ARM Instruction Sets and Program

Jin-Fu Li

Department of Electrical Engineering

National Central University

Adopted from National Chiao-Tung University
IP Core Design

Outline

- Programmer’s model
- 32-bit instruction set
- 16-bit instruction set
- Summary

Programmer’s model

ARM Ltd

- ARM was originally developed at Acron Computer Limited, of Cambridge, England between 1983 and 1985.
  - 1980, RISC concept at Stanford and Berkeley universities.
  - First RISC processor for commercial use
- 1990 Nov, ARM Ltd was founded
- ARM cores
  - Licensed to partners who fabricate and sell to customers.
- Technologies assist to design in the ARM application
  - Software tools, boards, debug hardware, application software, bus architectures, peripherals etc…
- Modification of the acronym expansion to **Advanced RISC Machine**.

RISC architecture

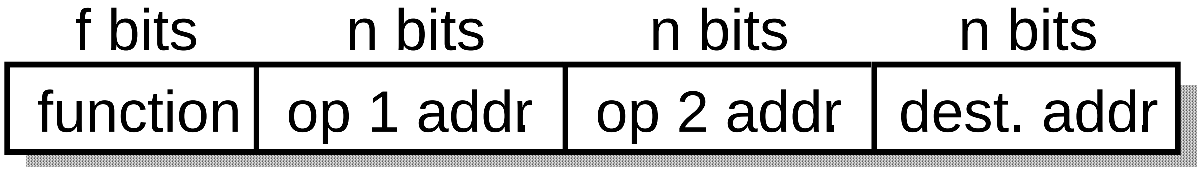
- Berkeley incorporated a Reduced Instruction Set Computer (RISC) architecture.
- It had the following key features:
  - A fixed (32-bit) instruction size with few formats;
    - CISC processors typically had variable length instruction sets with many formats.
  - A load–store architecture were instructions that process data operate only on registers and are separate from instructions that access memory;
    - CISC processors typically allowed values in memory to be used as operands in data processing instructions.
  - A large register bank of thirty-two 32-bit registers, all of which could be used for any purpose, to allow the load-store architecture to operate efficiently;
    - CISC register sets were getting larger, but none was this large and most had different registers for different purposes

RISC organization

- Hard-wired instruction decode logic
  - CISC processor used large microcode ROMs to decode their instructions
- Pipelined execution
  - CISC processors allowed little, if any, overlap between consecutive instructions (though they do now)
- Single-cycle execution
  - CISC processors typically took many clock cycles to completes a single instruction

ARM Architecture vs. Berkeley RISC (1/2)

- Features used
  - Load/Store architecture
  - Fixed-length 32-bit instructions
  - 3-address instruction formats



```
ADD   d, S1, S2          ; d := S1 + S2
```

ARM Architecture vs. Berkeley RISC (2/2)

❑ Features rejected
- Register windows → costly
  - Use shadow registers in ARM
- Delay branch
  - Badly with branch prediction
- Single-cycle execution of all instructions
  - Most single cycle, many other take multiple clock cycles

Data Size and Instruction set

- ARM processor is a 32-bit architecture
- Most ARM's implement two instruction sets
  – 32-bit *ARM* instruction set
  – 16-bit *Thumb* instruction set

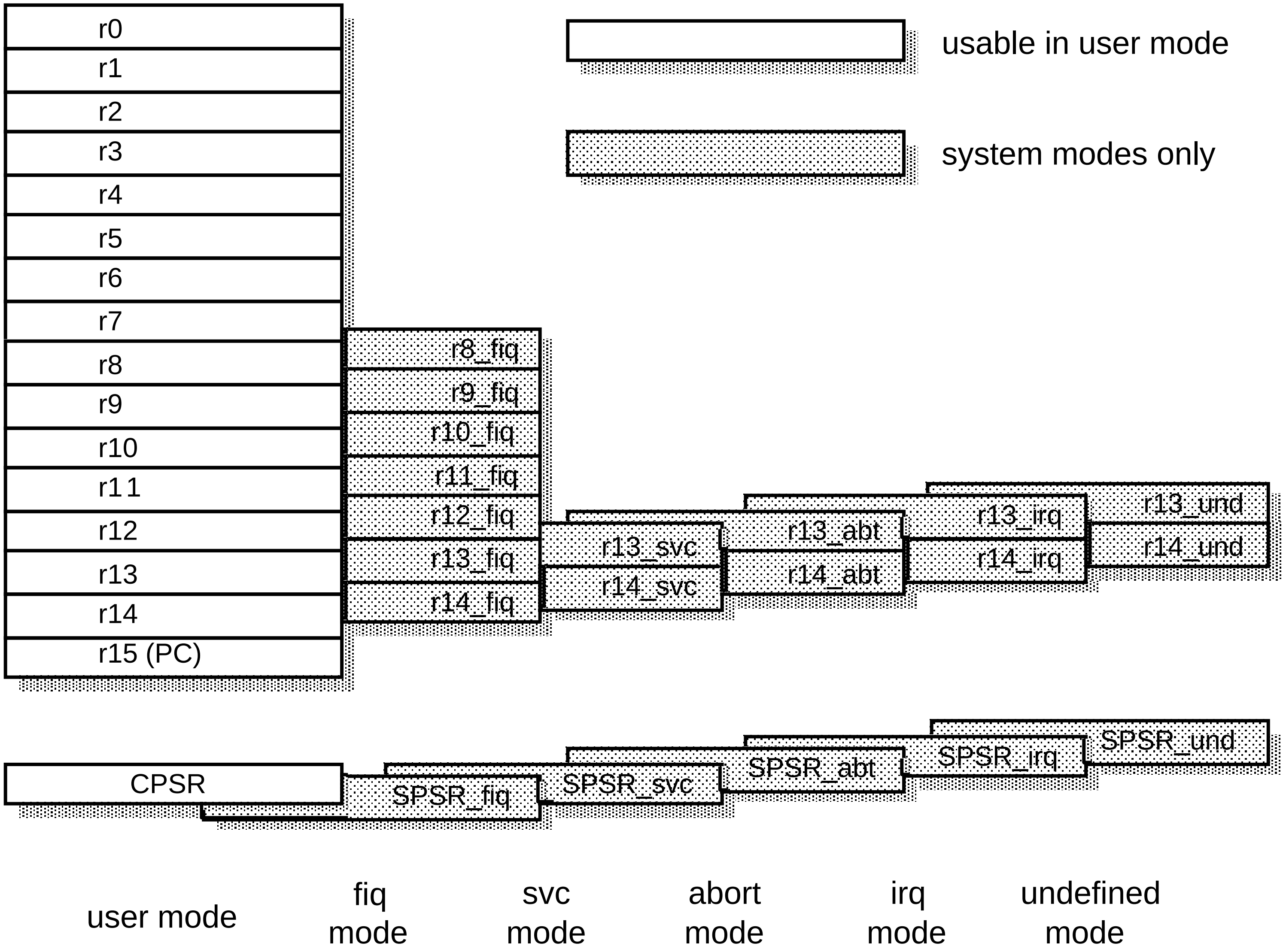
Data Types

- ARM processor supports 6 data types
  - 8-bits signed and unsigned bytes
  - 16-bits signed and unsigned half-word, aligned on 2-byte boundaries
  - 32-bits signed and unsigned words, aligned on 4-byte boundaries
- ARM instructions are all 32-bit words, word-aligned; Thumb instructions are half-words, aligned on 2-byte boundaries
- ARM coprocessor supports floating-point values

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
  - – 31 general purpose registers
- The current processor mode governs which bank is accessible
  - – User mode can access
    - A particular set of r0 – r12 registers
    - A particular r13 (*stack pointer*, SP) and r14 (*link register*. LR)
    - The *program counter*, r15 (PC)
    - The *curent program status register*, CPSR
  - – Privileged modes (except system) can access
    - A particular SPSR (*Saved Program Status Register*)

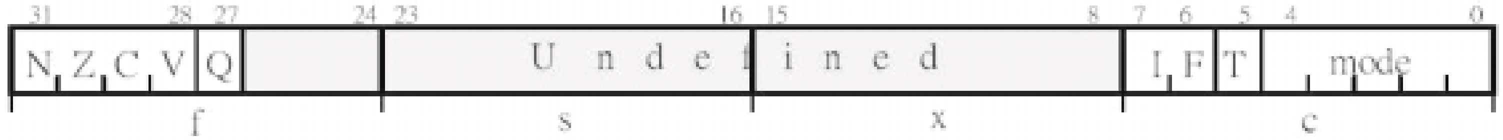
Register Banking



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 [32:2] with bits [1:0] undefined (as instruction cannot be halfword)
- 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 [32:1] with bits [0] undefined (as instruction cannot be byte-aligned)

Current Program Status Registers (CPSR)



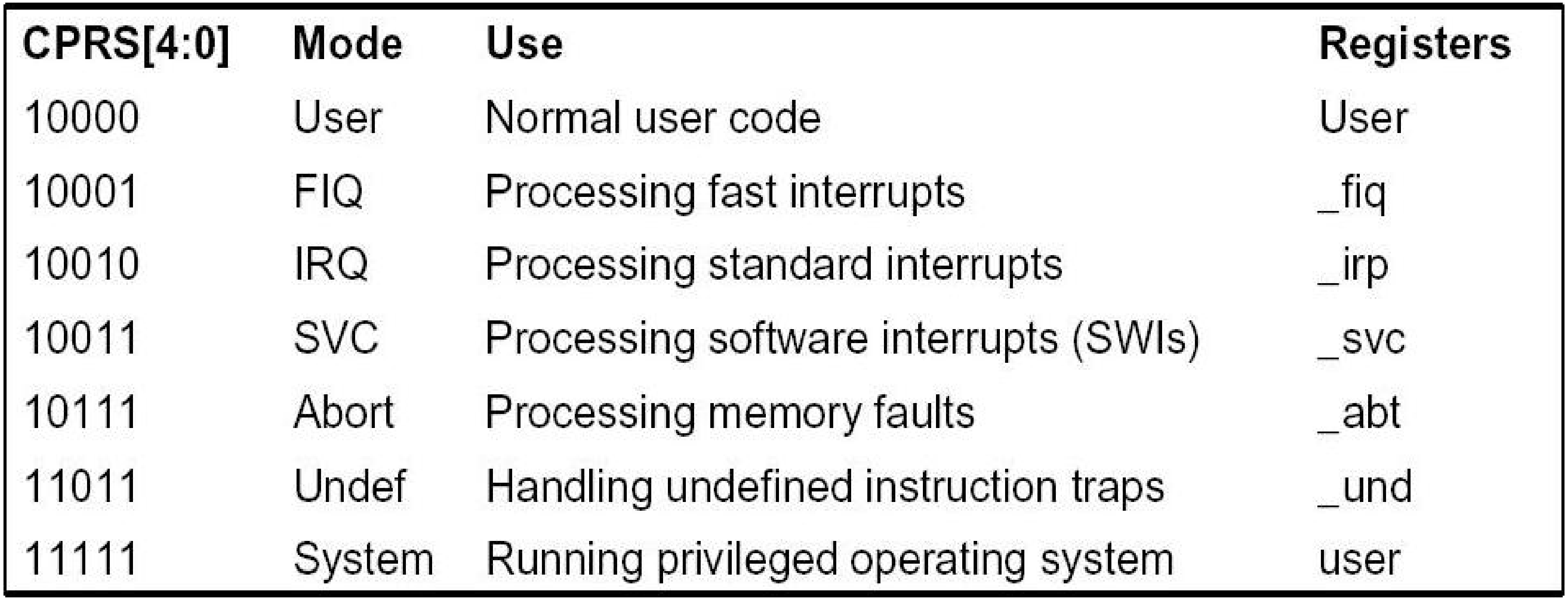
- ❑ Condition code flags
  - **N**: Negative result form ALU
  - **Z**: Zero result from ALU
  - **C**: ALU Operation Carried out
  - **V**: ALU operation oVerflowed
- ❑ Sticky overflow flag – Q flag
  - Architecture 5TE only
  - Indicates if saturation has occurred during certain operations
- ❑ Interrupt disable bits
  - **I** = 1, disable the IRQ
  - **F** = 1, disable 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

Saved Program Status Register (SPSR)

- Each privileged mode (except system mode) has associated with it a SPSR
- This SPSR is used to save the state of CPSR when the privileged mode is entered in order that the user state can be fully restored when the user process is resumed
- Often the SPSR may be untouched from the time the privileged mode is entered to the time it is used to restore the CPSR, but if the privileged supervisor calls to itself the SPSR must be copied into a general register and saved

Processor Modes

- ARM has seven basic operation modes
- Mode changes by software control or external interrupts



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

Privileged Modes

- ❑ Most programs operate in user mode. ARM has other privileges operating modes which are used to handle exceptions, supervisor calls (software interrupt), and system mode.
- ❑ More access rights to memory systems and coprocessors.
- ❑ Current operating mode is defined by CPSR[4:0].

Exceptions

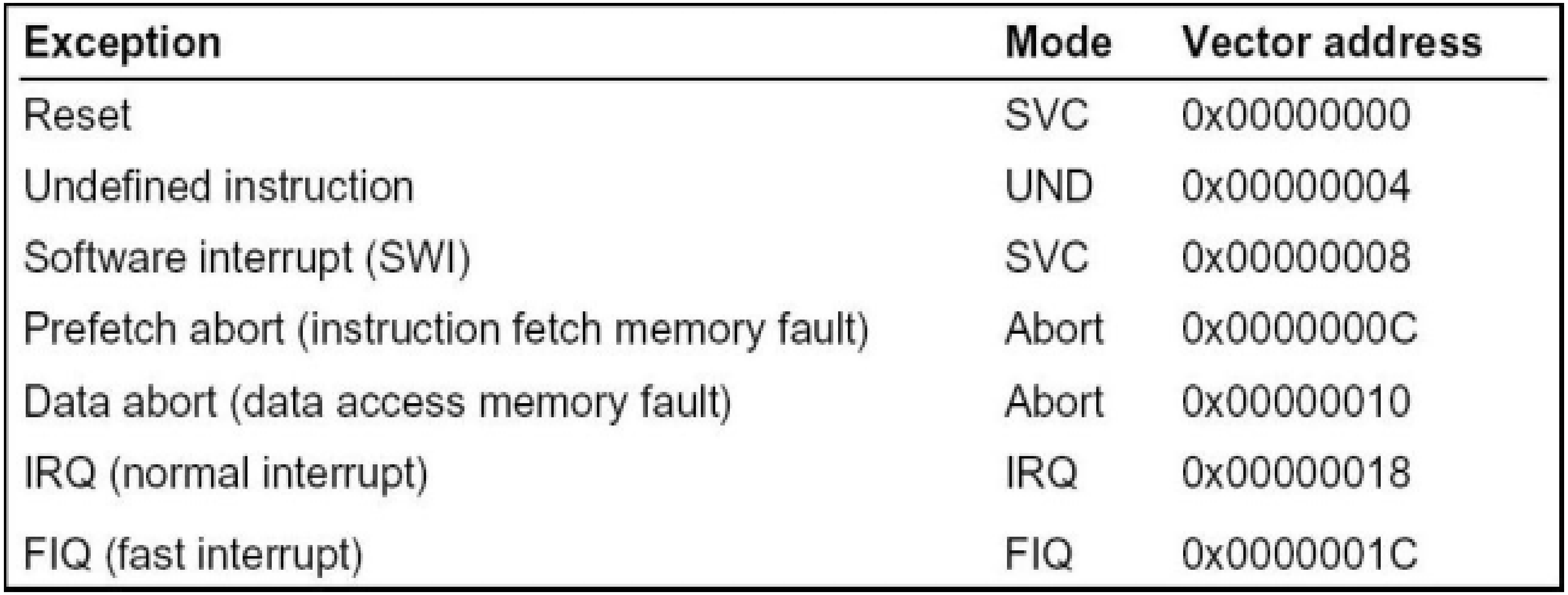
- ❑ Exceptions are usually used to handle unexpected events which arise during the execution of a program, such as interrupts or memory faults, also cover software interrupts, undefined instruction traps, and the system reset
- ❑ Three groups:
  - – Exceptions generated as the direct effect of execution an instruction
    - Software interrupts, undefined instructions, and prefetch abort
  - – Exceptions generated as a side effect of an instruction
    - Data aborts
  - – Exceptions generated externally
    - Reset, IRQ and FIQ

