

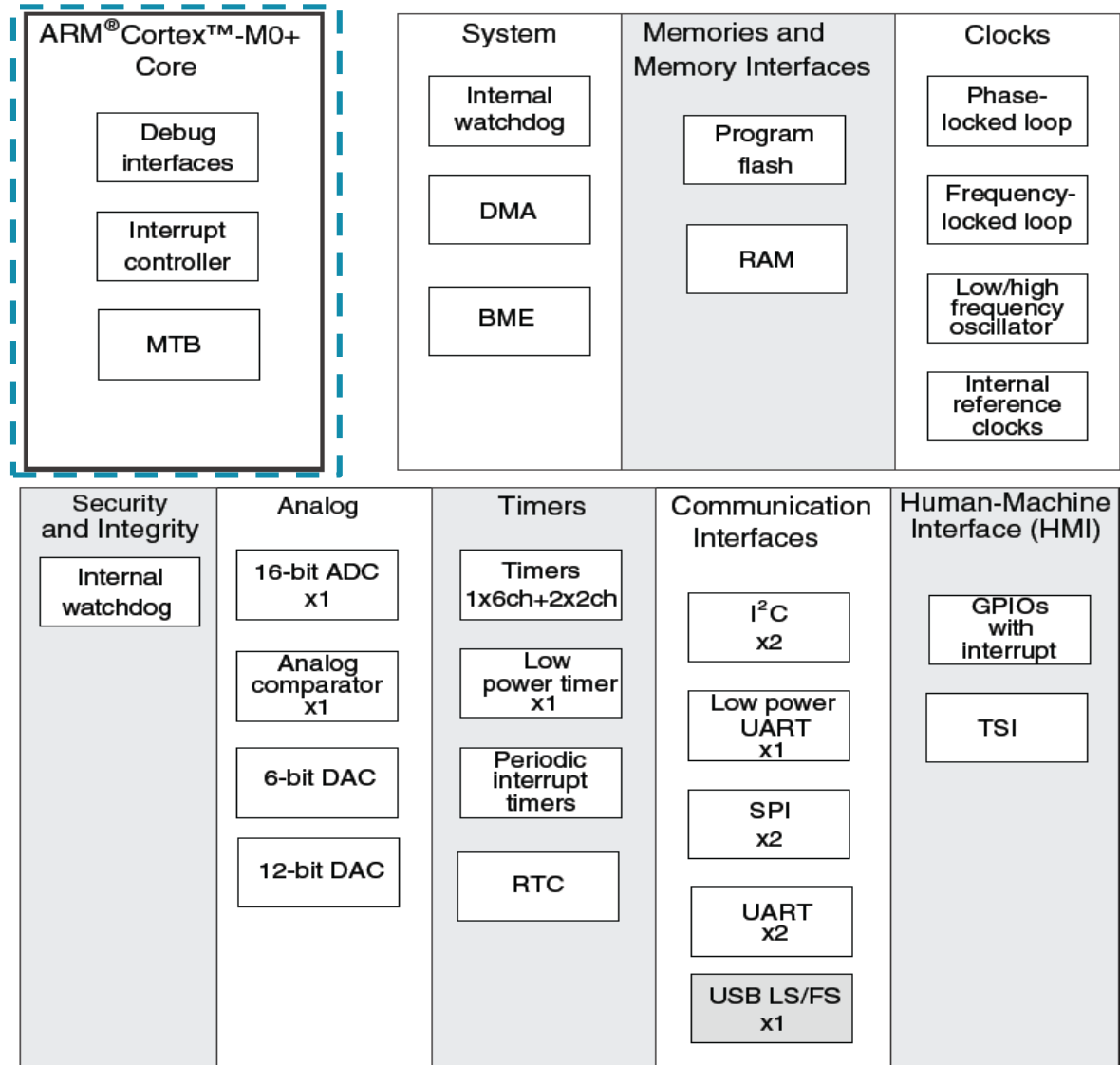
Module 1B - Cortex-M0+ CPU Core

Overview

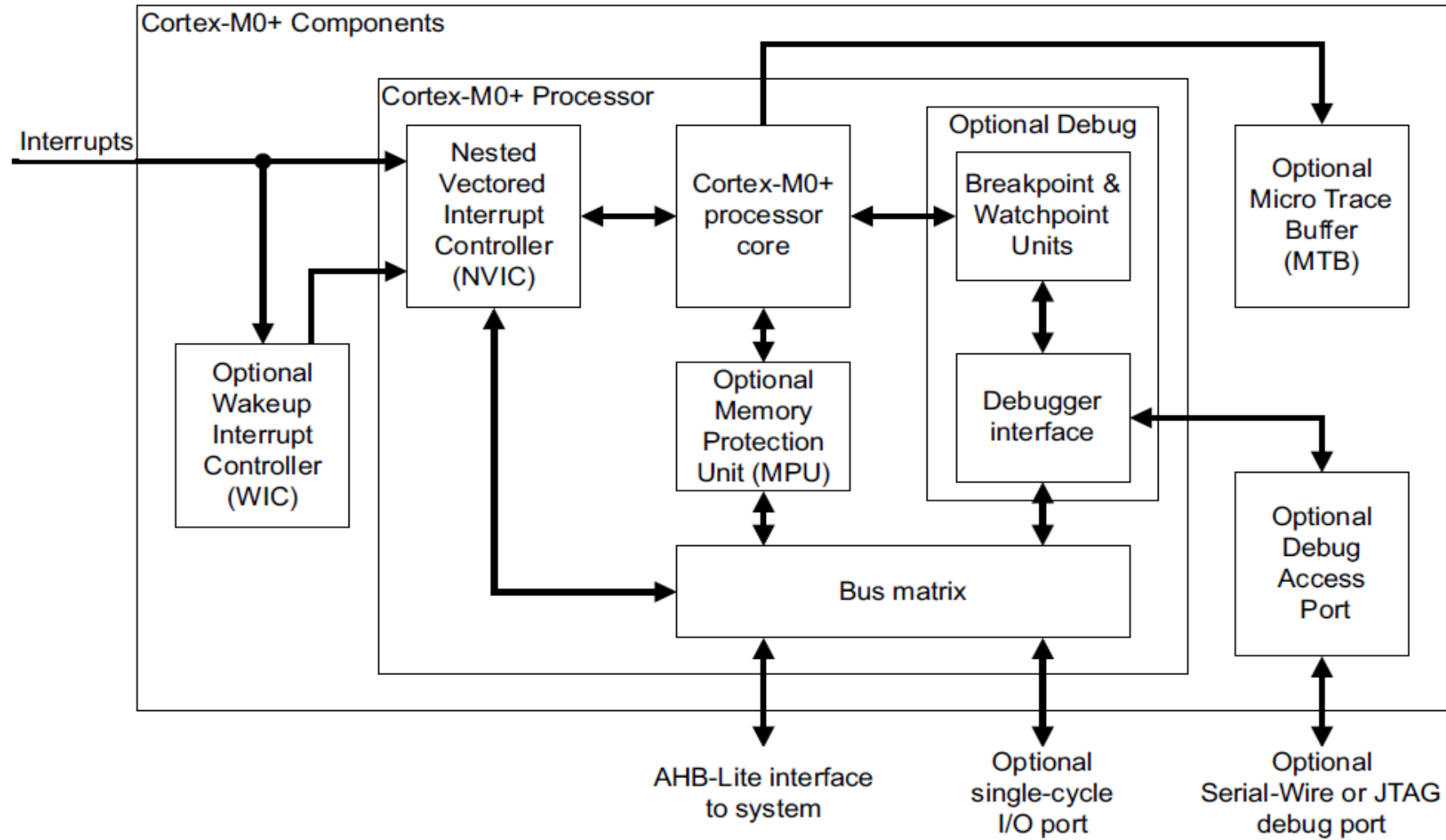
- Cortex-M0+ Processor Core Registers
- Memory System and Addressing
- Thumb Instruction Set
- References
 - DDI0419C Architecture ARMv6-M Reference Manual

