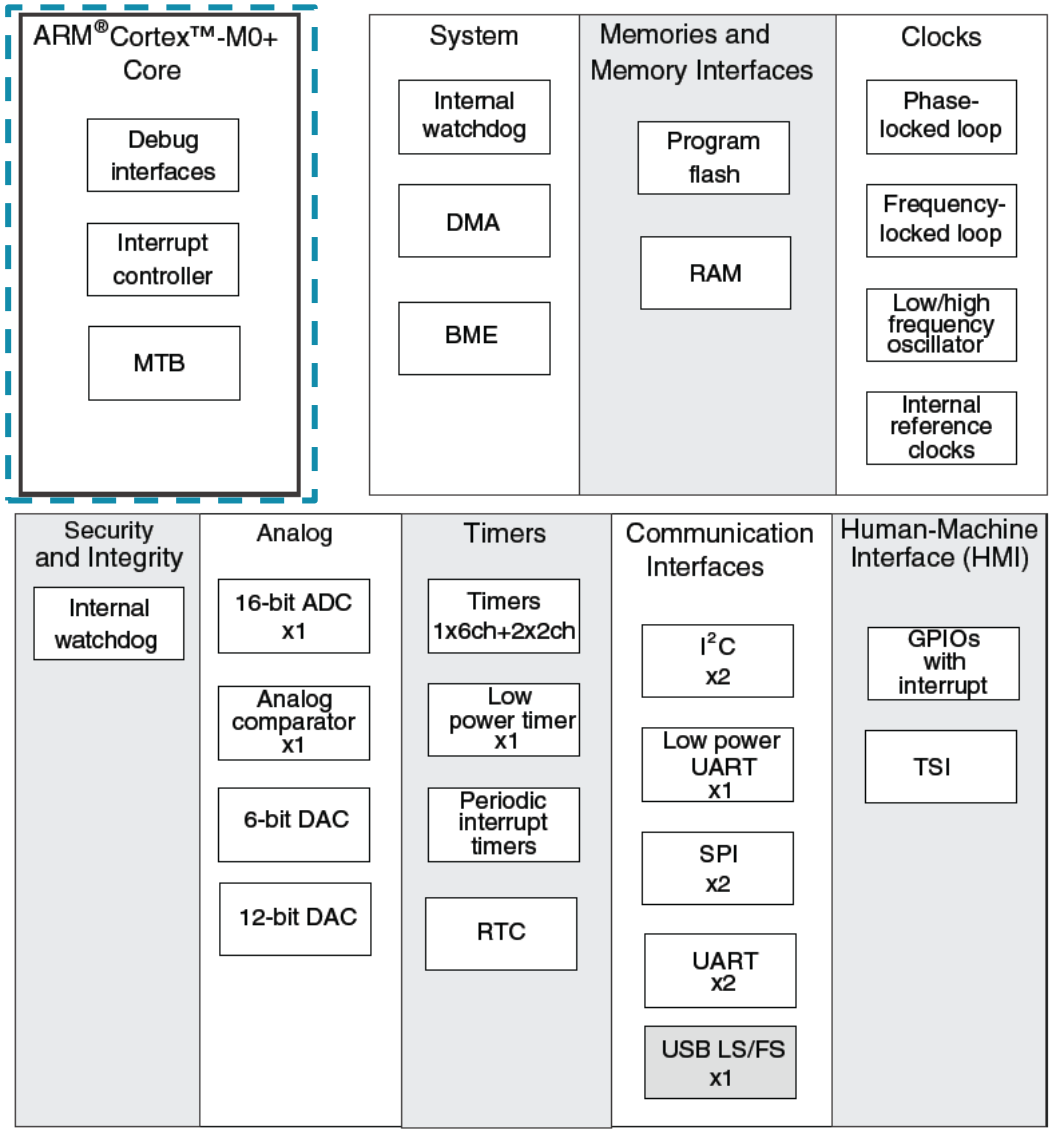
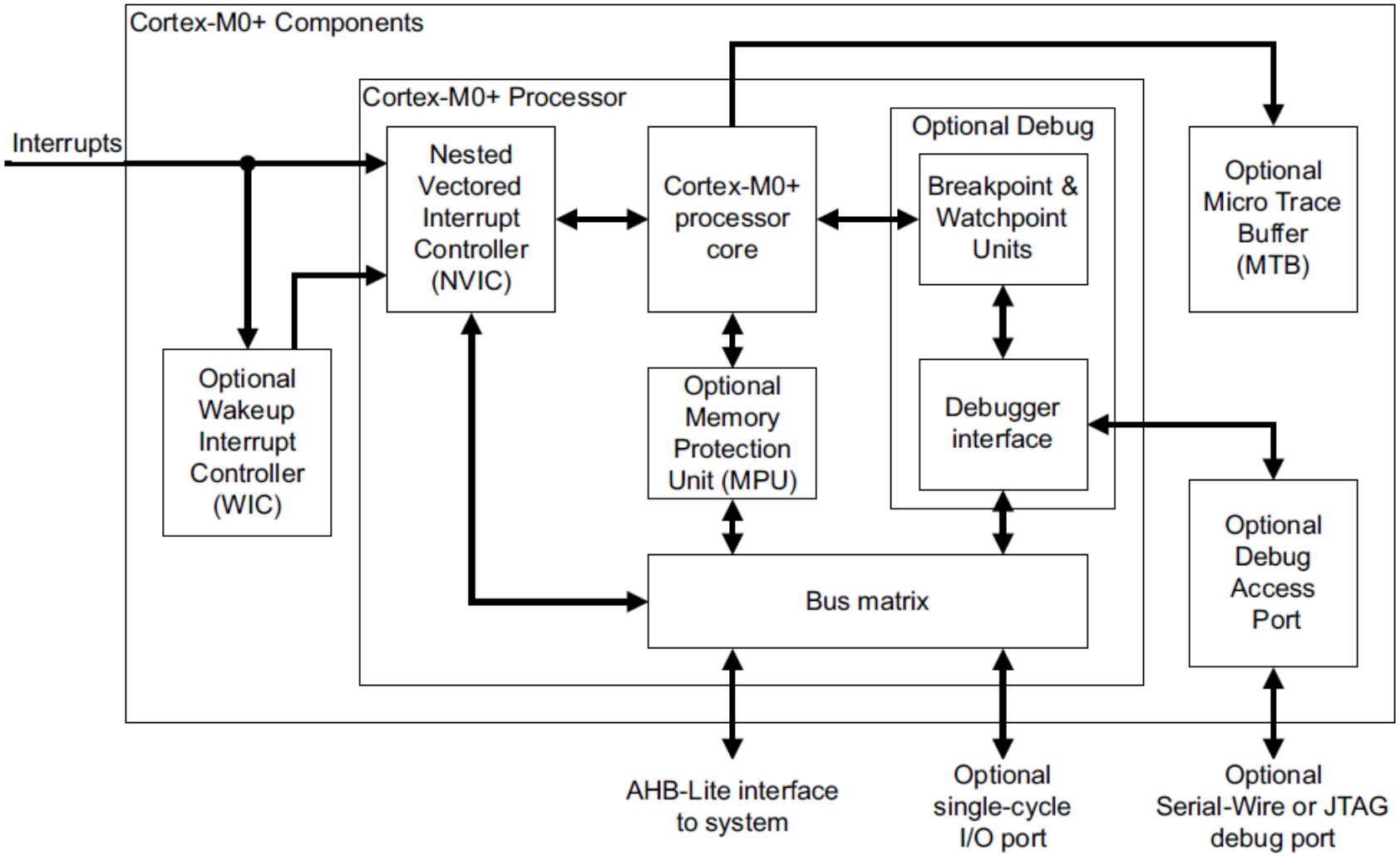
Cortex-M0+ CPU Core and ARM Instruction Set Architecture

Microcontroller vs. Microprocessor

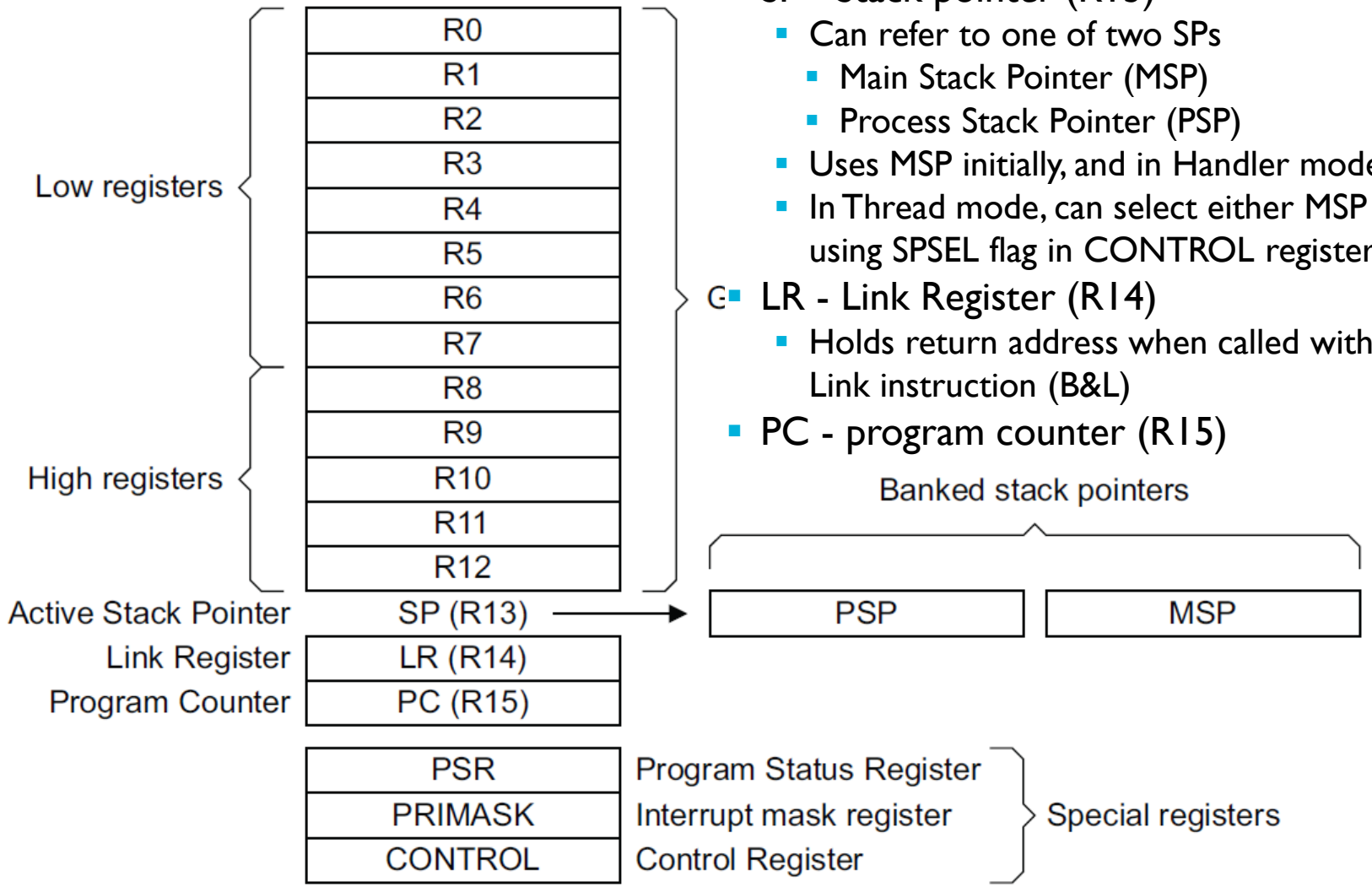
- Both have a CPU core to execute instructions
- Microcontroller has peripherals for embedded interfacing and control
  - Analog
  - Non-logic level signals
  - Timing
  - Clock generators
  - Communications
    - point to point
    - network
  - Reliability and safety



