ARM Processor Architecture

Humayun Kabir
Professor, CS, Vancouver Island University, BC, Canada

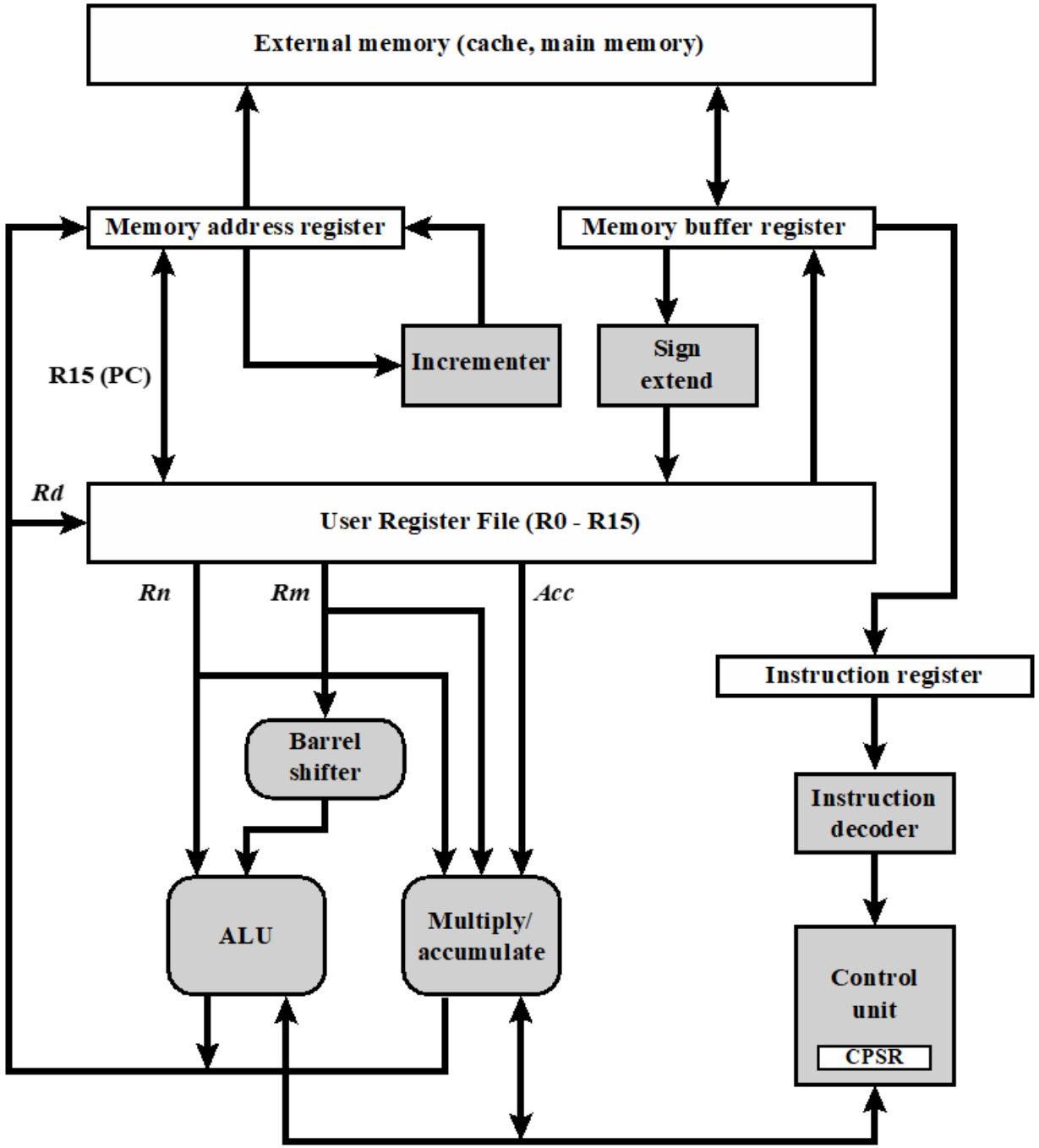
ARM Processor Architecture: Outline

- ARM7 Processor Organization
- ARM7 Registers
- ARM7 Modes, Status, and Control Flags
- ARM8 Registers
- ARM8 Instruction Set Architecture
  - Data Transfer: Load and Store
  - Data Processing: Arithmetic, Logical, and Shift
  - Branch: Conditional and Unconditional