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



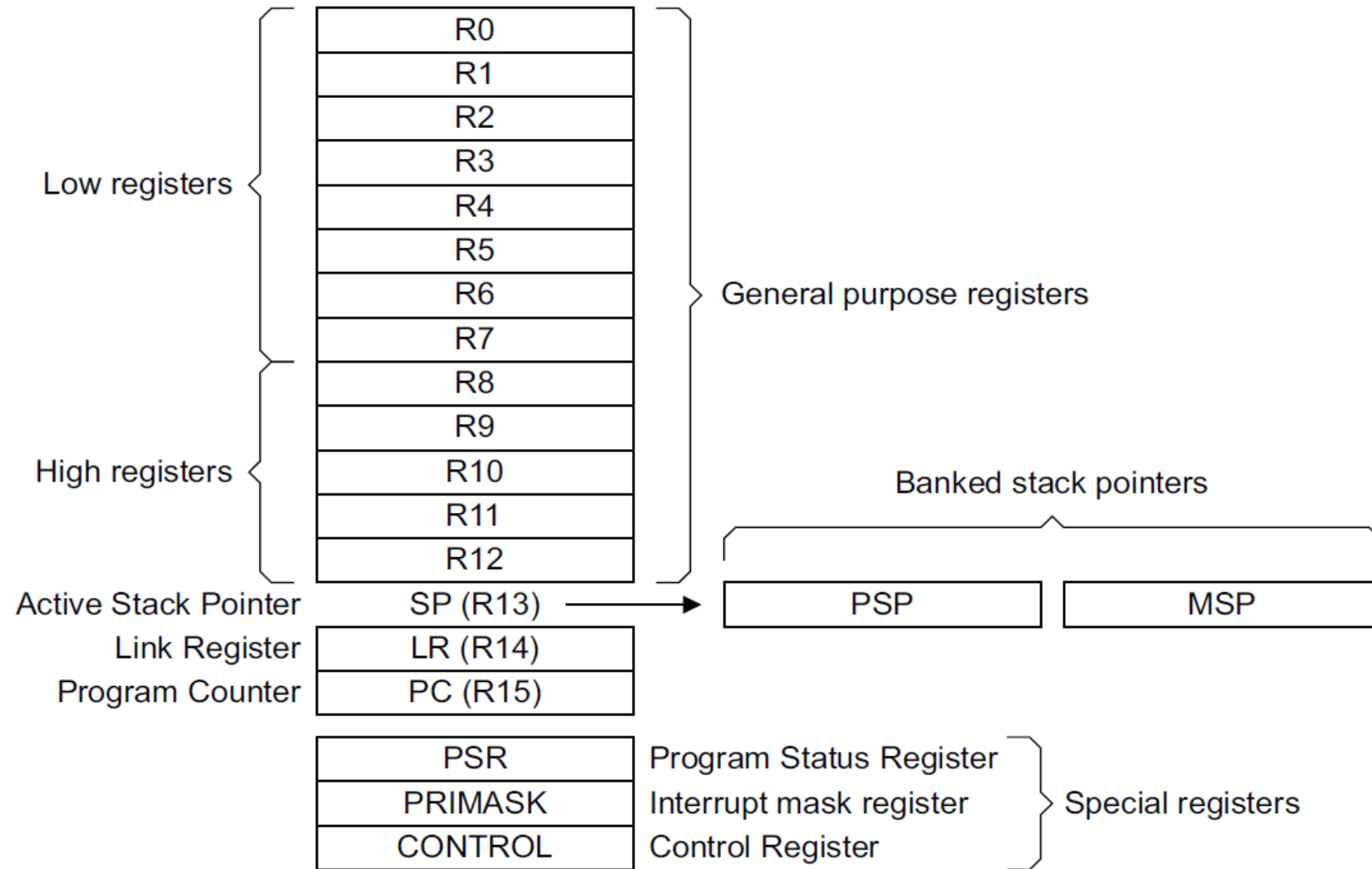
Cortex-M0+ Core



Architectures and Memory Speed

- Load/Store Architecture
 - Developed to simplify CPU design and improve performance
 - *Memory wall*: CPUs keep getting faster than memory
 - Memory accesses slow down CPU, limit compiler optimizations
 - Change instruction set to make most instructions *independent* of memory
 - Data processing instructions can access registers only
 1. Load data into the registers
 2. Process the data
 3. Store results back into memory
 - More effective when more registers are available
- Register/Memory Architecture
 - Data processing instructions can access memory or registers
 - Memory wall is not very high at lower CPU speeds (e.g. under 50 MHz)

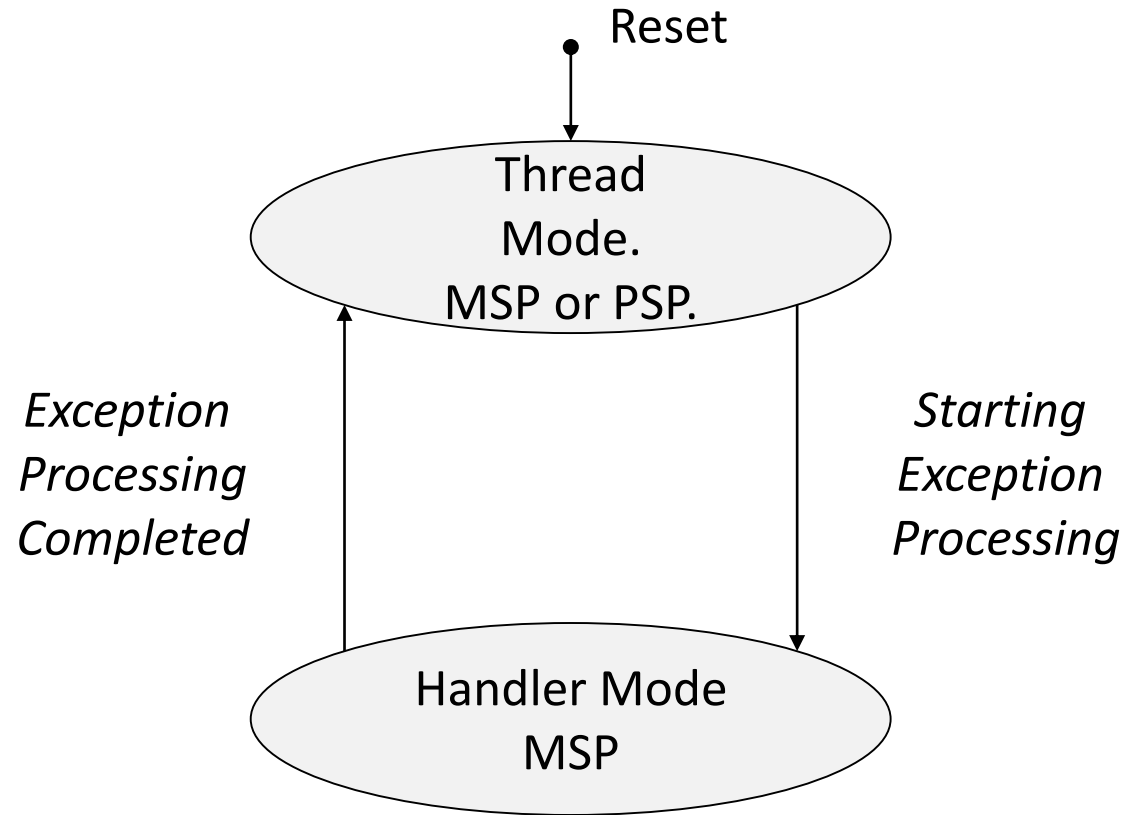
ARM Processor Core Registers



ARM Processor Core Registers (32 bits each)

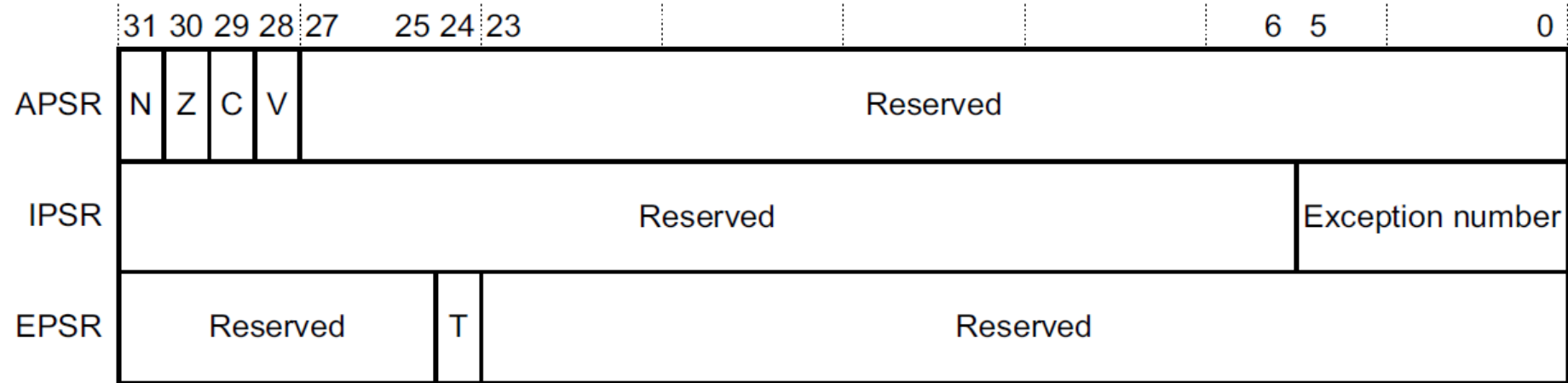
- R0-R12 - General purpose registers for data processing
- SP - Stack pointer (R13)
 - Can refer to one of two SPs
 - Main Stack Pointer (MSP)
 - Process Stack Pointer (PSP)
 - Uses MSP initially, and whenever in Handler mode
 - When in Thread mode, can select either MSP or PSP using SPSEL flag in CONTROL 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 Processor Core Registers

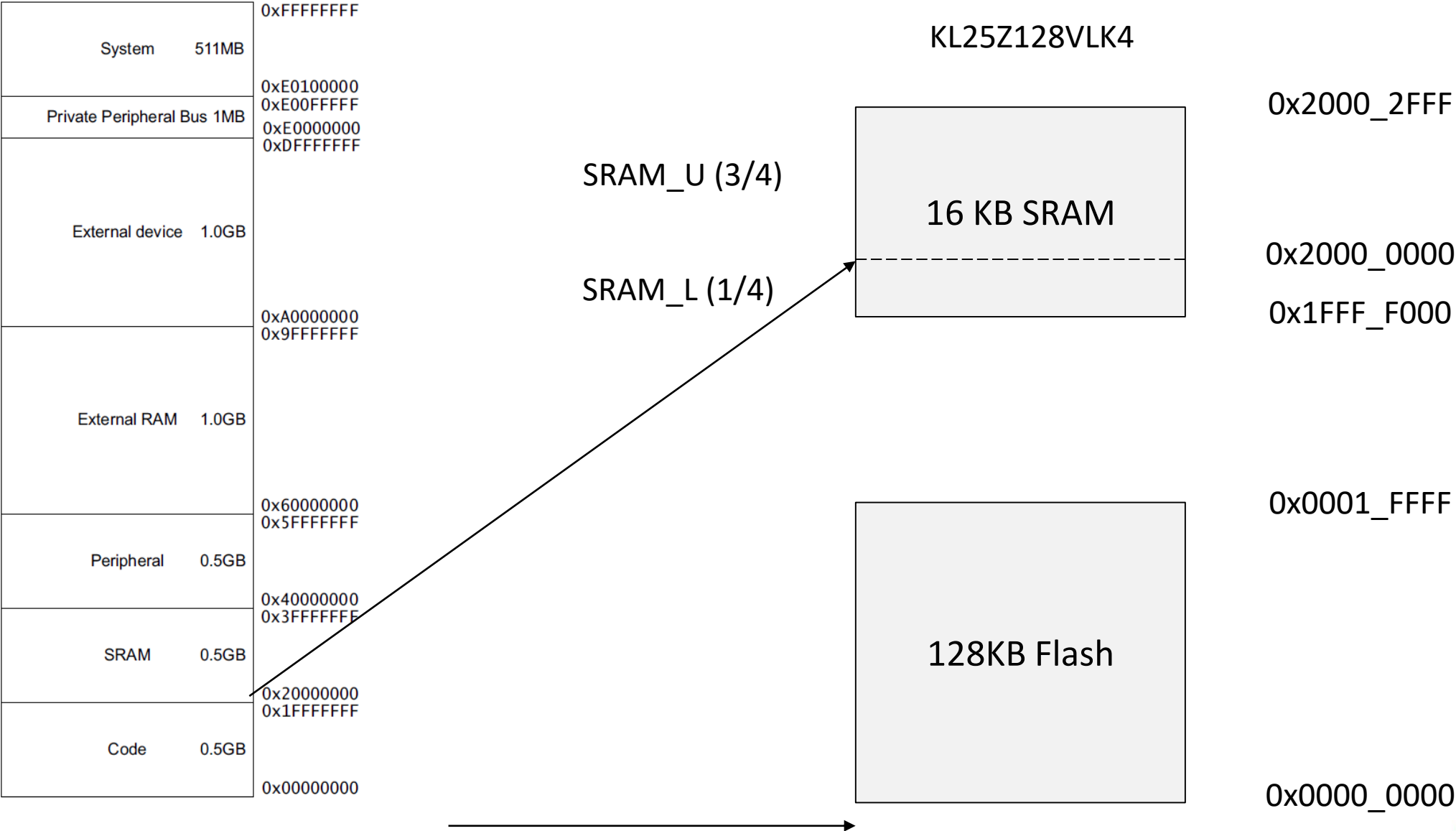


- Program Status Register (PSR) is three views of same register
 - Application PSR (APSR)
 - Condition code flag bits Negative, Zero, oVerflow, Carry
 - Interrupt PSR (IPSR)
 - Holds exception number of currently executing ISR
 - Execution PSR (EPSR)
 - Thumb state