Cortex-M0+ Core

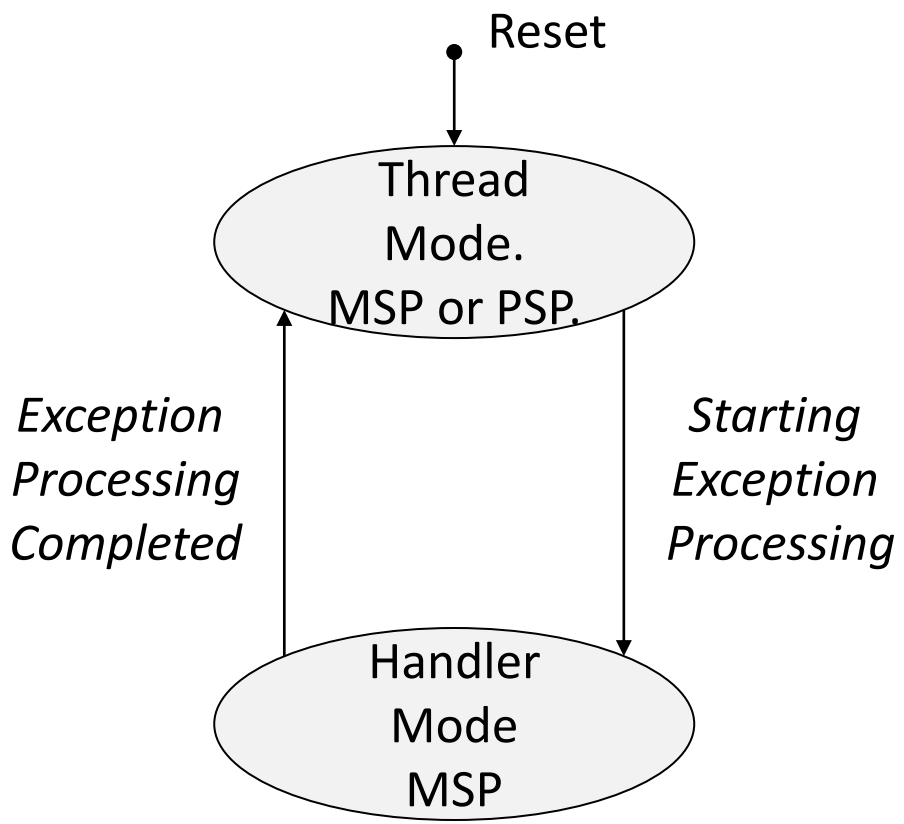


ARM Processor Core Registers



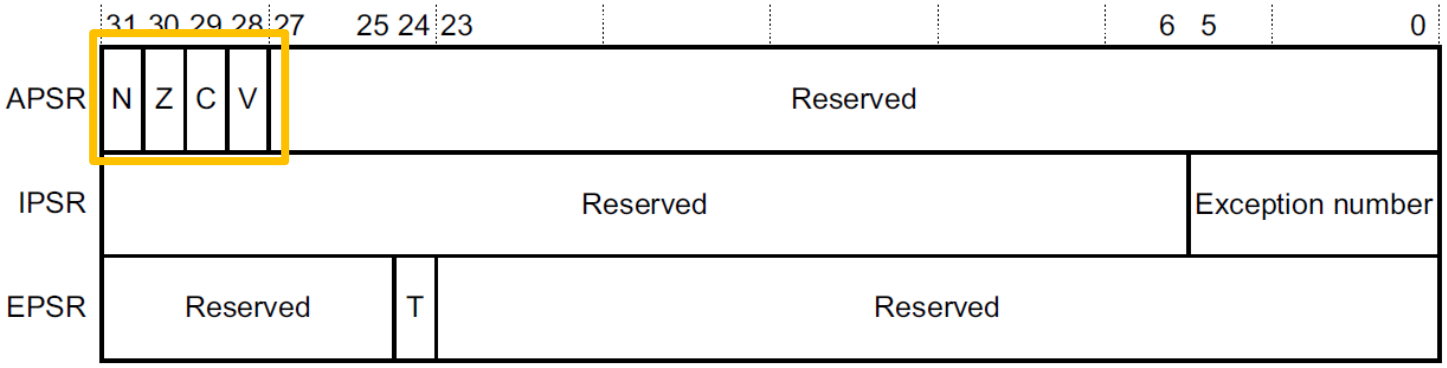
- R0-R12 - General purpose, for data processing
- SP - Stack pointer (R13)
  - Can refer to one of two SPs
    - Main Stack Pointer (MSP)
    - Process Stack Pointer (PSP)
  - Uses MSP initially, and in Handler mode
  - In Thread mode, can select either MSP or PSP using SPSEL flag in CONTROL register.
- LR - Link Register (R14)
  - Holds return address when called with Branch & Link instruction (B&L)
- PC - program counter (R15)

Operating Modes



- Which SP is active depends on operating mode, and SPSEL (CONTROL register bit 1)
  - SPSEL == 0: MSP
  - SPSEL == 1: PSP

ARM Program Status Register



- Three views of same register
  - Application PSR (APSR)
    - Condition code flag bits Negative, Zero, oVerflow, Carry used for conditional branches, extended precision math, error detection
  - Interrupt PSR (IPSR)
    - Holds exception number of currently executing ISR
  - Execution PSR (EPSR)
    - Thumb state

Mnemonic extension	Meaning	Condition flags
EQ	Equal	Z == 1
NE	Not equal	Z == 0
CS [a]	Carry set	C == 1
CC [b]	Carry clear	C == 0
MI	Minus, negative	N == 1
PL	Plus, positive or zero	N == 0
VS	Overflow	V == 1
VC	No overflow	V == 0
HI	Unsigned higher	C == 1 and Z == 0
LS	Unsigned lower or same	C == 0 or Z == 1
GE	Signed greater than or equal	N == V
LT	Signed less than	N != V
GT	Signed greater than	Z == 0 and N == V
LE	Signed less than or equal	Z == 1 or N != V
None (AL) [d]	Always (unconditional)	Any

ARM Processor Core Registers

- PRIMASK - Exception mask register
  - Bit 0: PM Flag
    - Set to 1 to prevent activation of all exceptions with configurable priority
  - Access using CPS, MSR and MRS instructions
  - Use to prevent data race conditions with code needing atomicity

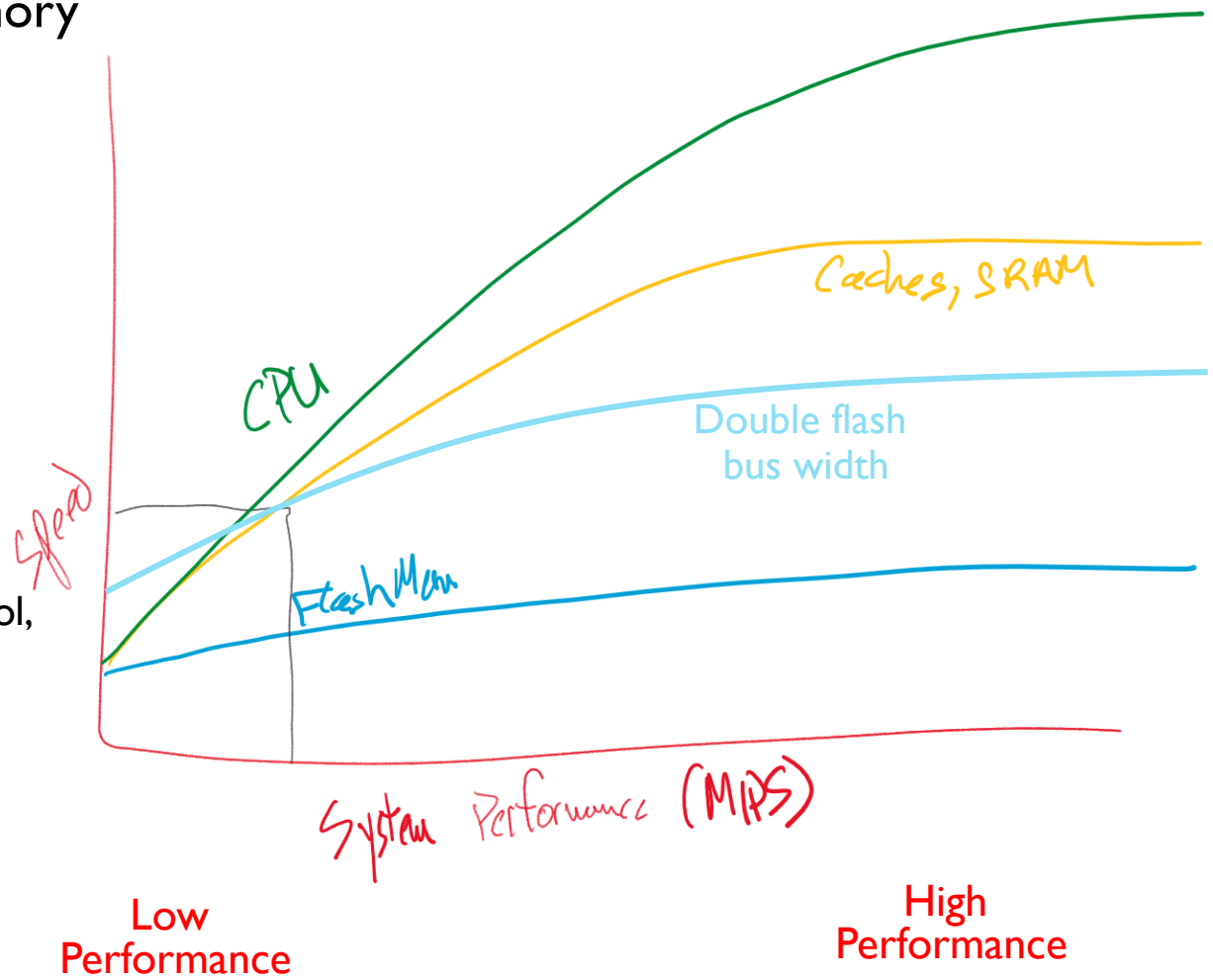
- CONTROL
  - Bit 1: SPSEL flag
    - Selects SP when in thread mode: MSP (0) or PSP (1)
  - Bit 0: nPRIV flag
    - Defines whether thread mode is privileged (0) or unprivileged (1)
  - With OS environment,
    - Threads use PSP
    - OS and exception handlers (ISRs) use MSP

Different Instruction Sets for Different Design Spaces?

