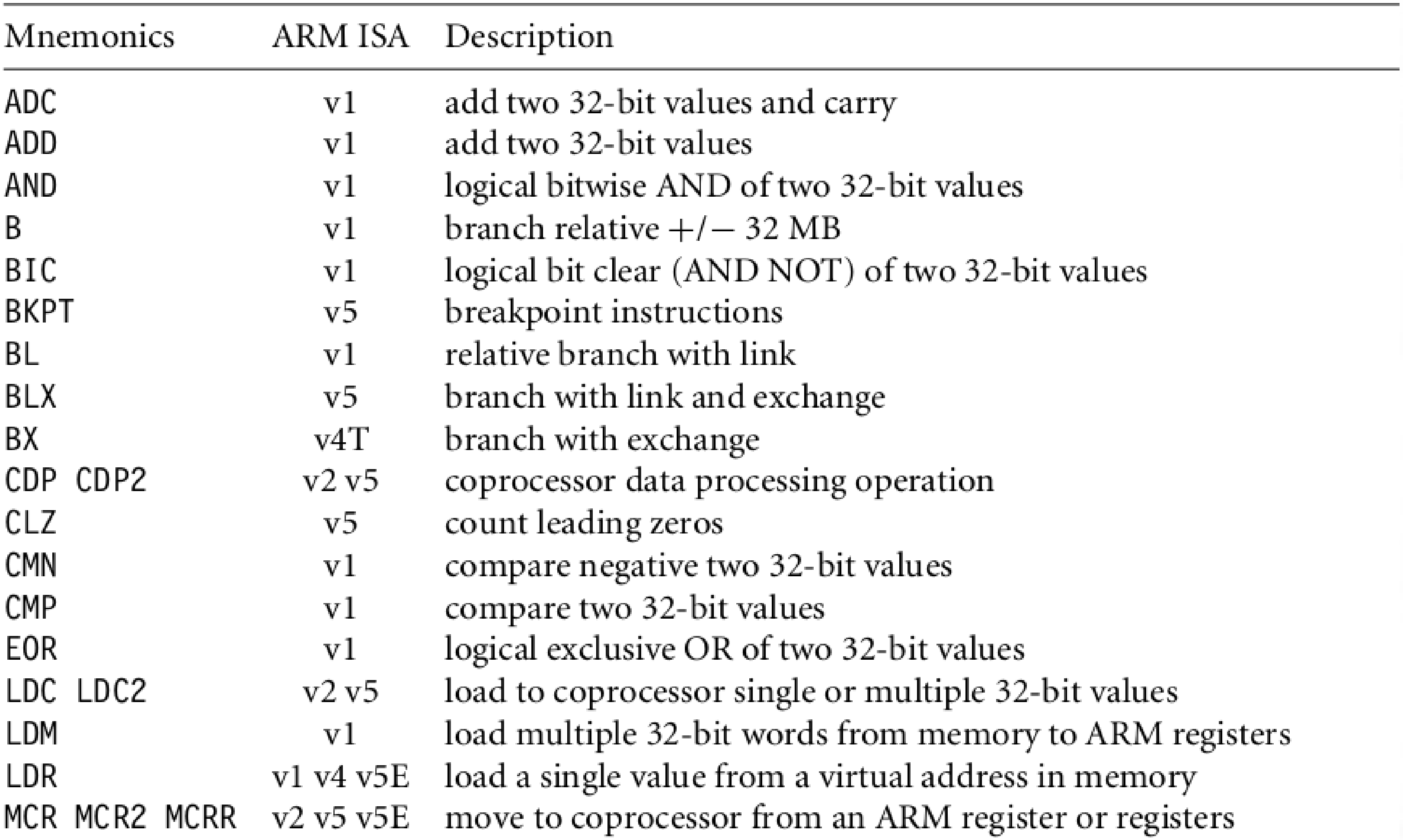
Arguments into function Result(s) from function otherwise corruptible (Additional parameters passed on stack)	r0 r1 r2 r3	
Register variables Must be preserved	r4 r5 r6 r7 r8 r9/sb r10/sl r11	**- Stack base** **- Stack limit if software stack checking selected**
Scratch register (corruptible)	r12	
Stack Pointer Link Register Program Counter	r13/sp r14/lr r15/pc	**- SP should always be 8-byte (2 word) aligned** **- R14 can be used as a temporary once value stacked**

The compiler has a set of rules known as a Procedure Call Standard that determine how to pass parameters to a function (see **AAPCS**)

CPSR flags may be corrupted by function call.
Assembler code which links with compiled code must follow the AAPCS at external interfaces

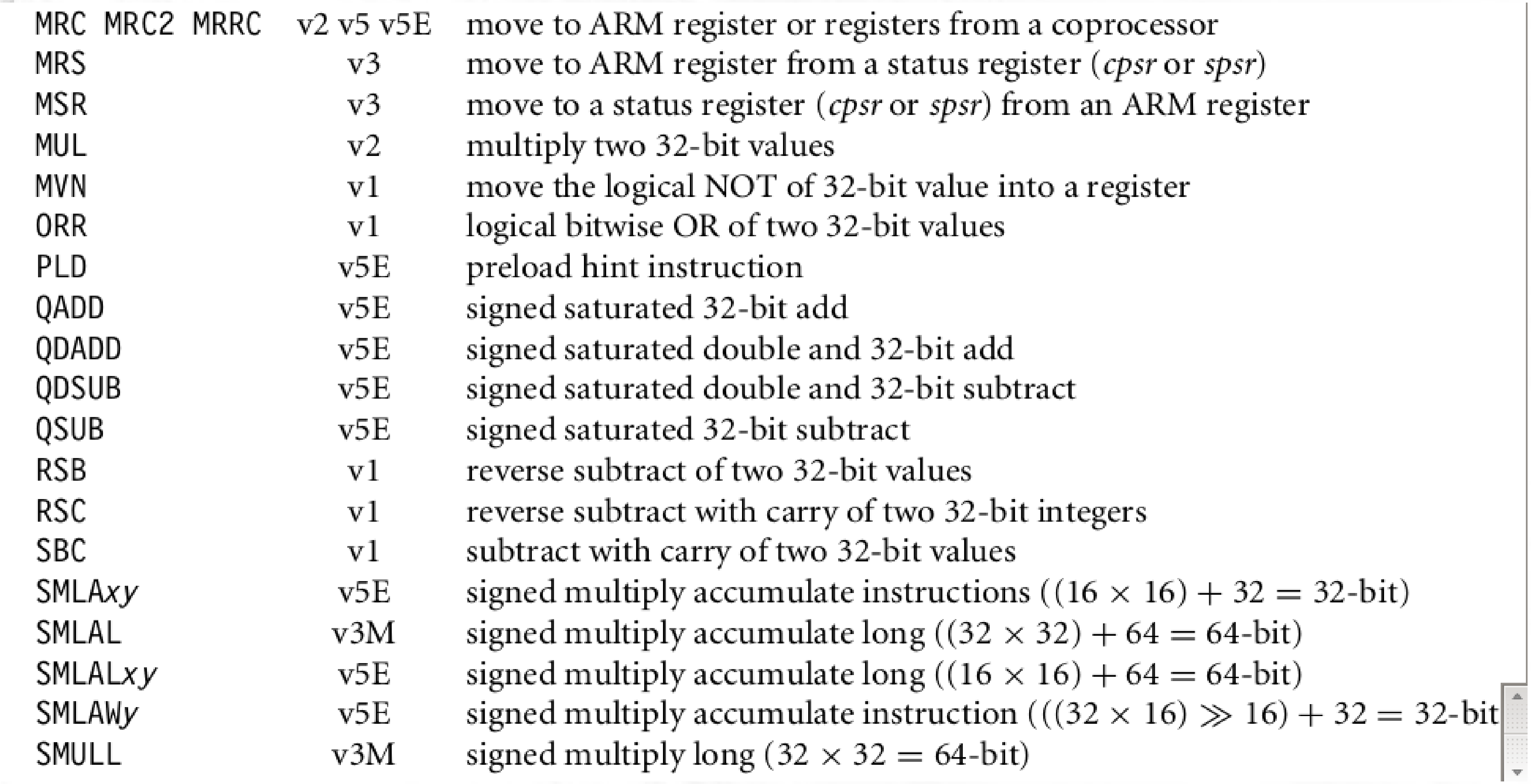
The AAPCS is part of the new ABI for the ARM Architecture