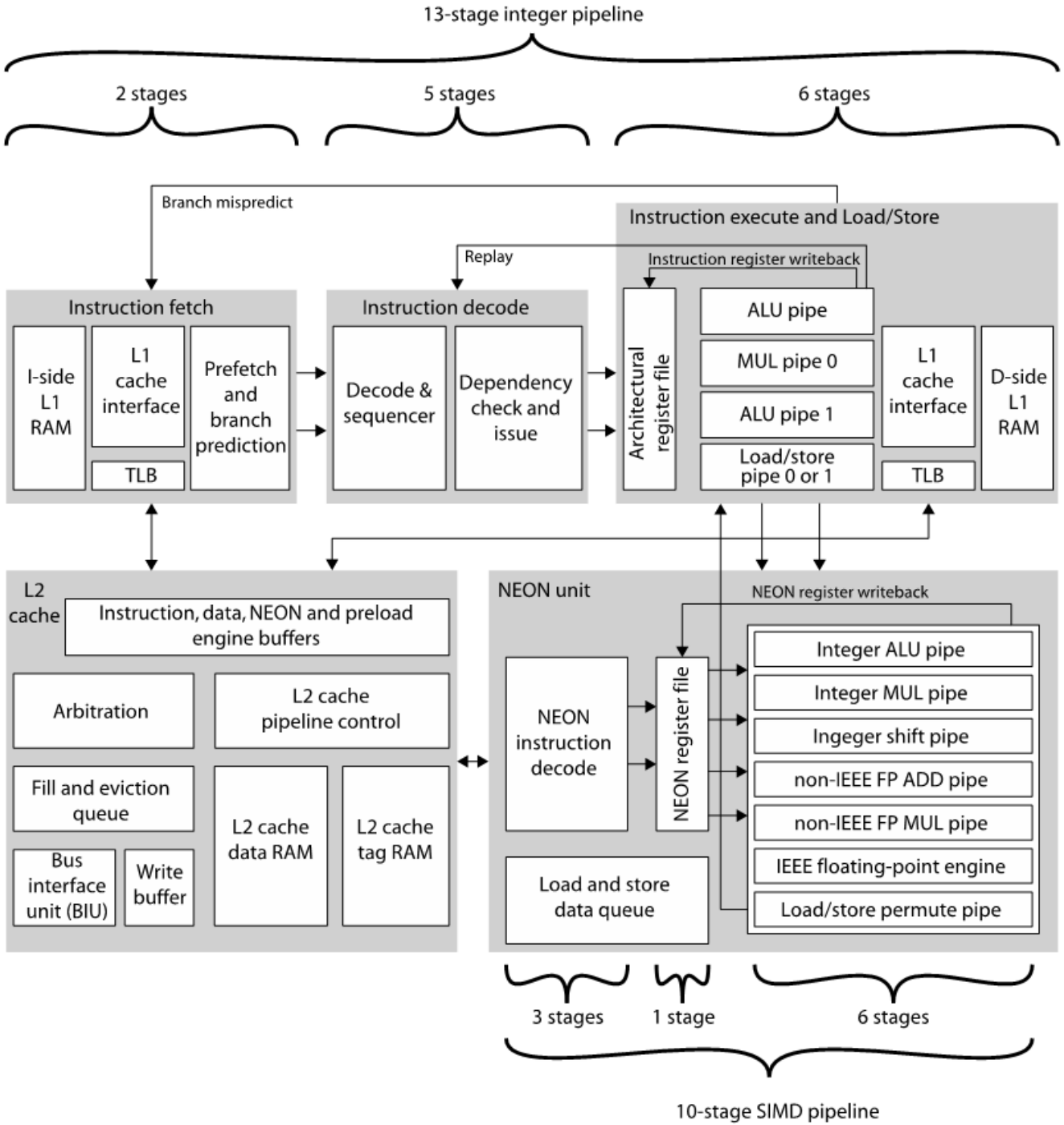
ARM Processor Architecture

- ARM, primarily a RISC system with the following attributes:
  - Moderate array of uniform registers
  - A load/store model of data processing in which operations only perform on operands in registers and not directly in memory
  - A uniform fixed-length instruction of 32 bits for the standard set and 16 bits for the Thumb instruction set
  - Separate arithmetic logic unit (ALU), Multiply, and Shifter units
  - A small number of addressing modes with all load/store addresses determined from registers and instruction fields
  - Auto-increment and auto-decrement addressing modes are used to improve the operation of program loops

Simplified ARM7 Processor Organization



ARM Cortex-A8 Processor Organization



ARM Processor Modes

ARM supports 1 **non-privileged** (user mode) and 6 **privileged** modes.

User mode: Most *application programs* execute in this mode, the program in user mode is unable to access protected system resources or to change mode, other than by causing an exception to occur.

Privileged Modes: These modes are used to run system software, the program in any privileged mode is able to access *protected system resources* and to *change mode*. Privileged modes consist of 1 **system mode** and 5 **exception modes**. A program enters into an exception mode when corresponding *exception* occurs.

ARM Processor Modes

System Mode: This mode is not entered by any exception and uses the same register set available in User mode. The System mode is used for running certain *privileged operating system tasks*. System mode tasks may be interrupted by any of the five exception categories.

ARM Processor Modes

Exception Modes: The exception modes have full access to system resources and can change modes freely. These are entered when specific *exceptions* occur. Each of these modes has some dedicated registers that substitute for some of the user mode registers, and which are used to avoid corrupting User mode state information when the exception occurs.

ARM Exception Modes

Supervisor mode: It is entered when the processor encounters a *software interrupt* instruction. Software interrupts are a standard way to invoke operating system services on ARM.

Abort mode: Entered in response to *memory faults*.

Undefined mode: Entered when the processor attempts to execute an instruction that is *not supported* either by the main integer core or by one of the coprocessors.

ARM Exception Modes

Fast interrupt mode: Entered whenever the processor receives an interrupt signal from the *designated fast interrupt source*. A fast interrupt cannot be interrupted, but a fast interrupt may interrupt a normal interrupt.

Interrupt mode: Entered whenever the processor receives an interrupt signal from any *interrupt source* (other than fast interrupt). An interrupt may only be interrupted by a fast interrupt.