- ARM instructions optimized for resource-rich high-performance computing systems
  - Deeply pipelined processor, high clock rate, wide (e.g. 32-bit) memory bus
  - https://en.wikipedia.org/wiki/ARM_Cortex-M#Instruction_sets
- Low-end embedded computing systems are different
  - Slower clock rates, shallow pipelines
  - Different cost factors – e.g. code size matters much more
  - Bit and byte operations critical

The Memory Wall

- It has been easier to speed up the CPU than the memory
- Facts of life
  - Off-chip memory is slower than on-chip memory. May not want to put all memory on-chip, even if possible.
  - Flash is slower to read or write than RAM.
  - Fast RAM is more expensive than slow RAM. Same for flash.
- Design for high-performance CPUs
  - Use caches (small fast RAM) to make main memory (large slow RAM, flash) look faster at a low cost.
  - Put cache(s) on chip if possible.
  - Increase bandwidth by widening memory bus, improving protocol, reducing overhead, split transactions, using page mode, etc.)
- Design for low-performance CPUs
  - Put memory on-chip with CPU. RAM, flash ROM
  - Increase flash ROM bandwidth by widening memory bus, adding prefetch buffer, branch target buffer, etc.
  - Add cache
  - *Change instruction set size to reduce instruction bandwidth needed*



ARM and Thumb Instructions

- Thumb reduces program memory size and bandwidth requirements
  - Subset of instructions re-encoded into fewer bits (most 16 bits, some 32 bits)
    - Not all 32-bit instructions available
    - Most 16-bit instructions can only access low registers (R0-R7), but a few can access high registers (R8-R15)
  - 1995: Thumb-1 instruction set
    - 16-bit instructions
  - 2003: Thumb-2 instruction set
    - Adds some 32 bit instructions
    - Improves speed with little memory overhead
- Arm Architecture supports different *instruction set states*, which define:
  - How memory contents are decoded into instructions
  - Which instructions are available
- Instruction set states
  - ARM state: full 32-bit ARM instruction set
  - Thumb state: Thumb instruction set
- CPUs and states
  - Cortex-M CPUs support only Thumb instruction set, are always in Thumb state
  - Cortex-A CPUs support both instruction sets and states, can switch between them
- State selection – "interworking"
  - Some instructions (BX, BLX, POP {PC}) also can *exchange* instruction set
  - Last bit of target program counter indicates desired state (Thumb = 1, ARM = 0)
- See ARMv6-M Architecture Reference Manual for more (Section A1.1.1, A4.1, A4.1.1, A6.7)

Cortex-M Instruction Groups

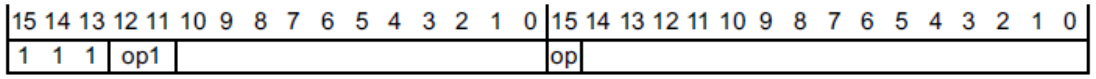
Group	Instr bits	Instructions	M0,M0+,M1	M3	M4	M7	M23	M33,M35P
Thumb-1	16	ADC, ADD, ADR, AND, ASR, B, BIC, BKPT, BLX, BX, CMN, CMP, CPS, EOR, LDM, LDR, LDRB, LDRH, LDRSB, LDRSH, LSL, LSR, MOV, MUL, MVN, NOP, ORR, POP, PUSH, REV, REV16, REVSH, ROR, RSB, SBC, SEV, STM, STR, STRB, STRH, SUB, SVC, SXTB, SXTH, TST, UXTB, UXTH, WFE, WFI, YIELD	Yes	Yes	Yes	Yes	Yes	Yes
Thumb-1	16	CBNZ, CBZ	No	Yes	Yes	Yes	Yes	Yes
Thumb-1	16	IT	No	Yes	Yes	Yes	No	Yes
Thumb-2	32	BL, DMB, DSB, ISB, MRS, MSR	Yes	Yes	Yes	Yes	Yes	Yes
Thumb-2	32	SDIV, UDIV	No	Yes	Yes	Yes	Yes	Yes
Thumb-2	32	ADC, ADD, ADR, AND, ASR, B, BFC, BFI, BIC, CDP, CLREX, CLZ, CMN, CMP, DBG, EOR, LDC, LDM, LDR, LDRB, LDRBT, LDRD, LDREX, LDREXB, LDREXH, LDRH, LDRHT, LDRSB, LDRSBT, LDRSH, LDRSHT, LDRT, LSL, LSR, MCR, MCRR, MLA, MLS, MOV, MOVT, MRC, MRRC, MUL, MVN, NOP, ORN, ORR, PLD, PLDW, PLI, POP, PUSH, RBIT, REV, REV16, REVSH, ROR, RRX, RSB, SBC, SBFX, SEV, SMLAL, SMULL, SSAT, STC, STM, STR, STRB, STRBT, STRD, STREX, STREXB, STREXH, STRH, STRHT, STRT, SUB, SXTB, SXTH, TBB, TBH, TEQ, TST, UBFX, UMLAL, UMULL, USAT, UXTB, UXTH, WFE, WFI, YIELD	No	Yes	Yes	Yes	No	Yes
DSP	32	PKH, QADD, QADD16, QADD8, QASX, QDADD, QDSUB, QSAX, QSUB, QSUB16, QSUB8, SADD16, SADD8, SASX, SEL, SHADD16, SHADD8, SHASX, SHSAX, SHSUB16, SHSUB8, SMLABB, SMLABT, SMLATB, SMLATT, SMLAD, SMLALBB, SMLALBT, SMLALTB, SMLALTT, SMLALD, SMLAWB, SMLAWT, SMLSD, SMLSLD, SMMLA, SMMLS, SMMUL, SMUAD, SMULBB, SMULBT, SMULTT, SMULTB, SMULWT, SMULWB, SMUSD, SSAT16, SSAX, SSUB16, SSUB8, SXTAB, SXTAB16, SXTAH, SXTB16, UADD16, UADD8, UASX, UHADD16, UHADD8, UHASX, UHSAX, UHSUB16, UHSUB8, UMAAL, UQADD16, UQADD8, UQASX, UQSAX, UQSUB16, UQSUB8, USAD8, USADA8, USAT16, USAX, USUB16, USUB8, UXTAB, UXTAB16, UXTAH, UXTB16	No	No	Yes	Yes	No	Optional
SP Float	32	VABS, VADD, VCMP, VCMPE, VCVT, VCVTR, VDIV, VLDM, VLDR, VMLA, VMLS, VMOV, VMRS, VMSR, VMUL, VNEG, VNMLA, VNMLS, VNMUL, VPOP, VPUSH, VSQRT, VSTM, VSTR, VSUB	No	No	Optional	Optional	No	Optional
DP Float	32	VCVTA, VCVTM, VCVTN, VCVTP, VMAXNM, VMINNM, VRINTA, VRINTM, VRINTN, VRINTP, VRINTR, VRINTX, VRINTZ, VSEL	No	No	No	Optional	No	No
TrustZone	16	BLXNS, BXNS	No	No	No	No	Optional	Optional
TrustZone	32	SG, TT, TTT, TTA, TTAT	No	No	No	No	Optional	Optional
Co-processor	16	CDP, CDP2, MCR, MCR2, MCRR, MCRR2, MRC, MRC2, MRRC, MRRC2	No	No	No	No	No	Optional

https://en.wikipedia.org/wiki/ARM_Cortex-M#Instruction_sets

Reference for ARM Instruction Set Architecture

- **ARM V6-M Architecture Reference Manual,** Chapter A5. The Thumb Instruction Set Encoding
- 16- or 32-bit instruction?
  - Bits [15:11]
    - 0b11101, 0b11110, 0b11111: 32-bit instruction. Page A5-91
    - Else 16-bit instruction. Page A5-84

The encoding of 32-bit Thumb instructions is:



For 32-bit Thumb encoding, op1 != 0b00. If op1 == 0b00, a 16-bit instruction is encoded, see *16-bit Thumb instruction encoding* on page A5-84.

Table A5-9 shows the allocation of ARMv6-M Thumb encodings in this space.

Table A5-9 32-bit Thumb encoding

op1	op	Instruction class
x1	x	UNDEFINED
10	1	See *Branch and miscellaneous control*
10	0	UNDEFINED

The encoding of 16-bit Thumb instructions is:

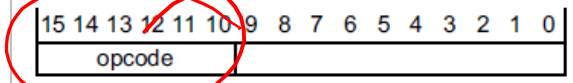


Table A5-1 shows the allocation of 16-bit instruction encodings.

Table A5-1 16-bit Thumb instruction encoding

opcode	Instruction or instruction class
00xxxx	*Shift (immediate), add, subtract, move, and compare* on page A5-85
010000	*Data processing* on page A5-86
010001	*Special data instructions and branch and exchange* on page A5-87
01001x	Load from Literal Pool, see *LDR (literal)* on page A6-141
0101xx 011xxx 100xxx	*Load/store single data item* on page A5-88
10100x	Generate PC-relative address, see *ADR* on page A6-115
10101x	Generate SP-relative address, see *ADD (SP plus immediate)* on page A6-111
1011xx	*Miscellaneous 16-bit instructions* on page A5-89
11000x	Store multiple registers, see *STM, STMIA, STMEA* on page A6-175
11001x	Load multiple registers, see *LDM, LDMIA, LDMFD* on page A6-137
1101xx	*Conditional branch, and Supervisor Call* on page A5-90
11100x	Unconditional Branch, see *B* on page A6-119

Example Instruction Encoding: ADC (register)

ADC (register)

Add with Carry (register) adds a register value, the carry flag value, and an optionally-shifted register value, and writes the result to the destination register. It updates the condition flags based on the result.