Introduction to the ARM Instruction Set



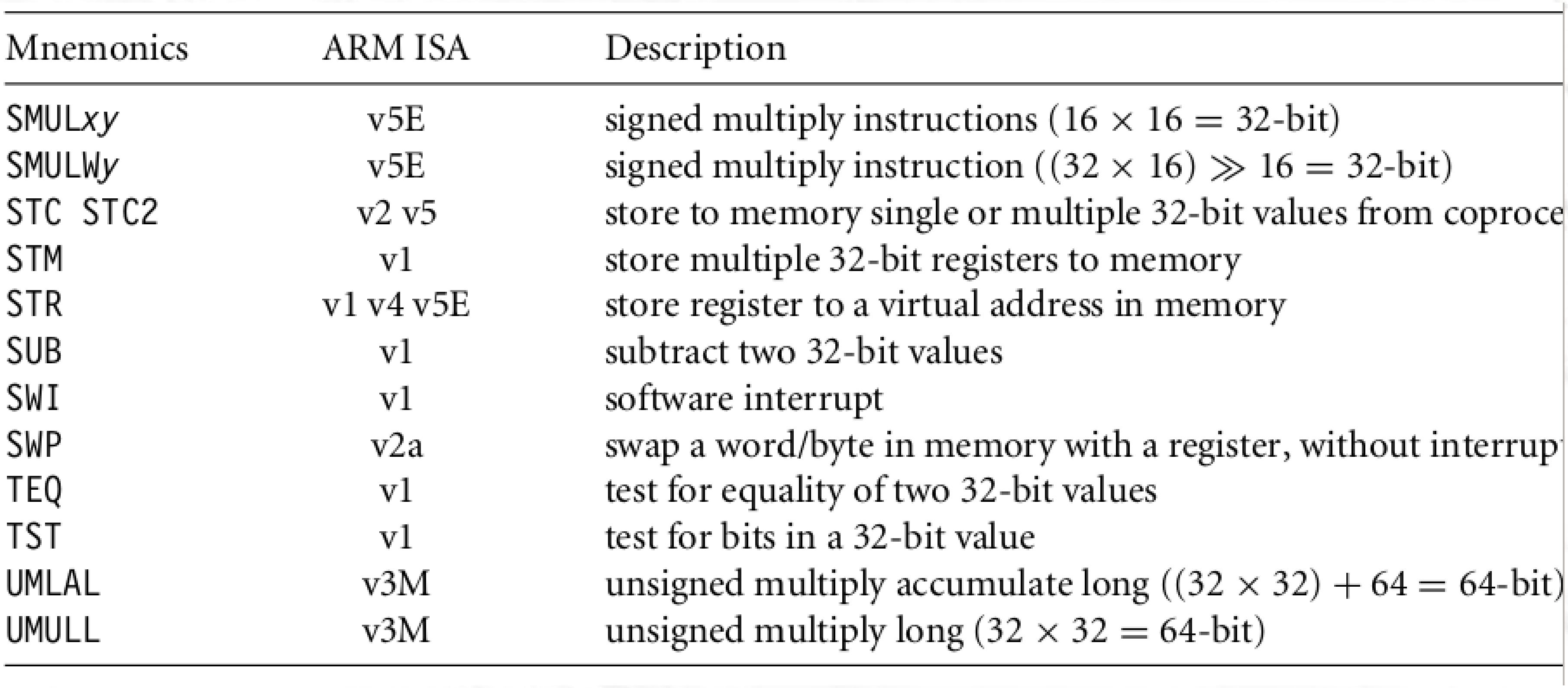
Mnemonics	ARM ISA	Description
ADC	v1	add two 32-bit values and carry
ADD	v1	add two 32-bit values
AND	v1	logical bitwise AND of two 32-bit values
B	v1	branch relative +/− 32 MB
BIC	v1	logical bit clear (AND NOT) of two 32-bit values
BKPT	v5	breakpoint instructions
BL	v1	relative branch with link
BLX	v5	branch with link and exchange
BX	v4T	branch with exchange
CDP CDP2	v2 v5	coprocessor data processing operation
CLZ	v5	count leading zeros
CMN	v1	compare negative two 32-bit values
CMP	v1	compare two 32-bit values
EOR	v1	logical exclusive OR of two 32-bit values
LDC LDC2	v2 v5	load to coprocessor single or multiple 32-bit values
LDM	v1	load multiple 32-bit words from memory to ARM registers
LDR	v1 v4 v5E	load a single value from a virtual address in memory
MCR MCR2 MCRR	v2 v5 v5E	move to coprocessor from an ARM register or registers

Introduction to the ARM Instruction Set



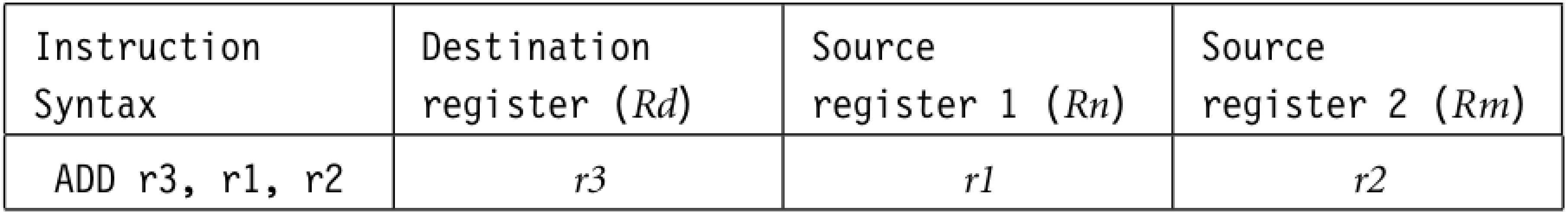
MRC MRC2 MRRC	v2 v5 v5E	move to ARM register or registers from a coprocessor
MRS	v3	move to ARM register from a status register (*cpsr* or *spsr*)
MSR	v3	move to a status register (*cpsr* or *spsr*) from an ARM register
MUL	v2	multiply two 32-bit values
MVN	v1	move the logical NOT of 32-bit value into a register
ORR	v1	logical bitwise OR of two 32-bit values
PLD	v5E	preload hint instruction
QADD	v5E	signed saturated 32-bit add
QDADD	v5E	signed saturated double and 32-bit add
QDSUB	v5E	signed saturated double and 32-bit subtract
QSUB	v5E	signed saturated 32-bit subtract
RSB	v1	reverse subtract of two 32-bit values
RSC	v1	reverse subtract with carry of two 32-bit integers
SBC	v1	subtract with carry of two 32-bit values
SMLA*xy*	v5E	signed multiply accumulate instructions ((16 × 16) + 32 = 32-bit)
SMLAL	v3M	signed multiply accumulate long ((32 × 32) + 64 = 64-bit)
SMLAL*xy*	v5E	signed multiply accumulate long ((16 × 16) + 64 = 64-bit)
SMLAW*y*	v5E	signed multiply accumulate instruction ((($32 \times 16) \gg 16) + 32 = 32$-bit
SMULL	v3M	signed multiply long (32 × 32 = 64-bit)

Introduction to the ARM Instruction Set



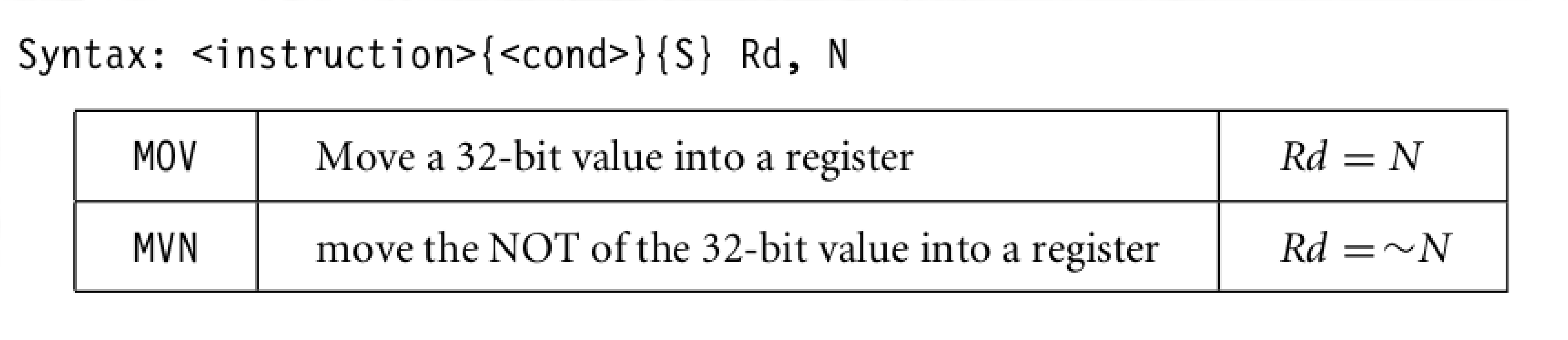
Mnemonics	ARM ISA	Description
SMUL*xy*	v5E	signed multiply instructions ($16 \times 16 = 32$-bit)
SMULW*y*	v5E	signed multiply instruction ($(32 \times 16) \gg 16 = 32$-bit)
STC STC2	v2 v5	store to memory single or multiple 32-bit values from coproce
STM	v1	store multiple 32-bit registers to memory
STR	v1 v4 v5E	store register to a virtual address in memory
SUB	v1	subtract two 32-bit values
SWI	v1	software interrupt
SWP	v2a	swap a word/byte in memory with a register, without interrup
TEQ	v1	test for equality of two 32-bit values
TST	v1	test for bits in a 32-bit value
UMLAL	v3M	unsigned multiply accumulate long ($(32 \times 32) + 64 = 64$-bit)
UMULL	v3M	unsigned multiply long ($32 \times 32 = 64$-bit)

Introduction to the ARM Instruction Set



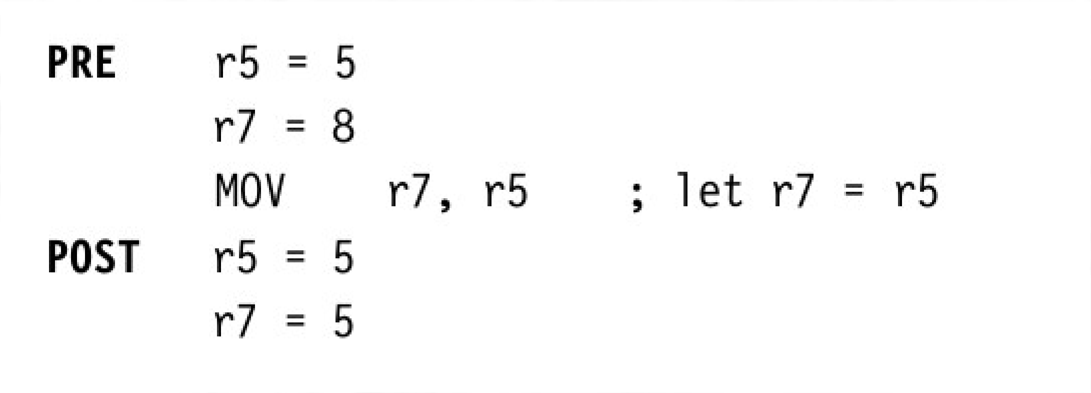
Instruction Syntax	Destination register (*Rd*)	Source register 1 (*Rn*)	Source register 2 (*Rm*)
ADD r3, r1, r2	*r3*	*r1*	*r2*