Exception Entry (1/2)

- When an exception arises, ARM completes the current instruction as best it can (except that *reset* exception terminates the current instruction immediately) and then departs from the current instruction sequence to handle the exception which starts from a specific location (exception vector).
- Processor performs the following sequence:
  - Change to the operating mode corresponding to the particular exception
  - Save the address of the instruction following the exception entry instruction in r14 of the new mode
  - Save the old value of CPSR in the SPSR of the new mode
  - Disable IRQs by setting bit 7 of the CPSR and, if the exception is a fast interrupt, disable further faster interrupt by setting bit 6 of the CPSR

Exception Entry (2/2)

– Force the PC to begin execution at the relevant vector address



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

- ❑ Normally the vector address contains a branch to the relevant routine
- ❑ Two banked registers in each of the privilege modes are used to hold the return address and stack point

Exception Return

- Once the exception has been handled, the user task is normally resumed
- The sequence is
  - Any modified user registers must be restored from the handler’s stack
  - CPSR must be restored from the appropriate SPSR
  - PC must be changed back to the relevant instruction address
- The last two steps happen atomically as part of a single instruction

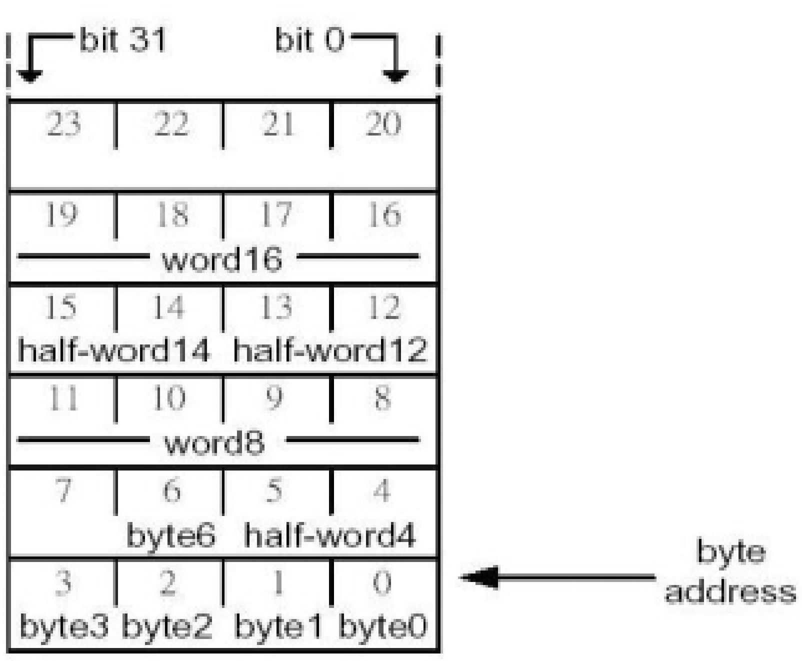
ARM Exceptions

- Exception handler use r13_<mode> which will normally have been initialized to point a dedicated stack in memory, to save some user register for use as work registers

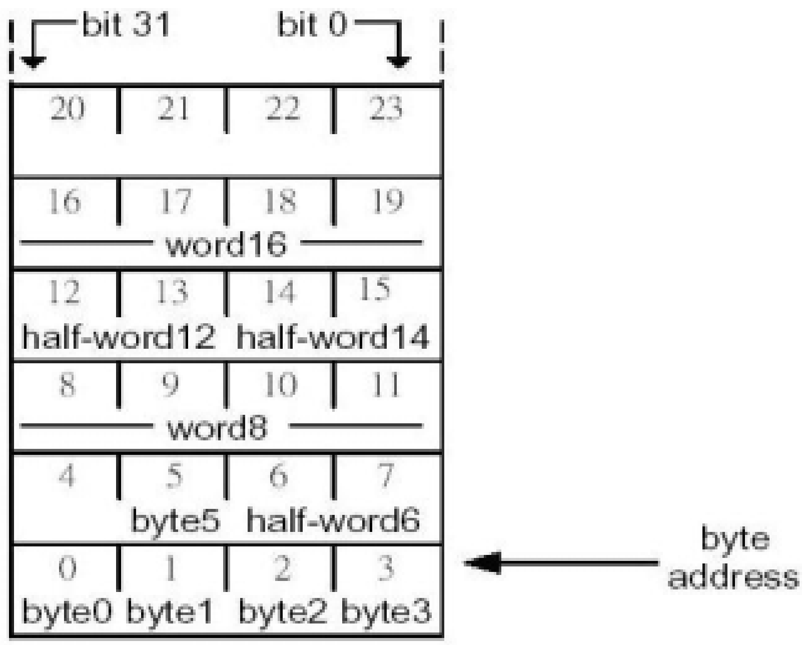
Exception Priorities

- Priority order
  - Reset (highest priority)
  - Data abort
  - FIQ
  - IRQ
  - Prefetch abort
  - SWI, undefined instruction

Memory Organization



(a) Little-endian memory organization



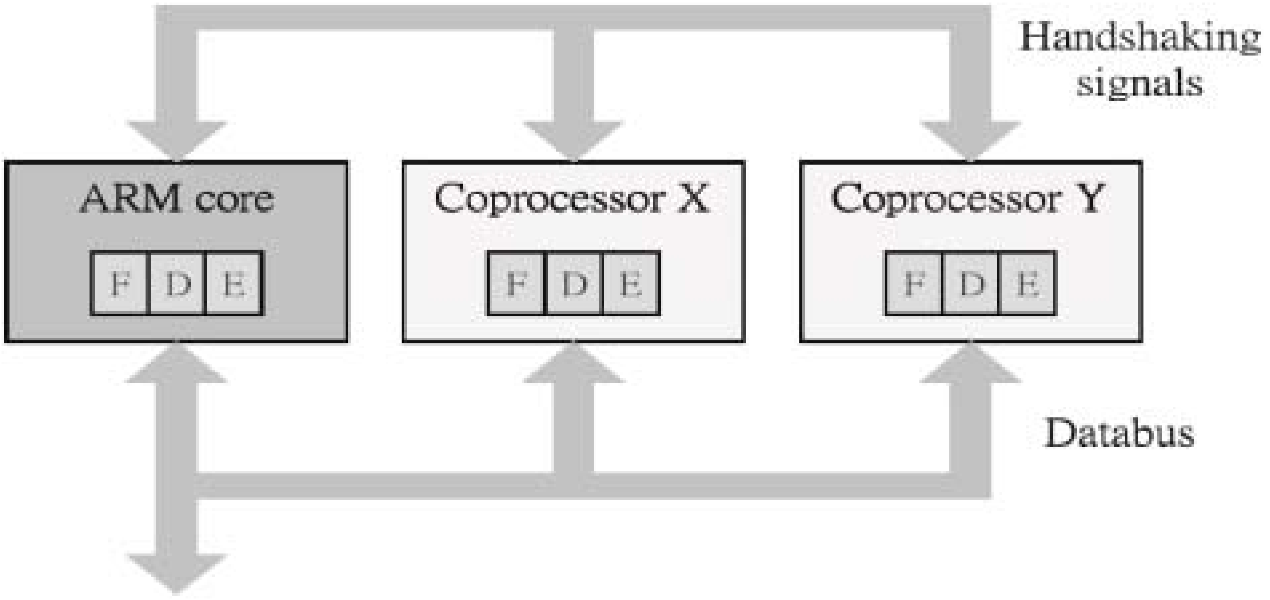
(b) Big-endian memory organization

- ❑ Word, half-word alignment (xxxx00 or xxxxx0)
- ❑ ARM can be set up to access data in either *little-endian* or *big-endian* format, through they default to **little-endian**.

Features of the ARM Instruction Set

- Load-store architecture
  - Process values which are in registers
  - Load, store instructions for memory data accesses
- 3-address data processing instructions
- Conditional execution of every instruction
- Load and store multiple registers
- Shift, ALU operation in a single instruction
- Open instruction set extension through the coprocessor instruction
- Very dense 16-bit compressed instruction set (Thumb)

Coprocessors



– Up to *16* coprocessors can be defined

– Expands the ARM instruction set

– Each coprocessor can have up to 16 private registers of any reasonable size

– Load-store architecture

Thumb

- Thumb is a 16-bit instruction set
  - Optimized for code density from C code
  - Improved performance form narrow memory
  - Subset of the functionality of the ARM instruction set
- Core has two execution states – ARM and Thumb
  - Switch between them using ***BX*** instruction
- Thumb has characteristic features:
  - Most Thumb instruction are executed unconditionally
  - Many Thumb data process instruction use a 2-address format
  - Thumb instruction formats are less regular than ARM instruction formats, as a result of the dense encoding.

I/O System

- ARM handles input/output peripherals as ***memory-mapped*** with interrupt support
- Internal registers in I/O devices as addressable locations with ARM's memory map read and written using load-store instructions
- Interrupt by normal interrupt (*IRQ*) or fast interrupt (*FIQ*)
- Input signals are *level-sensitive* and *maskable*
- May include Direct Memory Access (DMA) hardware

ARM Processor Cores (1/2)

- ARM Processor core + cache + MMU
  → ARM CPU cores
- ARM6 → ARM7
  - 3-stage pipeline
  - Keep its instructions and data in the same memory system
  - **T**humb 16-bit compressed instruction set
  - on-chip **D**ebug support, enabling the processor to halt in response to a debug request
  - enhanced **M**ultiplier, 64-bit result
  - Embedded**I**CE hardware, give on-chip breakpoint and watchpoint support

ARM Processor Cores (2/2)

- ARM8 → ARM9

  → ARM10
- ARM9
  - 5-stage pipeline (130 MHz or 200MHz)
  - Using separate instruction and data memory ports
- ARM 10 (1998. Oct.)
  - High performance, 300 MHz
  - Multimedia digital consumer applications
  - Optional vector floating-point unit

ARM Architecture Version (1/5)

- Version 1
  - The first ARM processor, developed at Acorn Computers Limited 1983-1985
  - 26-bit address, no multiply or coprocessor support
- Version 2
  - Sold in volume in the Acorn Archimedes and A3000 products
  - 26-bit addressing, including 32-bit result multiply and coprocessor
- Version 2a
  - Coprocessor 15 as the system control coprocessor to manage cache
  - Add the atomic load store (SWP) instruction

ARM Architecture Version (2/5)

- Version 3
  - First ARM processor designed by ARM Limited (1990)
  - ARM6 (macro cell)

    ARM60 (stand-alone processor)

    ARM600 (an integrated CPU with on-chip cache, MMU, write buffer)

    ARM610 (used in Apple Newton)
  - 32-bit addressing, separate CPSR and SPSRs
  - Add the undefined and abort modes to allow coprocessor emulation and virtual memory support in supervisor mode
- Version 3M
  - Introduce the signed and unsigned multiply and multiply-accumulate instructions that generate the full 64-bit result

ARM Architecture Version (3/5)

- Version 4
  - Add the signed, unsigned half-word and signed byte load and store instructions
  - Reserve some of SWI space for architecturally defined operation
  - System mode is introduced
- Version 4T
  - 16-bit Thumb compressed form of the instruction set is introduced

ARM Architecture Version (4/5)

- ❑ Version 5T
  - – Introduced recently, a superset of version 4T adding the BLX, CLZ and BRK instructions
- ❑ Version 5TE
  - – Add the signal processing instruction set extension

ARM Architecture Version (5/5)

Core	Architecture
ARM1	v1
ARM2	v2
ARM2as, ARM3	v2a
ARM6, ARM600, ARM610	v3
ARM7, ARM700, ARM710	v3
ARM7TDMI, ARM710T, ARM720T, ARM740T	v4T
StrongARM, ARM8, ARM810	v4
ARM9TDMI, ARM920T, ARM940T	V4T
ARM9E-S, ARM10TDMI, ARM1020E	v5TE
ARM10TDMI, ARM1020E	v5TE

32-bit instruction set

- ARM assembly language program
  - – ARM development board or ARM emulator
- ARM instruction set
  - – Standard ARM instruction set
  - – A compressed form of the instruction set, a subset of the full ARM instruction set is encoded into 16-bit instructions – Thumb instruction
  - – Some ARM cores support instruction set extensions to enhance signal processing capabilities

Instructions

- ❑ Data processing instructions
- ❑ Data transfer instructions
- ❑ Control flow instructions

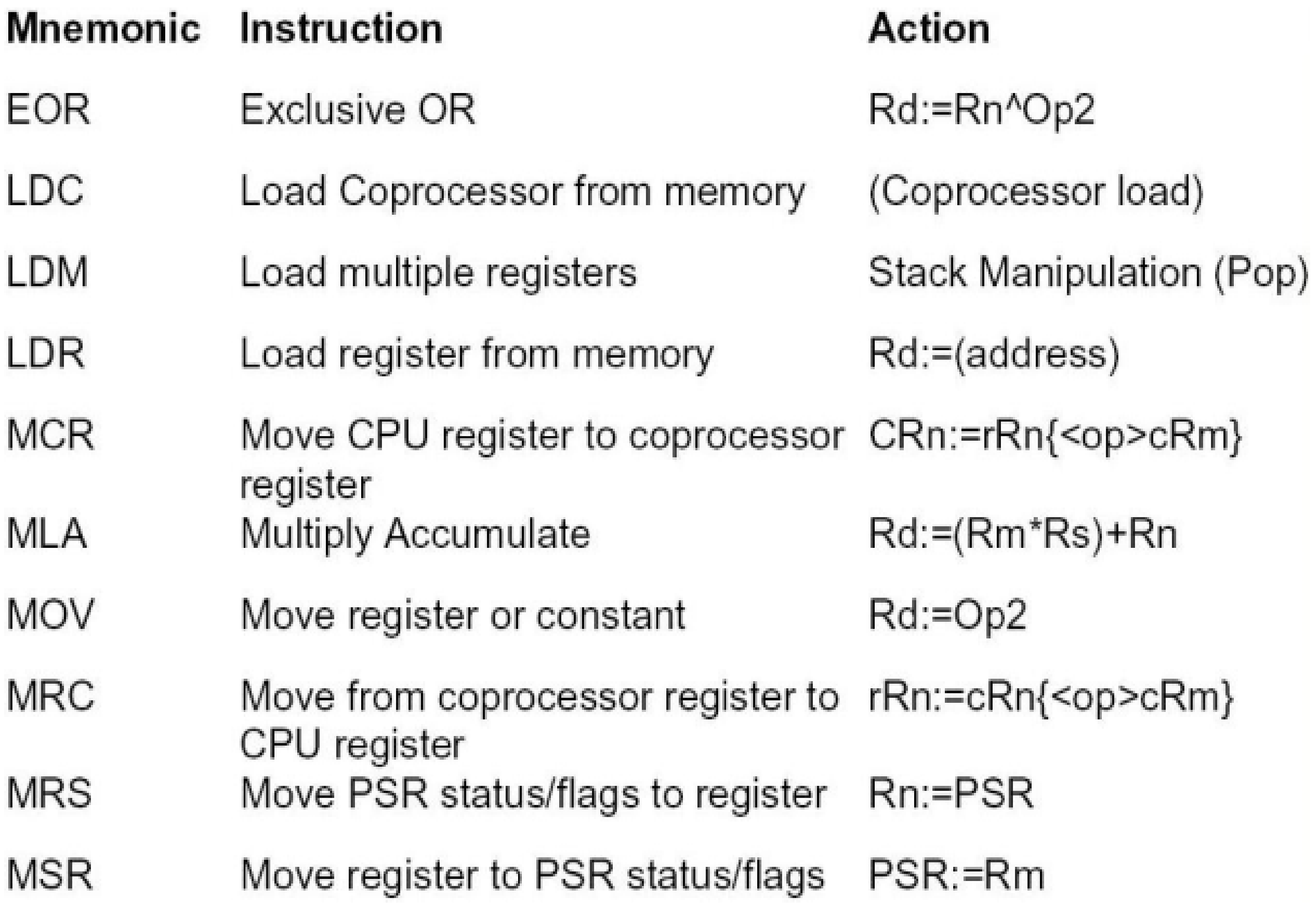
ARM Instruction Set Summary (1/4)