Encoding T1 All versions of the Thumb instruction set.

ADCS <Rdn>,<Rm>

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	0	1	0	1	Rm			Rdn		

```
d = UInt(Rdn);  n = UInt(Rdn);  m = UInt(Rm);  setflags = !InITBlock();
(shift_t, shift_n) = (SRType_LSL, 0);
```

Assembler syntax

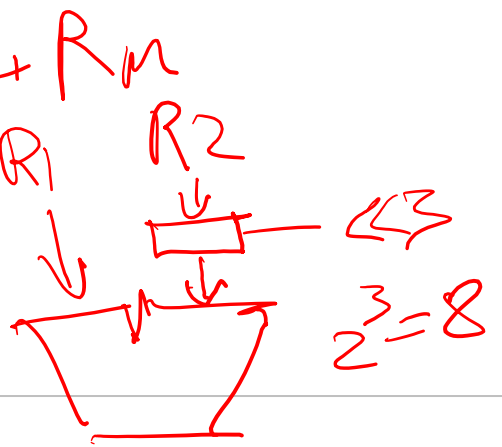
ADCS{<q>} {<Rd>,} <Rn>, <Rm>

where:

S	The instruction updates the flags.
{<q>}	See *Standard assembler syntax fields* on page A6-98.
<Rd>	The destination register. If <Rd> is omitted, this register is the same as <Rn>.
<Rn>	The register that contains the first operand.
<Rm>	The register that is optionally shifted and used as the second operand.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    shifted = Shift(R[m], shift_t, shift_n, APSR.C);
    (result, carry, overflow) = AddWithCarry(R[n], shifted, APSR.C);
    R[d] = result;
    if setflags then
        APSR.N = result<31>;
        APSR.Z = IsZeroBit(result);
        APSR.C = carry;
        APSR.V = overflow;
```



- Page A6-106 of **ARM-V6M ARM**

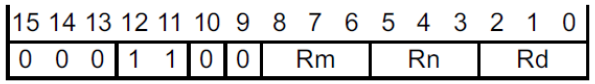
Example Instruction Encoding: ADD (register)

ADD (register)

This instruction adds a register value and an optionally-shifted register value, and writes the result to the destination register. Encoding T1 updates the condition flags based on the result.

Encoding T1 All versions of the Thumb instruction set.

```
ADDS <Rd>,<Rn>,<Rm>
```

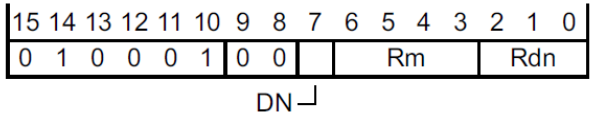


Rd ← Rn + Rm

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = !InITBlock();
(shift_t, shift_n) = (SRType_LSL, 0);
```

Encoding T2 All versions of the Thumb instruction set.

```
ADD <Rdn>,<Rm>
```




```
if (DN:Rdn) == '1101' || Rm == '1101' then SEE ADD (SP plus register);
d = UInt(DN:Rdn);  n = d;  m = UInt(Rm);  setflags = FALSE;  (shift_t, shift_n) = (SRType_LSL, 0);
if n == 15 && m == 15 then UNPREDICTABLE;
if d == 15 && InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```

Assembler syntax

```
ADD{S}{<q>}  {<Rd>,} <Rn>, <Rm>
```

where:

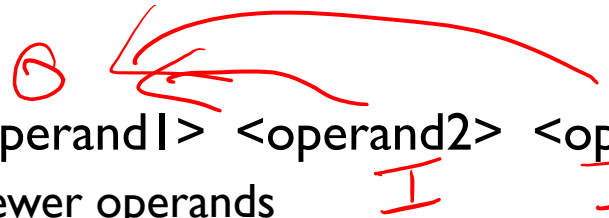
S	If present, specifies that the instruction updates the flags. Otherwise, the instruction does not update the flags.
{<q>}	See *Standard assembler syntax fields* on page A6-98.
<Rd>	The destination register. If <Rd> is omitted, this register is the same as <Rn> and encoding T2 is preferred to encoding T1 if both are available. If <Rd> is specified, encoding T1 is preferred to encoding T2. If R<m> is not the PC, the PC can be used in encoding T2.
<Rn>	The register that contains the first operand. If the SP is specified for <Rn>, see *ADD (SP plus register)* on page A6-113. If R<m> is not the PC, the PC can be used in encoding T2.
<Rm>	The register that is used as the second operand. The PC can be used in encoding T2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    shifted = Shift(R[m], shift_t, shift_n, APSR.C);
    (result, carry, overflow) = AddWithCarry(R[n], shifted, '0');
    if d == 15 then
        ALUWritePC(result);  // setflags is always FALSE here
    else
        R[d] = result;
        if setflags then
            APSR.N = result<31>;
            APSR.Z = IsZeroBit(result);
            APSR.C = carry;
            APSR.V = overflow;
```

- Page A6-109 of ARM-V6M ARM

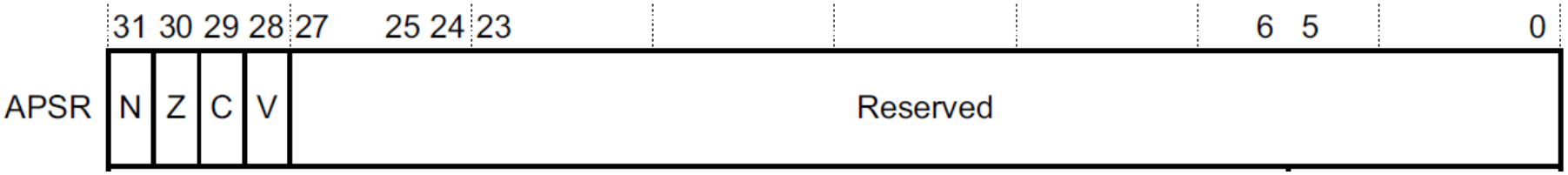
Assembler Instruction Format



- <operation> <operand1> <operand2> <operand3>
  - There may be fewer operands
  - First operand is typically destination (<Rd>)
  - Other operands are sources (<Rn>, <Rm>)

- Examples
  - ADDS <Rd>, <Rn>, <Rm>
    - Add registers: <Rd> = <Rn> + <Rm>
  - AND <Rdn>, <Rm>
    - Bitwise and: <Rdn> = <Rdn> & <Rm>
  - CMP <Rn>, <Rm>
    - Compare: Set condition flags based on result of computing <Rn> - <Rm>

Update Condition Codes in APSR?



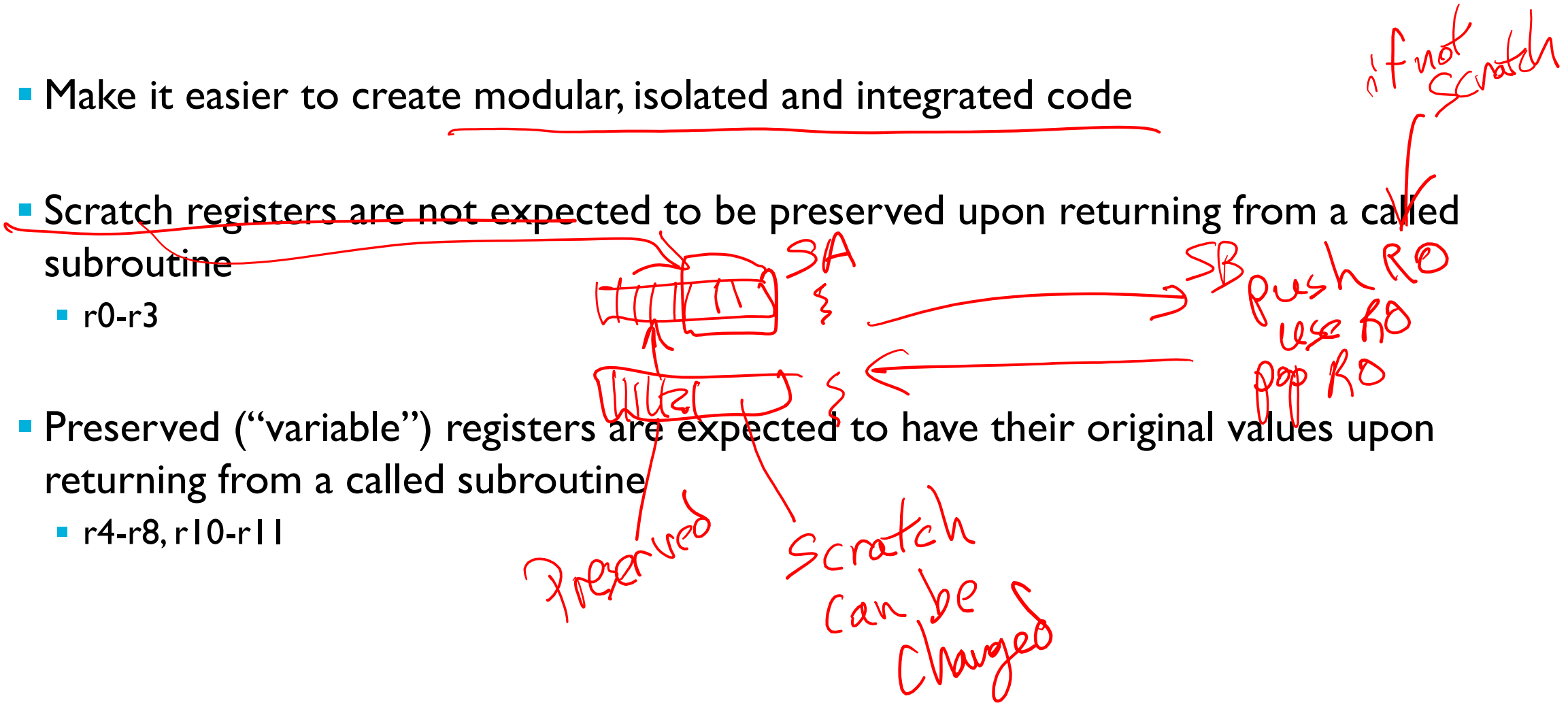
- "S" suffix indicates the instruction updates APSR
  - ADD vs. ADDS
  - ADC vs. ADCS
  - SUB vs. SUBS
  - MOV vs. MOVS

RISC
Relegate Important Stuff to Compiler

USING REGISTERS

AAPCS Register Use Conventions

- Make it easier to create modular, isolated and integrated code
- Scratch registers are not expected to be preserved upon returning from a called subroutine
  - r0-r3
- Preserved ("variable") registers are expected to have their original values upon returning from a called subroutine
  - r4-r8, r10-r11



AAPCS Core Register Use

Register	Synonym	Special	Role in the procedure call standard
r15		PC	The Program Counter.
r14		LR	The Link Register.
r13		SP	The Stack Pointer.
r12		IP	The Intra-Procedure-call scratch register.
r11	v8		Variable-register 8.
r10	v7		Variable-register 7.
r9		v6 SB TR	Platform register. The meaning of this register is defined by the platform standard.
r8	v5		Variable-register 5.
r7	v4		Variable register 4.
r6	v3		Variable register 3.
r5	v2		Variable register 2.
r4	v1		Variable register 1.
r3	a4		Argument / scratch register 4.
r2	a3		Argument / scratch register 3.
r1	a2		Argument / result / scratch register 2.
r0	a1		Argument / result / scratch register 1.