Data Processing Instructions



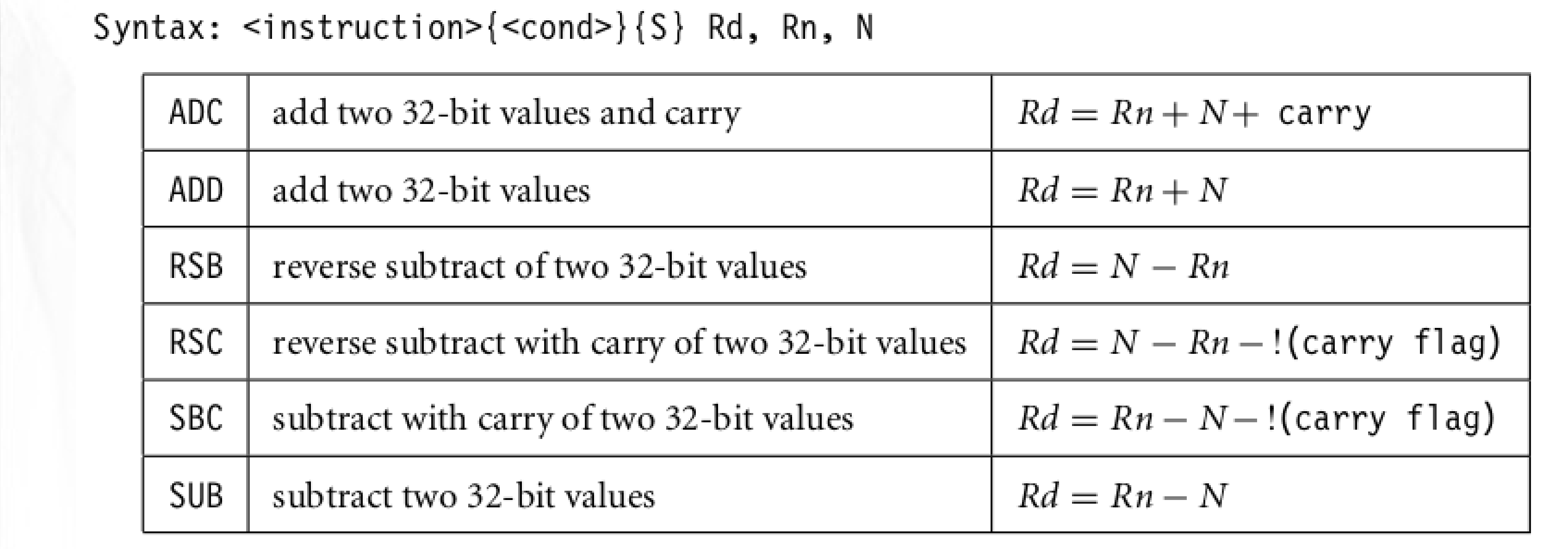
```
Syntax: <instruction>{<cond>}{S} Rd, N
```

MOV	Move a 32-bit value into a register	$Rd = N$
MVN	move the NOT of the 32-bit value into a register	$Rd = \sim N$



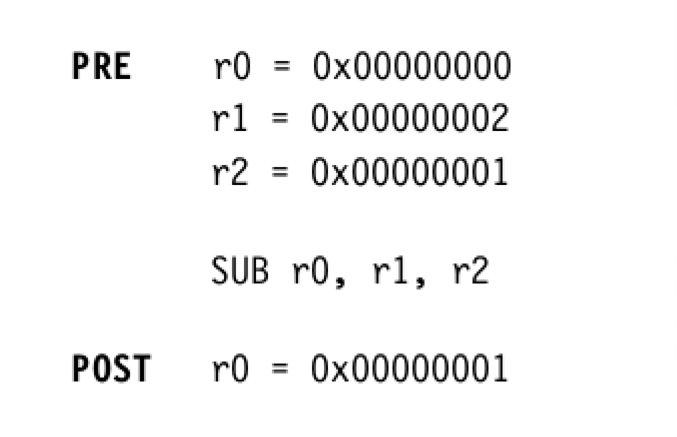
```
PRE     r5 = 5
        r7 = 8
        MOV    r7, r5    ; let r7 = r5
POST    r5 = 5
        r7 = 5
```

Arithmetic Instructions



Syntax: `<instruction>{<cond>}{S} Rd, Rn, N`

ADC	add two 32-bit values and carry	$Rd = Rn + N +$ carry
ADD	add two 32-bit values	$Rd = Rn + N$
RSB	reverse subtract of two 32-bit values	$Rd = N - Rn$
RSC	reverse subtract with carry of two 32-bit values	$Rd = N - Rn -$!(carry flag)
SBC	subtract with carry of two 32-bit values	$Rd = Rn - N -$!(carry flag)
SUB	subtract two 32-bit values	$Rd = Rn - N$

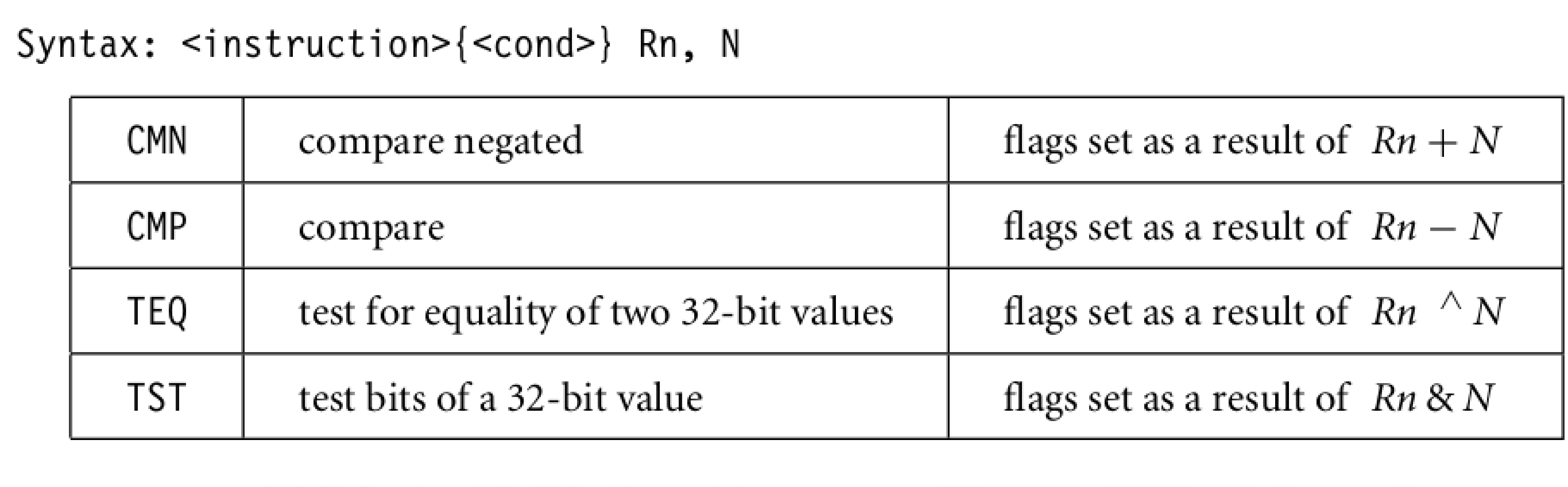


```
PRE     r0 = 0x00000000
        r1 = 0x00000002
        r2 = 0x00000001

        SUB r0, r1, r2

POST    r0 = 0x00000001
```

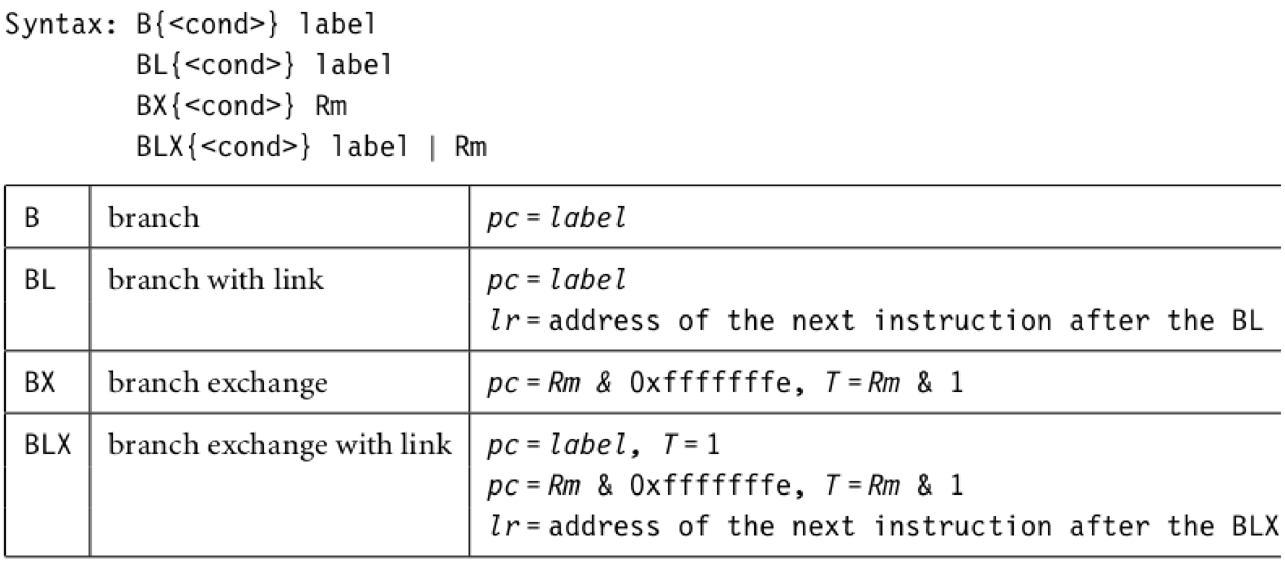
Comparison Instructions



```
Syntax: <instruction>{<cond>} Rn, N
```

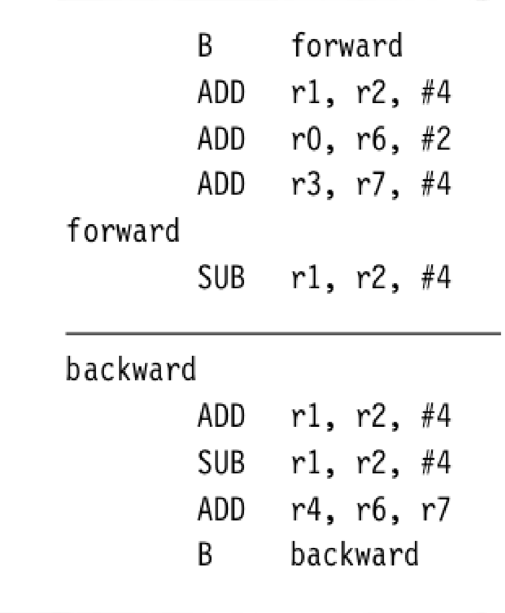
CMN	compare negated	flags set as a result of $Rn + N$
CMP	compare	flags set as a result of $Rn - N$
TEQ	test for equality of two 32-bit values	flags set as a result of Rn ^ N
TST	test bits of a 32-bit value	flags set as a result of $Rn \& N$