Mnemonic	Instruction	Action
ADC	Add with carry	Rd:=Rn+Op2+Carry
ADD	Add	Rd:=Rn+Op2
AND	AND	Rd:=Rn AND Op2
B	Branch	R15:=address
BIC	Bit Clear	Rd:=Rn AND NOT Op2

BL	Branch with Link	R14:=R15
		R15:=address

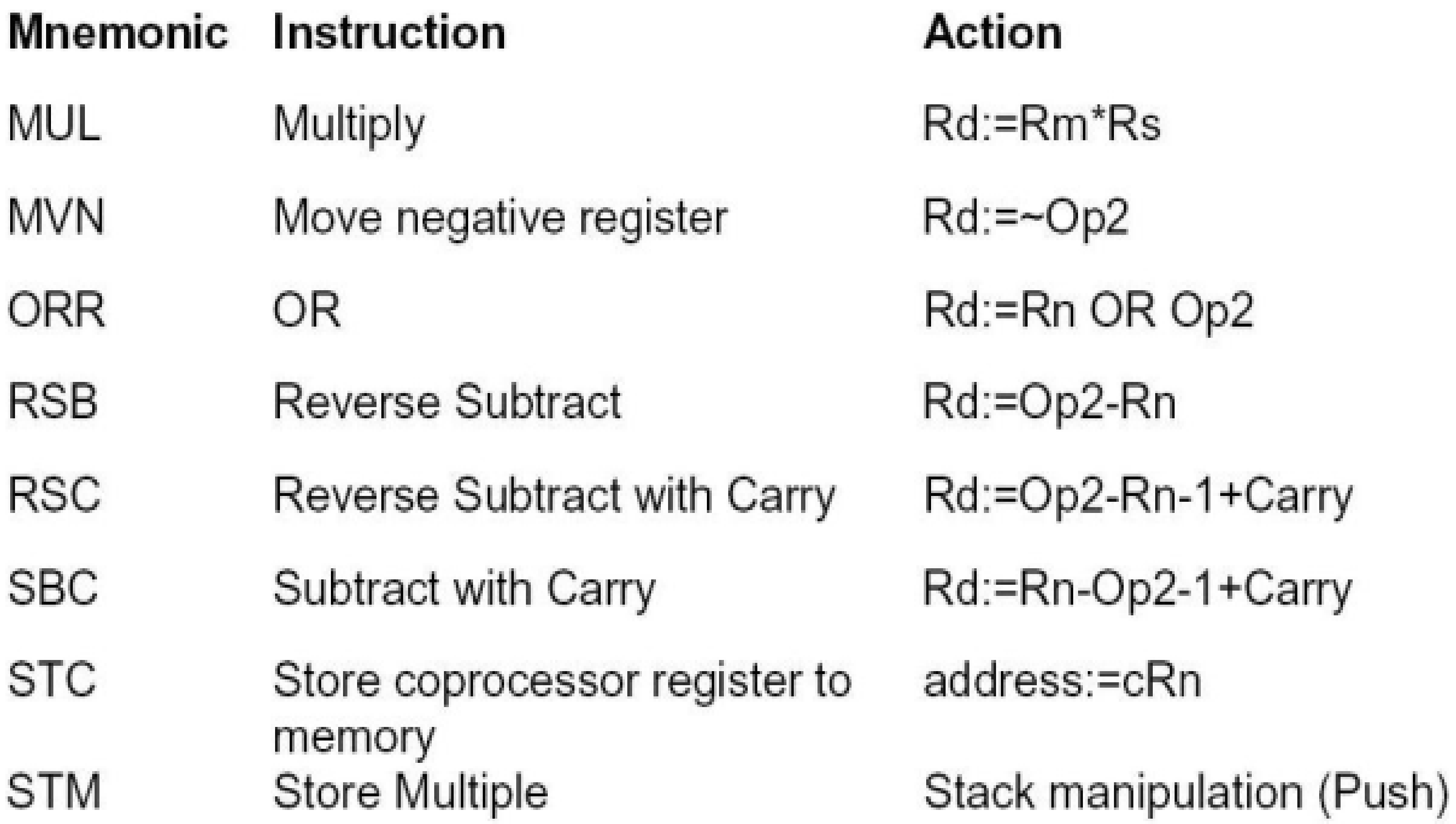
BX	Branch and Exchange	R15:=Rn T bit:=Rn[0]
CDP	Coprocessor Data Processing	(Coprocessor-specific)
CMN	Compare Negative	CPSR flags:=Rn+Op2
CMP	Compare	CPSR flags:=Rn-Op2

ARM Instruction Set Summary (2/4)



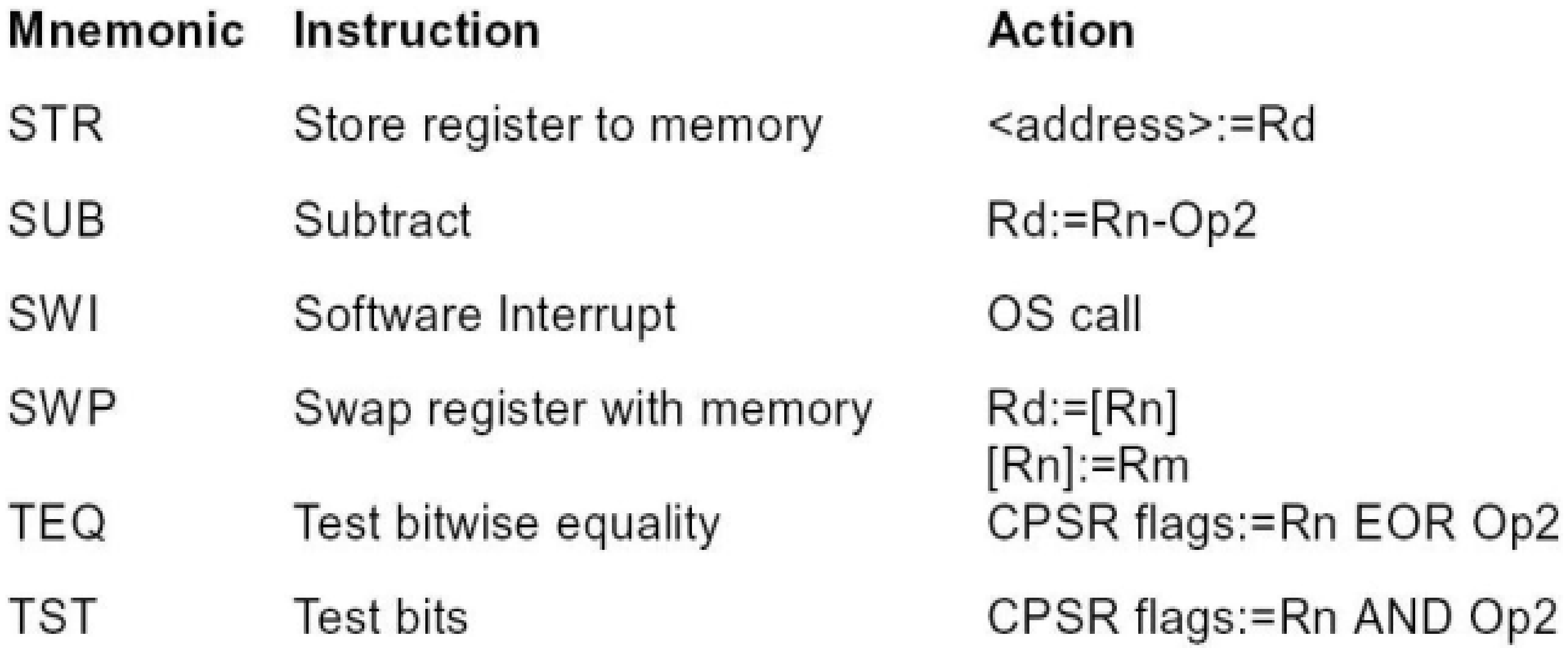
Mnemonic	Instruction	Action
EOR	Exclusive OR	Rd:=Rn^Op2
LDC	Load Coprocessor from memory	(Coprocessor load)
LDM	Load multiple registers	Stack Manipulation (Pop)
LDR	Load register from memory	Rd:=(address)
MCR	Move CPU register to coprocessor register	CRn:=rRn{<op>cRm}
MLA	Multiply Accumulate	Rd:=(Rm*Rs)+Rn
MOV	Move register or constant	Rd:=Op2
MRC	Move from coprocessor register to CPU register	rRn:=cRn{<op>cRm}
MRS	Move PSR status/flags to register	Rn:=PSR
MSR	Move register to PSR status/flags	PSR:=Rm

ARM Instruction Set Summary (3/4)



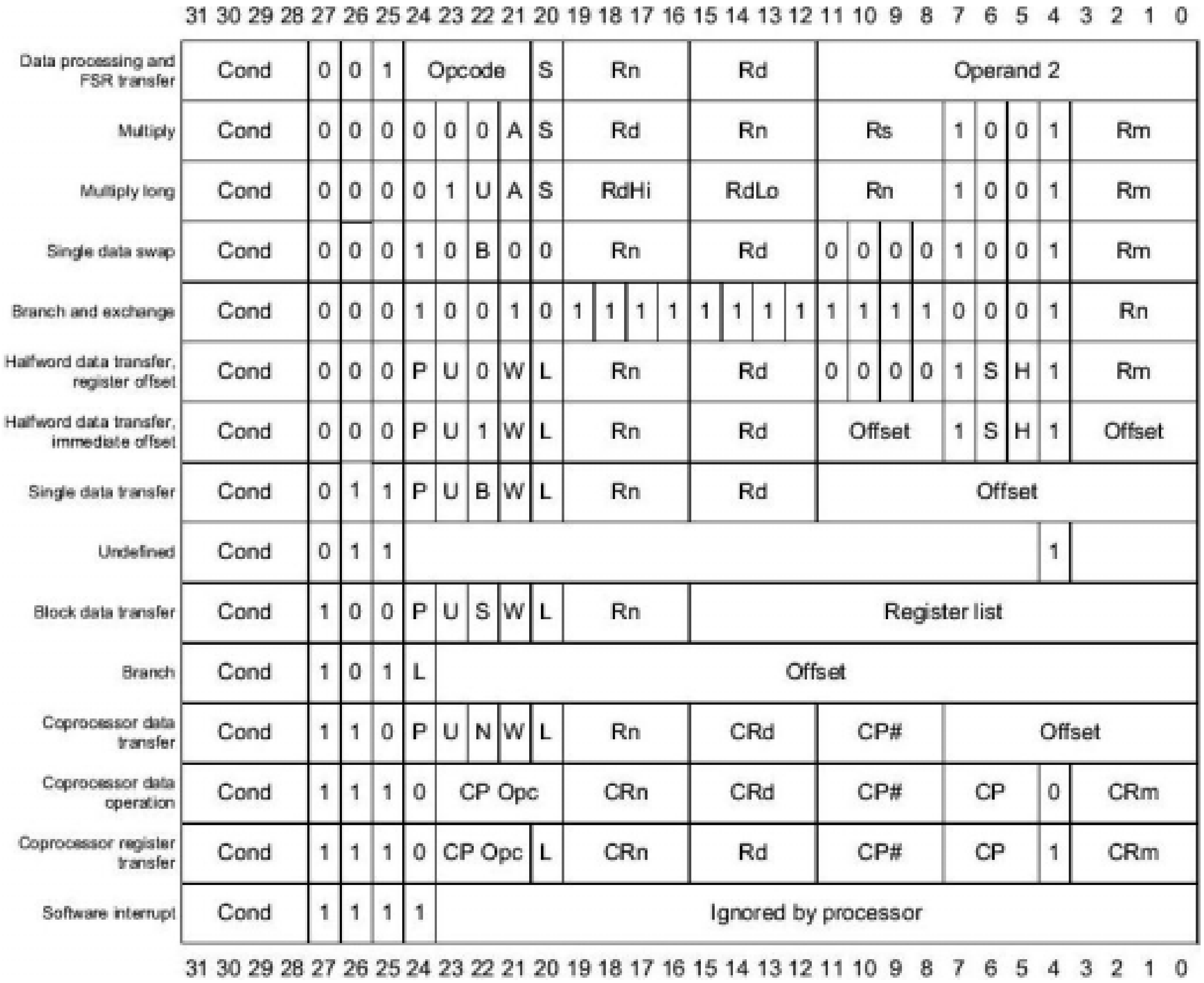
Mnemonic	Instruction	Action
MUL	Multiply	Rd:=Rm*Rs
MVN	Move negative register	Rd:=~Op2
ORR	OR	Rd:=Rn OR Op2
RSB	Reverse Subtract	Rd:=Op2-Rn
RSC	Reverse Subtract with Carry	Rd:=Op2-Rn-1+Carry
SBC	Subtract with Carry	Rd:=Rn-Op2-1+Carry
STC	Store coprocessor register to memory	address:=cRn
STM	Store Multiple	Stack manipulation (Push)

ARM Instruction Set Summary (4/4)



Mnemonic	Instruction	Action
STR	Store register to memory	<address>:=Rd
SUB	Subtract	Rd:=Rn-Op2
SWI	Software Interrupt	OS call
SWP	Swap register with memory	Rd:=[Rn] [Rn]:=Rm
TEQ	Test bitwise equality	CPSR flags:=Rn EOR Op2
TST	Test bits	CPSR flags:=Rn AND Op2

ARM Instruction Set Format



	31 30 29 28	27	26	25	24	23	22	21	20	19 18 17 16	15 14 13 12	11 10 9 8	7	6	5	4	3 2 1 0
Data processing and FSR transfer	Cond	0	0	1	Opcode				S	Rn	Rd	Operand 2					
Multiply	Cond	0	0	0	0	0	0	A	S	Rd	Rn	Rs	1	0	0	1	Rm
Multiply long	Cond	0	0	0	0	1	U	A	S	RdHi	RdLo	Rn	1	0	0	1	Rm
Single data swap	Cond	0	0	0	1	0	B	0	0	Rn	Rd	0 0 0 0	1	0	0	1	Rm
Branch and exchange	Cond	0	0	0	1	0	0	1	0	1 1 1 1	1 1 1 1	1 1 1 1	0	0	0	1	Rn
Halfword data transfer, register offset	Cond	0	0	0	P	U	0	W	L	Rn	Rd	0 0 0 0	1	S	H	1	Rm
Halfword data transfer, immediate offset	Cond	0	0	0	P	U	1	W	L	Rn	Rd	Offset	1	S	H	1	Offset
Single data transfer	Cond	0	1	1	P	U	B	W	L	Rn	Rd	Offset					
Undefined	Cond	0	1	1												1	
Block data transfer	Cond	1	0	0	P	U	S	W	L	Rn	Register list						
Branch	Cond	1	0	1	L	Offset											
Coprocessor data transfer	Cond	1	1	0	P	U	N	W	L	Rn	CRd	CP#	Offset				
Coprocessor data operation	Cond	1	1	1	0	CP Opc				CRn	CRd	CP#	CP			0	CRm
Coprocessor register transfer	Cond	1	1	1	0	CP Opc			L	CRn	Rd	CP#	CP			1	CRm
Software interrupt	Cond	1	1	1	1	Ignored by processor											