ARM7 or ARM8 32-bit Architecture Register Set

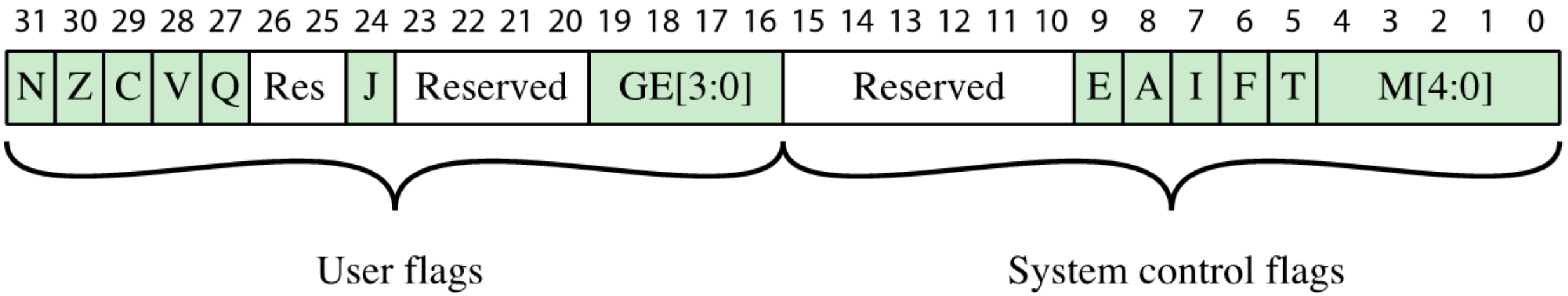
Modes						
	Privileged modes					
		Exception modes				
User	**System**	**Supervisor**	**Abort**	**Undefined**	**Interrupt**	**Fast Interrupt**
R0	R0	R0	R0	R0	R0	R0
R1	R1	R1	R1	R1	R1	R1
R2	R2	R2	R2	R2	R2	R2
R3	R3	R3	R3	R3	R3	R3
R4	R4	R4	R4	R4	R4	R4
R5	R5	R5	R5	R5	R5	R5
R6	R6	R6	R6	R6	R6	R6
R7	R7	R7	R7	R7	R7	R7
R8	R8	R8	R8	R8	R8	R8_fiq
R9	R9	R9	R9	R9	R9	R9_fiq
R10	R10	R10	R10	R10	R10	R10_fiq
R11	R11	R11	R11	R11	R11	R11_fiq
R12	R12	R12	R12	R12	R12	R12_fiq
R13 (SP)	R13 (SP)	R13_svc	R13_abt	R13_und	R13_irq	R13_fiq
R14 (LR)	R14 (LR)	R14_svc	R14_abt	R14_und	R14_irq	R14_fiq
R15 (PC)	R15 (PC)	R15 (PC)	R15 (PC)	R15 (PC)	R15 (PC)	R15 (PC)
CPSR	CPSR	CPSR	CPSR	CPSR	CPSR	CPSR
		SPSR_svc	SPSR_abt	SPSR_und	SPSR_irq	SPSR_fiq

Shading indicates that the normal register used by User or System mode has been replaced by an alternative register specific to the exception mode.

SP = stack pointer
LR = link register
PC = program counter

CPSR = current program status register
SPSR = saved program status register

ARM CPSR and SPSR Register Format



ARM Condition Code Flag bits in CPSR

N bit: indicates that the computation result is negative.
Z bit: indicates that the computation result is zero.
C bit: indicates that carry out occurred in computation.
V bit: indicates that overflow occurred in computation.

ARM Special Flag bits in CPSR

Q bit: used to indicate whether overflow and/or saturation has occurred in some Single Instruction Multiple Data (SIMD) instructions.

J bit: indicates the use of special 8-bit instructions, known as Jazelle instructions.

GE[3:0] bits: SIMD instructions use bits [19:16] as Greater than or Equal (GE) flags for individual bytes or halfwords of the result.

ARM System Control Flag bits in CPSR

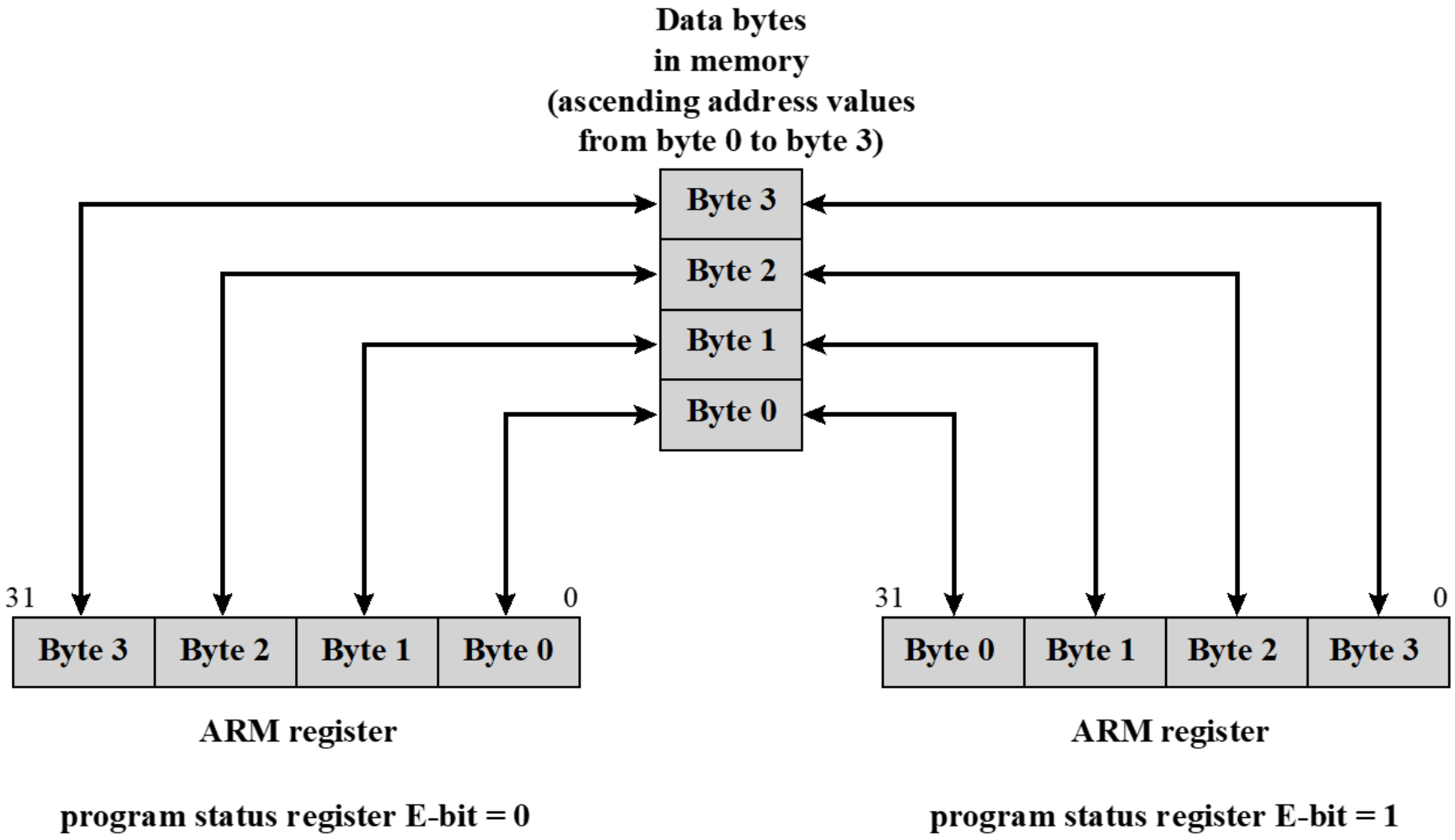
E bit: Controls load and store endianness for data; ignored for instruction fetches.
Interrupt disable bits: When set, **A** bit disables imprecise data aborts; **I** bit disables IRQ interrupts; and **F** bit disables FIQ interrupts.
T bit: Indicates whether instructions should be interpreted as normal ARM instructions or Thumb instructions.
Mode bits [4:0]: Indicates the processor mode.