Branch Instructions



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

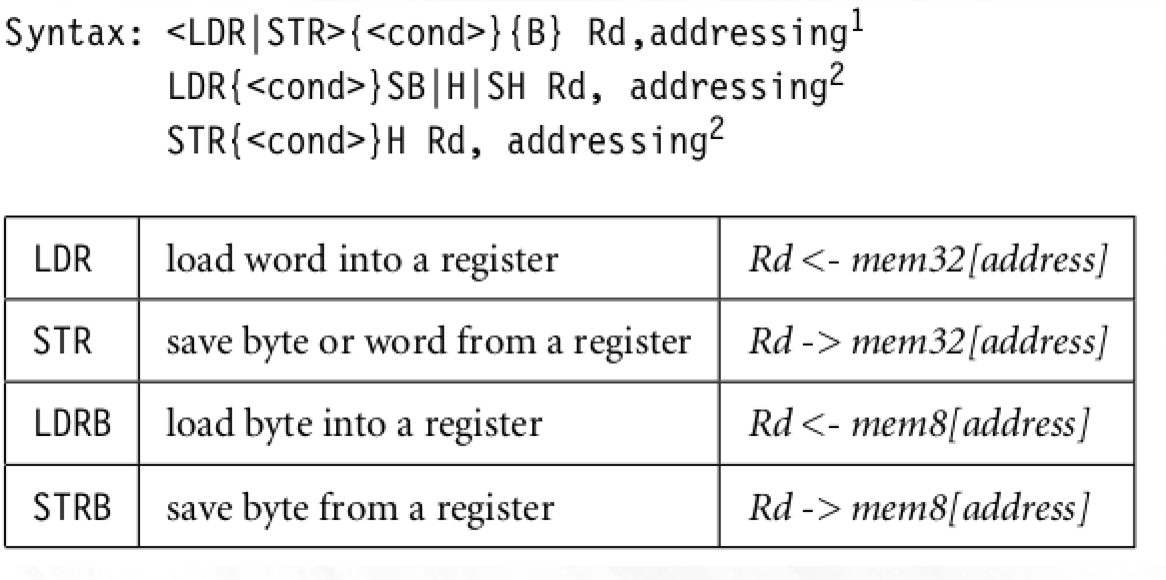
B	branch	*pc* = *label*
BL	branch with link	*pc* = *label* *lr* = address of the next instruction after the BL
BX	branch exchange	*pc* = *Rm* & 0xfffffffe, *T* = *Rm* & 1
BLX	branch exchange with link	*pc* = *label*, *T* = 1 *pc* = *Rm* & 0xfffffffe, *T* = *Rm* & 1 *lr* = address of the next instruction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



```
        B     forward
        ADD   r1, r2, #4
        ADD   r0, r6, #2
        ADD   r3, r7, #4
forward
        SUB   r1, r2, #4
```

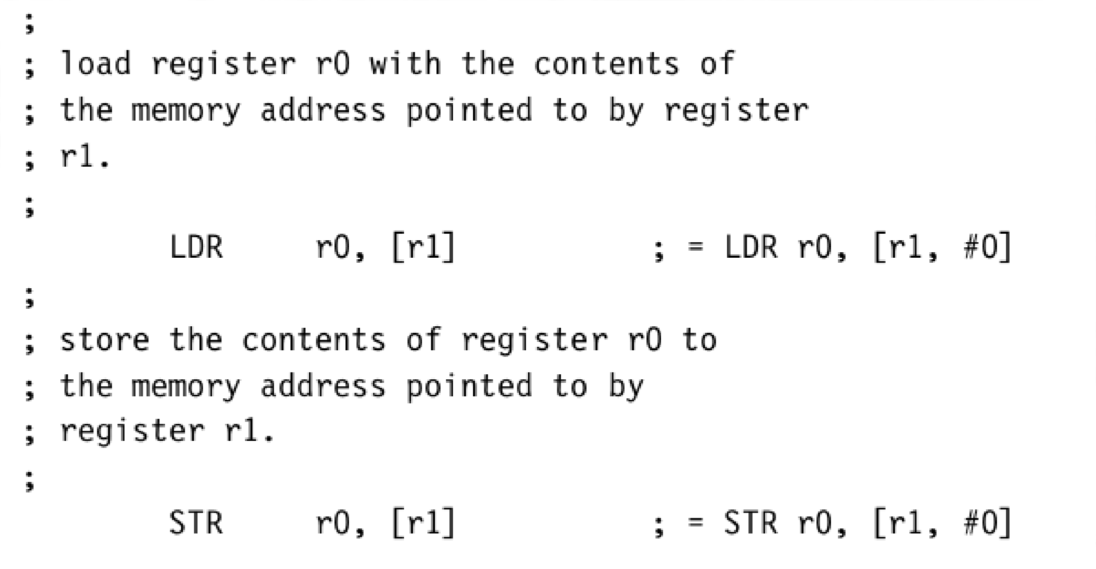
```
backward
        ADD   r1, r2, #4
        SUB   r1, r2, #4
        ADD   r4, r6, r7
        B     backward
```

Single-Register Transfer



```
Syntax: <LDR|STR>{<cond>}{B} Rd,addressing[1]
        LDR{<cond>}SB|H|SH Rd, addressing[2]
        STR{<cond>}H Rd, addressing[2]
```

LDR	load word into a register	*Rd <- mem32[address]*
STR	save byte or word from a register	*Rd -> mem32[address]*
LDRB	load byte into a register	*Rd <- mem8[address]*
STRB	save byte from a register	*Rd -> mem8[address]*



```
;
; load register r0 with the contents of
; the memory address pointed to by register
; r1.
;
        LDR     r0, [r1]            ; = LDR r0, [r1, #0]
;
; store the contents of register r0 to
; the memory address pointed to by
; register r1.
;
        STR     r0, [r1]            ; = STR r0, [r1, #0]
```

Introduction to the Thumb Instruction Set

- Thumb encodes a subset of the 32-bit ARM instructions into a 16-bit instruction set space
- Since Thumb has higher performance than ARM on a processor with a 16-bit data bus
- but lower performance than ARM on a 32-bit data bus, use Thumb for memory-constrained systems
- Thumb has higher code density

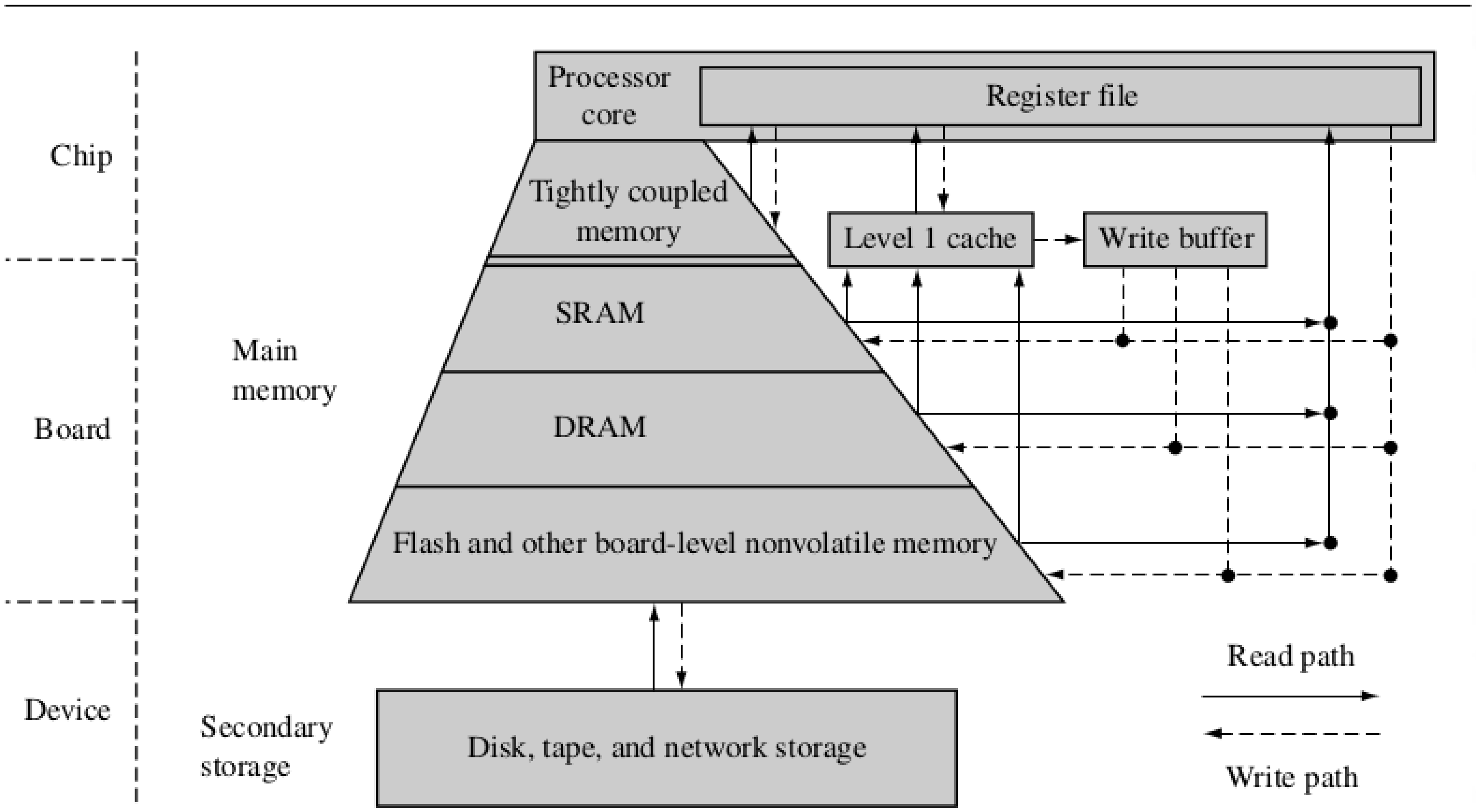
Caches

- Cache
  - a small, fast array of memory placed between the processor core and main memory that stores portions of recently referenced main memory.
- Often used with a cache is a write buffer
  - a very small first-in-first-out (FIFO) memory placed between the processor core and main memory.