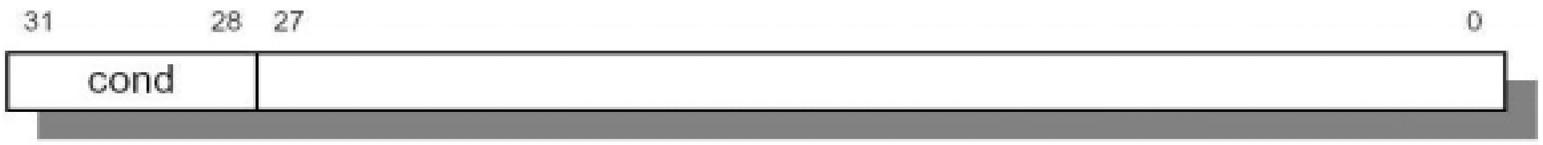
Data Processing Instruction

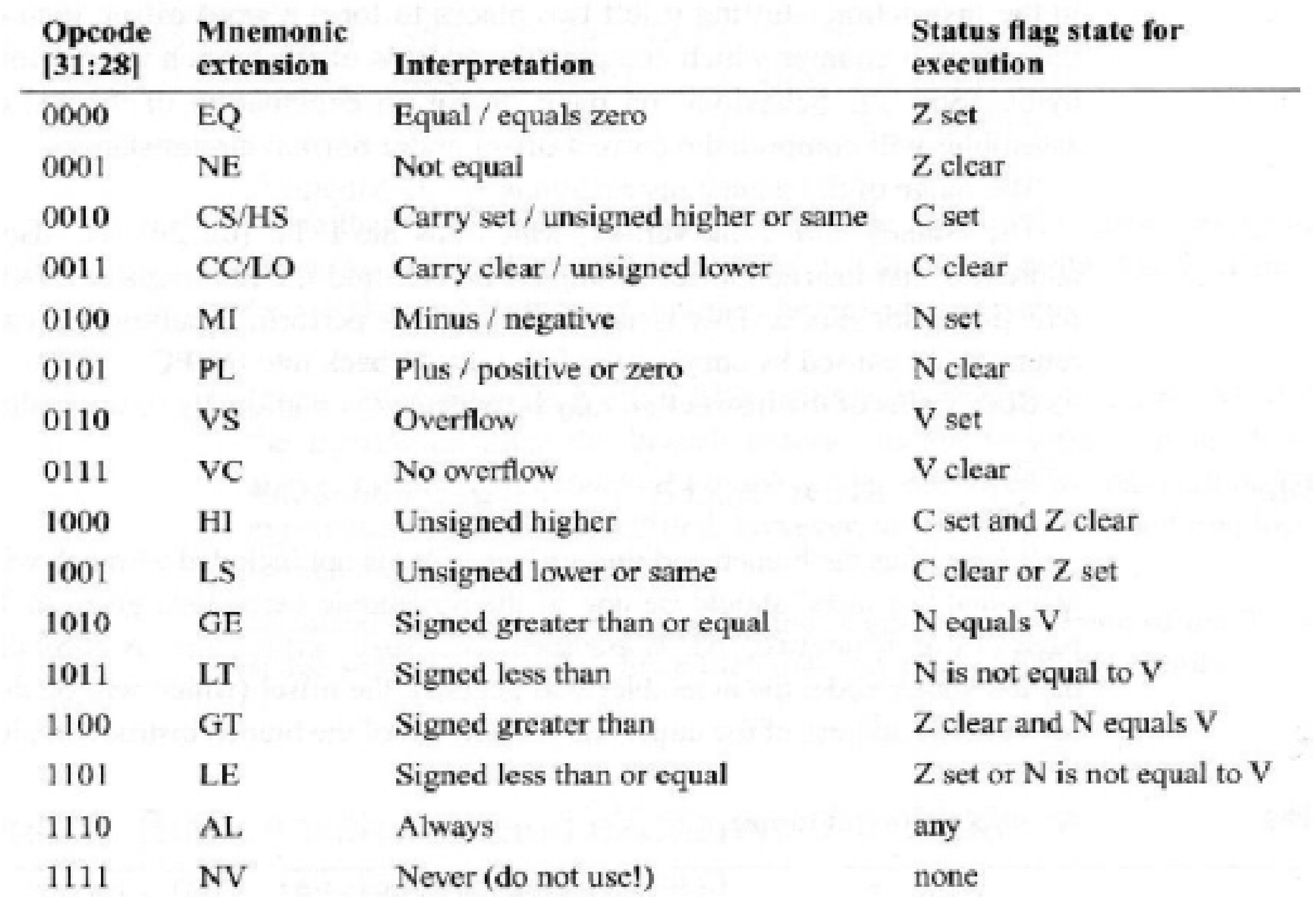
- Consist of
  - Arithmetic (ADD, SUB, RSB)
  - Logical (BIC, AND)
  - Compare (CMP, TST)
  - Register movement (MOV, MVN)
- All operands are 32-bit wide; come from registers or specified as literal in the instruction itself
- Second operand sent to ALU via barrel shifter
- 32-bit result placed in register; long multiply instruction produces 64-bit result
- 3-address instruction format

Conditional Execution (1/2)

- Most instruction sets only allow branches to be executed conditionally.
- However by reusing the condition evaluation hardware, ARM effectively increase number of instruction
  - All instructions contain a condition field which determines whether the CPU will execute them
  - Non-executed instruction still take up 1 cycle
    - To allow other stages in the pipeline to complete
- This reduces the number of branches which would stall the pipeline
  - Allows very dense in-line code
  - The time penalty of not executing several conditional instructions is frequently less than overhead of the branch or instruction call that would otherwise be needed

Conditional Execution (2/2)





Opcode [31:28]	Mnemonic extension	Interpretation	Status flag state for execution
0000	EQ	Equal / equals zero	Z set
0001	NE	Not equal	Z clear
0010	CS/HS	Carry set / unsigned higher or same	C set
0011	CC/LO	Carry clear / unsigned lower	C clear
0100	MI	Minus / negative	N set
0101	PL	Plus / positive or zero	N clear
0110	VS	Overflow	V set
0111	VC	No overflow	V clear
1000	HI	Unsigned higher	C set and Z clear
1001	LS	Unsigned lower or same	C clear or Z set
1010	GE	Signed greater than or equal	N equals V
1011	LT	Signed less than	N is not equal to V
1100	GT	Signed greater than	Z clear and N equals V
1101	LE	Signed less than or equal	Z set or N is not equal to V
1110	AL	Always	any
1111	NV	Never (do not use!)	none

Data Processing Instructions

- ❑ Simple register operands
- ❑ Immediate operands
- ❑ Shifted register operands
- ❑ Multiply

Simple Register Operands (1/2)

- 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, reverse subtraction
RSC r0,r1,r2  ;r0:=r2–r1+C–1
```

  – By default data processing operations do no affect the condition flags

- Bit-wise Logical Operations

```
AND r0,r1,r2  ;r0:=r1ANDr2
ORR r0,r1,r2  ;r0:=r1ORr2
EOR r0,r1,r2  ;r0:=r1XORr2
BIC r0,r1,r2  ;r0:=r1AND (NOT r2), bit clear
```

Simple Register Operands (2/2)

- Register Movement Operations
  - Omit 1st source operand from the format

```
MOV r0,r2      ;r0:=r2
MVN r0,r2      ;r0:=NOT r2, move 1’s complement
```

- Comparison Operations
  - Not produce result; omit the destination from the format
  - Just set the condition code bits (N, Z, C and V) in CPSR

```
CMP r1,r2      ;set cc on r1 - r2, compare
CMN r1,r2      ;set cc on r1 + r2, compare negated
TST r1,r2      ;set cc on r1 AND r2, bit test
TEQ r1,r2      ;set cc on r1 XOR r2, test equal
```

Immediate Operands

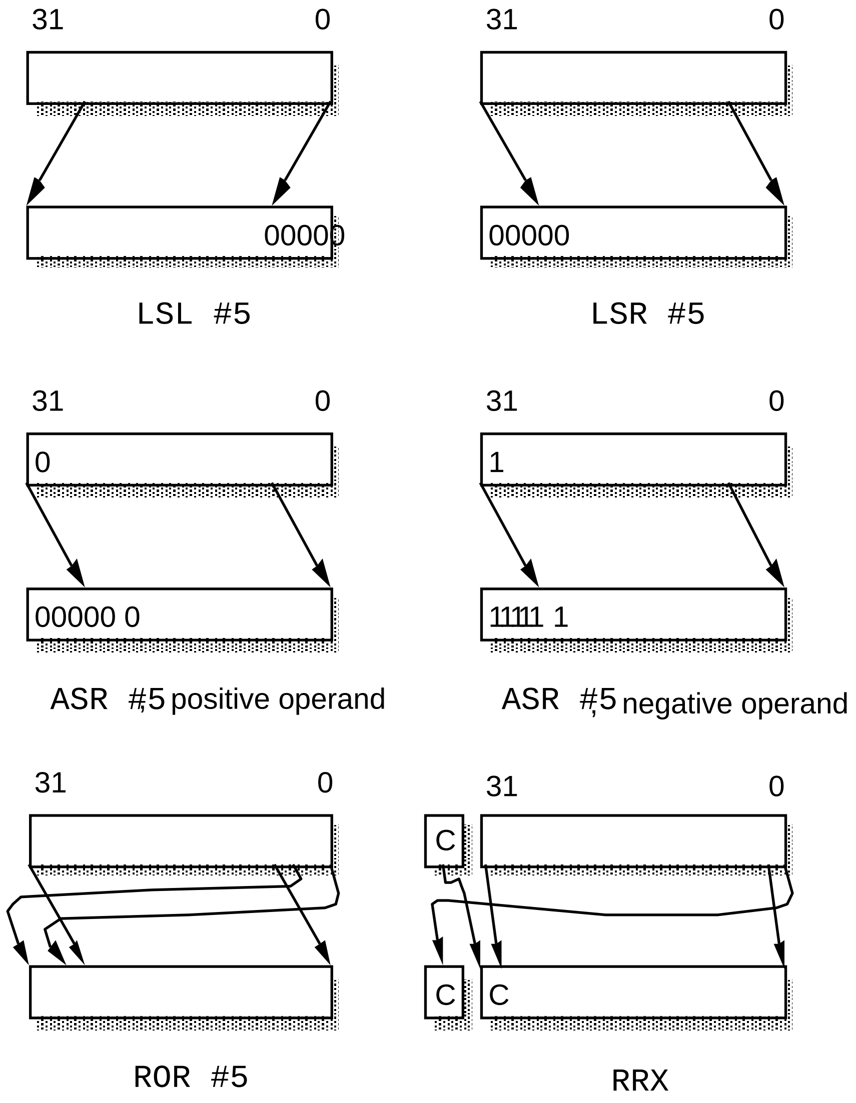
- Replace the second source operand with an immediate operand, which is a literal constant, preceded by “#”

```
    ADD r3,r3,#1          ;r3:=r3+1
    AND r8,r7,#&FF        ;r8:=r7[7:0], &:hexadecimal
```

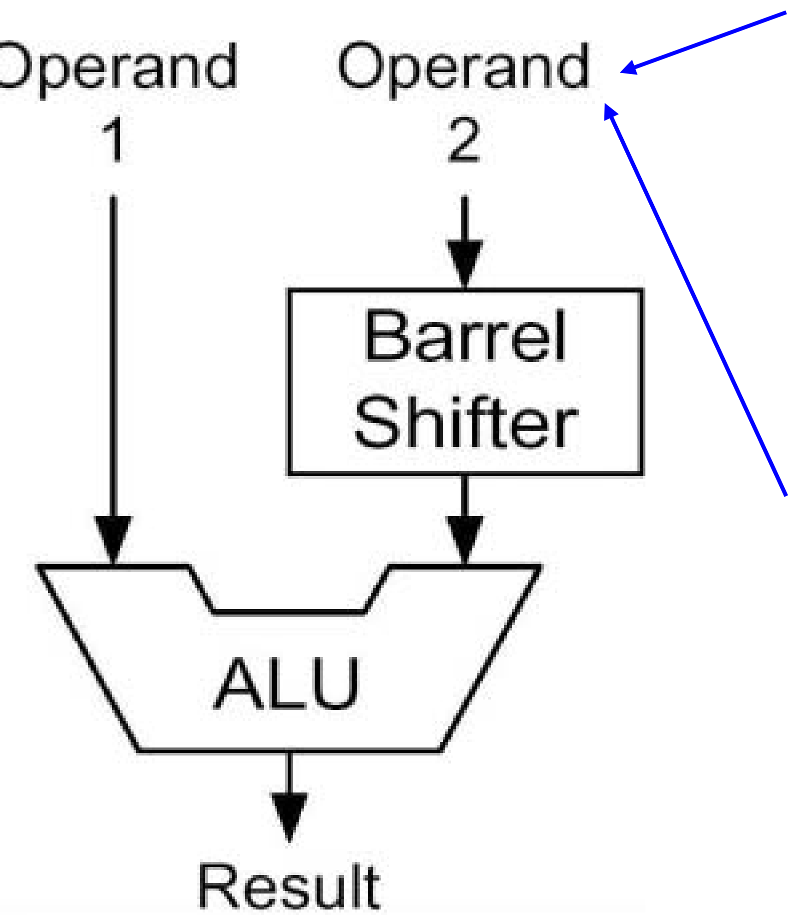
- Since the immediate value is coded within the 32 bits of the instruction, it is not possible to enter every possible 32-bit value as an immediate.

Shift Register Operands

- ADD r3,r2,r2,LSL#3 ;r3 := r2 + 8 * r1
  - A single instruction executed in a single cycle
- ❑ LSL: Logical Shift Left by 0 to 31 places, 0 filled at the lsb end
- ❑ LSR, ASL (Arithmetic Shift Left), ASR, ROR (Rotate Right), RRX (Rotate Right eXtended by 1 place)
  - ADD r5,r5,r3,LSL r2 ; r5:=r5+r3*2^{r2}
  - MOV r12,r4,ROR r3 ;r12:=r4 rotated right by value of r3



Using the Barrel Shifter: the 2nd Operand



- ❑ Register, optionally with shift operation applied
  - Shift value can be either
    - 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

Multiply

- Multiply