These system control bits can only be altered in the privileged mode.

ARM Processor Mode Bits

M[4:0]	Mode	Accessible registers
0b10000	User	PC, R14 to R0, CPSR
0b10001	FIQ	PC, R14_fiq to R8_fiq, R7 to R0, CPSR, SPSR_fiq
0b10010	IRQ	PC, R14_irq, R13_irq, R12 to R0, CPSR, SPSR_irq
0b10011	Supervisor	PC, R14_svc, R13_svc, R12 to R0, CPSR, SPSR_svc
0b10111	Abort	PC, R14_abt, R13_abt, R12 to R0, CPSR, SPSR_abt
0b11011	Undefined	PC, R14_und, R13_und, R12 to R0, CPSR, SPSR_und
0b11111	System	PC, R14 to R0, CPSR (ARM architecture v4 and above)

ARM Processor Big and Little Endian Support



ARM Condition Codes based on Condition Code Flags in CPSR

Code	Symbol	Condition Tested	Comment
0000	EQ	Z = 1	Equal
0001	NE	Z = 0	Not equal
0010	CS/HS	C = 1	Carry set/unsigned higher or same
0011	CC/LO	C = 0	Carry clear/unsigned lower
0100	MI	N = 1	Minus/negative
0101	PL	N = 0	Plus/positive or zero
0110	VS	V = 1	Overflow
0111	VC	V = 0	No overflow
1000	HI	C = 1 AND Z = 0	Unsigned higher
1001	LS	C = 0 OR Z = 1	Unsigned lower or same
1010	GE	N = V [(N = 1 AND V = 1) OR (N = 0 AND V = 0)]	Signed greater than or equal
1011	LT	N ≠ V [(N = 1 AND V = 0) OR (N = 0 AND V = 1)]	Signed less than
1100	GT	(Z = 0) AND (N = V)	Signed greater than
1101	LE	(Z = 1) OR (N ≠ V)	Signed less than or equal
1110	AL	—	Always (unconditional)
1111	—	—	This instruction can only be execut unconditionally

ARM Processor 64-bit Architecture Double Word Register Set

Name	Register number	Usage	Preserved on call?
X0–X7	0–7	Arguments/Results	no
X8	8	Indirect result location register	no
X9–X15	9–15	Temporaries	no
X16 (IP0)	16	May be used by linker as a scratch register; other times used as temporary register	no
X17 (IP1)	17	May be used by linker as a scratch register; other times used as temporary register	no
X18	18	Platform register for platform independent code; otherwise a temporary register	no
X19–X27	19–27	Saved	yes
X28 (SP)	28	Stack Pointer	yes
X29 (FP)	29	Frame Pointer	yes
X30 (LR)	30	Link Register (return address)	yes
XZR	31	The constant value 0	n.a.

ARM Processor 64-bit Architecture Double Word Register Set

X0 – X7: Arguments or Results Registers
Arguments to a function are passed through x0 – x7 registers.
Results from a function are returned through x0 – x7 registers.

X8: Indirect Result Location Register
Results from a function is usually returned from a function to its caller through x0 to x7 registers if the function is returning directly to the caller.
There are some situations where a function does not call another function directly and the called function also does not return to the caller directly. For example, a user mode function calls a kernel function indirectly using a syscall and the kernel function does not return to the user mode function. User mode function passes a syscall number using x8 register to a trap handler. Trap handler calls an appropriate kernel function based on the syscall number. The kernel function returns the result to the user function using x8 register.

ARM Processor 64-bit Architecture Double Word Register Set

X9 - X15: Temporary Registers
Functions are free to use these registers as temporary or scratch registers. Functions do not need to save and retrieve the current value of these registers before use and before return respectively.

X16 and X17: Intra-Procedure Call Scratch Registers
Linker often inserts codes before and after function codes in order to facilitate a function call, which are called prologue and epilogue respectively.
Registers x16 and x17 can be used by the linker as the scratch registers in both prologue and epilogue codes.
Regular function codes should avoid using both x16 and x17.
If a regular function uses x16 and x17 as the temporary registers, it should be aware of the fact that the value it writes on these registers might not be seen by its called functions since the linker might have used them as the scratch registers and modifies their values in prologue or epilogue codes.