Must be saved, restored by callee-procedure if it will modify them. Calling subroutine expects these to retain their value.

Must be saved, restored by callee-procedure if it will modify them. Calling subroutine expects these to retain their value.

Don't need to be saved. May be used for arguments, results, or temporary values.

INSTRUCTION SUMMARY

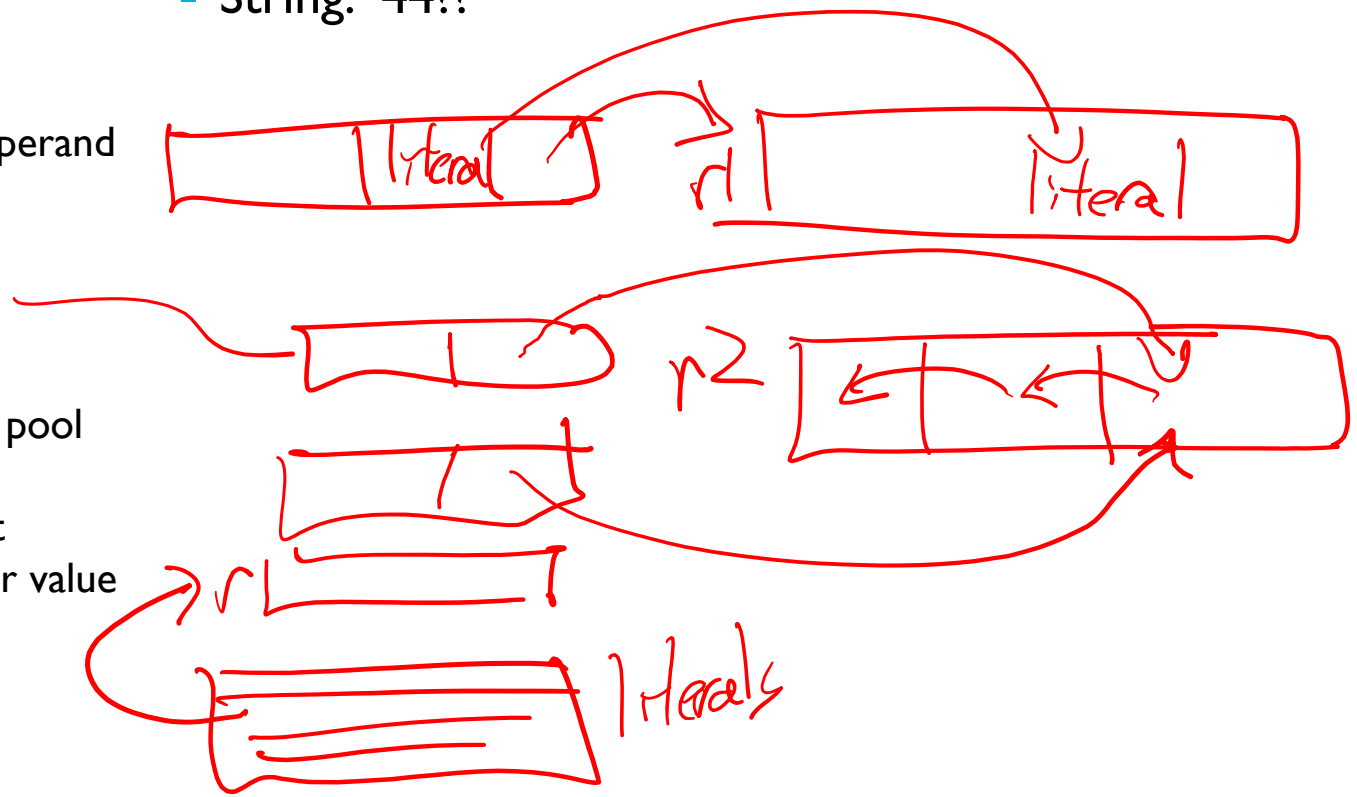
Instruction Set Summary

Instruction Type	Instructions
Move	MOV
Load/Store	LDR, LDRB, LDRH, LDRSH, LDRSB, LDM, STR, STRB, STRH, STM
Add, Subtract, Multiply	ADD, ADDS, ADCS, ADR, SUB, SUBS, SBCS, RSBS, MULS
Compare	CMP, CMN
Logical	ANDS, EORS, ORRS, BICS, MVNS, TST
Shift and Rotate	LSLS, LSRS, ASRS, RORS
Stack	PUSH, POP
Branch	B, BL, B{cond}, BX, BLX
Extend	SXTH, SXTB, UXTH, UXTB
Reverse	REV, REV16, REVSH
Processor State	SVC, CPSID, CPSIE, SETEND, BKPT
No Operation	NOP
Hint	SEV, WFE, WFI, YIELD
Barriers	DMB, DSB, ISB

PSEUDO-INSTRUCTIONS

Load Literal Value into Register

- Assembly pseudo-instruction: LDR <rd>, =value
  - Assembler generates code to load <rd> with value
- Assembler selects best approach depending on value
  - Load immediate
    - MOV instruction provides 8-bit unsigned immediate operand (0-255)
  - Load and shift immediate values
    - Can use MOV, shift, rotate, sign extend instructions
  - Load from literal pool
    - 1. Place value as a 32-bit literal in the program's literal pool (table of literal values to be loaded into registers)
    - 2. Use instruction LDR <rd>, [pc,#offset] where offset indicates position of literal relative to program counter value
- Example formats for literal values (depends on compiler and toolchain used)
  - Decimal: 3909
  - Hexadecimal: 0xa7ee
  - Character: 'A'
  - String: "44??"



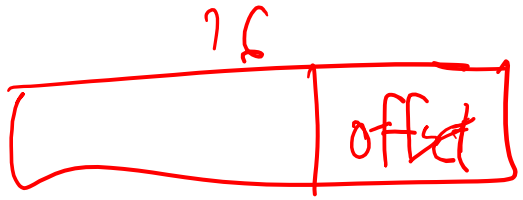
Move (Pseudo-)Instructions

- Copy data from one register to another without updating condition flags
  - MOV <Rd>, <Rm>
- Assembler translates pseudo-instructions into equivalent instructions (shifts, rotates)
  - Copy data from one register to another and update condition flags
    - MOVS <Rd>, <Rm>
  - Copy immediate literal value (0-255) into register and update condition flags
    - MOVS <Rd>, #<imm8>

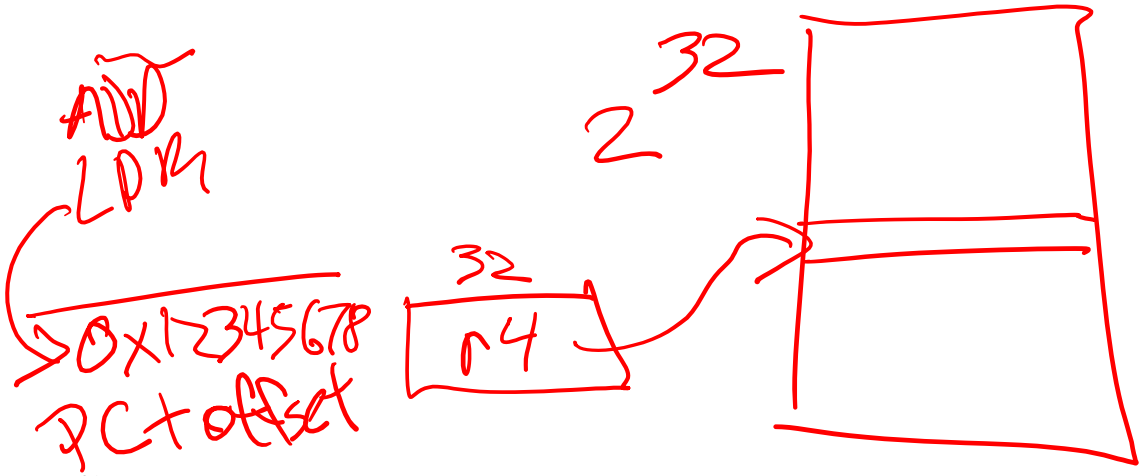
MOV instruction	Canonical form
MOVS <Rd>,<Rm>,ASR #<n>	ASRS <Rd>,<Rm>,#<n>
MOVS <Rd>,<Rm>,LSL #<n>	LSLS <Rd>,<Rm>,#<n>
MOVS <Rd>,<Rm>,LSR #<n>	LSRS <Rd>,<Rm>,#<n>
MOVS <Rd>,<Rm>,ASR <Rs>	ASRS <Rd>,<Rm>,<Rs>
MOVS <Rd>,<Rm>,LSL <Rs>	LSLS <Rd>,<Rm>,<Rs>
MOVS <Rd>,<Rm>,LSR <Rs>	LSRS <Rd>,<Rm>,<Rs>
MOVS <Rd>,<Rm>,ROR <Rs>	RORS <Rd>,<Rm>,<Rs>