ARM Processor Core Registers

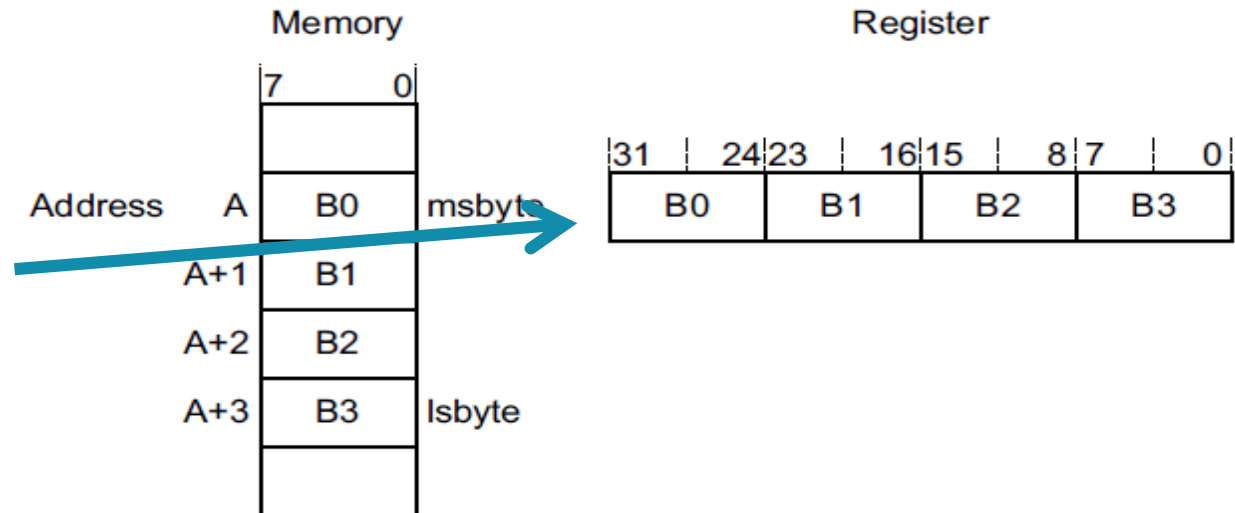
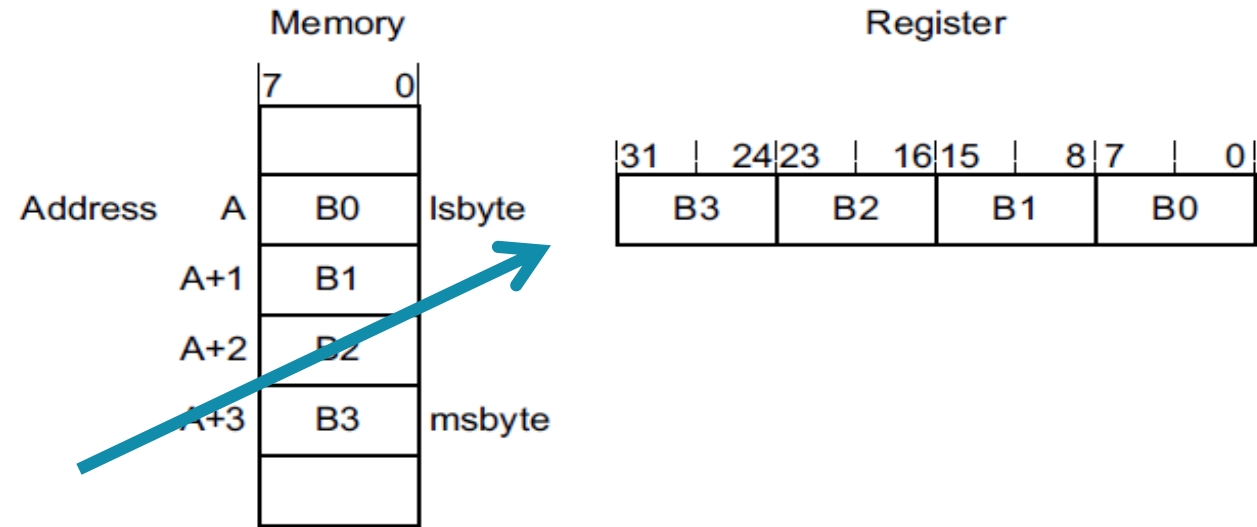
- PRIMASK - Exception mask register
 - Bit 0: PM Flag
 - Set to 1 to prevent activation of all exceptions with configurable priority
 - Access using CPS, MSR and MRS instructions
 - Use to prevent data race conditions with code needing atomicity
- CONTROL
 - Bit 1: SPSEL flag
 - Selects SP when in thread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

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

ARM, Thumb and Thumb-2 Instructions

- ARM instructions optimized for resource-rich high-performance computing systems
 - Deeply pipelined processor, high clock rate, wide (e.g. 32-bit) memory bus
- 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
- Modifications to ARM ISA to fit low-end embedded computing
 - 1995: Thumb instruction set
 - 16-bit instructions
 - Reduces memory requirements (and performance slightly)
 - 2003: Thumb-2 instruction set
 - Adds some 32 bit instructions
 - Improves speed with little memory overhead
 - CPU decodes instructions based on whether in Thumb state or ARM state - controlled by T bit

Instruction Set

- Cortex-M0+ core implements ARMv6-M Thumb instructions
- Only uses Thumb instructions, always in Thumb state
 - Most instructions are 16 bits long, some are 32 bits
 - Most 16-bit instructions can only access low registers (R0-R7), but some can access high registers (R8-R15)
- Conditional execution only supported for 16-bit branch
- 32 bit address space
- Half-word aligned instructions
- See ARMv6-M Architecture Reference Manual for specifics per instruction (Section A.6.7)

Assembly Instructions

- Arithmetic and logic
 - Add, Subtract, Multiply, Divide, Shift, Rotate
- Data movement
 - Load, Store, Move
- Compare and branch
 - Compare, Test, If-then, Branch, compare and branch on zero
- Miscellaneous
 - Breakpoints, wait for events, interrupt enable/disable, data memory barrier, data synchronization barrier

Instruction Format: Labels

label **mnemonic operand1, operand2, operand3 ; comments**

- ▶ Place marker, marking the memory address of the current instruction
- ▶ Used by branch instructions to implement **if-then** or **goto**
- ▶ Must be unique

Instruction Format: Mnemonic

label **mnemonic** operand1, operand2, operand3 ; comments

- ▶ The name of the instruction
- ▶ Operation to be performed by processor core

Instruction Format: Operands

label **mnemonic** **operand1, operand2, operand3** ; **comments**

- ▶ Operands
 - ▶ Registers
 - ▶ Constants (called *immediate values*)
- ▶ Number of operands varies
 - ▶ No operands: **DSB**
 - ▶ One operand: **BX LR**
 - ▶ Two operands: **CMP R1, R2**
 - ▶ Three operands: **ADD R1, R2, R3**
 - ▶ Four operands: **MLA R1, R2, R3, R4**
- ▶ Normally
 - ▶ **operand1** is the destination register, and operand2 and operand3 are source operands.
 - ▶ **operand2** is usually a register, and the first source operand
 - ▶ **operand3** may be a register, an immediate number, a register shifted to a constant number of bits, or a register plus an offset (used for memory access).

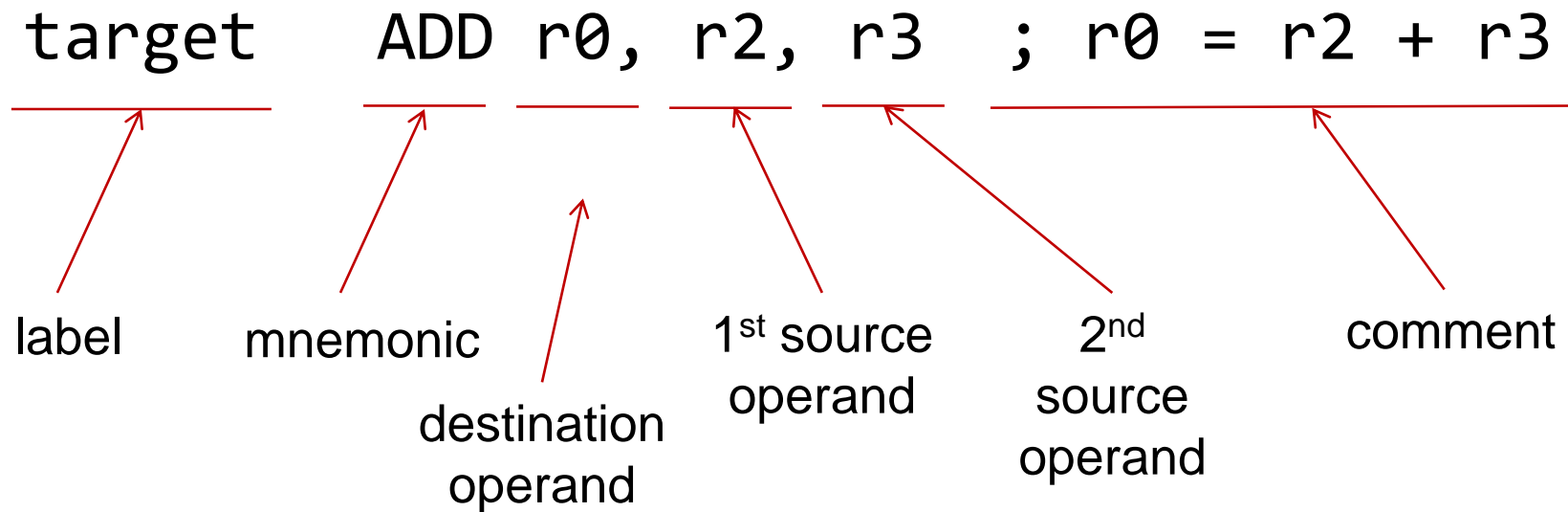
Instruction Format: Comments

`label mnemonic operand1, operand2, operand3 ; comments`

- ▶ Everything after the semicolon (;) is a comment
- ▶ Explain programmers' intentions or assumptions

ARM Instruction Format

label mnemonic operand1, operand2, operand3 ; comments



ARM Instruction Format

target ADD r0, r2, r3 ; r0 = r2 + r3



poor comment!