ARM Processor 64-bit Architecture Double Word Register Set

X18: Platform Register
Register x18 is reserved for individual platform or OS specific codes in order to use it in platform specific way. Platform independent codes must avoid using it.
If a platform chooses not to use x18 as its platform specific special register, platform independent codes can use x18 register as a temporary register.

X19 – X27: Saved Register
A function must save the current value of these registers before using them and restore the saved value before returning. It guarantees that the values in these registers are preserved across multiple function calls.

ARM Processor 64-bit Architecture Word Register Set

- **PC** is **not** a **general purpose register** in ARM64.

- ARM64 also supports **32-bit word registers** and are represented by ***Wn*** instead of ***Xn*** in assembly code.
  - Stack pointer ***sp*** (64-bit) is represented by ***wsp*** (32-bit).
  - Zero register ***xzr*** (64-bit) is represented by ***wzr*** (32-bit).

ARM Processor 64-bit Architecture Word Register Set

- ARM64 assembly language *overloads* instruction mnemonics, and distinguishes between the different forms of an instruction based on the operand register names.
- For example the ADD instructions below all have different opcodes, but the programmer only has to remember one mnemonic and the assembler automatically chooses the correct opcode based on the operands.

```
ADD W0, W1, W2          // add 32-bit register
ADD X0, X1, X2          // add 64-bit register
ADD X0, X1, #42         // add 64-bit immediate
```

ARM8 A64 Data Transfer Instructions

Load and Store Instructions		
LDP	rt, rt2, [addr]	$rt2{:}rt = [addr]_{2N}$
LDPSW	Xt, Xt2, [addr]	$Xt = [addr]^{\pm}_{32}$; $Xt2 = [addr+4]^{\pm}_{32}$
LD{U}R	rt, [addr]	$rt = [addr]_{N}$
LD{U}R{B,H}	Wt, [addr]	$Wt = [addr]^{\emptyset}_{N}$
LD{U}RS{B,H}	rt, [addr]	$rt = [addr]^{\pm}_{N}$
LD{U}RSW	Xt, [addr]	$Xt = [addr]^{\pm}_{32}$
PRFM	prfop, addr	Prefetch(addr, prfop)
STP	rt, rt2, [addr]	$[addr]_{2N} = rt2{:}rt$
ST{U}R	rt, [addr]	$[addr]_{N} = rt$
ST{U}R{B,H}	Wt, [addr]	$[addr]_{N} = Wt_{N0}$

Addressing Modes (addr)		
xxP,LDPSW	[Xn{, #i_{7+s}}]	$addr = Xn + i^{\pm}_{6+s:s}{:}0_{s}$
xxP,LDPSW	[Xn], #i_{7+s}	$addr{=}Xn$; $Xn{+}{=}i^{\pm}_{6+s:s}{:}0_{s}$
xxP,LDPSW	[Xn, #i_{7+s}]!	$Xn{+}{=}i^{\pm}_{6+s:s}{:}0_{s}$; $addr{=}Xn$
xxR*,PRFM	[Xn{, #i_{12+s}}]	$addr = Xn + i^{\emptyset}_{11+s:s}{:}0_{s}$
xxR*	[Xn], #i_{9}	$addr = Xn$; $Xn \mathrel{+}= i^{\pm}$
xxR*	[Xn, #i_{9}]!	$Xn \mathrel{+}= i^{\pm}$; $addr = Xn$
xxR*,PRFM	[Xn,Xm{, LSL #0|s}]	$addr = Xn + Xm \ll s$
xxR*,PRFM	[Xn,Wm,{S,U}XTW{ #0|s}]	$addr = Xn + Wm^{?} \ll s$
xxR*,PRFM	[Xn,Xm,SXTX{ #0|s}]	$addr = Xn + Xm^{\pm} \ll s$
xxUR*,PRFM	[Xn{, #i_{9}}]	$addr = Xn \mathrel{+}= i^{\pm}$
LDR{SW},PRFM	$\pm rel_{21}$	$addr = PC + rel^{\pm}_{20:2}{:}0_{2}$

ARM8 A64 Data Transfer Instructions: Unscaled and Scaled

- Unscaled
  - **LDUR** Wt/Xt, [Xn|SP{, #**simm**}]
  - **STUR** Wt/Xt, [Xn|SP{, #**simm**}]
  - **simm** value range: **-256 ~ 255**
    - NO need to be a multiple of data access size 4 or 8
- Scaled
  - **LDR** Wt/Xt, [Xn|SP{, **#pimm**}]
  - **STR** Wt/Xt, [Xn|SP{, **#pimm**}]
  - **pimm** value range:
    - 32-bit: **0 ~ 16380**, and **pimm % 4 == 0** (is a multiple of 4)
    - 64-bit: **0 ~ 32760**, and **pimm % 8 == 0** (is a multiple of 8)
    - NEED to be a multiple of data access size 4 or 8

ARM8 A64 Data Transfer Instructions: 64-bit Unscaled Load

```
ldur x7, [x3, x2]      //Load x7 with double word at the address (x3 + x2)

ldur x7, [x3, #2]      //Load x7 with double word at the address (x3 + 2)

ldur x7, [x3, x2]!     //Load x7 with double word at the address (x3 + x2), then
                       //store the address in x3, pre-indexed

ldur x9, [x2, #2]!     //Load x9 with double word at the address (x2 + 2), then
                       //store the address in x2, pre-indexed

ldur x7, [x3], #2      //Load x7 with double word at the address in x3 then
                       //increment x3 by 2, post-indexed
```