```
MUL r4,r3,r2  ;r4:=(r3*r2)[31:0]
```

- Multiply-Accumulate

```
MLA r4,r3,r2,r1    ;r4:=(r3*r2+r1)[31:0]
```

Multiplication by a Constant

- Multiplication by a constant equals to a ((power of 2) +/- 1) can be done in a single cycle
  - Using MOV, ADD or RSBs with an inline shift
- Example: r0 = r1 * 5
- Example: r0 = r1 + (r1 * 4)
  - `ADD r0,r1,r1,LSL #2`
- Can combine several instruction to carry out other multiplies
- Example: r2 = r3 * 119
- Example: r2 = r3 * 17 * 7
- Example: r2 = r3 * (16 + 1) * (8 - 1)
  - `ADD r2,r3,r3,LSL #4  ;r2:=r3*17`
  - `RSB r2,r2,r2,LSL #3  ;r2:=r2*7`

Data Processing Instructions (1/3)

- `<op>{<cond>}{S} Rd,Rn,#<32-bit immediate>`
- `<op>{<cond>}{S} Rd,Rn,Rm,{<shift>}`
  - Omit Rn when the instruction is monadic (MOV, MVN)
  - Omit Rd when the instruction is a comparison, producing only condition code outputs (CMP, CMN, TST, TEQ)
  - <shift> specifies the shift type (LSL, LSR, ASL, ASR, ROR or RRX) and in all cases but RRX, the shift amount which may be a 5-bit immediate (# < # shift>) or a register Rs
- 3-address format
  - 2 source operands and 1 destination register
  - One source is always a register, the second may be a register, a shifted register or an immediate value

Data Processing Instructions (2/3)

Opcode [24:21]	Mnemonic	Meaning	Effect
0000	AND	Logical bit-wise AND	Rd := Rn AND Op2
0001	EOR	Logical bit-wise exclusive OR	Rd := Rn EOR Op2
0010	SUB	Subtract	Rd := Rn - Op2
0011	RSB	Reverse subtract	Rd := Op2 - Rn
0100	ADD	Add	Rd := Rn + Op2
0101	ADC	Add with carry	Rd := Rn + Op2 + C
0110	SBC	Subtract with carry	Rd := Rn - Op2 + C - 1
0111	RSC	Reverse subtract with carry	Rd := Op2 - Rn + C - 1
1000	TST	Test	Scc on Rn AND Op2
1001	TEQ	Test equivalence	Scc on Rn EOR Op2
1010	CMP	Compare	Scc on Rn - Op2
1011	CMN	Compare negated	Scc on Rn + Op2
1100	ORR	Logical bit-wise OR	Rd := Rn OR Op2
1101	MOV	Move	Rd := Op2
1110	BIC	Bit clear	Rd := Rn AND NOT Op2
1111	MVN	Move negated	Rd := NOT Op2

Data Processing Instructions (3/3)

- Allows direct control of whether or not the condition codes are affected by S bit (condition code unchanged when S = 0)
  - N = 1 if the result is negative; 0 otherwise (i.e. N = bit 31 of the result)
  - Z = 1 if the result is zero; 0 otherwise
  - C = 1 carry out from the ALU when ADD, ADC, SUB, SBC, RSB, RSC, CMP, or CMN; carry out from the shifter
  - V = 1 if overflow from bit 30 to bit 31; 0 if no overflow

    (V is preserved in non-arithmetic operations)
- PC may be used as a source operand (address of the instruction plus 8) except when a register-specified shift amount is used
- PC may be specified as the destination register, the instruction is a form of branch (return from a subroutine)

Multiply Instructions (1/2)

- 32-bit product (Least Significant)
  - `MUL{<cond>}{S} Rd,Rm,Rs`
  - `MLA{<cond>}{S} Rd,Rm,Rs,Rn`
- 64-bit Product
  - `<mul>{<cond>}{S} RdHi,RdLo,Rm,Rs`
  - `<mul> is UMULL,UMLAC,SMULL,SMLAL`

Opcode [23:21]	Mnemonic	Meaning	Effect
000	MUL	Multiply (32-bit result)	Rd := (Rm * Rs) [31:0]
001	MLA	Multiply-accumulate (32-bit result)	Rd := (Rm * Rs + Rn) [31:0]
100	UMULL	Unsigned multiply long	RdHi:RdLo := Rm * Rs
101	UMLAL	Unsigned multiply-accumulate long	RdHi:RdLo += Rm * Rs
110	SMULL	Signed multiply long	RdHi:RdLo := Rm * Rs
111	SMLAL	Signed multiply-accumulate long	RdHi:RdLo += Rm * Rs

Multiply Instructions (2/2)

- Accumulation is denoted by “+=”
- Example: form a scalar product of two vectors

```
      MOV r11,#20          ;initialize loop counter
      MOV r10,#0           ;initialize total
Loop  LDR r0,[r8],#4       ;get first component
      LDR r1,[r9],#4       ;get second component
      MLA r10,r0,r1,r10    ;accumulate product
      SUBS r11,r11,#1      ;decrement loop counter
      BNE Loop
```

Data Transfer Instructions

- Three basic forms to move data between ARM registers and memory
  - Single register load and store instruction
    - A byte, a 16-bit half word, a 32-bit word
  - Multiple register load and store instruction
    - To save or restore workspace registers for procedure entry and exit
    - To copy clocks of data
  - Single register swap instruction
    - A value in a register to be exchanged with a value in memory
    - To implement semaphores to ensure mutual exclusion on accesses

Single Register Data Transfer

- ❑ Word transfer
  - `LDR / STR`
- ❑ Byte transfer
  - `LDRB / STRB`
- ❑ Halfword transfer
  - `LDRH / STRH`
- ❑ Load singled byte or halfword-load value and sign extended to 32 bits
  - `LDRSB / LDRSH`
- ❑ All of these can be conditionally executed by inserting the appropriate condition code after STR/LDR
  - `LDREQB`

Addressing

- ❑ Register-indirect addressing
- ❑ Base-plus-offset addressing
  - – Base register
    - r0 – r15
  - – Offset, and or subtract an unsigned number
    - Immediate
    - Register (not PC)
    - Scaled register (only available for word and unsigned byte instructions)
- ❑ Stack addressing
- ❑ Block-copy addressing

Register-Indirect Addressing

- Use a value in one register (base register) as a memory address

LDR r0,[r1]   ;r0:=mem_{32}[r1]

STR r0,[r1]   ;mem_{32}[r1]:=r0

- Other forms
  - Adding immediate or register offsets to the base address

Initializing an Address Pointer

- A small offset to the program counter, r15
  - ARM assembler has a “pseudo” instruction, ADR
- As an example, a program which must copy data from TABLE1 to TABLE2, both of which are near to the code

```
Copy    ADR r1,TABLE1 ;r1 points to TABLE1

        ADR r2,TABLE2 ;r2 points to TABLE2

        …

TABLE1

        …             ;<source>

TABLE2

        …             ;<destination>
```

Single Register Load and Store

- A base register, and offset which may be another register or an immediate value

```
Copy    ADR r1,TABLE1
        ADR r2,TABLE2
Loop    LDR r0,[r1]
        STR r0,[r2]
        ADD r1,r1,#4
        ADD r2,r2,#4
        ???
        …
TABLE1
        …
TABLE2
        …
```

Base-plus-offset Addressing (1/2)

- Pre-indexing

```
LDR r0,[r1,#4]        ;r0:=mem32[r1+4]
```

  – Offset up to 4K, added or subtracted, (# -4)

- Post-indexing

```
LDR r0,[r1],#4        ;r0:=mem32[r1], r1:=r1+4
```

  – Equivalent to a simple register-indirect load, but faster, less code space

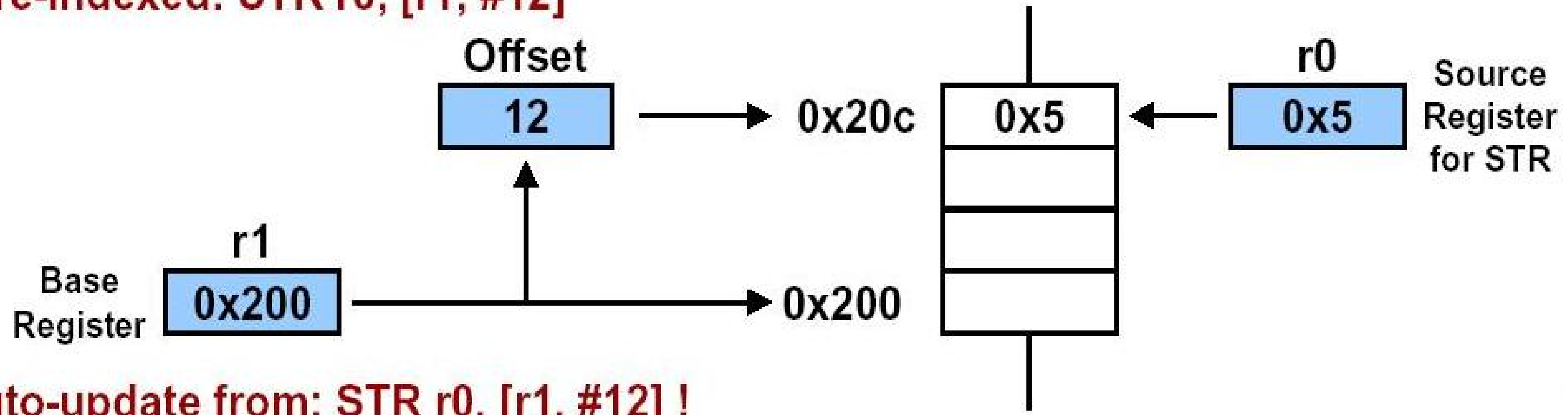
- Auto-indexing

```
LDR r0, [r1,#4]!      ;r0:=mem32[r1+4], r1:=r1+4
```

  – No extra time, auto-indexing performed while the data is being fetched from memory

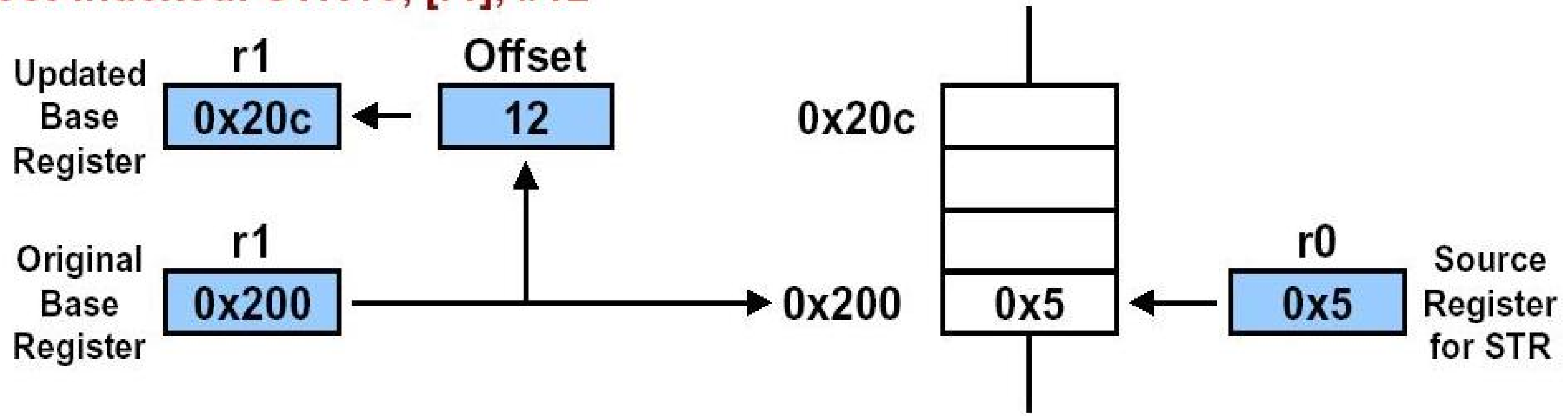
Base-plus-offset Addressing (2/2)

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



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

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



Loading Constants (1/2)

- No single ARM instruction can load a 32-bit immediate constant directly into a register
  - All ARM instructions are 32-bit long
  - ARM instructions do not use the instruction stream as data
- The data processing instruction format has 12 bits available for operand 2
  - If used directly, this would only give a range of 4096
- Instead it is used to store 8-bit constants, give a range of 0-255
- These 8 bits can then be rotated right through an even number of positions
- This gives a much larger range of constants that can be directly loaded, through some constants will still need to be loaded from memory

Loading Constant (2/2)

- To load a constant, simply move the required value into a register – the assembler will convert to the rotate form for us
  - `MOV r0,#4096 ;MOV r0,#0x1000 (0x40 ror 26)`

- The bitwise complements can also be formed using MVN:
  - `MOV r0,#&FFFFFFFF           ;MVN r0,#0`

- Value 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

```
– MOV r0,=&FF               ;MOV r0,#0xFF
– LDR r0,=&55555555         ;LDR r0,[PC,#Imm10]
```

Multiple Register Data Transfer (1/2)

- The load and store multiple instructions (LDM/STM) allow between 1 and 16 registers to be transferred to or from memory
  - Order of register transfer cannot be specified, order in the list is insignificant
  - Lowest register number is always transferred to/form lowest memory location accessed
- The transferred registers can be either
  - Any subset of the current bank of registers (default)
  - Any subset of the user mode bank of registers when in a privileged mode (postfix instruction with a “^”)
- Base register used to determine where memory access should occur
  - 4 different addressing modes
  - Base register can e optionally updated following the transfer (using “!”)

Multiple Register Data Transfer (2/2)

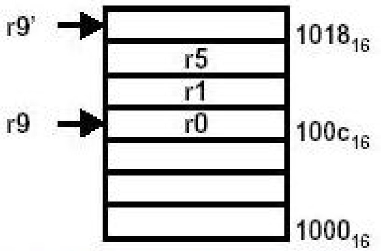
- These instruction are very efficient for
  - Moving block of data around memory
  - Saving and restoring context – stack
- Allow any subset (or all, r0 to r15) of the 16 registers to be transferred with a single instruction

LDMIA r1,{r0,r2,r5} ;r0:=mem_{32}[r1]
;r2:=mem_{32}[r1+4]
;r5:=mem_{32}[r1+8]

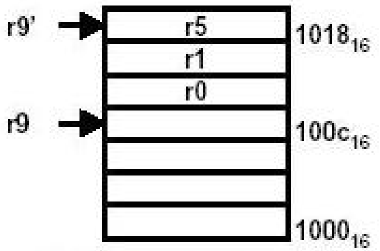
Stack Processing

- A stack is usually implemented as a linear data structure which grows up (an ascending stack) or down (a descending stack) memory
- A stack pointer holds the address of the current top of the stack, either by pointing to the last valid data item pushed onto the stack (a full stack), or by pointing to the vacant slot where the next data item will be placed (an empty stack)
- ARM multiple register transfer instructions support all four forms of stacks
  - **Full ascending**: grows up; base register points to the highest address containing a valid item
  - **empty ascending**: grows up; base register points to the first empty location above the stack
  - **Full descending**: grows down; base register points to the lowest address containing a valid data
  - **empty descending**: grows down; base register points to the first empty location below the stack

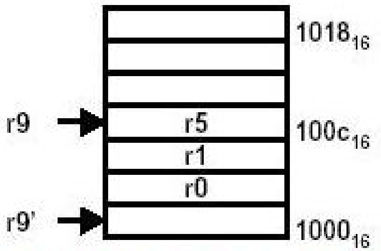
Block Copy Addressing



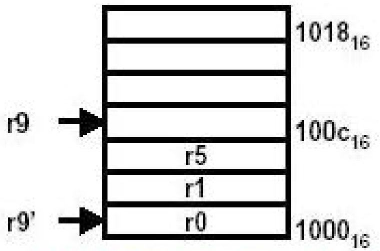
STMIA r9!, {r0, r1, r5}



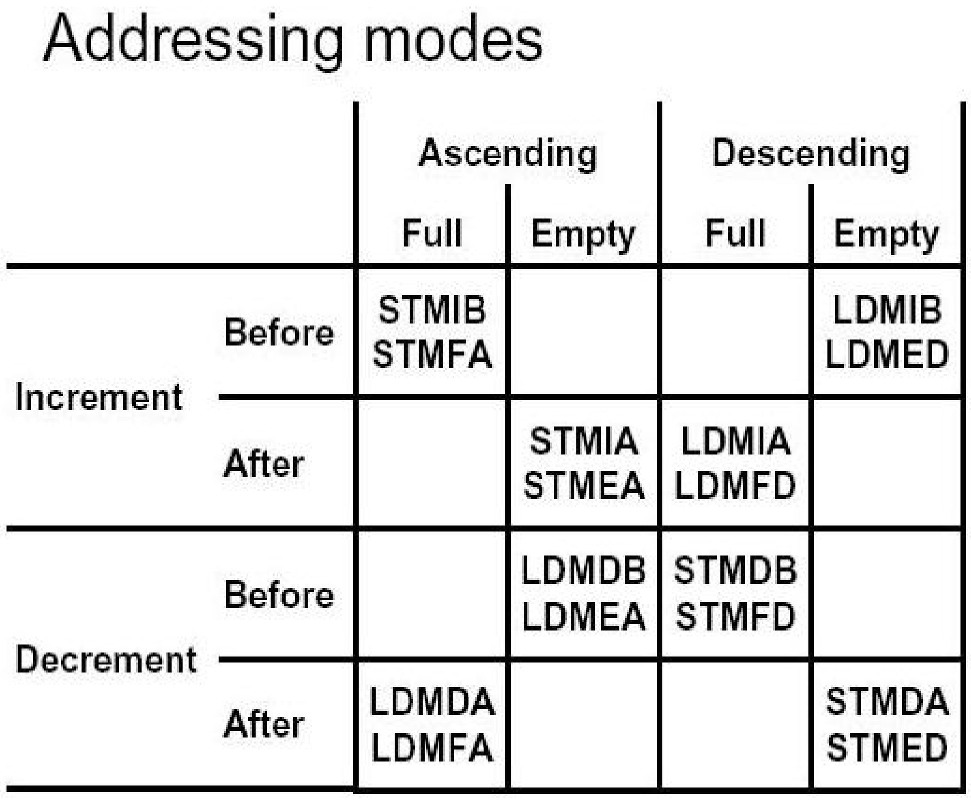
STMIB r9!, {r0, r1, r5}



STMDA r9!, {r0, r1, r5}



STMDB r9!, {r0, r1, r5}



Addressing modes

		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

Single Word and Unsigned Byte Data Transfer instructions

- ❑ Pre-indexed form
  - `LDR|STR{<cond>}{B} Rd, [Rn, <offset>]{!}`
- ❑ Post-indexed form
  - `LDR|STR{<cond>}{B} Rd, [Rn], <offset>`
- ❑ PC-relative form
  - `LDR|STR{<cond>}{B} Rd, LABEL`

  - LDR: ’load register’; STR: ’store register’
  - ‘B’ unsigned byte transfer, default is word;
  - <offset> may be # +/-<12-bit immediate> or +/- Rm{, shift}
  - !: auto-indexing
  - T flag selects the user view of the memory translation and protection system

Example

- Store a byte in r0 to a peripheral