A better example:

; Increment angle r2 by step
size r3

ARM Instruction Format

Examples: Variants of the ADD instruction

ADD r1, r2, r3 ; r1 = r2 + r3

ADD r1, r3 ; r1 = r1 + r3

ADD r1, r2, #4 ; r1 = r2 + 4

ADD r1, #15 ; r1 = r1 + 15

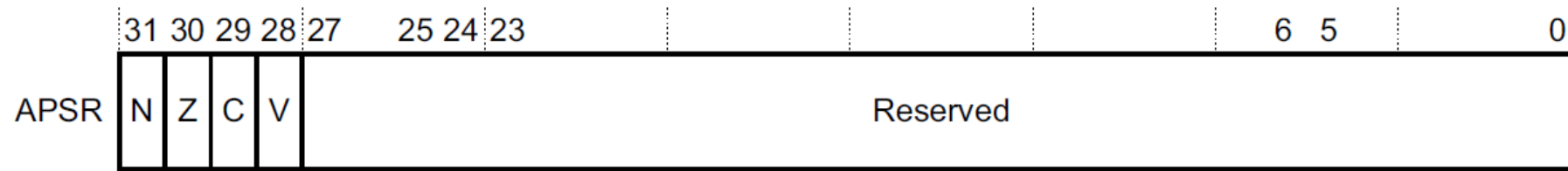
Assembler Instruction Format

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

Where Can the Operands Be Located?

- In a general-purpose register R
 - Destination: Rd
 - Source: Rm, Rn
 - Both source and destination: Rdn
 - Target: Rt
 - Source for shift amount: Rs
- An immediate value encoded in instruction word
- In a condition code flag
- In memory
 - Only for load, store, push and pop instructions

Update Condition Codes in APSR?



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

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
Conditional branch	IT, 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

Load/Store Register

- 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

Load-Modify-Store

C statement

$$X = X + 1;$$

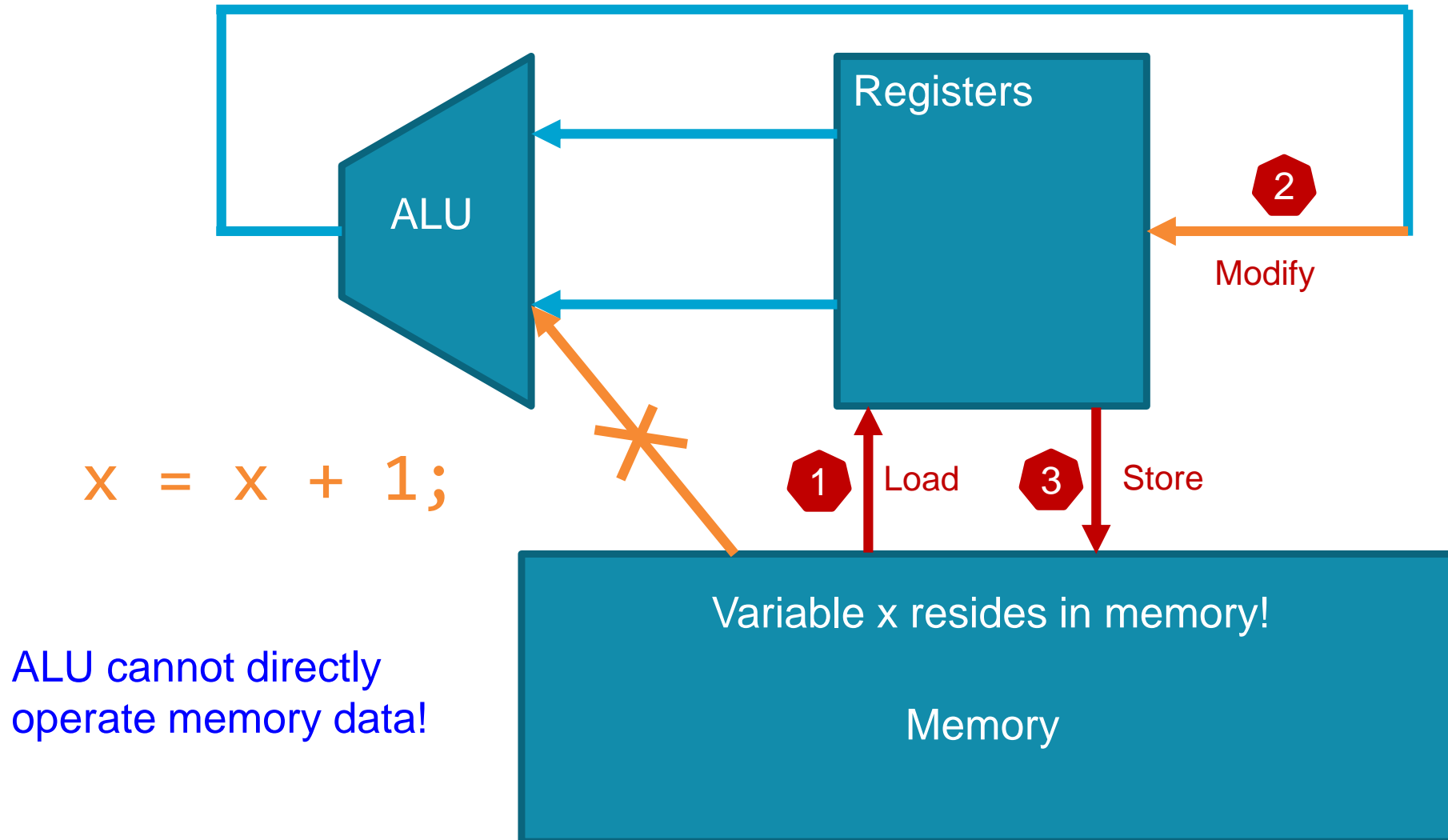


Assume variable X resides in memory
and is a 32-bit integer

**; Assume the memory address of x is stored in
r1**

```
LDR r0, [r1]      ; load value of x from memory  
ADD r0, r0, #1    ; x = x + 1  
STR r0, [r1]      ; store x into memory
```

3 Steps: Load, Modify, Store



Load Instructions

- **LDR Rt, [Rs]**
 - **Read from memory**
 - Mnemonic: LoaD to Register (**LDR**)
 - rs specifies the memory address
 - rt holds the 32-bit value fetched from memory

Example:

```
; Assume r0 = 0x08200004  
; Load a word:  
LDR r1, [r0]           ; r1 = Memory.word[0x08200004]
```

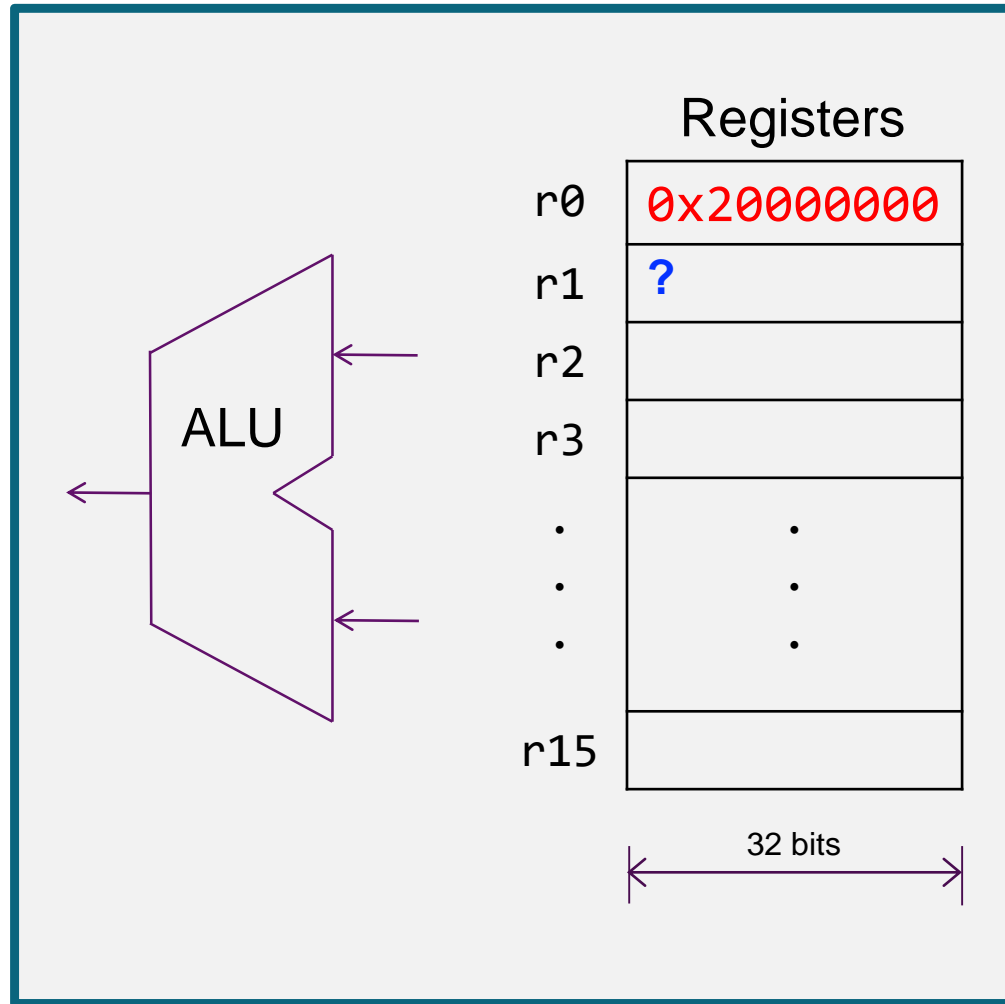
Store Instructions

- **STR Rt, [Rs]**
 - **Write into memory**
 - Mnemonic: **ST**ore from **R**egister (**STR**)
 - rs specifies memory address
 - Save the content of rt into memory