INSTRUCTIONS FOR MEMORY

Load and Store Register Instructions



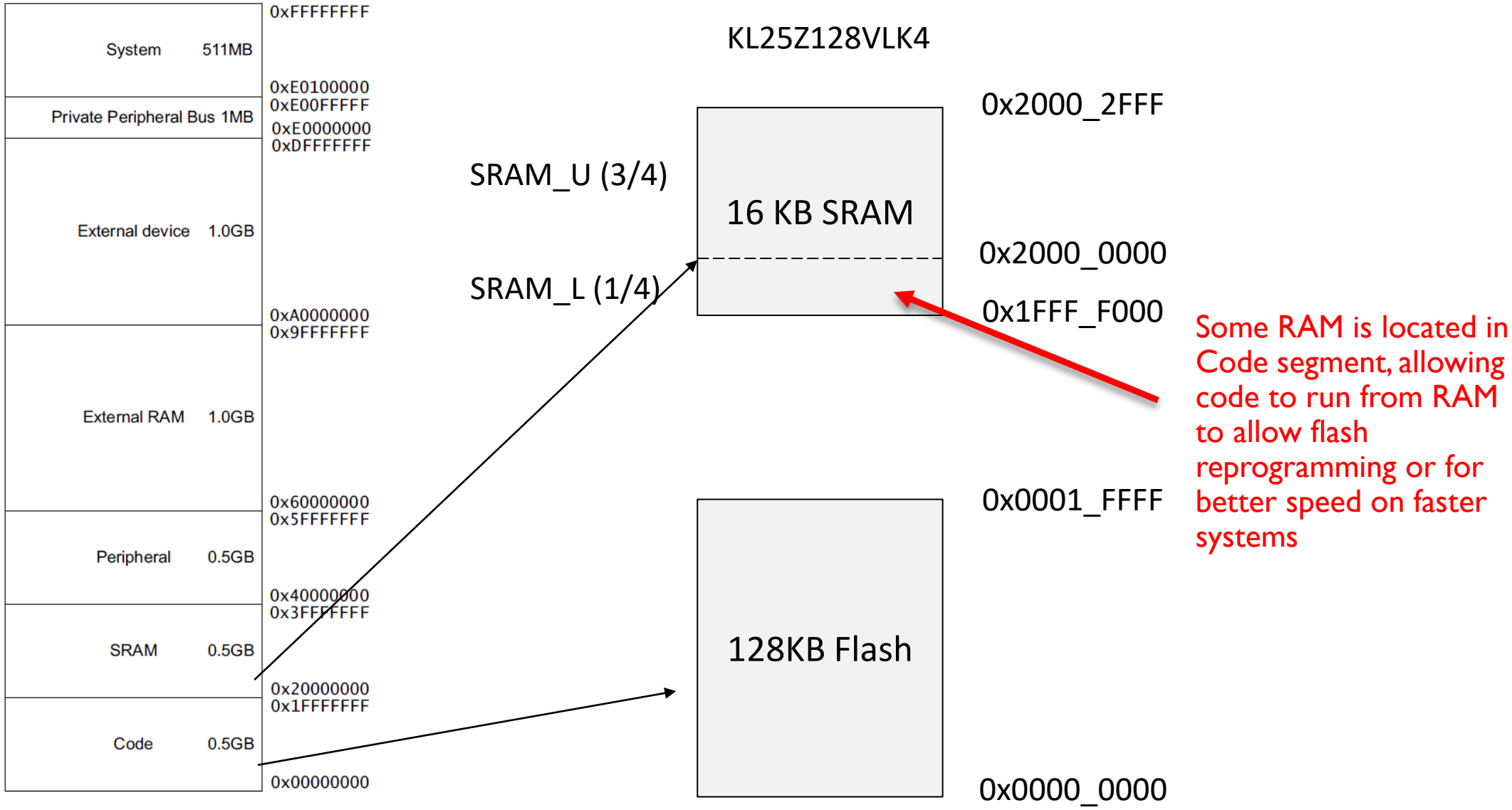
- ARM is a load/store architecture, so must process data in registers (not memory)
- LDR: load register with word (32 bits) from memory
  - LDR <Rt>, source address
- STR: store register contents (32 bits) to memory
  - STR <Rt>, destination address



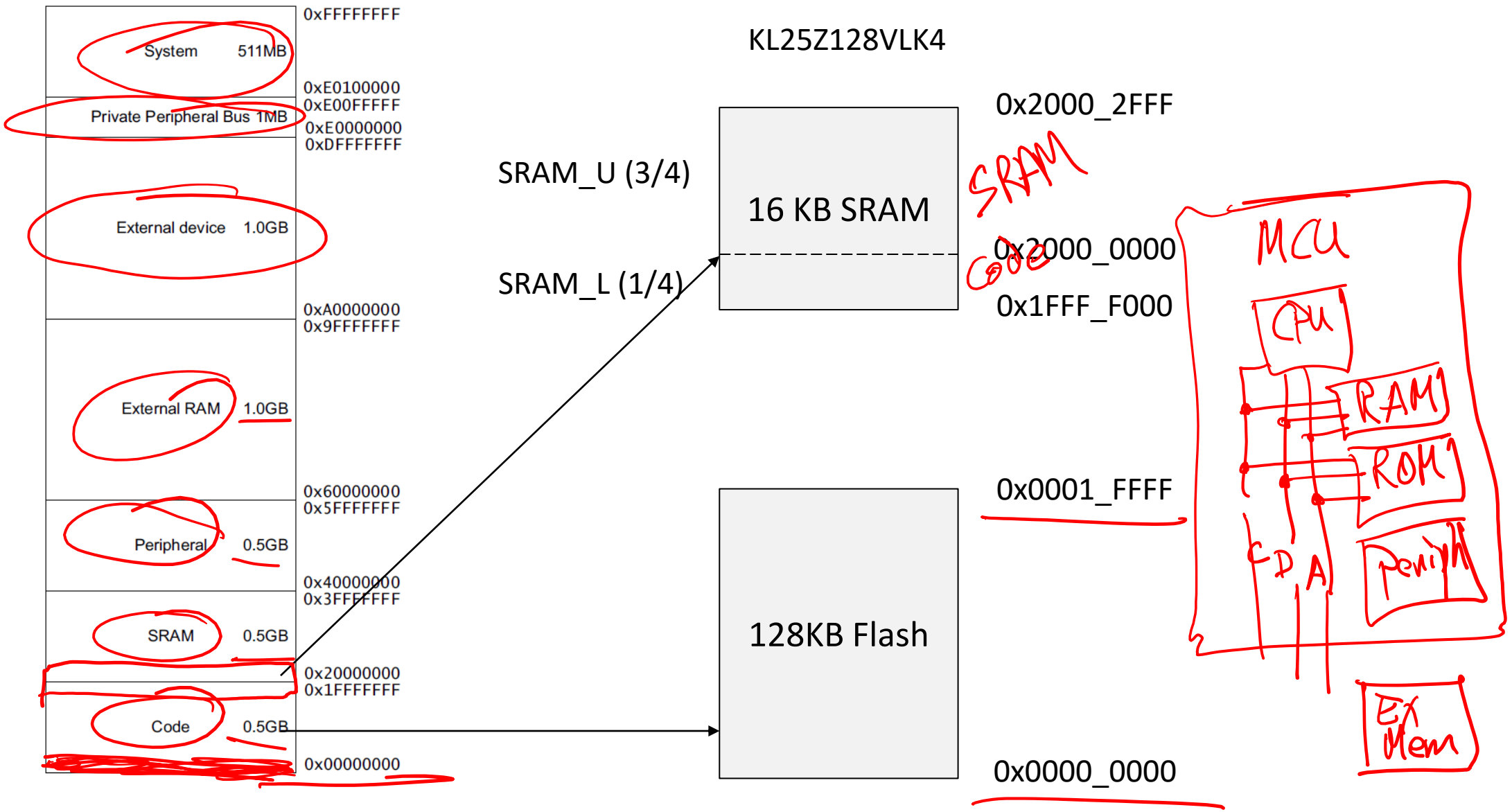
- Source and destination addresses are specified using available addressing modes
  - Offset Addressing mode: [<Rn>, <offset>] accesses address <Rn>+<offset>
  - Base Register <Rn> can be R0-R7, SP or PC
  - <offset> is added or subtracted from base register to create effective address
    - Can be an immediate constant
    - Can be another register, used as index <Rm>

- Auto-update: Can write effective address back to base register
  - Pre-indexing: use **effective address** to access memory, then update base register
  - Post-indexing: use **base register** to access memory, then update base register