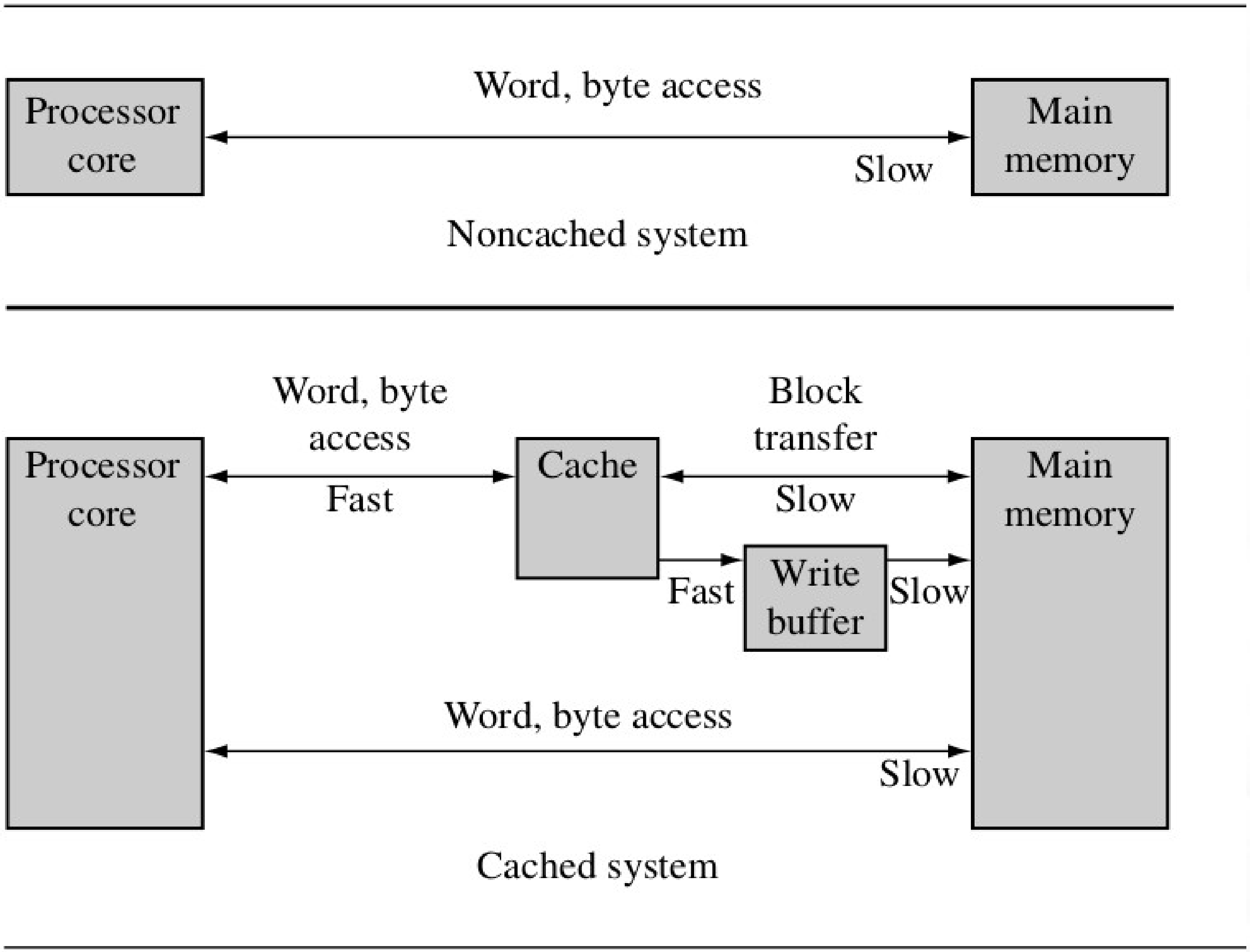
Advantage and Drawbacks

- Advantage
  - Help you create programs that run faster on a specific ARM core
- Drawbacks
  - The difficulty of determining the execution time of a program