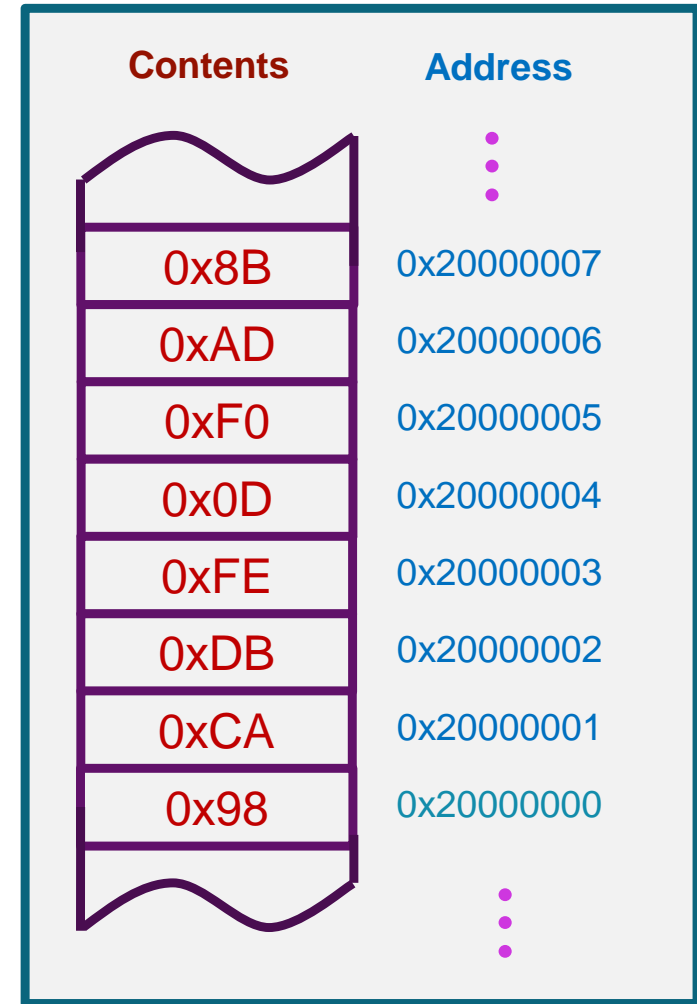
Example:

```
; Assume r0 = 0x08200004  
; Store a word  
STR r1, [r0]      ; Memory.word[0x08200004] = r1
```