Memory Maps For Cortex M0+ and MCU

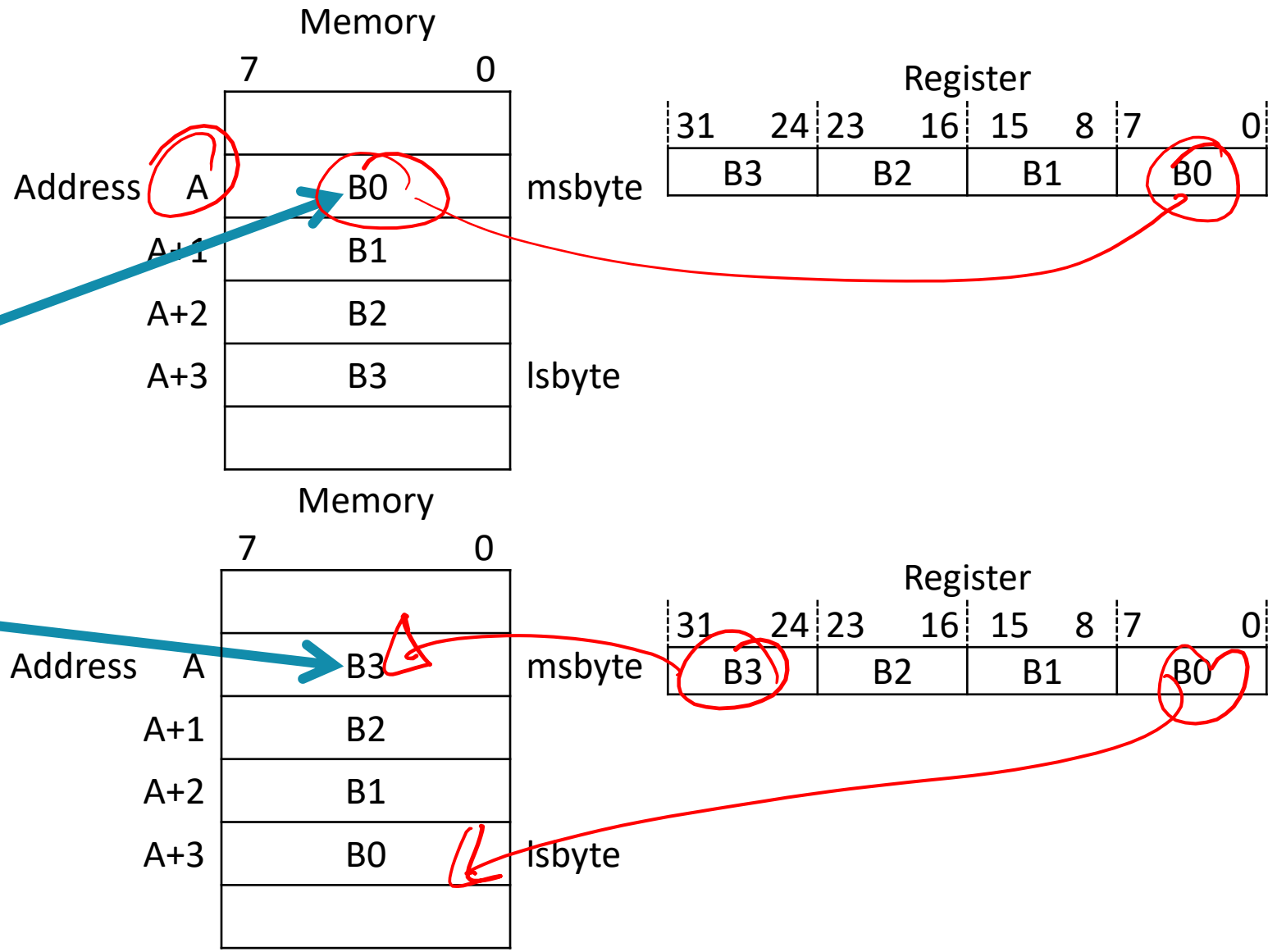


Memory Maps For Cortex M0+ and MCU



Endianness

- For a multi-byte value, in what order are the bytes stored?
- Little-Endian: Start with least-significant byte
- Big-Endian: Start with most-significant byte



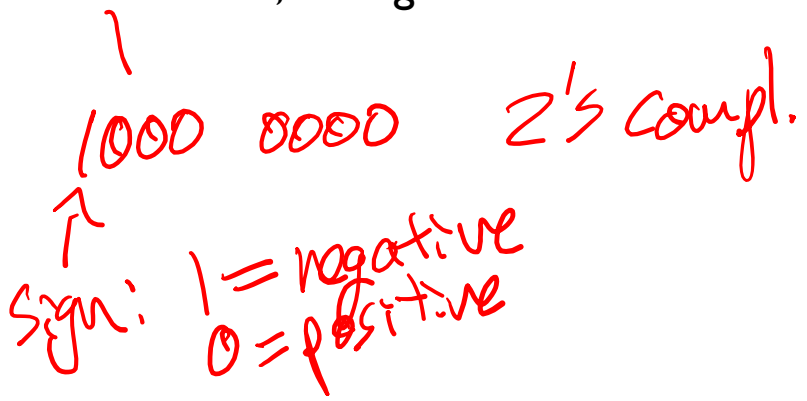
ARMv6-M Endianness

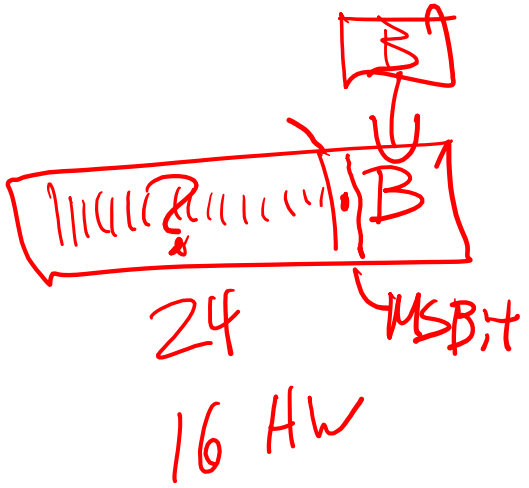
- Instructions are always little-endian
- Loads and stores to Private Peripheral Bus are always little-endian
- Data: Depends on implementation, or from reset configuration
  - Kinetis processors are little-endian

Loading/Storing Smaller Data Sizes

	Signed	Unsigned
Byte	LDRSB	LDRB
Half-word	LDRSH	LDRH

- Some load and store instructions can handle half-word (16 bits) and byte (8 bits)
- Store just writes to half-word or byte
  - STRH, STRB
- Loading a byte or half-word requires padding or extension: What do we put in the upper bits of the register?
  - Example: How do we extend 0x80 into a full word?
    - Unsigned? Then 0x80 = 128, so zero-pad to extend to word 0x0000_0080 = 128
    - Signed? Then 0x80 = -128, so sign-extend to word 0xFFFF_FF80 = -128





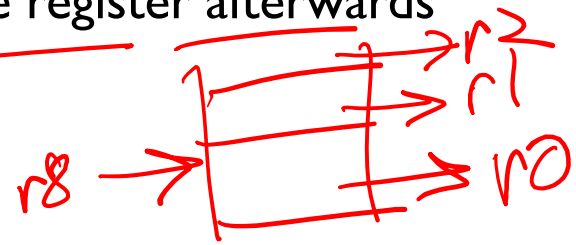
In-Register Size Extension

	Signed	Unsigned
Byte	SXTB	UXTB
Half-word	SXTH	UXTH

- Can also extend byte or half-word already in a register
  - Signed or unsigned (zero-pad)
- How do we extend 0x80 into a full word?
  - Unsigned? Then 0x80 = 128, so zero-pad to extend to word 0x0000_0080 = 128
  - Signed? Then 0x80 = -128, so sign-extend to word 0xFFFF_FF80 = -128

Load/Store Multiple

- LDM/LDMIA: load multiple registers starting from [base register], update base register afterwards
  - LDM <Rn>!,<registers>
  - LDM <Rn>,<registers>



- STM/STMIA: store multiple registers starting at [base register], update base register after
  - STM <Rn>!, <registers>

- LDMIA and STMIA are pseudo-instructions, translated by assembler

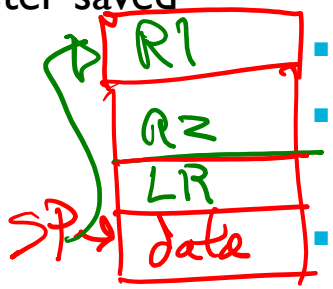
STACK INSTRUCTIONS

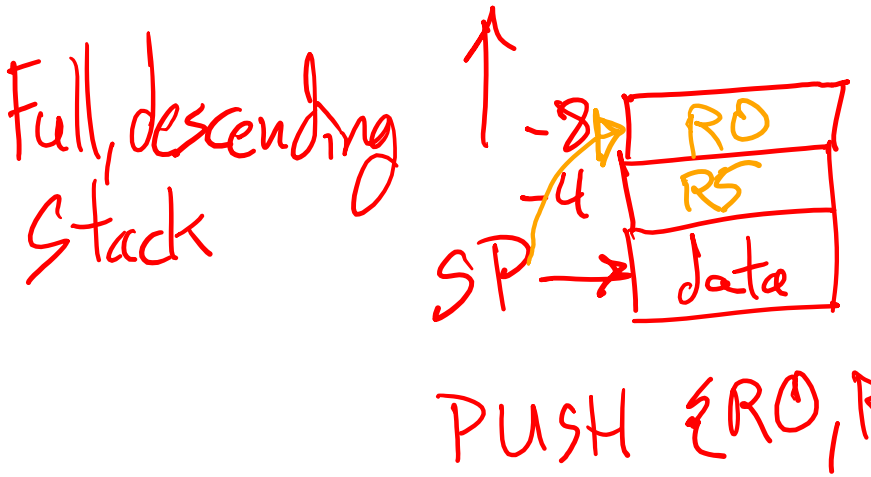
Stack Operations

- Push some or all of registers (R0-R7, LR) to stack
  - PUSH {<registers>}
  - **Decrements** SP by 4 bytes for each register saved
  - Pushing LR saves return address
  - Always pushes registers in same order:
    - Smaller register number -> smaller memory address. LR = r14
    - So larger number register numbers are pushed before smaller (i.e. LR to r0)
  - PUSH {r1, r2, LR}
- Pop some or all of registers (R0-R7, PC) from stack
  - POP {<registers>}
  - **Increments** SP by 4 bytes for each register restored
  - If PC is popped, then execution will branch to new PC value after this POP instruction (e.g. return address)
  - Always pops registers in same order (opposite of pushing)
    - Smaller memory address -> register number. PC = r15
    - So smaller number registers are popped before larger (i.e. r0 to PC)
  - POP {r5, r6, r7}

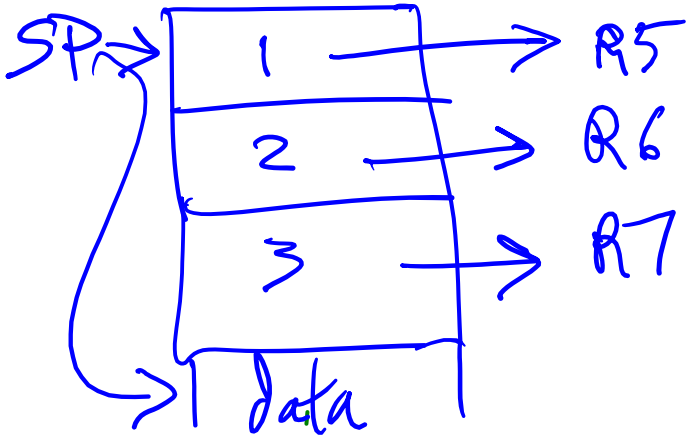
Stack Operations

- **Push** some or all of registers (R0-R7, LR) to stack
  - PUSH {<registers>}
  - Always pushes registers in same order:
    - Smaller register number -> smaller memory address. LR = r14
    - So **larger** number registers are pushed first (i.e. LR to r0)
  - **Decrements** SP by 4 bytes for each register saved
  - Pushing LR saves return address
  - Example: PUSH {r1, r2, LR}





- **Pop** some or all of registers (R0-R7, PC) from stack
  - POP {<registers>}
  - Always pops registers in same order (opposite of pushing)
    - Smaller memory address -> register number. PC = r15
    - So **smaller** number registers are popped first (i.e. r0 to PC)
  - **Increments** SP by 4 bytes for each register restored
  - If PC is popped, then execution will branch to new PC value after this POP instruction (e.g. return address)
  - Example: POP {r5, r6, r7}