```
          LDR r1, UARTADD      ; UART address into r1
          STRB r0, [r1]        ; store data to UART
UARTADD   &    &10000000       ; address literal
```

Half-word and Signed Byte Data Transfer Instructions

- Pre-indexed form
  - `LDR|STR{<cond>}H|SH|SB Rd,[Rn,<offset>]{!}`
- Post-indexed form
  - `LDR|STR{<cond>}H|SH|SB Rd,[Rn],<offset>`

  - <offset> is # +/-<8-bit immediate> or +/- Rm
  - H|SH|SB selects the data type
    - Unsigned half-word
    - Signed half-word and
    - Signed byte
    - Otherwise the assumble format is for word and unsigned byte transfer

Example

- Expand an array of signed half-words into an array of words

```
        ADR  r1,ARRAY1       ;half-word array start
        ADR  r2,ARRAY2       ;word array start
        ADR  r3,ENDARR1      ;ARRAY1 end + 2
Loop    LDRSH     r0,[r1],#2;get signed half-word
        STR  r0,[r2],#4      ;save word
        CMP  r1,r3           ;check for end of array
        BLT  Loop            ;if not finished, loop
```

Multiple Register Transfer instructions

- `LDR|STR{<cond>}{B}<add mode> Rn{!}, <register>`
  - <add mode> specifies one of the addressing modes
  - '!': auto-indexing
  - <registers> a list of registers, e.g., {r0, r3-r7, pc}
- In non-user mode, the CPSR may be restored by

  `LDM{<cond>}<add mode> Rn{!}, <registers + PC>^`
- In non-user mode, the user registers may be saved or restored by

  `LDM|STM{<cond>}<add mode> Rn, <registers - PC>^`
  - The register list must not contain PC and write-back is no allowed

Example

- Save 3 work registers and the return address upon entering a subroutine (assume r13 has been initialized for use as a stack pointer)

```
    STMFD r13!,{r0-r2,r14}
```

- Restore the work registers and return

```
    LDMFD r13!,{r0-r2,PC}
```

Swap Memory and Register Instructions

- `SWP{<cond>}{B} Rd,Rm,[Rn]`
- `Rd <- [Rn], [Rn] <- Rm`

- Combine a load and a store of a word or an unsigned byte in a single instruction
- Example

```
ADR r0,SEMAPHORE
SWPB r1,r1,[r0]    ;exchange byte
```

Status Register to General Register Transfer instructions

- ❑ `MRS{<cond>} Rd,CPSR|SPSR`

- ❑ The CPSR or the current mode SPSR is copied into the destination register. All 32 bits are copied.

- ❑ Example

```
MRS r0,CPSR
MRS r3,SPSR
```

General Register to Status Register Transfer instructions

- `MSR{<cond>} CPSR_<field>|SPSR_<field>,#<32-bit immediate>`

  `MSR{<cond>} CPSR_<field>|SPSR_<field>,Rm`
  - <field> is one of
    - c – the control field PSR[7:0]
    - x – the extension field PSR[15:8]
    - s – the status field PSR[23:16]
    - f – the flag field PSR[31:24]
- Example
  - Set N, X, C, V flags
    - MSR CPSR_f,#&f0000000

Control Flow Instructions

- Branch instructions
- Conditional branches
- Conditional execution
- Branch and link instructions
- Subroutine return instructions
- Supervisor calls
- Jump tables

Branch Instructions

```
        B  LABEL
        …
LABEL            …
```

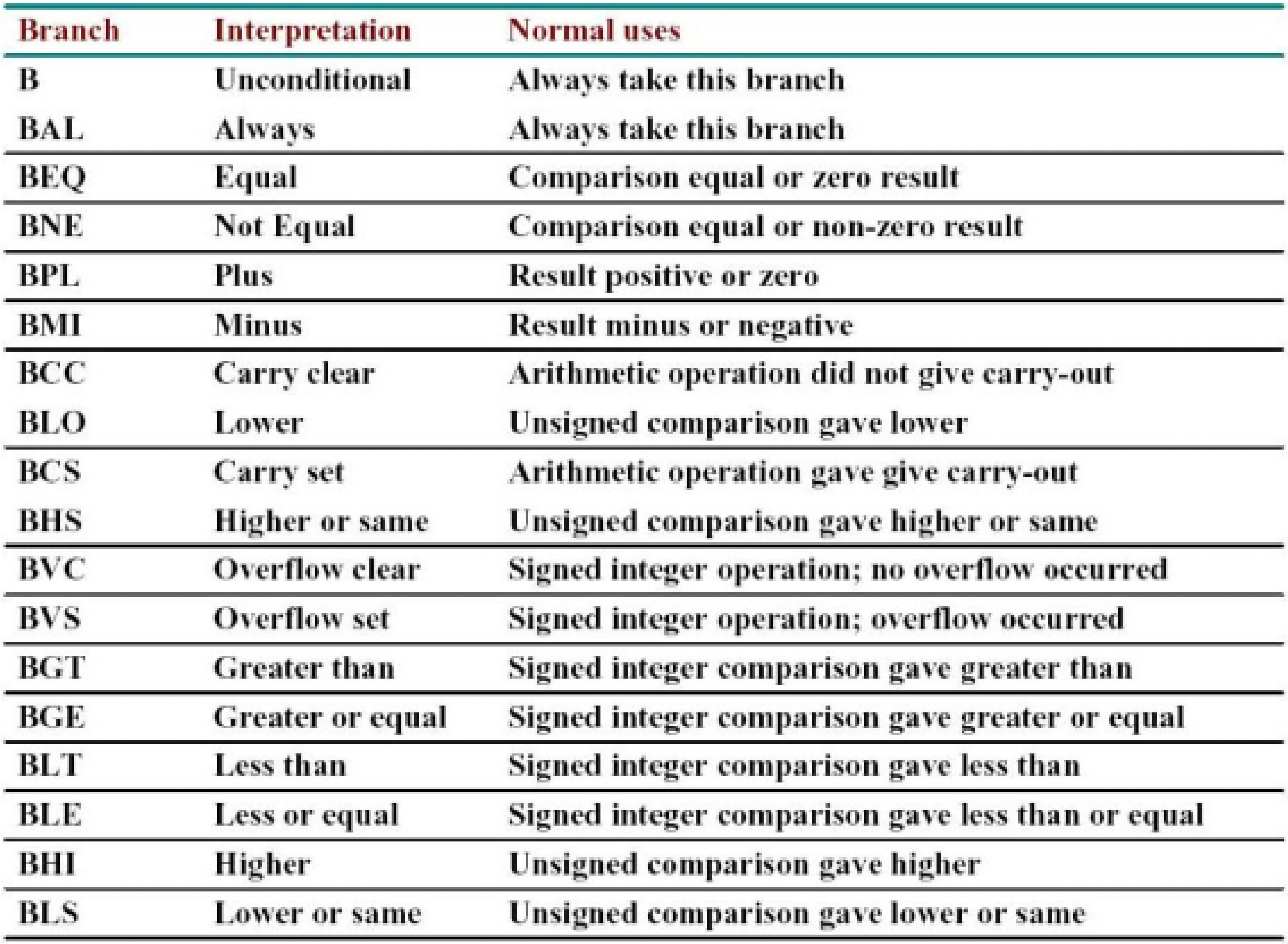
- LABEL comes after or before the branch instruction

Conditional Branches

- The branch has a condition associated with it and it is only executed if the condition codes have the correct value – taken or not taken

```
     MOV r0,#0      ;initialize counter
Loop …
     ADD r0,r0,#1   ;increment loop counter
     CMP r0,#10     ;compare with limit
     BNE  Loop      ;repeat if not equal
                    ;else fail through
```

Conditional Branch



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

Conditional Execution

- An unusual feature of the ARM instruction set is that conditional execution applies no only to branches but to all ARM instructions

```
        CMP r0,#5
        BEQ Bypass   ;if (r0!=5)
        ADD r1,r1,r0 ;{r1=r1+r0}
        SUB r1,r1,r2
Bypass …
```

$\Rightarrow$

```
CMP r0,#5
ADDNE r1,r1,r0
SUBNE r1,r1,r2
```

- Whenever the conditional sequence is 3 instructions for fewer it is better (smaller and faster) to exploit conditional execution than to use a branch

```
if((a==b)&&(c==d)) e++;
```

$\Rightarrow$

```
CMP r0,r1
CMPEQ r2,r3
ADDEQ r4,r4,#1
```

Branch and Link Instructions

- Perform a branch, save the address following the branch in the link register, r14

```
        BL SUBR           ;branch to SUBR
        …                 ;return here
SUBR    …                 ;subroutine entry point
        MOV PC,r14        ;return
```

- For nested subroutine, push r14 and some work registers required to be saved onto a stack in memory

```
        BL SUB1
        …
SUB1    STMFD r13!,{r0-r2,r14};save work and link regs
        …
SUB2
```

Subroutine Return Instructions

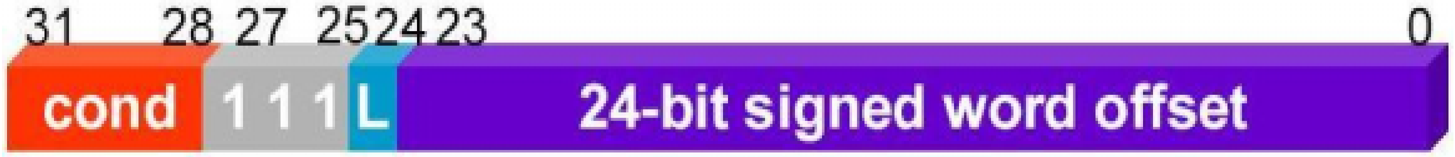
```
SUB         …
        MOV PC,r14     ;copy r14 into r15 to return
```

- Where the return address has been pushed onto a stack

```
SUB1 STMFD r13!,{r0-r2,r14} ;save work regs and link
     BL SUB2
    …
     LDMFD r13!,{r0-r2,PC}  ;restore work regs &
                            ;return
```

Branch and Branch with Link (B,BL)

- B {L} {<cond>} <target address>
  - <target address> is normally a label in the assembler code.



24-bit offset, sign-extended, shift left 2 places
+ PC (address of branch instruction + 8)

target address

Examples

- Unconditional jump

```
        B     LABEL
        …
LABEL   …
```

- Loop ten times

```
        MOV r0,#10
Loop          …
        SUBS r0,#1
        BNE Loop
        …
```

- Call a subroutine

```
        BL SUB
        …
SUB     …
        MOV PC,r14
```

- Conditional subroutine call

```
CMP r0,#5
BLLT SUB1 ;if r0<5,
          ;call sub1
BLGE SUB2 ;else call
          ;SUB2
```

Branch, Branch with Link and eXchange

- `B{L}X{<cond>} Rm`
  - The branch target is specified in a register, Rm
  - Bit[0] of Rm is copied into the T bit in CPSR; bit[31:1] is moved into PC
  - If Rm[0] is 1, the processor switches to execute Thumb instructions and begins executing at the address in Rm aligned to a half-word boundary by clearing the bottom bit
  - If Rm[0] is 0, the processor continues executing ARM instructions and begins executing at the address in Rm aligned to a word boundary by clearing Rm[1]
- `BLX <target address>`
  - Call Thumb subroutine from ARM
  - The H bit (bit 24) is also added into bit 1 of the resulting addressing, allowing an odd half-word address to be selected for the target instruction which will always be a Thumb instruction

Example

- A call to a Thumb subroutine

```
     CODE32
     …
     BLX TSUB  ;call Thumb subroutine
     …
     CODE16    ;start of Thumb code
TSUB …
     BX r14    ;return to ARM code
```

Supervisor Calls

- The supervisor is a program which operates at a privileged level, which means that it can do things that a use-level program cannot do directly (e.g. input or output)
- SWI instruction
  - Software interrupt or supervisor call

```
SWI SWI_WriteC          ;output r0[7:0]
SWI SWI_Exit            ;return to monitor program
```

Software Interrupt (SWI)

- ❑ `SWI{<cond>}<24-bit immediate>`
  - – Used for calls to the operating system and is often called a “supervisor call”
  - – It puts the processor into supervisor mode and begins executing instruction from address 0x08
    - Save the address of the instruction after SWI in r14_svc
    - Save the CPSR in SPSR_svc
    - Enter supervisor mode and disable IRQs by setting CPSR[4:0] to 10011_2 and CPSR[7] to 1
    - Set PC to 08_{16} and begin executing the instruction there
  - – The 24-bit immediate does not influence the operation of the instruction but may be interpreted by the system code

Examples

- ❑ Output the character ‘A’