ARM8 A64 Data Transfer Instructions: 32-bit Unscaled Load

```
ldur w7, [w3, w2]      //Load w7 with word at the address (w3 + w2)

ldur w7, [w3, #2]      //Load w7 with word at the address (w3 + 2)

ldur w7, [w3, w2]!     //Load w7 with word at the address (w3 + w2), then
                       //store the address in w3, pre-indexed

ldur w9, [w2, #2]!     //Load w9 with word at the address (w2 + 2), then
                       //store the address in w2, pre-indexed

ldur w7, [w3], #2      //Load w7 with word at the address in w3 then
                       //increment w3 by 2, post-indexed
```

ARM8 A64 Data Transfer Instructions: 64-bit Scaled Load

```
ldr x7, [x3, x2]        //Load x7 with double word at the address (x3 + x2)

ldr x7, [x3, #8]        //Load x7 with double word at the address (x3 + 8)

ldr x7, [x3, x2]!       //Load x7 with double word at the address (x3 + x2), then
                        //store the address in x3, pre-indexed

ldr x9, [x2, #8]!       //Load x9 with double word at the address (x2 + 8), then
                        //store the address in x2, pre-indexed

ldr x7, [x3], #8        //Load x7 with double word at the address in x3 then
                        //increment x3 by 8, post-indexed
```

ARM8 A64 Data Transfer Instructions: 64-bit Unscaled Half-word and Byte Load

```
ldurh x9, [x2, #2]!     //Load x9 with the half-word at the address (x2 + 2) and
                        //zero extend x9, then store the address in x2, pre-indexed

ldursh x5, [x2]         //Load x5 with the half-word at the address in x2 and sign
                        //extend x5.

ldurb x9, [x2, #1]!     //Load x9 with the byte at the address (x2 + 1) and zero
                        //extend x9, then store the address in x2, pre-indexed

ldursb x5, [x2]         //Load x5 with the byte at the address in x2 and sign
                        //extend x5.
```

ARM8 A64 Data Transfer Instructions: 64-bit Scaled Half-word and Byte Load

```
ldrh x9, [x2, #2]!      //Load x9 with the half-word at the address (x2 + 2) and
                        //zero extend x9 , then store the address in x2, pre-indexed

ldrsh x5, [x2]          //Load x5 with the half-word at the address in x2 and sign
                        //extend x5.

ldrb x9, [x2, #1]!      //Load x9 with the byte at the address (x2 + 1), then store
                        //the address in x2, pre-indexed

ldrsb x5, [x2]          //Load x5 with the byte at the address in x2 and sign
                        //extend x5.
```

ARM8 A64 Data Transfer Instructions: 64-bit Unscaled Store

```
stur x7, [x3, x2]       //Store x7 (double word) at the address (x3 + x2)

stur x7, [x3, #2]       //Store x7 (double word) at the address (x3 + 2)

stur x7, [x3, x2]!      //Store x7 (double word) at the address (x3 + x2), then
                        //store the address in x3, pre-indexed

stur x9, [x2, #2]!      //Store x9 (double word) at the address (x2 + 2), then store
                        //the address in x2, pre-indexed

stur x7, [x3], #2       //Store x7 (double word) at the address in x3 then
                        //increment x3 by 2, post indexed
```

ARM8 A64 Data Transfer Instructions: 64-bit Scaled Store

```
str x7, [x3, x2]        //Store x7 (double word) at the address (x3 + x2)

str x7, [x3, #8]        //Store x7 (double word) at the address (x3 + 8)

str x7, [x3, x2]!       //Store x7 (double word) at the address (x3 + x2), then
                        //store the address in x3, pre-indexed

str x9, [x2, #8]!       //Store x9 (double word) at the address (x2 + 8), then store
                        //the address in x2, pre-indexed

str x7, [x3], #8        //Store x7 (double word) at the address in x3 then
                        //increment x3 by 8, post-indexed
```

ARM8 A64 Data Transfer Instructions: 64-bit Unscaled Half-word and Byte Store

```
sturh x9, [x2, #2]!     //Store half-word of x9 at the address (x2 + 2), then store
                        //the address in x2, pre-indexed

sturb x9, [x2, #1]!     //Store byte of x9 at the address (x2 + 1), then store
                        //the address in x2, pre-indexed

strh x9, [x2, #2]!      //Store half-word of x9 at the address (x2 + 2), then store
                        //the address in x2, pre-indexed

strb x9, [x2, #1]!      //Store byte of x9 at the address (x2 + 1), then store
                        //the address in x2, pre-indexed
```

ARM8 A64 Data Transfer Instructions: Literal Pool Load

```
ldr x9, =label
        //loads memory address referred by label to X9
```

```
                .data
fname:          .asciz          “Humayun”

                .text
                ldr x1, =fname
```

ARM8 A64 Load/Store Instructions Addressing Modes

Type	Immediate Offset	Register Offset	Extended Register Offset
Simple register (exclusive)	`[base{,#0}]`	n/a	n/a
Offset	`[base{,#imm}]`	`[base,Xm{,LSL #imm}]`	`[base,Wm,(S|U)XTW {#imm}]`
Pre-indexed	`[base,#imm]!`	n/a	n/a
Post-indexed	`[base],#imm`	n/a	n/a
PC-relative (literal) load	`label`	n/a	n/a