INSTRUCTIONS FOR DATA PROCESSING

Add Instructions

- Add registers, update condition flags
  - ADDS <Rd>,<Rn>,<Rm>
- Add registers and carry bit, update condition flags
  - ADCS <Rdn>,<Rm>
- Add registers
  - ADD <Rdn>,<Rm>
- Add immediate value to register
  - ADDS <Rd>,<Rn>,#<imm3>
  - ADDS <Rdn>,#<imm8>

Add Instructions with Stack Pointer

- Add SP and immediate value
  - ADD <Rd>, SP, #<imm8>
  - ADD SP, SP, #<imm7>

- Add SP and register
  - ADD <Rdm>, SP, <Rdm>
  - ADD SP, <Rm>

Address to Register Pseudo-Instruction

- Add immediate value to PC, write result in register
  - ADR <Rd>,<label>
- How is this used?
  - Enables storage of constant data near program counter
  - First, load register R2 with address of const_data
  - ADR R2, const_data
  - Second, load const_data into R2
  - LDR R2, [R2]
- Value must be close to current PC value

Subtract

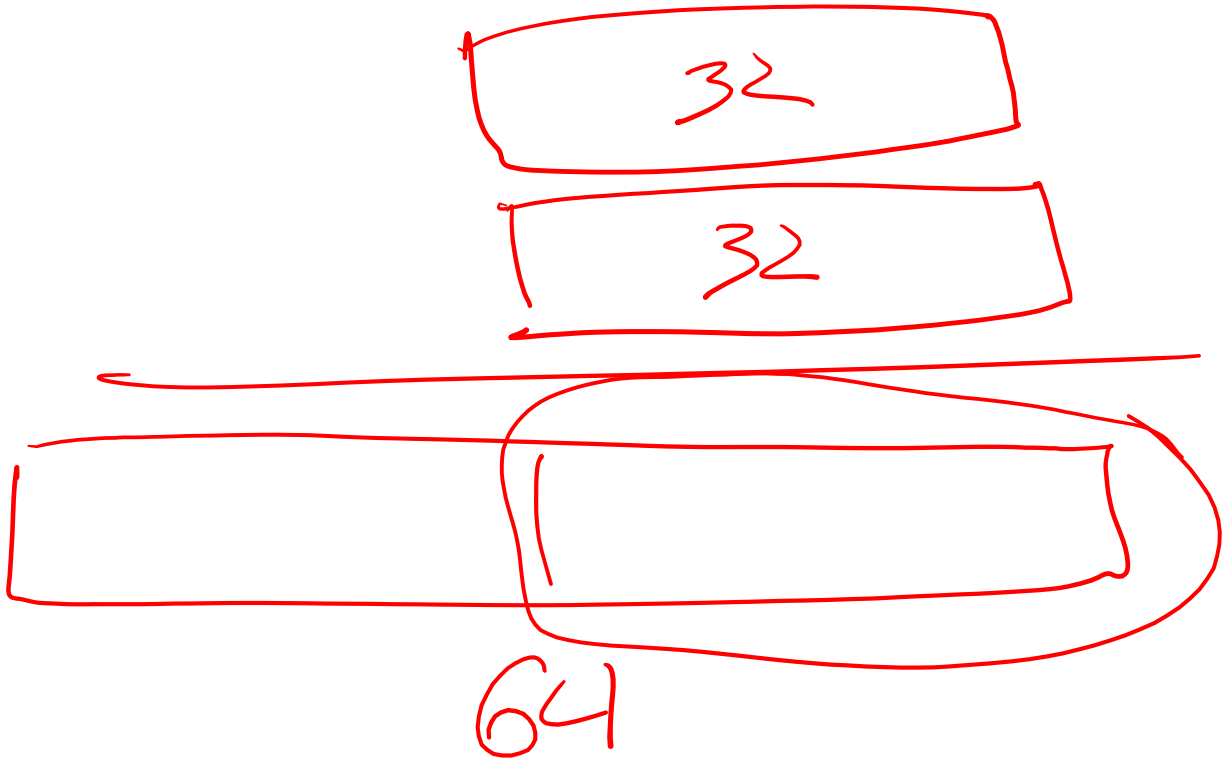
- Subtract immediate from register, update condition flags
  - SUBS <Rd>, <Rn>, #<imm3>
  - SUBS <Rdn>, #<imm8>
- Subtract registers, update condition flags
  - SUBS <Rd>, <Rn>, <Rm>
- Subtract registers with carry, update condition flags
  - SBCS <Rdn>, <Rm>
- Subtract immediate from SP
  - SUB SP, SP, #<imm7>

Multiply

- Multiply source registers, save lower word of result in destination register, update condition flags
  - MULS <Rdm>, <Rn>, <Rdm>
  - <Rdm> = <Rdm> * <Rn>

- Signed multiply

- Note:
  - 32-bit * 32-bit = 64-bit
  - Upper word of result is truncated



Logical Operations

- All of these instructions update the condition flags

- Bitwise AND registers
  - ANDS <Rdn>,<Rm>
- Bitwise OR registers
  - ORRS <Rdn>,<Rm>
- Bitwise Exclusive OR registers
  - EORS <Rdn>,<Rm>
- Bitwise AND register and complement of second register
  - BICS <Rdn>,<Rm>
- Move inverse of register value to destination
  - MVNS <Rd>,<Rm>
- Bitwise AND two registers, discard result
  - TST <Rn>, <Rm>

Compare

- Compare - subtracts second value from first, updates condition flags, discards result
  - CMP <Rn>,#<imm8>
  - CMP <Rn>,<Rm>

- Compare negative - **adds** two values, updates condition flags, discards result
  - CMN <Rn>,<Rm>

Shift and Rotate

- Common features
  - All of these instructions update APSR condition flags
  - Shift/rotate amount (in number of bits) specified by last operand
- Logical shift left - shifts in zeroes on right
  - LSLS <Rd>,<Rm>,#<imm5>
  - LSLS <Rdn>,<Rm>
- Logical shift right - shifts in zeroes on left
  - LSRS <Rd>,<Rm>,#<imm5>
  - LSRS <Rdn>,<Rm>
- Arithmetic shift right - shifts in copies of sign bit on left (to maintain arithmetic sign)
  - ASRS <Rd>,<Rm>,#<imm5>
- Rotate right
  - RORS <Rdn>,<Rm>

Reversing Bytes

- REV - reverse all bytes in word
  - REV <Rd>,<Rm>
- REV16 - reverse bytes in both half-words
  - REV16 <Rd>,<Rm>
- REVSH - reverse bytes in low half-word (signed) and sign-extend
  - REVSH <Rd>,<Rm>

INSTRUCTIONS FOR CONTROL FLOW

Changing Program Flow - Branches

- Unconditional Branches
  - B <label>
  - Target address must be within 2 KB of branch instruction (-2048 B to +2046 B)
  - Takes 2 cycles to execute (flush pipeline fetch stage)
- Conditional Branches
  - B<cond> <label>
  - <cond> is condition - see next page
  - B<cond> target address must be within of branch instruction
  - B target address must be within 256 B of branch instruction (-256 B to +254 B)
  - Not-taken branch (condition false) needs 1 cycle to execute
  - Taken branch (condition true) needs 2 cycles to execute (flush pipeline fetch stage)

Condition Codes

- Append to branch instruction (B) to make a conditional branch
- Full ARM instructions (not Thumb or Thumb-2) support conditional execution of arbitrary instructions
- Note: Carry bit = not-borrow for compares and subtractions

Suffix	Flags	Meaning
EQ	Z = 1	Equal, last flag setting result was zero.
NE	Z = 0	Not equal, last flag setting result was non-zero.
CS or HS	C = 1	Higher or same, unsigned.
CC or LO	C = 0	Lower, unsigned.
MI	N = 1	Negative.
PL	N = 0	Positive or zero.
VS	V = 1	Overflow.
VC	V = 0	No overflow.
HI	C = 1 and Z = 0	Higher, unsigned.
LS	C = 0 or Z = 1	Lower or same, unsigned.
GE	N = V	Greater than or equal, signed.
LT	N != V	Less than, signed.
GT	Z = 0 and N = V	Greater than, signed.
LE	Z = 1 or N != V	Less than or equal, signed.
AL	Can have any value	Always. This is the default when no suffix is specified.

Changing Program Flow - Subroutines

- Call
  - BL <label> - branch with link to return address
    - Call subroutine at <label>
      - PC-relative, range limited to PC+/-16MB
    - Save return address in LR
  - BLX <Rd> - branch with link to return address and instruction set exchange
    - Call subroutine at address in register Rd
      - Supports full 4GB address range
    - Save return address in LR
- Return
  - BX <Rd> branch and instruction set exchange
    - Branch to address specified by <Rd>
      - Supports full 4 GB address space
    - BX LR - Return from subroutine
  - POP {PC} with instruction set exchange
- *Interworking*: Changing instruction set state
  - Cortex-A processors can interwork with BX, BLX, POP by specifying state with LSB of target address (1: Thumb, 0: ARM)
  - Cortex-M processors must always be in Thumb state, so BX, BLX, POP {PC} must have odd target addresses (LSB = 1)

SPECIAL INSTRUCTIONS

Special Register Instructions

- Move to Register from Special Register
  - MSR <Rd>, <spec_reg>
- Move to Special Register from Register
  - MRS <spec_reg>, <Rd>
- Change Processor State - Modify PRIMASK register
  - CPSIE - Interrupt enable
  - CPSID - Interrupt disable

Special register	Contents
APSR	The flags from previous instructions.
IAPSR	A composite of IPSR and APSR.
EAPSR	A composite of EPSR and APSR.
XPSR	A composite of all three PSR registers.
IPSR	The Interrupt status register.
EPSR	The execution status register.[b]
IEPSR	A composite of IPSR and EPSR.
MSP	The Main Stack pointer.
PSP	The Process Stack pointer.
PRIMASK	Register to mask out configurable exceptions.[c]
CONTROL	The CONTROL register, see *The special-purpose CONTROL register* on page B1-215.

Other

- No Operation - does nothing!
  - NOP
- Breakpoint - causes hard fault or debug halt - used to implement software breakpoints
  - BKPT #<imm8>
- Wait for interrupt - Pause program, enter low-power state until a WFI wake-up event occurs (e.g. an interrupt)
  - WFI
- Supervisor call generates SVC exception (#11), same as software interrupt
  - SVC #<imm>