```
        MOV     r0,#’A’
        SWI     SWI_WriteC
```

- ❑ Finish executing the user program and return to the monitor

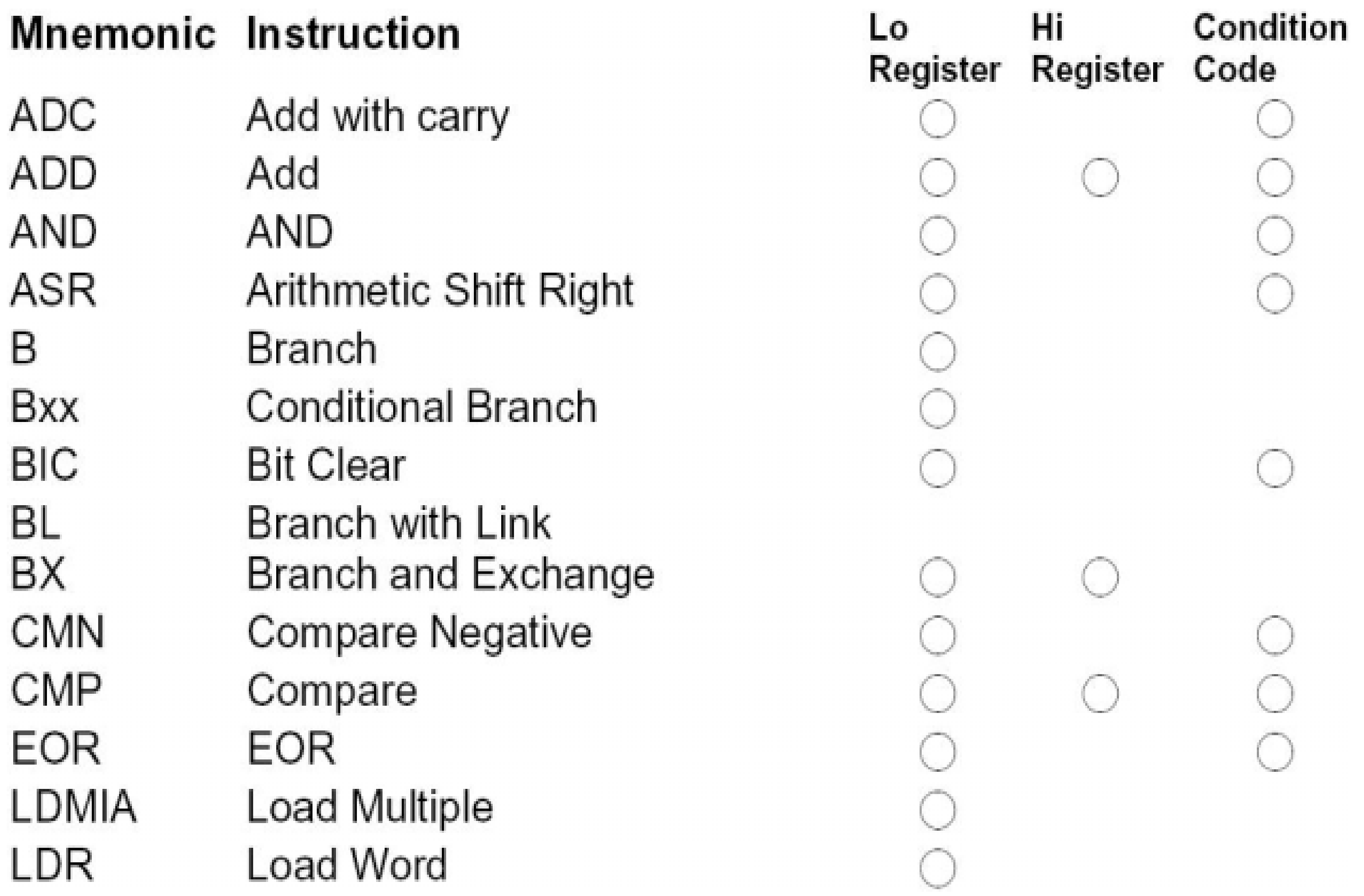
```
        SWI     SWI_EXIT
```

- ❑ A subroutine to output a text string

```
        BL STROUT
        =  “Hello World”, &0a, &0d,0
        …
STROUT  LDRB r0,[r14], #1       ;get character
        CMP r0,#0               ;check for end marker
        SWINE SWI_WriteC        if not end, print
        BNE STROUT              ; … ,loop
        ADD r14,#3              ;align to next word
        BIC r14,#3
        MOV PC,r14              ;return
```

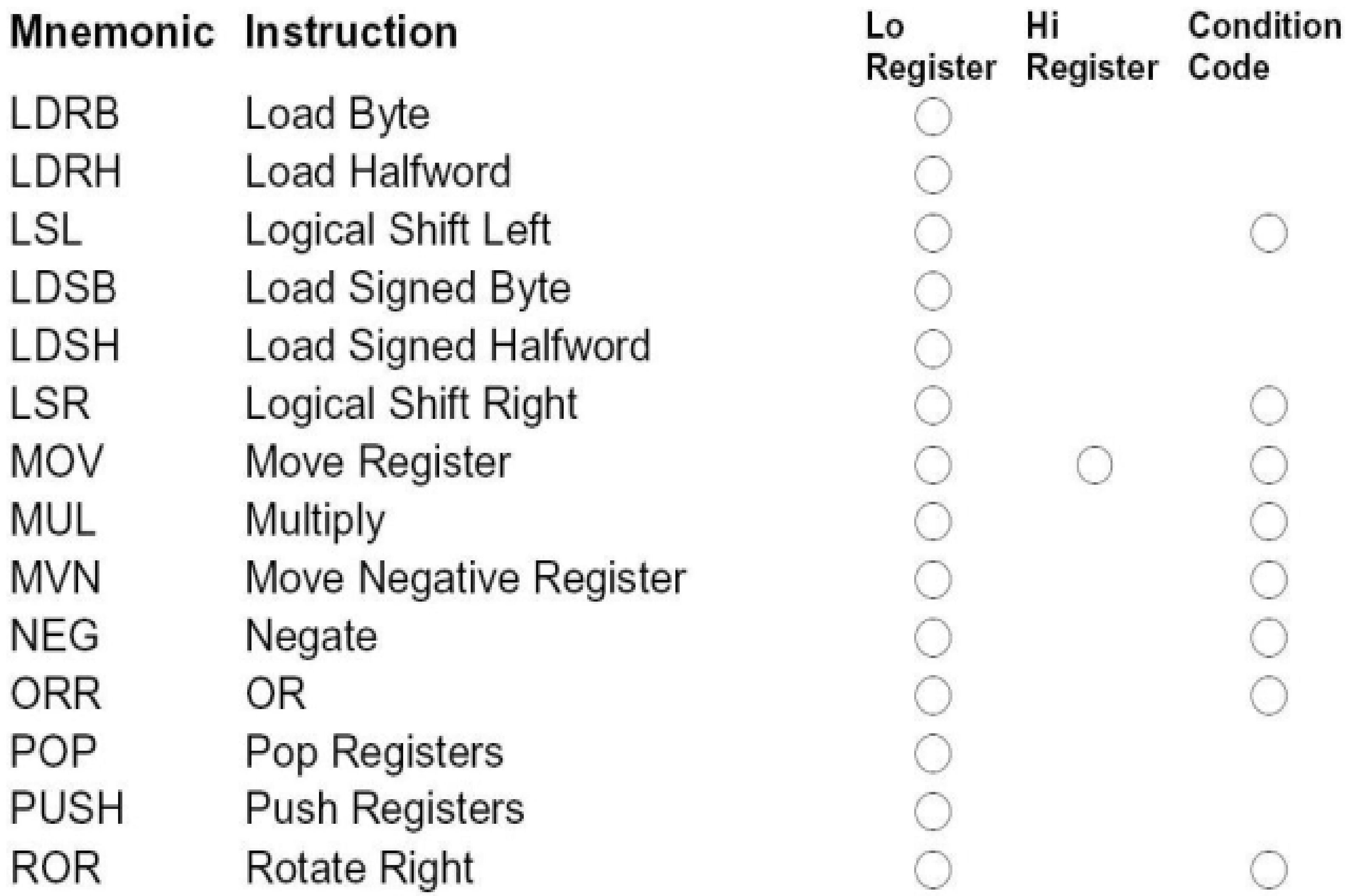
16-bit instruction set

Thumb Instruction Set (1/3)



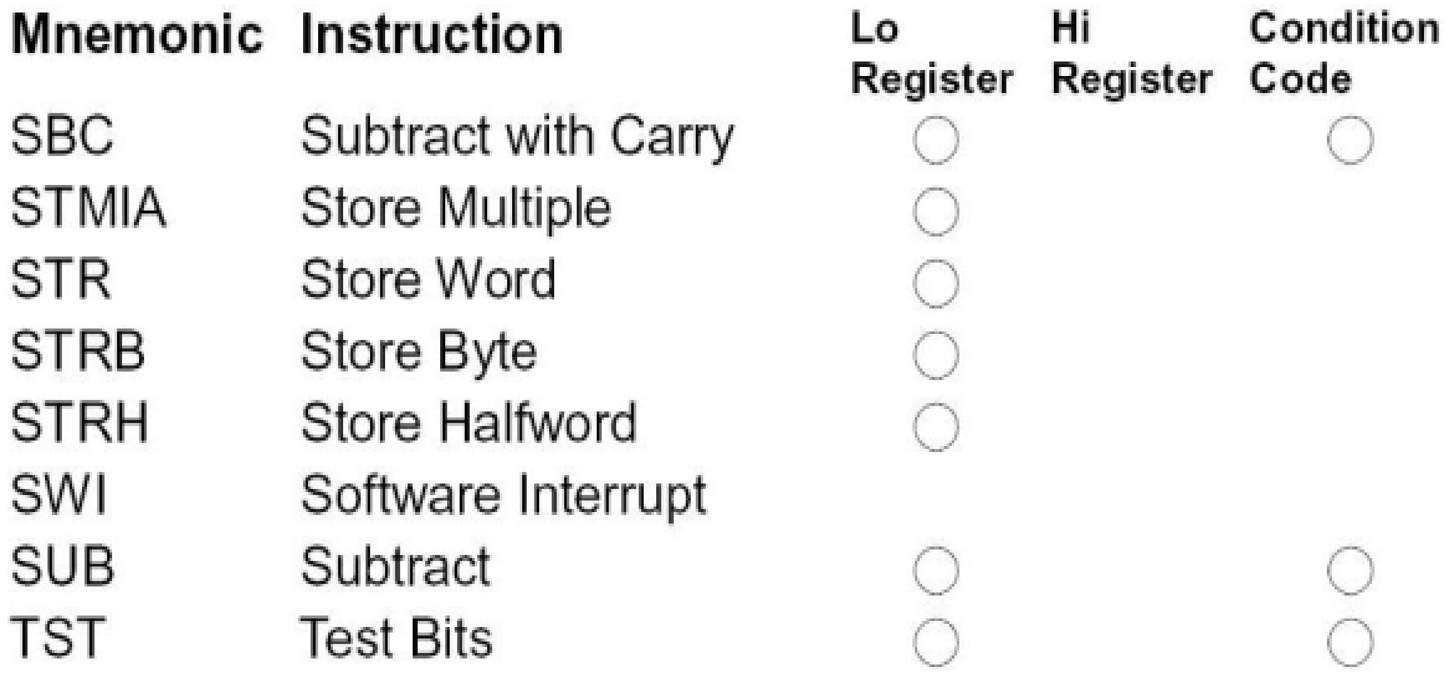
Mnemonic	Instruction	Lo Register	Hi Register	Condition Code
ADC	Add with carry	○		○
ADD	Add	○	○	○
AND	AND	○		○
ASR	Arithmetic Shift Right	○		○
B	Branch	○		
Bxx	Conditional Branch	○		
BIC	Bit Clear	○		○
BL	Branch with Link			
BX	Branch and Exchange	○	○	
CMN	Compare Negative	○		○
CMP	Compare	○	○	○
EOR	EOR	○		○
LDMIA	Load Multiple	○		
LDR	Load Word	○		

Thumb Instruction Set (2/3)



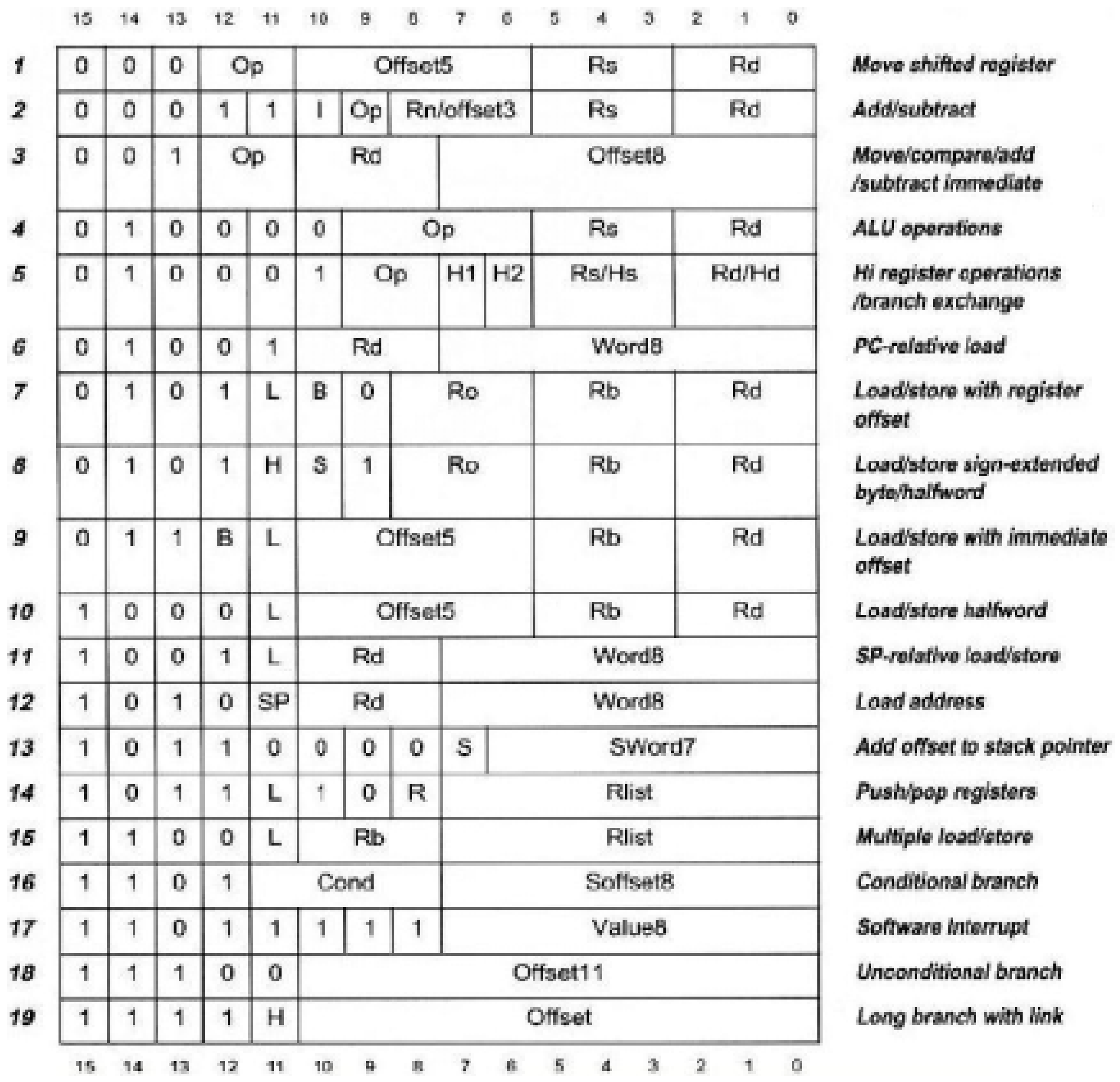
Mnemonic	Instruction	Lo Register	Hi Register	Condition Code
LDRB	Load Byte	○		
LDRH	Load Halfword	○		
LSL	Logical Shift Left	○		○
LDSB	Load Signed Byte	○		
LDSH	Load Signed Halfword	○		
LSR	Logical Shift Right	○		○
MOV	Move Register	○	○	○
MUL	Multiply	○		○
MVN	Move Negative Register	○		○
NEG	Negate	○		○
ORR	OR	○		○
POP	Pop Registers	○		
PUSH	Push Registers	○		
ROR	Rotate Right	○		○

Thumb Instruction Set (3/3)



Mnemonic	Instruction	Lo Register	Hi Register	Condition Code
SBC	Subtract with Carry	○		○
STMIA	Store Multiple	○		
STR	Store Word	○		
STRB	Store Byte	○		
STRH	Store Halfword	○		
SWI	Software Interrupt			
SUB	Subtract	○		○
TST	Test Bits	○		○

Thumb Instruction Format



	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	0	0	0	Op		Offset5					Rs			Rd			Move shifted register
2	0	0	0	1	1	I	Op	Rn/offset3			Rs			Rd			Add/subtract
3	0	0	1	Op		Rd			Offset8								Move/compare/add /subtract immediate
4	0	1	0	0	0	0	Op				Rs			Rd			ALU operations
5	0	1	0	0	0	1	Op		H1	H2	Rs/Hs			Rd/Hd			Hi register operations /branch exchange
6	0	1	0	0	1	Rd			Word8								PC-relative load
7	0	1	0	1	L	B	0	Ro			Rb			Rd			Load/store with register offset
8	0	1	0	1	H	S	1	Ro			Rb			Rd			Load/store sign-extended byte/halfword
9	0	1	1	B	L	Offset5					Rb			Rd			Load/store with immediate offset
10	1	0	0	0	L	Offset5					Rb			Rd			Load/store halfword
11	1	0	0	1	L	Rd			Word8								SP-relative load/store
12	1	0	1	0	SP	Rd			Word8								Load address
13	1	0	1	1	0	0	0	0	S	SWord7							Add offset to stack pointer
14	1	0	1	1	L	1	0	R	Rlist								Push/pop registers
15	1	1	0	0	L	Rb			Rlist								Multiple load/store
16	1	1	0	1	Cond				Soffset8								Conditional branch
17	1	1	0	1	1	1	1	1	Value8								Software Interrupt
18	1	1	1	0	0	Offset11											Unconditional branch
19	1	1	1	1	H	Offset											Long branch with link
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	

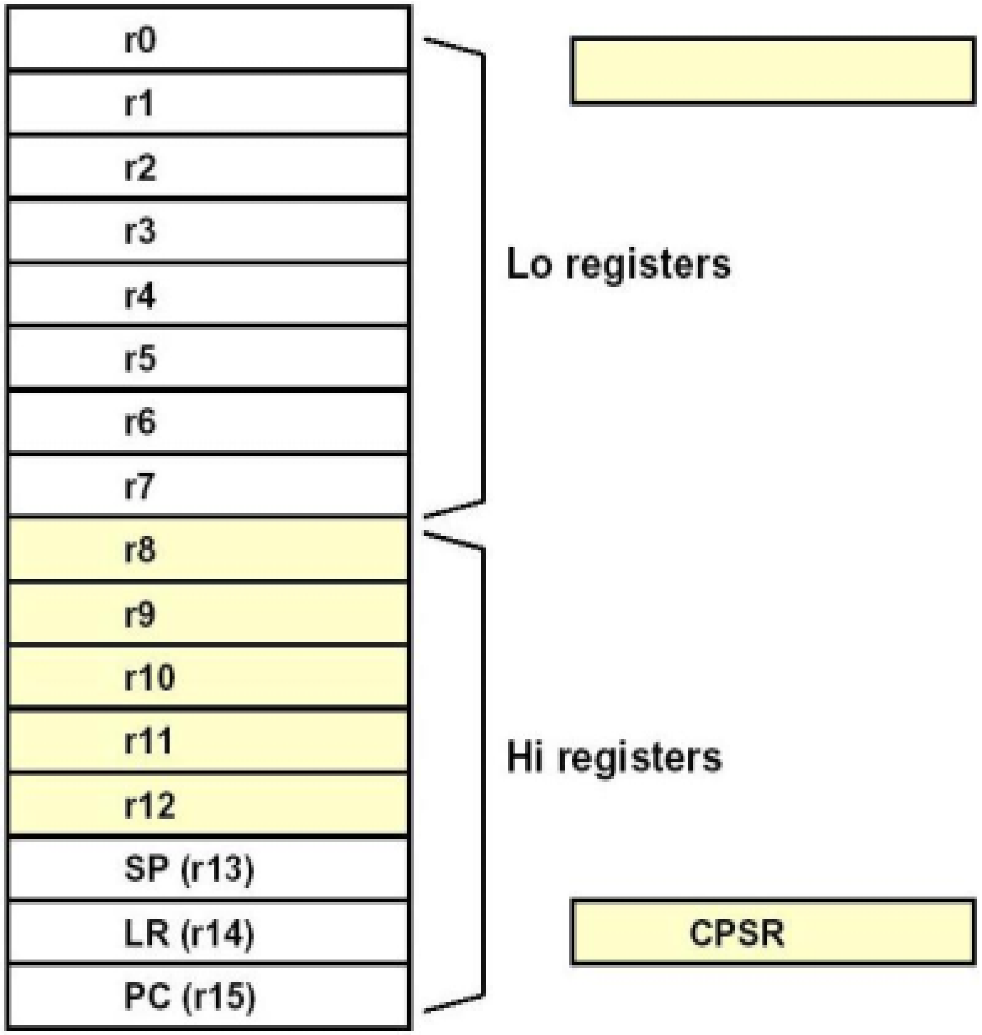
Register Access in Thumb

- Not all registers are directly accessible in Thumb
- Low register **r0 – r7**: fully accessible
- High register **r8 – r12**: only accessible with MOV, ADD, CMP; only CMP sets the condition code flags
- **SP** (Stack Pointer), **LR** (Link Register) & **PC** (Program Counter): limited accessibility, certain instructions have implicit access to these
- **CPSR**: only indirect access
- **SPSR**: no access

Thumb-ARM Difference

- Thumb instruction set is a subset of the ARM instruction set and the instructions operate on a restricted view of the ARM registers
- Most Thumb instructions are executed unconditionally (All ARM instructions are executed conditionally)
- Many Thumb data processing instructions use 2 2-address format, i.e. the destination register is the same as one of the source registers (ARM data processing instructions, with the exception of the 64-bit multiplies, use a 3-address format)
- Thumb instruction formats are less regular than ARM instruction formats => dense encoding

Thumb Accessible Registers



Shaded registers have restricted access

Branches

- Thumb defines three PC-relative branch instructions, each of which have different offset ranges
  - Offset depends upon the number of available bits
- Conditional Branches
  - `B<cond> label`
  - 8-bit offset: range of -128 to 127 instruction (+/-256 bytes)
  - Only conditional Thumb instructions
- Unconditional Branches
  - `B label`
  - 11-bit offset: range of -1024 to 1023 instructions (+/-2Kbytes)
- Long Branches with Link
  - `BL subroutine`
  - Implemented as a pair of instructions
  - 22-bit offset: range of -2097152 to 2097151 instruction (+/-4Mbytes)

Data Processing Instruction

- Subset of the ARM data processing instructions
- Separate shift instructions (e.g. LSL, ASR, LSR, ROR)