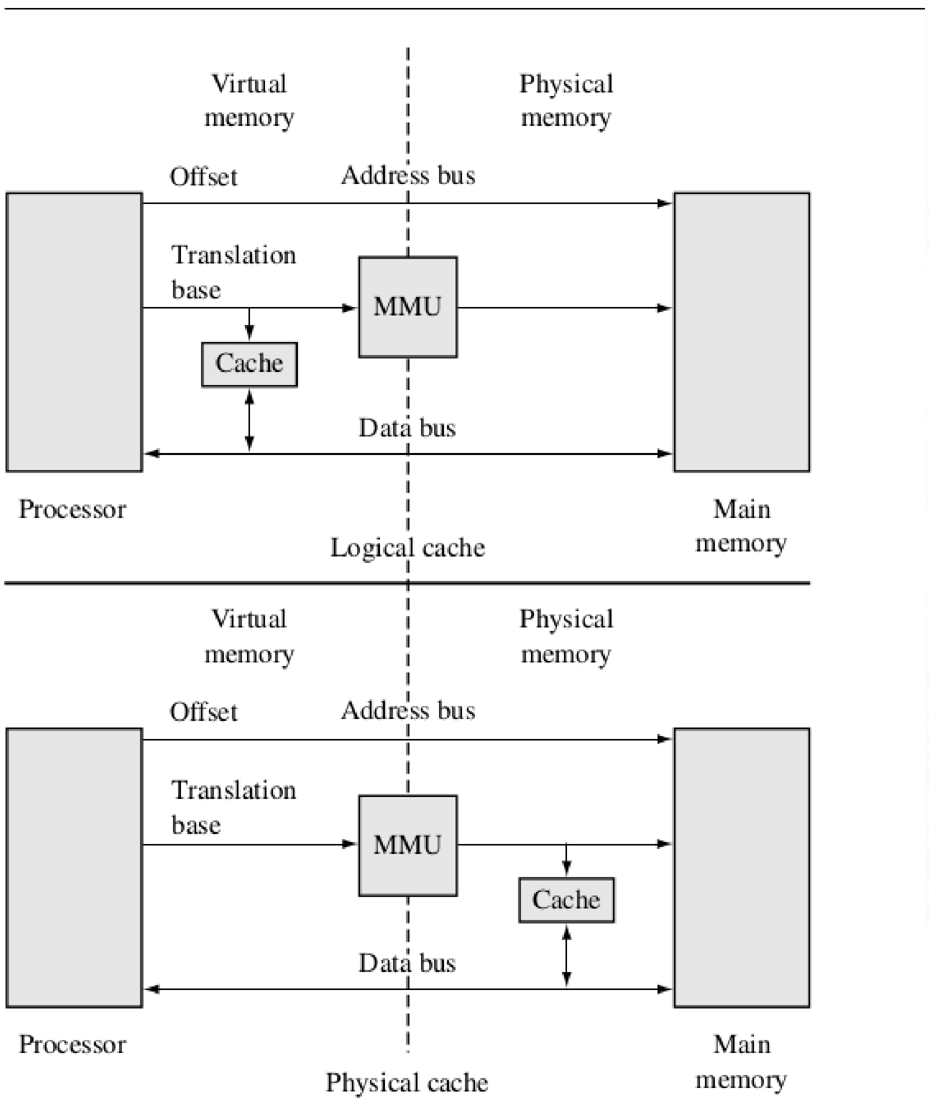
The Memory Hierarchy and Cache Memory



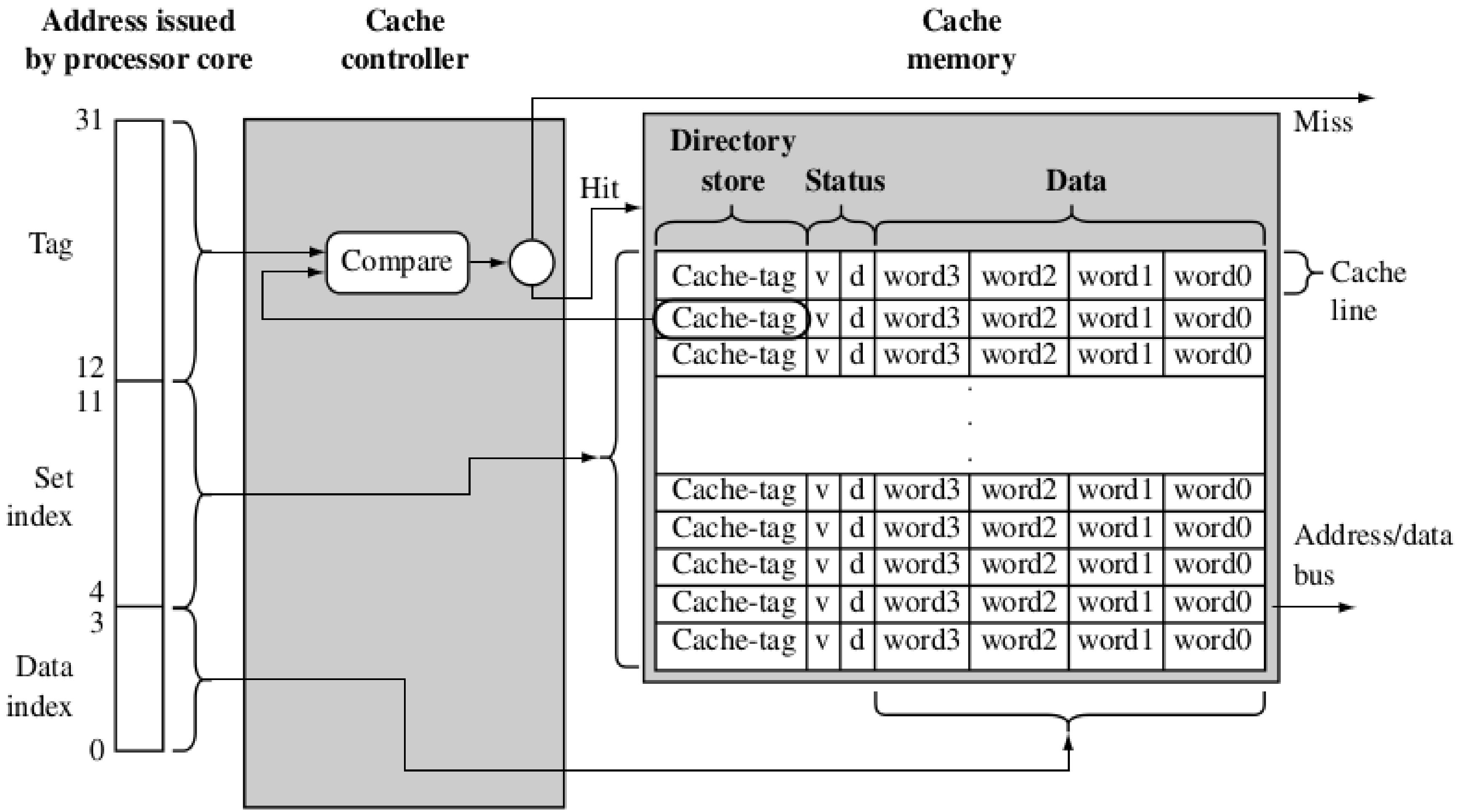
Cache, processor core and main memory



Logical and physical caches



Cache Architecture



Address issued by processor core

Cache controller

Cache memory

31

Tag

12

11

Set index

4

3

Data index

0

Compare

Hit

Miss

Directory store

Status

Data

Cache line

Address/data bus

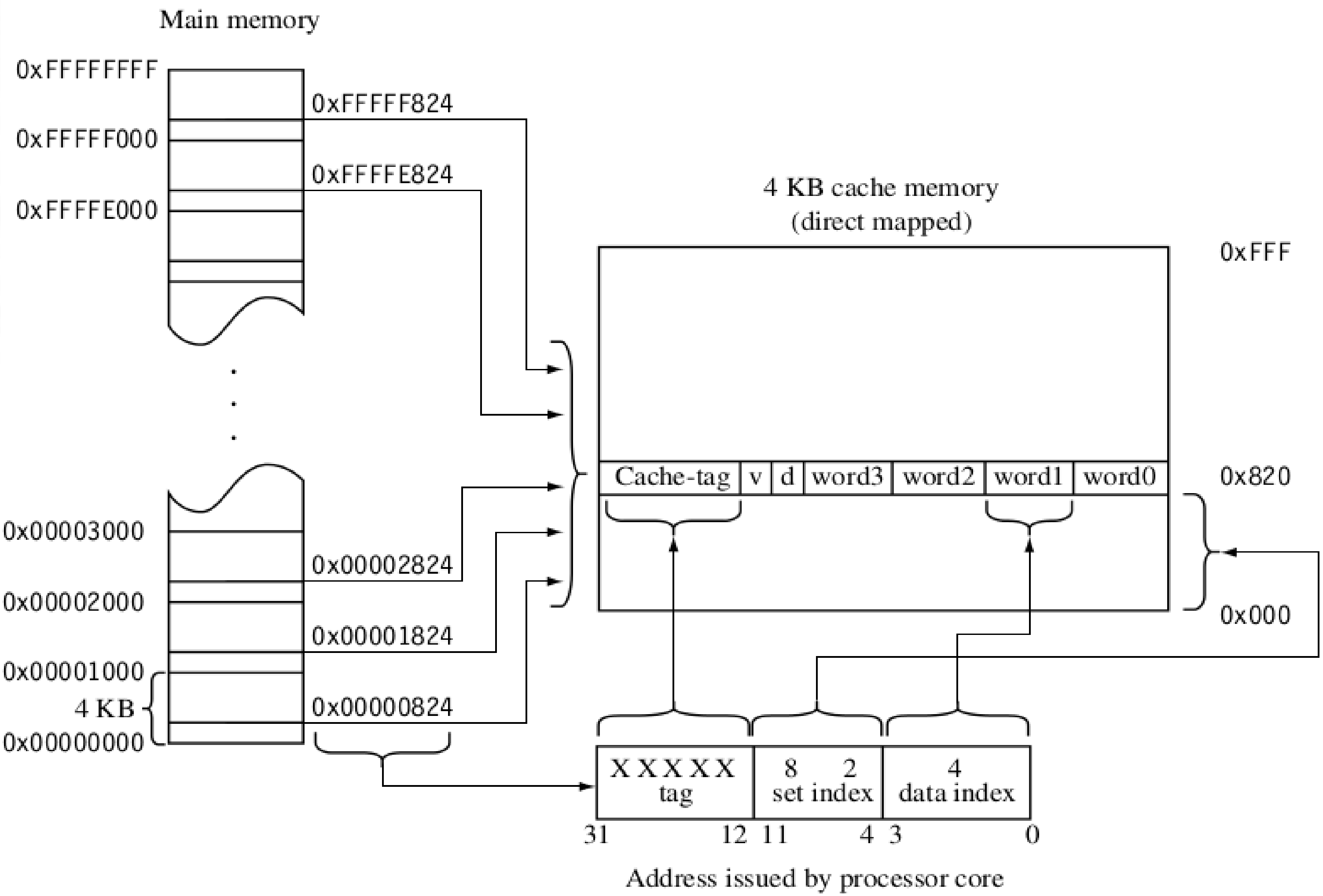
Cache-tag	v	d	word3	word2	word1	word0
Cache-tag	v	d	word3	word2	word1	word0
Cache-tag	v	d	word3	word2	word1	word0
.						
.						
.						
Cache-tag	v	d	word3	word2	word1	word0
Cache-tag	v	d	word3	word2	word1	word0
Cache-tag	v	d	word3	word2	word1	word0
Cache-tag	v	d	word3	word2	word1	word0
Cache-tag	v	d	word3	word2	word1	word0

A 4 KB cache consisting of 256 cache lines of four 32-bit words

Basic Operation of a Cache Controller

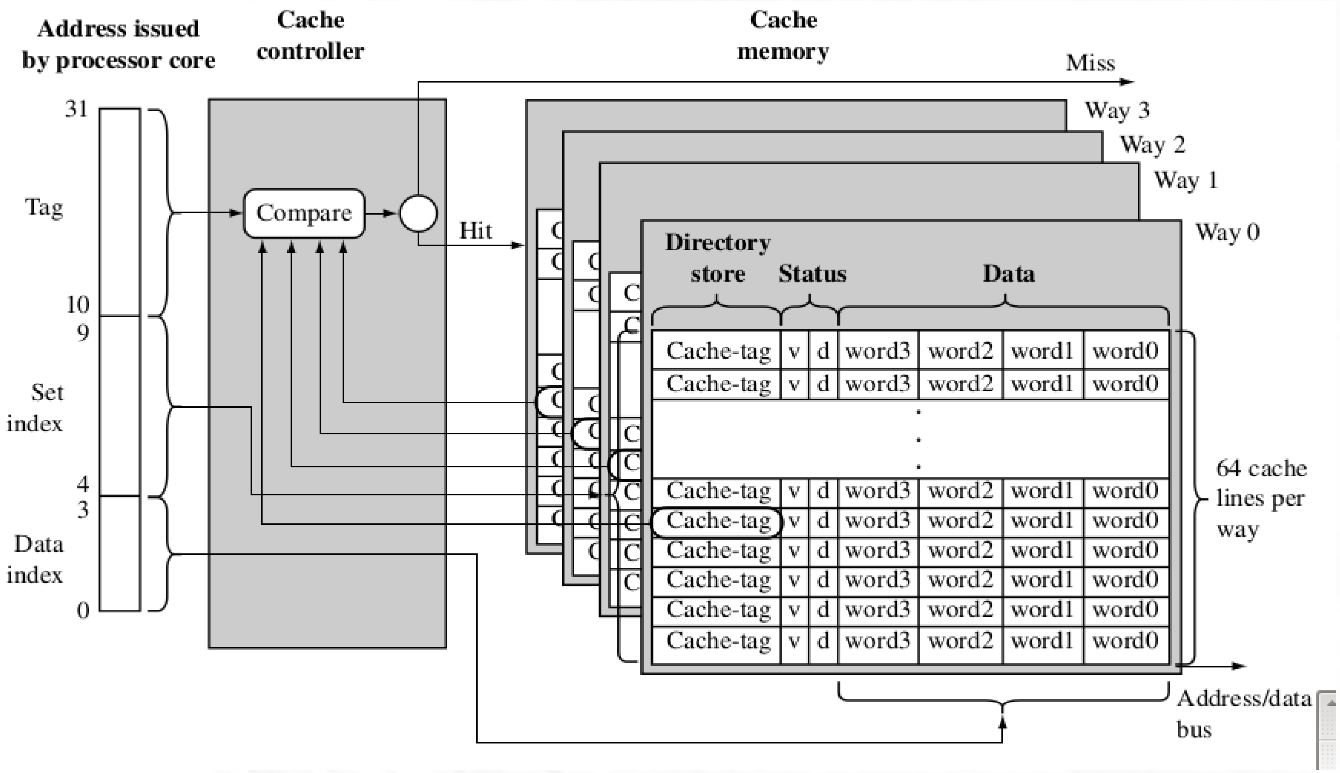
- The cache controller is hardware that copies code or data from main memory to cache memory automatically.