LDR r1, [r0] ; r1 = memory.word[r0]
; LDR stands for Load to Register

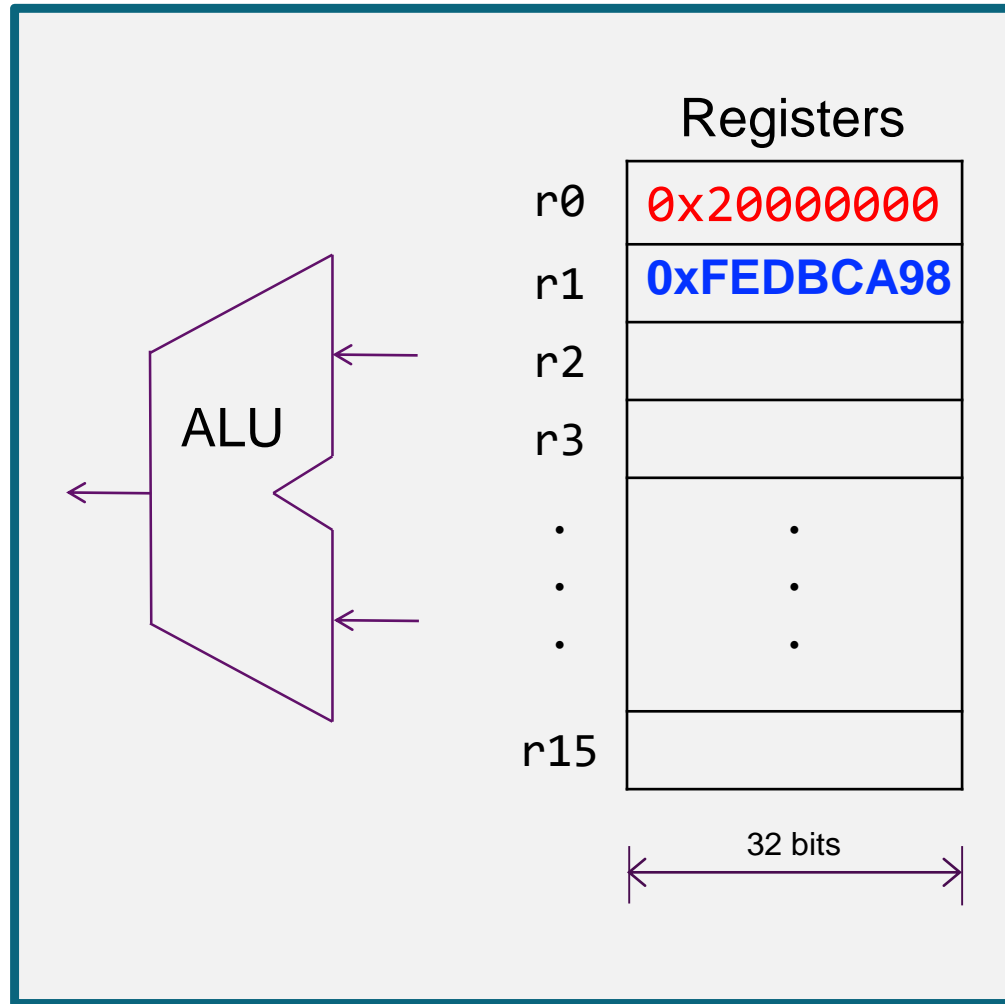


Processor
Core

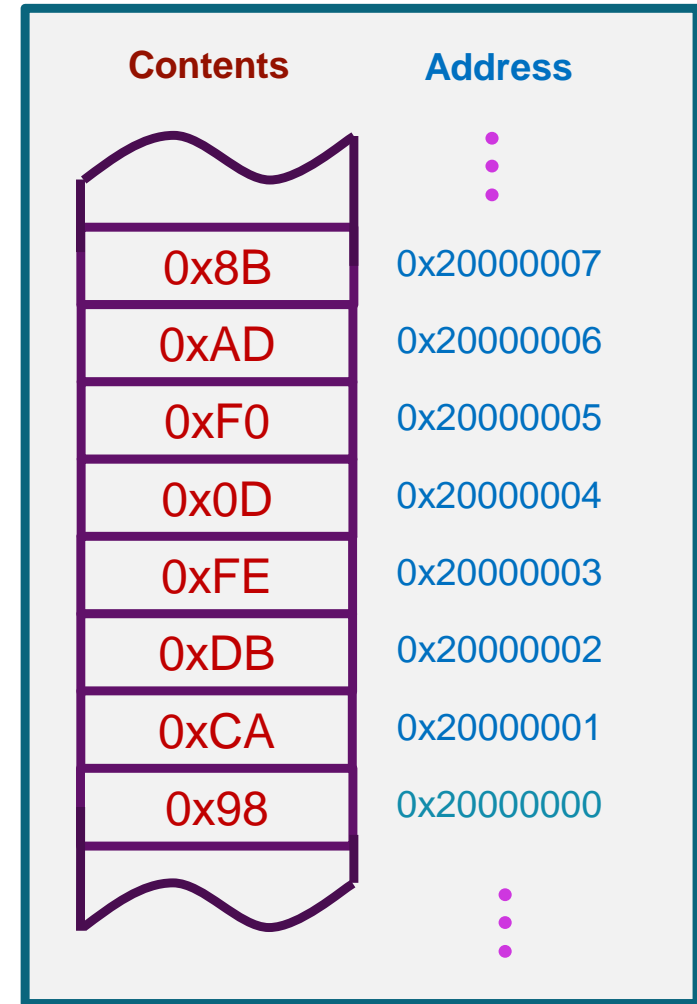


Memory

LDR r1, [r0] ; r1 = memory.word[r0]
; LDR stands for Load to Register

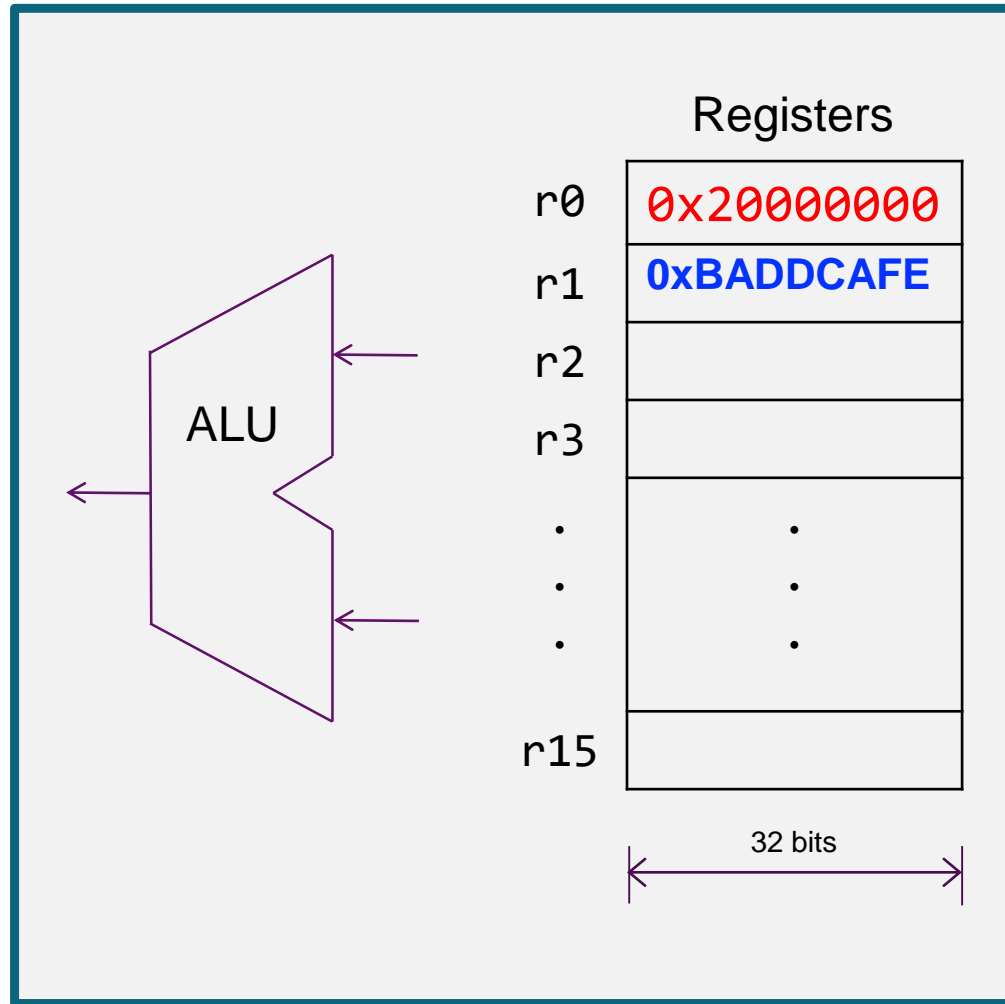


Processor
Core

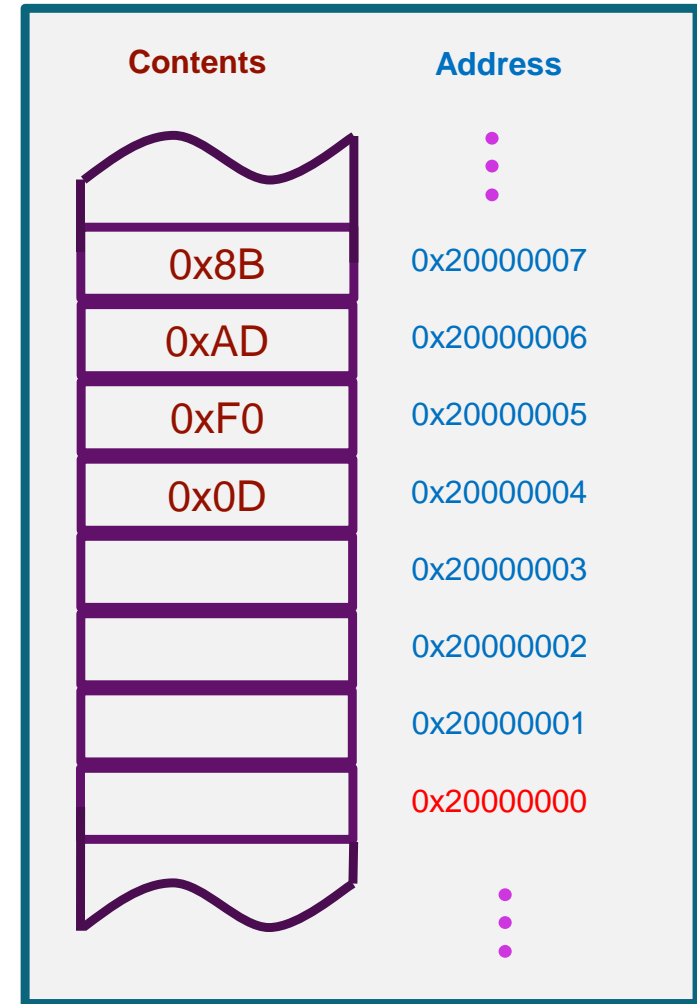


Memory

STR r1, [r0] ; memory.word[r0] = r1
; STR stands for Store Register

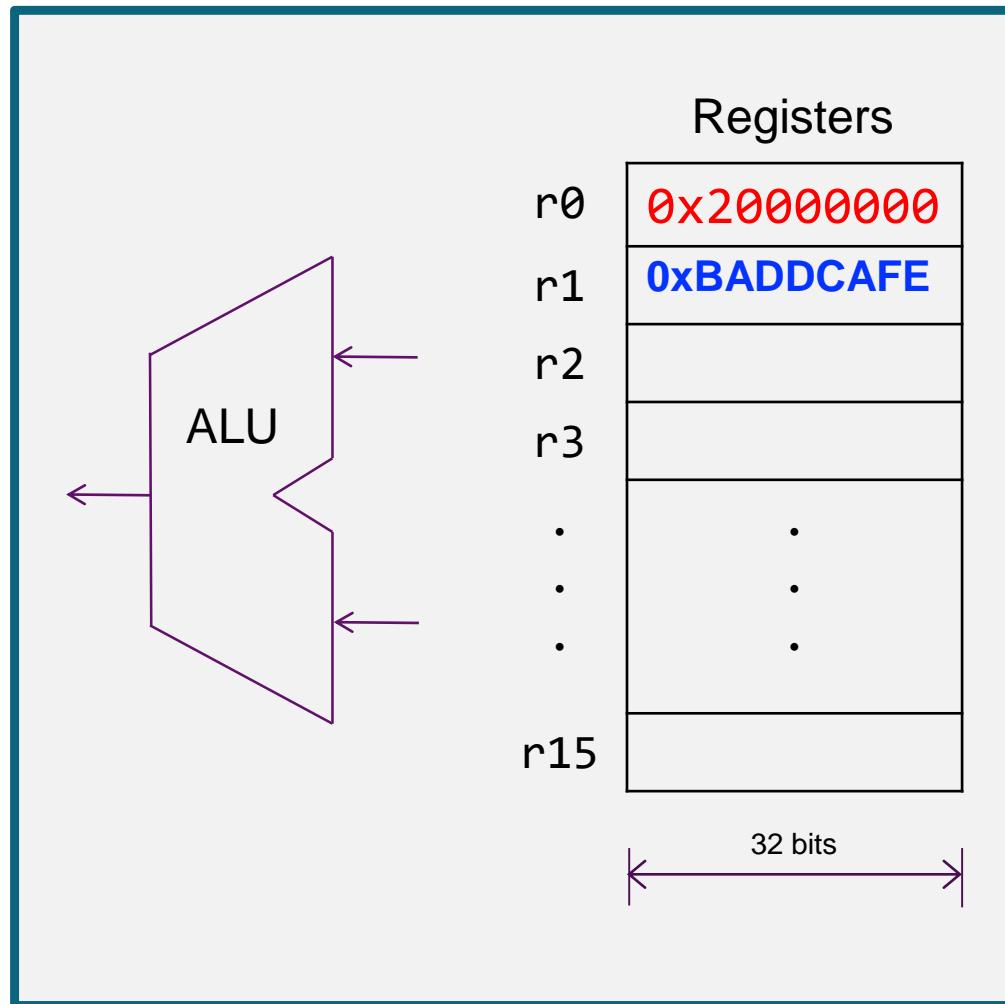


Processor
Core

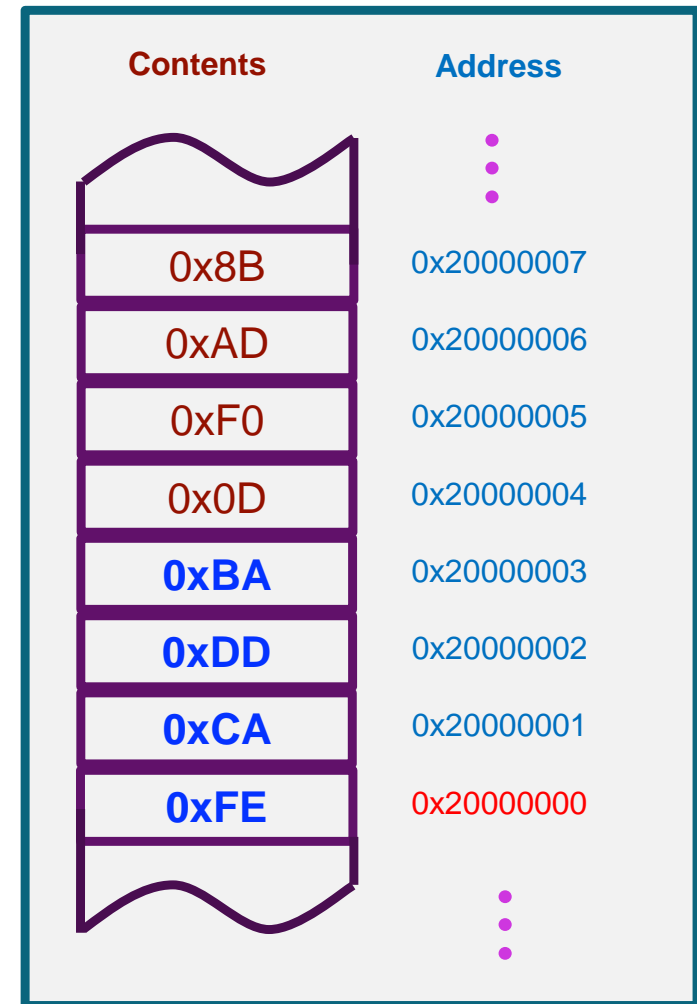


Memory

STR r1, [r0] ; memory.word[r0] = r1
; STR stands for Store Register



Processor
Core



Memory

Modes for Addressing Memory

- Offset Addressing mode: [$\langle Rn \rangle$, $\langle \text{offset} \rangle$] accesses address $\langle Rn \rangle + \langle \text{offset} \rangle$
- Base Register $\langle Rn \rangle$
 - Can be register R0-R7, SP or PC
- $\langle \text{offset} \rangle$ is added or subtracted from base register to create effective address
 - Can be an immediate constant
 - Can be another register, used as index $\langle Rm \rangle$
- Auto-update: Can write effective address back to base register
- Pre-indexing: use **effective address** to access memory, then update base register with that effective address
- Post-indexing: use **base register** to access memory, then update base register with effective address

Address Modes: Immediate Offset

- Address accessed by **LDR/STR** is specified by a base register **plus an offset**
- Offset can be **an immediate value**

LDR r0,[r1,#8]

- Base memory address hold in register r1
- Offset is an immediate value
- Target address = $r1 + 8$

Three modes for immediate offset:

- Pre-index,
- Post-index,
- Pre-index with Update

Addressing Mode: Pre-index vs Post-index

- Pre-index

`LDR r1, [r0, #4] ; r1 ← memory[r0 + 4], r0 is unchanged`

- Post-index

`LDR r1, [r0], #4 ; r1 ← memory[r0], r0 ← r0 + 4`

- Pre-index with Update

`LDR r1, [r0, #4]! ; r1 ← memory[r0+4], r0 ← r0 + 4`

Accessing an Array

► C code

```
uint32_t array[10];  
array[0] += 5;  
array[1] += 5;
```

Assume the memory address of the array starts at 0x20008000.