```
LSL Rd,Rs,#Imm5     ;Rd:=Rs <shift> #Imm5
ASR Rd,Rs           ;Rd:=Rd <shift> Rs
```

- Two operands for data processing instructions
  - Act on low registers

```
BIC Rd,Rs           ;Rd:=Rd AND NOT Rs
ADD Rd,#Imm8        ;Rd:=Rd+#Imm8
```

  - Also three operand forms of add, subtract and shifts

```
ADD Rd,Rs,#Imm3     ;Rd:=Rs+#Imm3
```

- Condition code always set by low register operations

Load or Store Register

- Two pre-indexed addressing modes
  - Base register + offset register
  - Base register + 5-bit offset, where offset scaled by
    - 4 for word accesses (range of 0-124 bytes / 0-31 words)
      - `STR Rd,[Rd,#Imm7]`
    - 2 for halfword accesses (range of 0-62 bytes / 0-31 halfwords)
      - `LDRH Rd,[Rb,#Imm6]`
    - 1 for bytes accesses (range of 0-31 bytes)
      - `LDRB Rd,[Rb,#Imm5]`
- Special forms:
  - Load with PC as base with 1Kbyte immediate offset (word aligned)
    - Used for loading a value from a literal pool
  - Load and store with SP as base with 1Kbyte immediate offset (word aligned)
    - Used for accessing local variables on the stack

Block Data Transfers

- ❑ Memory copy, incrementing base pointer after transfer
  - `STMIA Rb!, {Low Reg list}`
  - `LDMIA Rb!, {Low Reg list}`
- ❑ Full descending stack operations
  - `PUSH {Low Reg list}`
  - `PUSH {Low Reg List, LR}`
  - `POP {Low Reg list}`
  - `POP {Low Reg List, PC}`
- ❑ The optional addition of the LR/PC provides support for subroutine entry/exit

Thumb Instruction Entry and Exit

- T bit, bit 5 of CPSR
  - If T = 1, the processor interprets the instruction stream as 16-bit Thumb instruction
  - If T = 0, the processor interprets if as standard ARM instructions
- Thumb Entry
  - ARM cores startup, after reset, execution ARM instructions
  - Executing a branch and Exchange instruction (BX)
    - Set the T bit if the bottom bit of the specified register was set
    - Switch the PC to the address given in the remainder of the register
- Thumb Exit
  - Executing a thumb BX instruction

The Need for Interworking

- The code density of Thumb and its performance from narrow memory make it ideal for the bulk of C code in many systems. However there is still a need to change between ARM and Thumb state within most applications:
  - ARM code provides better performance from wide memory
    - Therefore ideal for speed-critical parts of an application
  - Some functions can only be performed with ARM instructions, e.g.
    - Access to CPSR (to enable/disable interrupts & to change mode)
    - Access to coprocessors
  - Exception Handling
    - ARM state is automatically entered for exception handling, but system specification may require usage of Thumb code for main handler
  - Simple standalone Thumb programs will also need an ARM assembler header to change state and call the Thumb routine

Interworking Instructions

- Interworking is achieved using the Branch Exchange instructions
  - In Thumb state

```
BX Rn
```

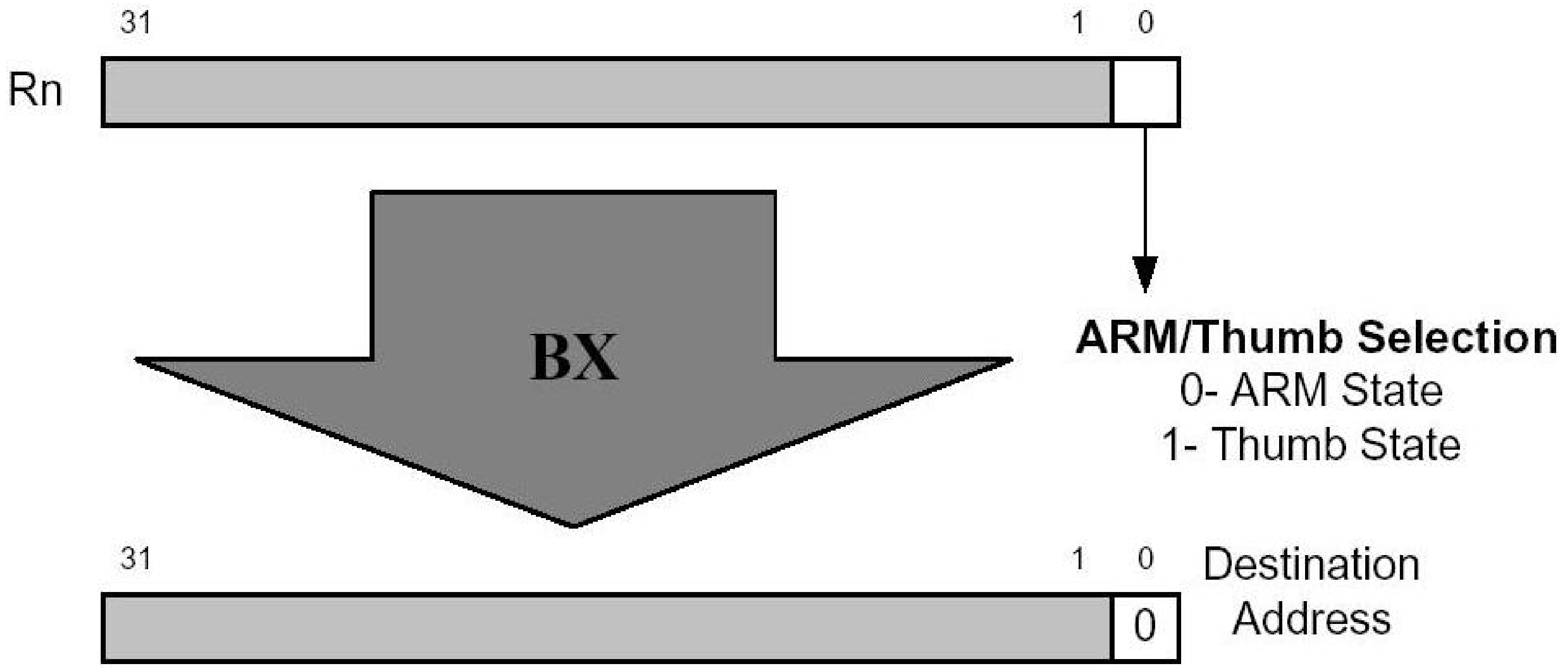
  - In ARM state (on Thumb-aware cores only)

```
BX<condition> Rn
```

  Where Rn can be any registers (R0 to R15)

- The performs a branch to an absolute address in 4GB address space by copying Rn to the program counter
- Bit 0 of Rn specifies the state to change to

Switching between States



Example

```
;start off in ARM state
     CODE32
     ADR r0,Into_Thumb+1 ;generate branch target
                         ;address & set bit 0
                         ;hence arrive Thumb state
     BX r0               ;branch exchange to Thumb
     …
     CODE16              ;assemble subsequent as Thumb
Into_Thumb  …
     ADR r5,Back_to_ARM  ;generate branch target to
                         ;word-aligned address,
                         ;hence bit 0 is cleared.
     BX r5               ;branch exchange to ARM
     …
     CODE32              ;assemble subsequent as ARM
Back_to_ARM     …
```

Summary

- ARM architecture
  - Load/Store architecture
  - 32-bit instructions
  - 3-address instruction formats
  - 37 registers
- Instruction set
  - 32-bit ARM instruction
  - 16-bit Thumb instruction
- ARM/Thumb Interworking

References

[1] http://twins.ee.nctu.edu.tw/courses/ip_core_02/index.html

[2] **ARM System-on-Chip Architecture** by S.Furber, Addison Wesley Longman: ISBN 0-201-67519-6.

[3] www.arm.com