ARM8 A64 Arithmetic Instructions

Arithmetic Instructions			
ADC{S}	rd, rn, rm	rd = rn + rm + C	
ADD{S}	rd, rn, op2	rd = rn + op2	S
ADR	Xd, $\pm rel_{21}$	Xd = PC + $rel^{\pm}$	
ADRP	Xd, $\pm rel_{33}$	Xd = $PC_{63:12}:0_{12}$ + $rel^{\pm}_{33:12}:0_{12}$	
CMN	rd, op2	rd + op2	S
CMP	rd, op2	rd − op2	S
MADD	rd, rn, rm, ra	rd = ra + rn × rm	
MNEG	rd, rn, rm	rd = − rn × rm	
MSUB	rd, rn, rm, ra	rd = ra − rn × rm	
MUL	rd, rn, rm	rd = rn × rm	
NEG{S}	rd, op2	rd = −op2	
NGC{S}	rd, rm	rd = −rm − ∼C	
SBC{S}	rd, rn, rm	rd = rn − rm − ∼C	
SDIV	rd, rn, rm	rd = rn $\bar{\div}$ rm	
SMADDL	Xd, Wn, Wm, Xa	Xd = Xa + Wn $\bar{\times}$ Wm	
SMNEGL	Xd, Wn, Wm	Xd = − Wn $\bar{\times}$ Wm	
SMSUBL	Xd, Wn, Wm, Xa	Xd = Xa − Wn $\bar{\times}$ Wm	
SMULH	Xd, Xn, Xm	Xd = $(Xn \bar{\times} Xm)_{127:64}$	
SMULL	Xd, Wn, Wm	Xd = Wn $\bar{\times}$ Wm	
SUB{S}	rd, rn, op2	rd = rn - op2	S
UDIV	rd, rn, rm	rd = rn ÷ rm	
UMADDL	Xd, Wn, Wm, Xa	Xd = Xa + Wn × Wm	
UMNEGL	Xd, Wn, Wm	Xd = − Wn × Wm	
UMSUBL	Xd, Wn, Wm, Xa	Xd = Xa − Wn × Wm	
UMULH	Xd, Xn, Xm	Xd = $(Xn \times Xm)_{127:64}$	
UMULL	Xd, Wn, Wm	Xd = Wn × Wm	

Operand 2 (op2)		
all	*rm*	rm
all	rm, LSL #i_6	rm ≪ i
all	rm, LSR #i_6	rm ≫ i
all	rm, ASR #i_6	rm $\bar{\gg}$ i
logical	rm, ROR #i_6	rm ⋙ i
arithmetic	Wm, {S,U}XTB{ #i_3}	$Wm^{?}_{B0}$ ≪ i
arithmetic	Wm, {S,U}XTH{ #i_3}	$Wm^{?}_{H0}$ ≪ i
arithmetic	Wm, {S,U}XTW{ #i_3}	$Wm^{?}$ ≪ i
arithmetic	Xm, {S,U}XTX{ #i_3}	$Xm^{?}$ ≪ i
arithmetic	#i_{12}	$i^{\emptyset}$
arithmetic	#i_{24}	$i^{\emptyset}_{23:12}:0_{12}$
AND,EOR,ORR,TST	#mask	mask

Keys	
N	Operand bit size (8, 16, 32 or 64)
s	Operand log byte size (0=byte,1=hword,2=word,3=dword)
rd, rn, rm, rt	General register of either size (Wn or Xn)
prfop	P{LD,LI,ST}L{1..3}{KEEP,STRM}
{,sh}	Optional halfword left shift (LSL #{16,32,48})
$val^{\pm}$, $val^{\emptyset}$, $val^{?}$	Value is sign/zero extended (? depends on instruction)
$\bar{\times}$ $\bar{\div}$ $\bar{\gg}$ $\bar{>}$ $\bar{<}$	Operation is signed

ARM8 A64 Arithmetic Logic Instructions

```
add x0, x1, x2        //x0=x1+x2
adds x0, x1, x2       //x0=x1+x2, and set cpsr flags
adc x0, x1, x2        //x0=x1+x2+carry
adcs x0, x1, x2       //x0=x1+x2+carry, and set cpsr flags

sub x0, x1, x2        //x0=x1-x2
subs x0, x1, x2       //x0=x1-x2, and set cpsr flags
sbc x0, x1, x2        //x0=x1-x2-1+carry
sbcs x0, x1, x2       //x0=x1-x2-1+carry, and set cpsr flags

cmp x0, #imm          //compare x0 with #imm
cmp x0, x1            //compare x0 with x1
```

ARM8 A64 Arithmetic Logic Instructions

```
mul w0, w1, w2      //w0=w1*w2
mul x0, x1, x2      //x0=x1*x2
smull x0, w1, w2    //x0=w1*w2, treats source operands as signed
umull x0, w1, w2    //x0=w1*w2, treats source operands as unsigned

sdiv w0, w1, w2     //w0=w1÷w2, treats source operadns as signed
sdiv x0, x1, x2     //x0=x1÷x2, treats source operadns as signed
udiv w0, w1, w2     //w0=w1÷w2, treats source oprands as unsigned
udiv x0, x1, x2     //x0=x1÷x2, treats source oprands as unsigned
```

ARM8 A64 Logical Instructions

Logical and Move Instructions			
AND{S}	rd, rn, op2	rd = rn & op2	
ASR	rd, rn, rm	rd = rn $\gg$ rm	
ASR	rd, rn, #i_6	rd = rn $\bar{\gg}$ i	
BIC{S}	rd, rn, op2	rd = rn & $\sim$op2	
EON	rd, rn, op2	rd = rn $\oplus$ $\sim$op2	
EOR	rd, rn, op2	rd = rn $\oplus$ op2	
LSL	rd, rn, rm	rd = rn $\ll$ rm	
LSL	rd, rn, #i_6	rd = rn $\ll$ i	
LSR	rd, rn, rm	rd = rn $\gg$ rm	
LSR	rd, rn, #i_6	rd = rn $\gg$ i	
MOV	rd, rn	rd = rn	S
MOV	rd, #i	rd = i	
MOVK	rd,#i_{16}{, sh}	$rd_{sh+15:sh}$ = i	
MOVN	rd,#i_{16}{, sh}	rd = $\sim(i^{\emptyset} \ll sh)$	
MOVZ	rd,#i_{16}{, sh}	rd = $i^{\emptyset} \ll sh$	
MVN	rd, op2	rd = $\sim$op2	
ORN	rd, rn, op2	rd = rn | $\sim$op2	
ORR	rd, rn, op2	rd = rn | op2	
ROR	rd, rn, #i_6	rd = rn $\ggg$ i	
ROR	rd, rn, rm	rd = rn $\ggg$ rm	
TST	rn, op2	rn & op2	