■ Pre-index

Assume r0 = 0x20008000.

```
LDR r1, [r0]      ; Read array[0]  
ADD r1, r1, #5  
STR r1, [r0]      ; Write to array[0]  
  
LDR r1, [r0, #4]  ; Read array[1]  
ADD r1, r1, #5  
STR r1, [r0, #4]  ; Write to array[1]
```


Loading/Storing Smaller Data Sizes

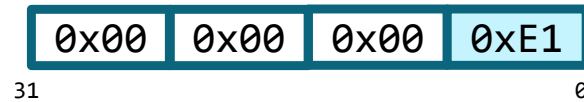
- Some load and store instructions can handle half-word (16 bits) and byte (8 bits)
- Store just writes to half-word or byte
 - STRH, STRB
- Loading a byte or half-word requires padding or extension: What do we put in the upper bits of the register?
 - Example: How do we extend 0x80 into a full word?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

	Signed	Unsigned
Byte	LDRSB	LDRB
Half-word	LDRSH	LDRH

Load a Byte, Half-word, Word

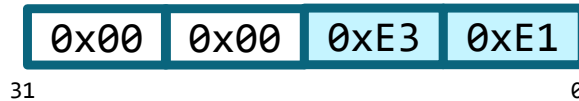
Load a Byte

LDRB r1, [r0]



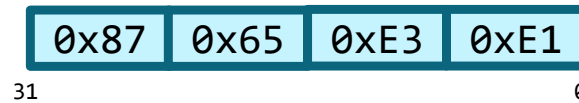
Load a Halfword

LDRH r1, [r0]



Load a Word

LDR r1, [r0]



0x02000003	0x87
0x02000002	0x65
0x02000001	0xE3
0x02000000	0xE1

Little Endian

Assume
r0 = 0x20000000

In-Register Size Extension

- 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 $0xFFFF_FF80 = -128$
- Example

SXTB R0, R1

	Signed	Unsigned
Byte	SXTB	UXTB
Half-word	SXTH	UXTH

Address Modes: Offset in Register

- Address accessed by **LDR/STR** is specified by a base register **plus an offset**
- Offset can be held in **a register**

LDR r0,[r1,r2]

- Base memory address hold in register r1
- Offset held at r2
- Target address = $r1 + r2$

LDR r0,[r1,r2,LSL #2]

- Base memory address hold in register r1
- Offset = r2, LSL #2
- Target address = $r1 + r2 * 4$

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

Example

```
LDM r4, {r0, r1, r2, r3}
```

- Here, it takes a base register (in this case, r4) and a register set (in this case, {r0, r1, r2, r3}). It loads consecutive words from the address in the base register into the registers in the set. In this example, the effect could be described using the following C-like pseudo-code:
- `r0 = r4[0]; r1 = r4[1]; r2 = r4[2]; r3 = r4[3];`
- The set notation also allows for ranges. We can rewrite the previous example as follows:

```
LDM r4, {r0-r3}
```

```
LDM r4!, {r0-r3}
```

- Fast transfer of 8 words of data

```
ldm r0, {r4-r11}
```

```
stm r1, {r4-r11}
```

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>

LDR Pseudo-instruction

LDR Rt, =expr

LDR Rt, =label

- If the value of expr can be loaded with **MOV**, **MVN** (16-bit instruction) or **MOVW** (32-bit instruction), the assembler uses that instruction.
- If a valid MOV, MVN, MOVW instruction cannot be used, or if the label_expr syntax is used, the assembler places the constant in a literal pool and generates a **PC-relative LDR** instruction that reads the constant from the literal pool.

```
LDR r1,=0xFF ; loads 0xFF0 into R1
               ; => MOV r1,#0xFF
LDR r2,=0xFF ; loads 0xFFF into R2
               ; => MOVW r2, #0xFF
LDR r3,=array ; loads the address of array into R3
               ; => LDR r3,[pc, offset_to_litpool]
               ;
               ; ...
               ; litpool DCD array
```

Software uses this pseudo instruction to set a register to some value without worrying about the size of the value.

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
 - PUSH {r1, r2, LR}
 - Always pushes registers in same order
- 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)
 - POP {r5, r6, r7}
 - Always pops registers in same order (opposite of pushing)

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 value to 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

ADR R3, MyMessage

HERE B HERE

MyMessage DCB "Hello"

Adding Two Integers

```
int x = 1;  
int y = 2;  
int z;
```

C Statement

```
z = x + y;
```

If values are in registers

- ▶ Value of *x* in *r0*
- ▶ Value of *y* in *r1*
- ▶ Value of *z* in *r2*

Assembly Statement

?

Adding Two Integers

```
int x = 1;  
int y = 2;  
int z;
```

C Statement

```
z = x + y;
```

If values are in registers

- ▶ Value of *x* in *r0*
- ▶ Value of *y* in *r1*
- ▶ Value of *z* in *r2*

Assembly Statement

```
ADD r2, r1, r0
```

Destination

Source Operand 2

Source Operand 1

ARM

Adding Two Integers

```
int x = 1;  
int y = 2;  
int z;
```

C Statement

```
z = x + y;
```

If addresses are in registers

- ▶ Address of x in r0
- ▶ Address of y in r1
- ▶ Address of z in r2

```
LDR r3, [r0] ; Read x  
LDR r4, [r1] ; Read y  
ADD r5, r3, r4  
STR r5, [r2] ; Write z
```


Subtract

- Subtract immediate from register, update condition flags
 - SUBS <Rd>,<Rn>,#<imm3> Ex. **SUBS R0, R1, #0x01**
 - SUBS <Rdn>,#<imm8> Ex. **SUBS R0, #0x01**
- 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>
 - $\text{<Rdm>} = \text{<Rdm>} * \text{<Rn>}$
- Signed multiply
- Note: upper word of result is truncated

Example Arithmetic Instructions

- **ADD** `r0, r1, r2 ; r0 = r1 + r2`
- **ADC** `r0, r1, r2 ; Add with carry, r0 = r1 + r2 + carry`
- **SUB** `r0, r1, r2 ; r0 = r1 - r2`
- **SBC** `r0, r1, r2 ; Subtract with borrow, r0 = r1 - r2 - (1 - carry)`
- **MUL** `r0, r1, r2 ; r0 = r1 * r2, product limited to 32 bits`
- **UDIV** `r0, r1, r2 ; Unsigned divide, r0 = r1 / r2`
- **SDIV** `r0, r1, r2 ; Signed divide, r0 = r1 / r2`
- **SMULL** `r0, r1, r2, r3 ; Signed multiply (64-bit product), r1:r0 = r2 * r3`
- **UMULL** `r0, r1, r2, r3 ; Unsigned multiply (64-bit product), r1:r0 = r2 * r3`

Compare

- Compare - subtracts second value from first, discards result, updates APSR
 - `CMP <Rn>,#<imm8>`
 - `CMP <Rn>,<Rm>`
- Compare negative - **adds** two values, updates APSR, 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>

Shift and rotate instructions

Logical Shift Left (LSL)



Logical Shift Right (LSR)



Rotate Right Extended (RRX)



Arithmetic Shift Right (ASR)



Rotate Right (ROR)



Why is there rotate right but no rotate left?

Rotate left can be replaced by a rotate right with a different rotate offset.

Logical Operations

- Bitwise AND registers, update condition flags
 - ANDS <Rdn>,<Rm>
- Bitwise OR registers, update condition flags
 - ORRS <Rdn>,<Rm>
- Bitwise Exclusive OR registers, update condition flags
 - EORS <Rdn>,<Rm>
- Bitwise AND register and complement of second register, update condition flags
 - BICS <Rdn>,<Rm>
- Move inverse of register value to destination, update condition flags
 - MVNS <Rd>,<Rm>
- Update condition flags by ANDing two registers, discarding result
 - TST <Rn>, <Rm>

Set a Bit in C

$$a \mid= (1 \ll k)$$

or

$$a = a \mid (1 \ll k)$$

Example: $k = 5$

a	a_7	a_6	a_5	a_4	a_3	a_2	a_1	a_0
$1 \ll k$	0	0	1	0	0	0	0	0
$a \mid (1 \ll k)$	a_7	a_6	1	a_4	a_3	a_2	a_1	a_0

The other bits should not be affected.

Set a Bit in Assembly

$a \mid= (1 \ll 5)$

Solution 1:

```
MOVS r4, #1      ; r4 = 1
LSLS r4, r4, #5   ; r4 = 1<<5
ORRS r0, r0, r4  ; r0 = r0 | 1<<5
```

Solution 2:

```
MOVS r4, #1      ; r4 = 1
ORRS r0, r0, r4, LSL #5 ; r0 = r0 | 1<<5
```

Clear a Bit in C

$$a \&= \sim(1 \ll k)$$

Example: $k = 5$

a	a_7	a_6	a_5	a_4	a_3	a_2	a_1	a_0
$\sim(1 \ll k)$	1	1	0	1	1	1	1	1
$a \& \sim(1 \ll k)$	a_7	a_6	0	a_4	a_3	a_2	a_1	a_0

The other bits should not be affected.