-

The Relationship between Cache and Main Memory



How main memory maps to a direct-mapped cache

Ways



A 4 KB, four-way set associative cache. The cache has 256 total cache lines, which are separated into four ways, each containing 64 cache lines. The cache line contains four words

Cache Policy

- Write Policy
  - Writethrough
  - Writeback

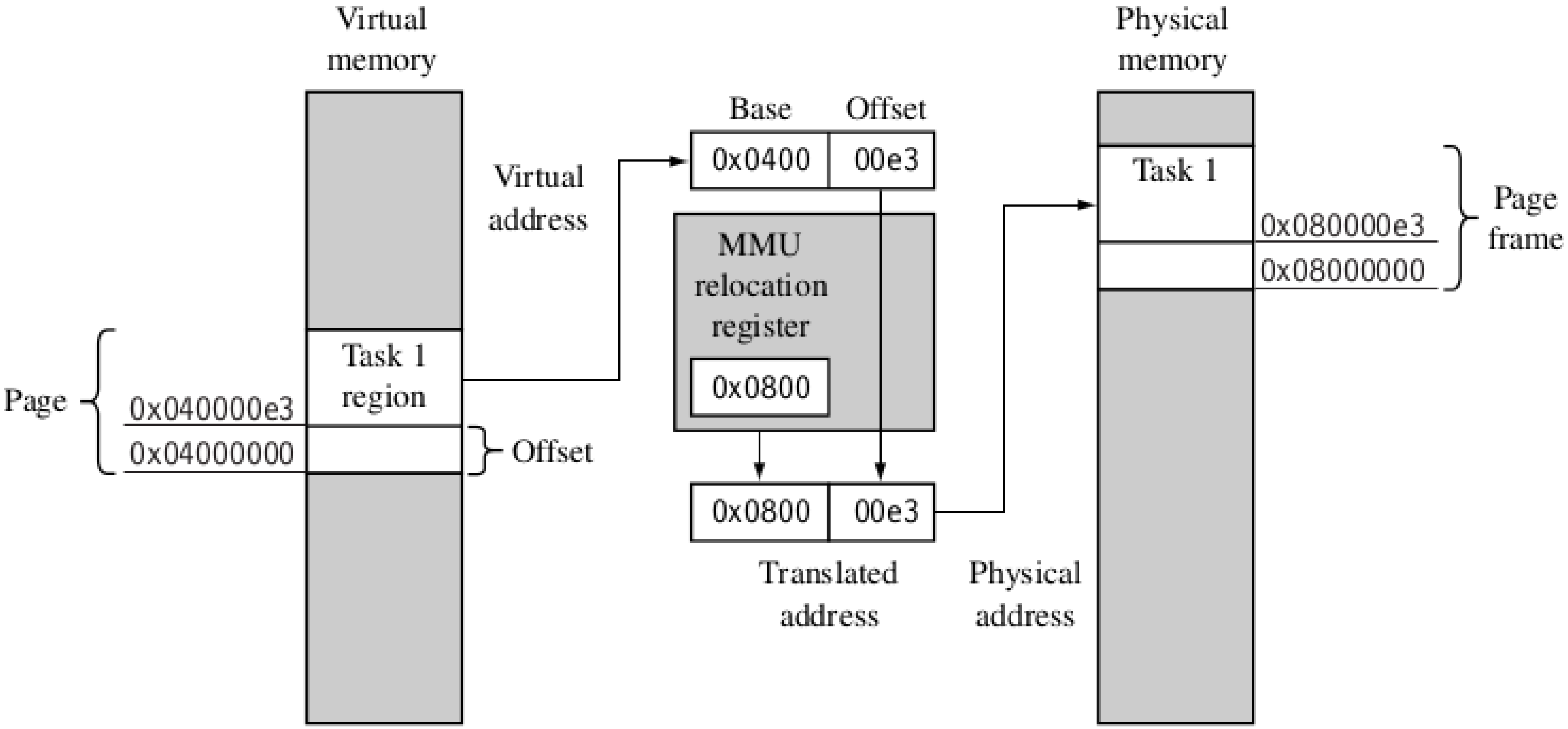
Flushing and Cleaning Cache Memory

- Flush a cache
  - To clear it of any stored data
- Clean a cache
  - To force a write of dirty cache lines from the cache out to main memory and clear the dirty bits in the cache line

Memory Management Units

- virtual memory
  - an additional memory space that is independent of the physical memory attached to the system
- **Virtual addresses** are assigned by the compiler and linker when locating a program in memory
- **Physical addresses** are used to access the actual hardware components of main memory where the programs are physically located.