Operand 2 (op2)		
all	*rm*	rm
all	rm, LSL #i_6	rm $\ll$ i
all	rm, LSR #i_6	rm $\gg$ i
all	rm, ASR #i_6	rm $\bar{\gg}$ i
logical	rm, ROR #i_6	rm $\ggg$ i
arithmetic	Wm, {S,U}XTB{ #i_3}	$Wm^{?}_{B0} \ll$ i
arithmetic	Wm, {S,U}XTH{ #i_3}	$Wm^{?}_{H0} \ll$ i
arithmetic	Wm, {S,U}XTW{ #i_3}	$Wm^{?} \ll$ i
arithmetic	Xm, {S,U}XTX{ #i_3}	$Xm^{?} \ll$ i
arithmetic	#i_{12}	$i^{\emptyset}$
arithmetic	#i_{24}	$i^{\emptyset}_{23:12}:0_{12}$
AND,EOR,ORR,TST	#mask	mask

Keys	
N	Operand bit size (8, 16, 32 or 64)
s	Operand log byte size (0=byte,1=hword,2=word,3=dword)
rd, rn, rm, rt	General register of either size (Wn or Xn)
prfop	P{LD,LI,ST}L{1..3}{KEEP,STRM}
{,sh}	Optional halfword left shift (LSL #{16,32,48})
$val^{\pm}$, $val^{\emptyset}$, $val^{?}$	Value is sign/zero extended (? depends on instruction)
$\bar{\times}$ $\bar{\div}$ $\bar{\gg}$ $\bar{>}$ $\bar{<}$	Operation is signed

ARM8 A64 Logical Instructions

```
and x0, x1, #bimm64      //x0 = x1 & #bimm64
and w0, w1, #bimm32      //w0 = w1 & #bimm32
and w0, w1, w2           //w0 = w1 & w2
ands x0, x1, x2          //x0 = x1 & x2, and set flags (clears C and V)

orr x0, x1, #bimm64      //x0 = x1 | #bimm64
orr x0, x1, x2           //x0 = x1 | x2

eor x0, x1, #bimm64      //x0 = x1 ⊕ #bimm64
eor x0, x1, x2           //x0 = x1 ⊕ x2
```

ARM8 A64 Logical Instructions

```
asr x0, x1, #uimm      //x0 = arithmetic shift right x1 #uimm bits
lsl x0, x1, #uimm      //x0 = logical shift left x1 #uimm bits
lsr x0, x1, #uimm      //x0 = logical shift right x1 #uimm bits
ror x0, x1, #uimm      //x0 = rotate right x1 #uimm bits

asr x0, x1, x2         //x0 = arithmetic shift right x1 (x2 & 0x3f) bits
lsl x0, x1, x2         //x0 = logical shift left x1 (x2 & 0x3f) bits
lsr x0, x1, x2         //x0 = logical shift right x1 (x2 & 0x3f) bits
ror x0, x1, x2         //x0 = rotate right x1 (x2 & 0x3f) bits
```

ARM8 A64 Branch Instructions

Branch Instructions		
B	rel_{28}	$\text{PC} = \text{PC} + \text{rel}^{\pm}_{27:2}{:}0_2$
Bcc	rel_{21}	$\text{if(cc) PC} = \text{PC} + \text{rel}^{\pm}_{20:2}{:}0_2$
BL	rel_{28}	$\text{X30} = \text{PC} + 4;\ \text{PC} \mathrel{+}= \text{rel}^{\pm}_{27:2}{:}0_2$
BLR	Xn	$\text{X30} = \text{PC} + 4;\ \text{PC} = \text{Xn}$
BR	Xn	$\text{PC} = \text{Xn}$
CBNZ	rn, rel_{21}	$\text{if(rn} \neq 0)\ \text{PC} \mathrel{+}= \text{rel}^{\emptyset}_{21:2}{:}0_2$
CBZ	rn, rel_{21}	$\text{if(rn} = 0)\ \text{PC} \mathrel{+}= \text{rel}^{\emptyset}_{21:2}{:}0_2$
RET	{Xn}	$\text{PC} = \text{Xn}$
TBNZ	rn, #i, rel_{16}	$\text{if(rn}_i \neq 0)\ \text{PC} \mathrel{+}= \text{rel}^{\pm}_{15:2}{:}0_2$
TBZ	rn, #i, rel_{16}	$\text{if(rn}_i = 0)\ \text{PC} \mathrel{+}= \text{rel}^{\pm}_{15:2}{:}0_2$

ARM8 A64 Conditional Branch Instructions

```
b.cond label          //Jump to program relative label if cond is true

cbz x1, label         //Jump to program relative label if x1 is equal to zero

cbnz x1, label        //Jump to program relative label if x1 is not equal to zero

tbz x1, #uimm6, label     //Jump to program relative label if bit number
                          //#uimm6 in x1 is equal to zero
tbnz x1, #uimm6, label    //Jump to program relative label if bit number
                          //#uimm6 in x1 is not equal to zero
```

ARM8 A64 Unconditional Branch Instructions

```
b label          //Jump to program relative label

bl label         //Jump to program relative label and write next instruction
                 //address in link register (lr) or x30

blr x1           //Jump to address in x1 and write next instruction
                 //address in link register (lr) or x30

br x1            //Jump to address in x1

ret {x30}        //Jump to address in x30 and hints the CPU that this a
                 //subroutine return. Jump to address in x30 if a register is
                 //omitted.
```