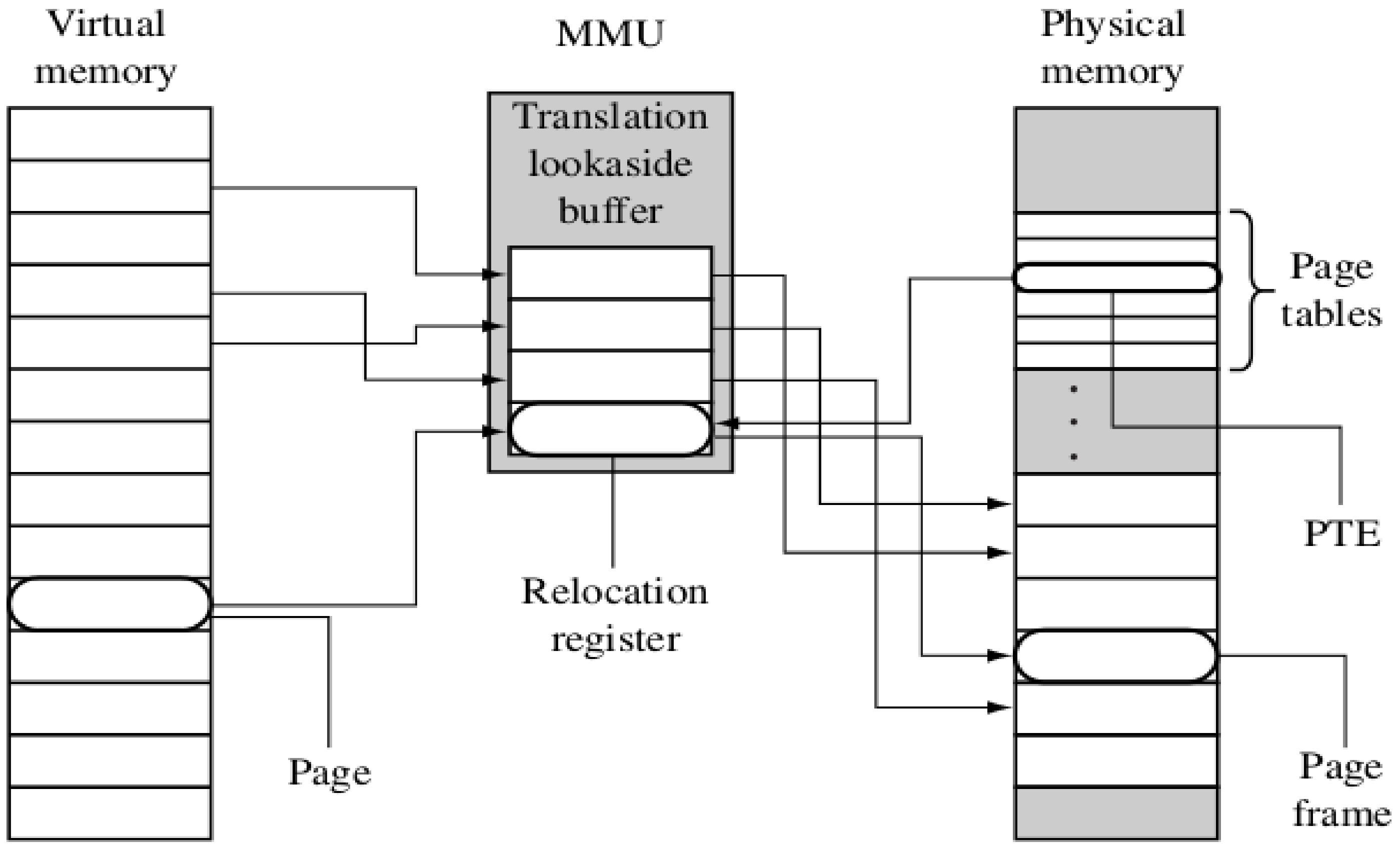
How Virtual Memory Works



Mapping a task in virtual memory to physical memory using a relocation register

TLB

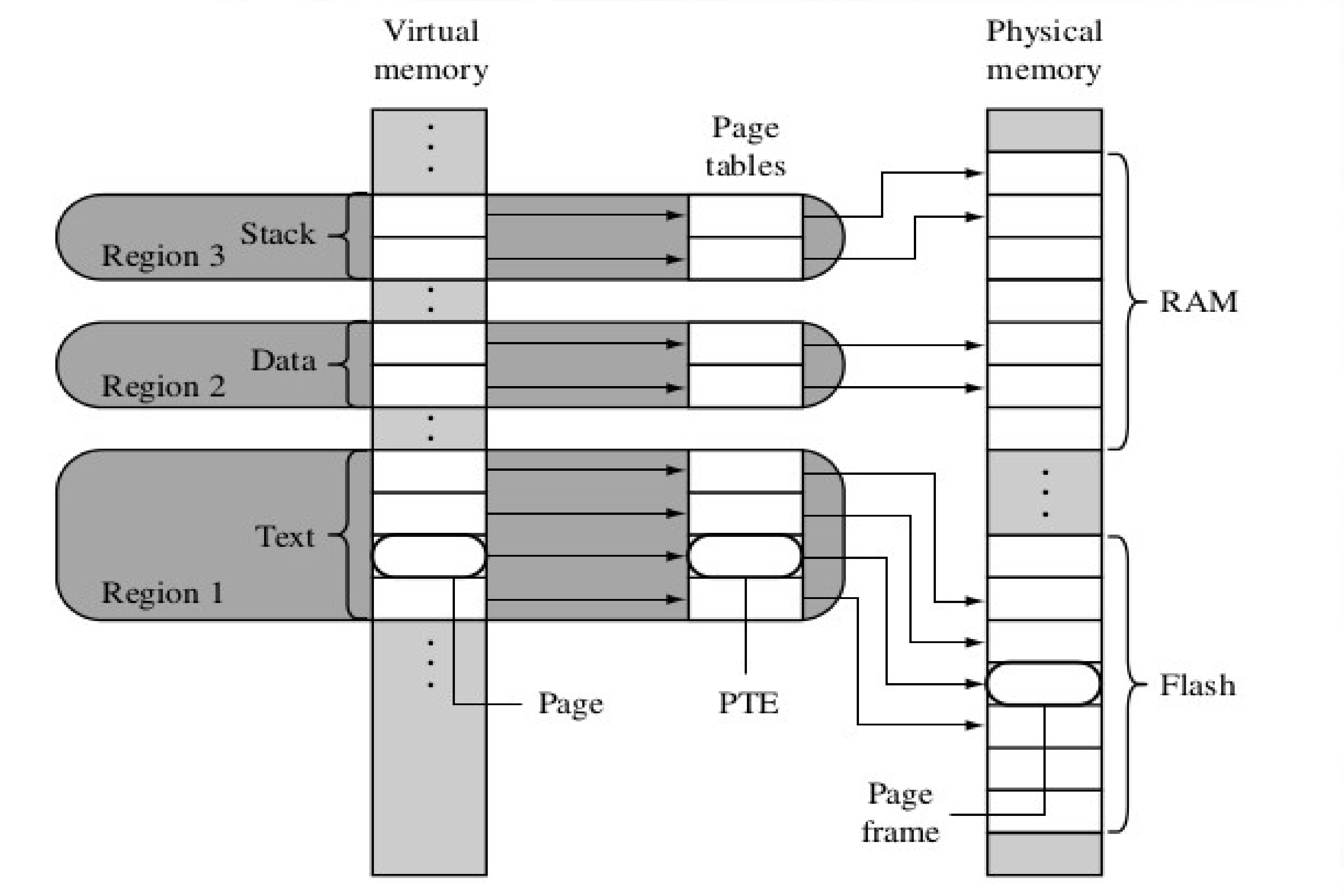
- TLB : Translation Lookaside Buffer



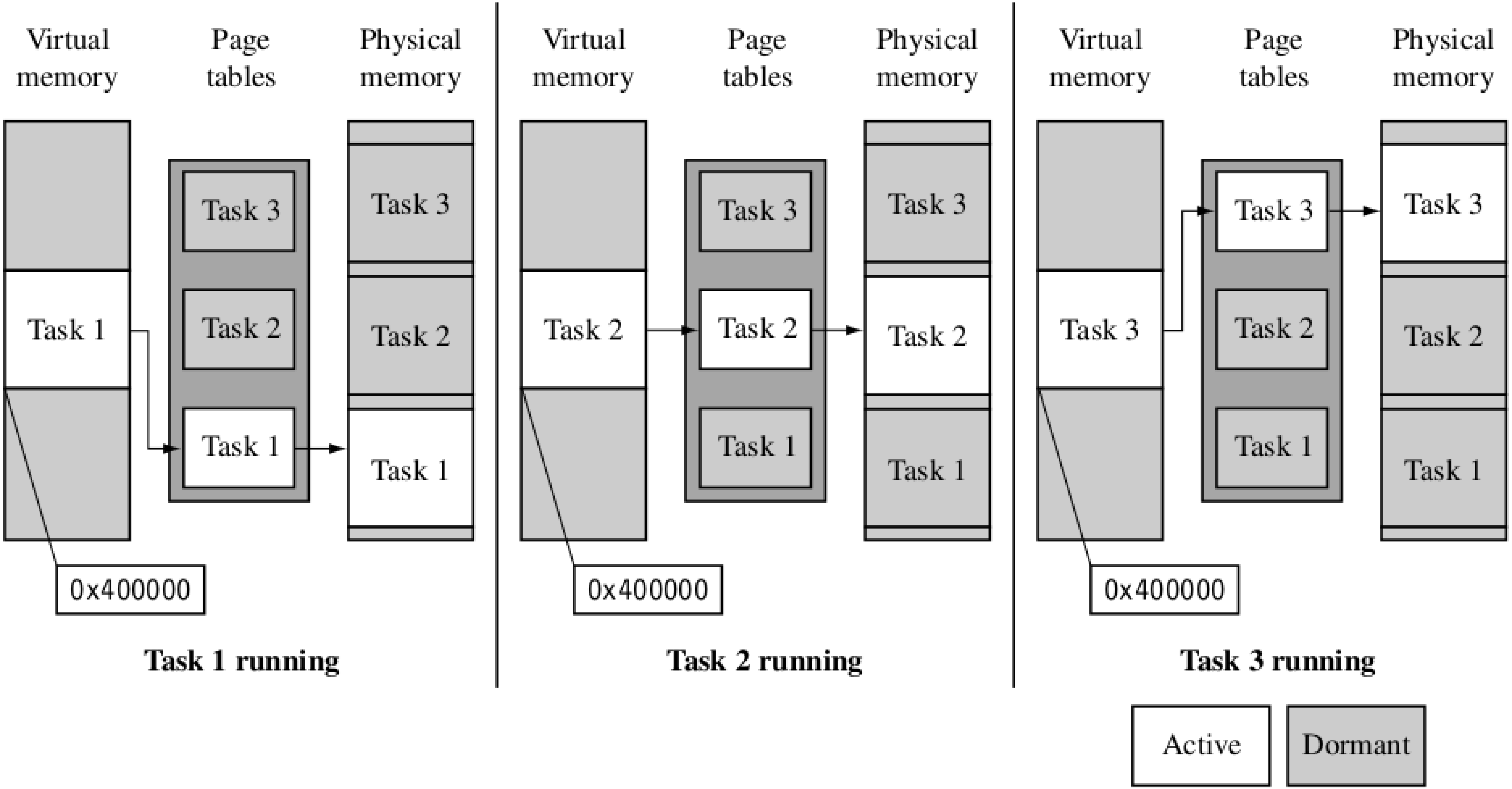
Page Tables

- the MMU uses tables in main memory to store the data describing the virtual memory maps used in the system
- PTE - page table entry

Example : Signal Task and MMU

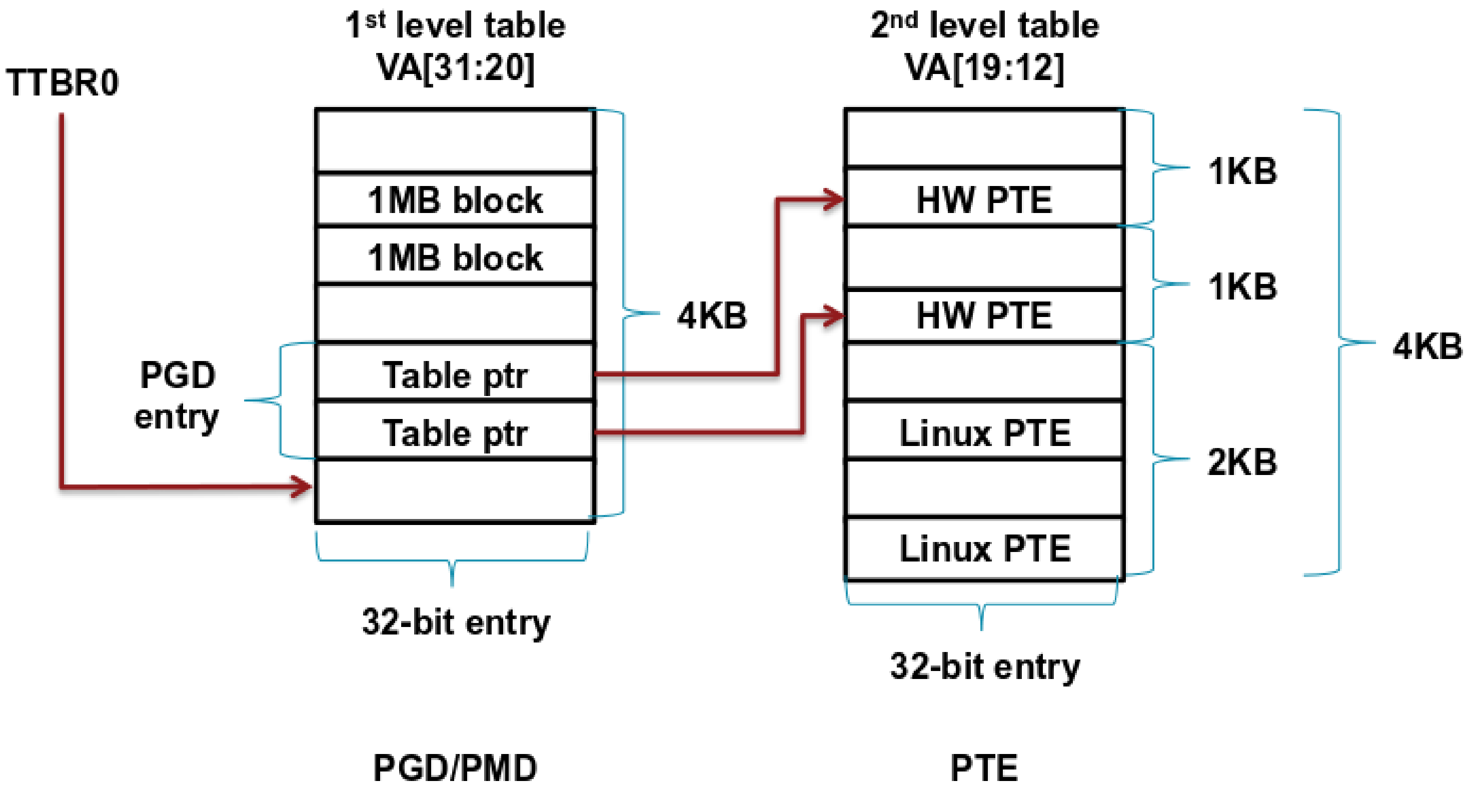


Example : Multi Task and MMU

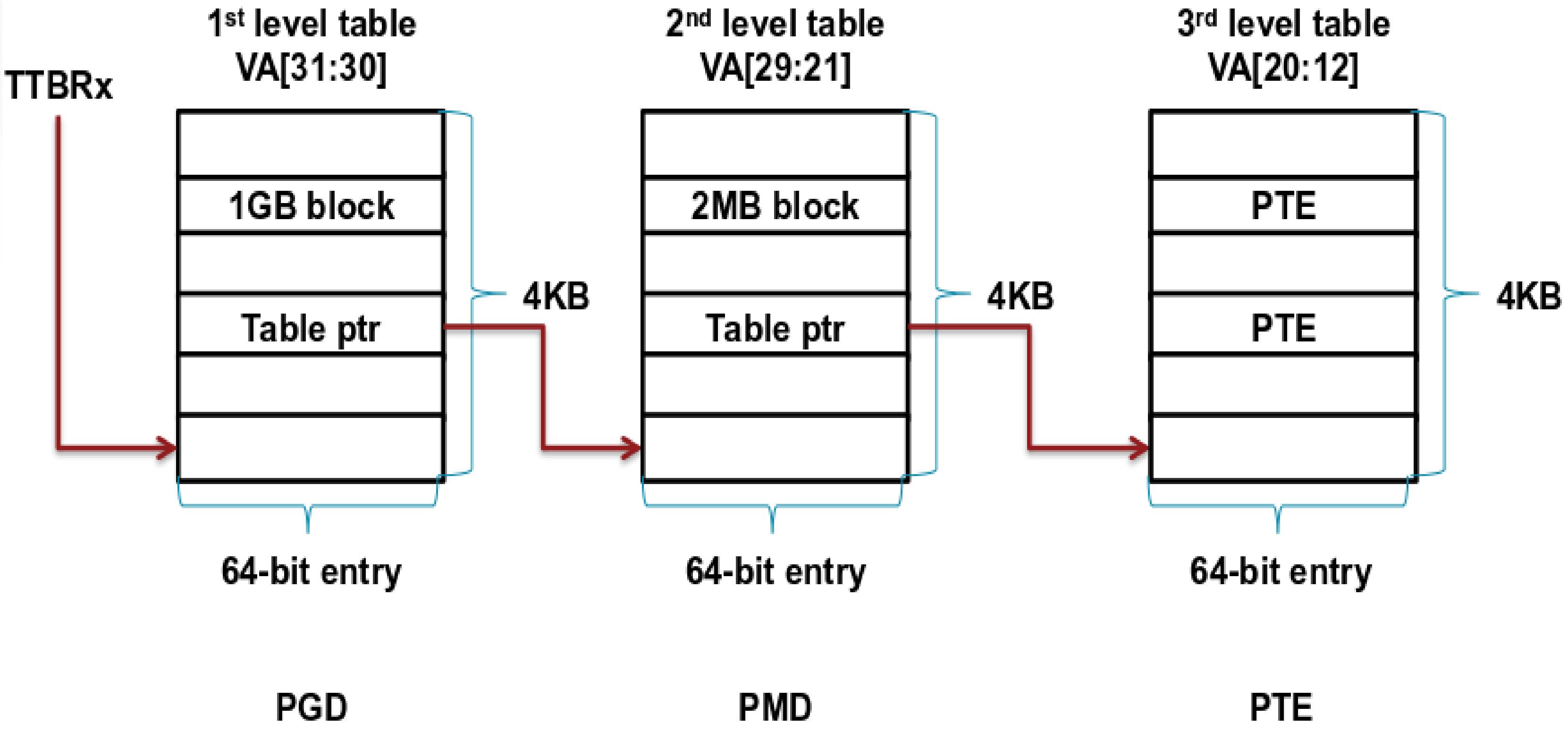


LPAE

Classic ARM MMU Limitations



LPAE


TTBRx
1st level table
VA[31:30]
2nd level table
VA[29:21]
3rd level table
VA[20:12]
1GB block
Table ptr
2MB block
Table ptr
PTE
PTE
4KB
4KB
4KB
64-bit entry
64-bit entry
64-bit entry
PGD
PMD
PTE