Clear a Bit in Assembly

a &= ~(1<<5)

Solution 1:

```
MOVS r4, #1          ; r4 = 1
LSLS r4, r4, #5       ; r4 = 1<<5
MVNS r4, r4           ; r4 = not (1<<5)
ANDS r0, r0, r4      ; r0 = r0 & not (1<<5)
```

Solution 2:

```
MOVS r4, #1          ; r4 = 1
MVNS r4, r4, LSL #5   ; r4 = not (1<<5)
ANDS r0, r0, r4      ; r0 = r0 & not (1<<5)
```

Solution 3:

```
MOVS r4, #1          ; r4 = 1
BICS r0, r0, r4, LSL #5
```

Toggle a Bit in C

Without knowing the initial value, a bit can be toggled by XORing it with a “1”

$$a \oplus = 1 \ll k$$

Example: $k = 5$

a	a_7	a_6	a_5	a_4	a_3	a_2	a_1	a_0
$1 \ll k$	0	0	1	0	0	0	0	0
$a \oplus (1 \ll k)$	a_7	a_6	NOT(a_5)	a_4	a_3	a_2	a_1	a_0

Toggle a Bit in Assembly

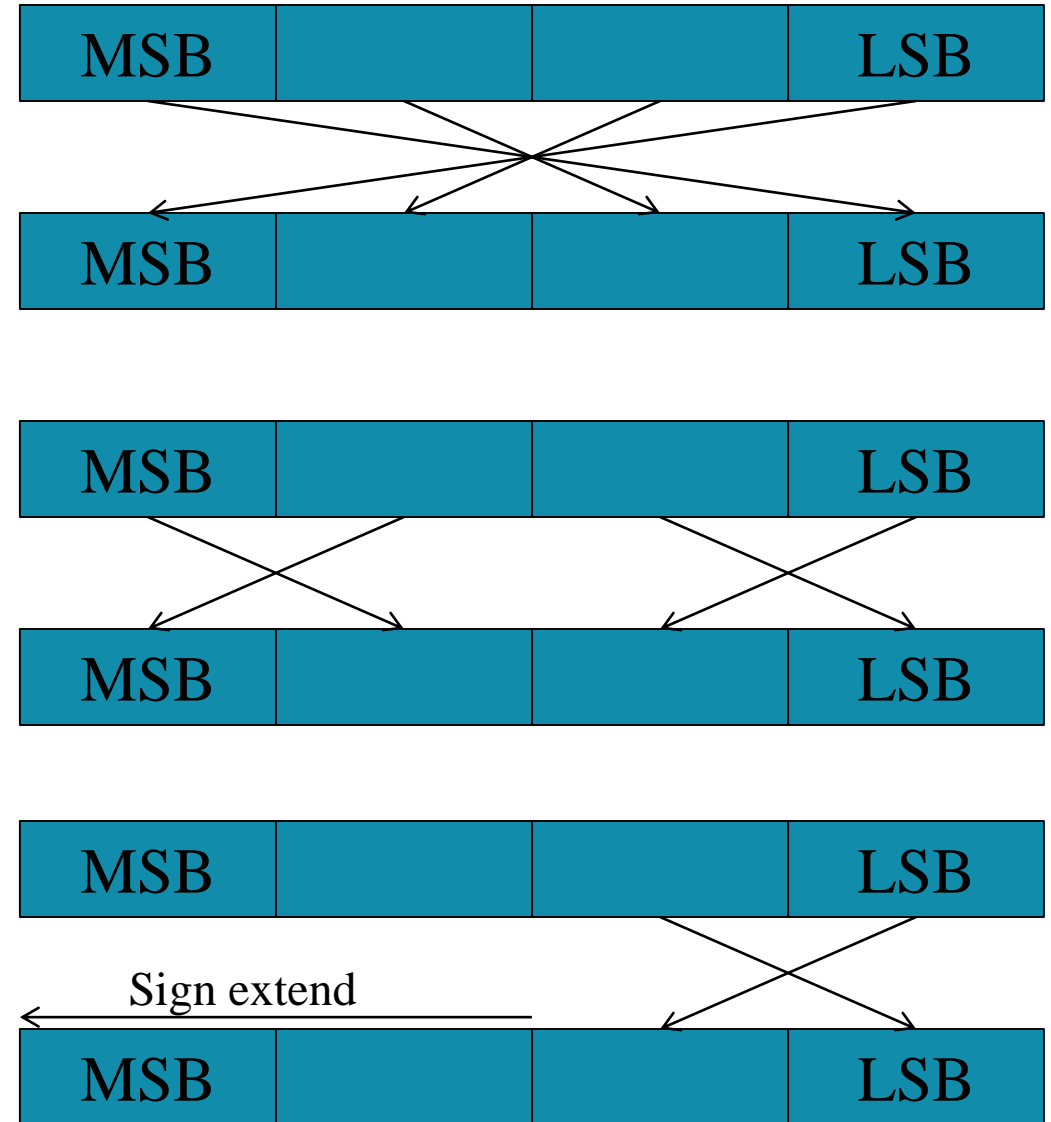
$a \oplus= 1 \ll 5$

Solution:

```
MOVS r4, #1           ; r4 = 1
EORS r0, r0, r4, LSL #5 ; r0 = r0 ^ 1<<5
```

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>



Changing Program Flow - Branches

- Unconditional Branches
 - B <label>
 - Target address must be within 2 KB of branch instruction (-2048 B to +2046 B)
- 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)

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

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	$N \neq V$
GT	Signed greater than	$Z = 0$ and $N = V$
LE	Signed less than or equal	$Z = 1$ or $N \neq V$
None (AL) ^d	Always (unconditional)	Any

Conditional Execution for ADD (not with Thumb)

Add instruction	Condition	Flag tested
ADDEQ r3, r2, r1	Add if EQual	Add if Z = 1
ADDNE r3, r2, r1	Add if Not Equal	Add if Z = 0
ADDHS r3, r2, r1	Add if Unsigned Higher or Same	Add if C = 1
ADDLO r3, r2, r1	Add if Unsigned LOwer	Add if C = 0
ADDMI r3, r2, r1	Add if Minus (Negative)	Add if N = 1
ADDPL r3, r2, r1	Add if PLus (Positive or Zero)	Add if N = 0
ADDVS r3, r2, r1	Add if oVerflow Set	Add if V = 1
ADDVC r3, r2, r1	Add if oVerflow Clear	Add if V = 0
ADDHI r3, r2, r1	Add if Unsigned HIgher	Add if C = 1 & Z = 0
ADDLS r3, r2, r1	Add if Unsigned Lower or Same	Add if C = 0 or Z = 1
ADDGE r3, r2, r1	Add if Signed Greater or Equal	Add if N = V
ADDLT r3, r2, r1	Add if Signed Less Than	Add if N != V
ADDGT r3, r2, r1	Add if Signed Greater Than	Add if Z = 0 & N = V
ADDLE r3, r2, r1	Add if Signed Less than or Equal	Add if Z = 1 or N = !V

Example – 64 Bit Addition

start

; C = A + B

; Two 64-bit integers A (r1,r0) and B (r3, r2).

; Result C (r5, r4)

; A = 00000002FFFFFFFF

; B = 0000000400000001

LDR r0, =0xFFFFFFFF ; A's lower 32 bits

LDR r1, =0x00000002 ; A's upper 32 bits

LDR r2, =0x00000001 ; B's lower 32 bits

LDR r3, =0x00000004 ; B's upper 32 bits

; Add A and B

ADD r4, r2, r0 ; C[31..0] = A[31..0] + B[31..0], update Carry

ADC r5, r3, r1 ; C[64..32] = A[64..32] + B[64..32] + Carry

stop B stop

Example – Block copy

The length of the data array is in memory location 0x20000000, the data originally starts in memory location 0x20004000 (Source Address), and the destination area for the data starts in memory location 0x20008000 (Destination Address). Write a program to transfer the data.

Label	Mnemonic	Comments
	LDR R0, =0x20000000	
	LDR R1, [R0]	; R1 holds the array length/counter
	LDR R2, =0x20004000	; R2 is a pointer to source area
	LDR R3, =0x20008000	; R3 is a pointer to destination area
Loop	LDR R4, [R2], #4	; Load the data and increment the source ptr
	STR R4, [R3], #4	; Store the data and increment destination ptr
	SUBS R1, #1	; Decrement length counter
	BNE Loop	; If there are more data to read go to Loop
End	B End	


Example – Finding Maximum

Length ($\neq 0$) of a memory array, which contains unsigned numbers, is in 0x20000044 and the array starts at 0x00000048. Write a program that places the maximum value in the array in 0x20000040.

label	mnemonic	comment
	LDR R0, =0x20000044	; R0 is the pointer to the length
	LDR R4, =0x20000040	; R4 is the pointer to the max storage
	LDR R1, [R0], #4	; R1 holds the length/count information
	LDR R2, [R0]	; R2 holds the maximum
	SUBS R1, #1	; Decrement counter
	BEQ Final	; If it is the end of the array, finish
Loop	LDR R3, [R0, #4]!	; R3 holds the next data
	CMP R2, R3	; R2 - R3
	BGE Cont	; If R2>R3, go to Cont
	LDR R2, [R0]	; Else load the new data to max (R2)
Cont	SUBS R1, #1	; Decrement counter
	BEQ Final	; If it is the end of the array, finish
	B Loop	; Else go to Loop
Final	STR R2, [R4]	; Store max
Forever	B Forever	

Conditional Execution Example

```
if (a <= 0)
    y = -1;
else
    y = 1;
```



a → r0
y → r1

```
CMP    r0, #0
MOVLE r1, #-1 ; executed if LE
MOVGT r1, #1  ; executed if GT
```

LE: Signed Less than or Equal

GT: Signed Greater Than

Conditional Execution Example

```
if (a==1 || a==7 || a==11)
    y = 1;
else
    y = -1;
```



a → r0
y → r1

```
CMP    r0, #1
CMPNE r0, #7 ; executed if r0 != 1
CMPNE r0, #11 ; executed if r0 != 7
MOVEQ r1, #1
MOVNE r1, #-1
```

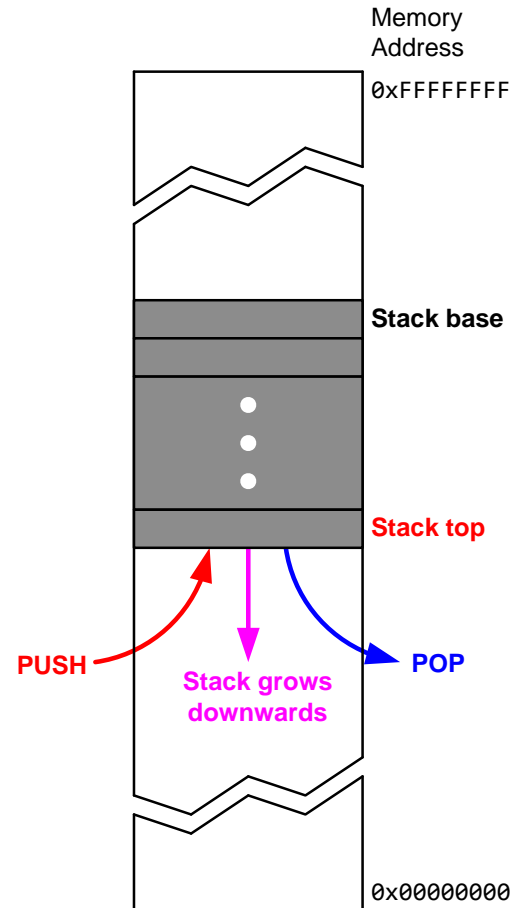
~~NE~~: Not Equal

~~EQ~~: Equal

Stack

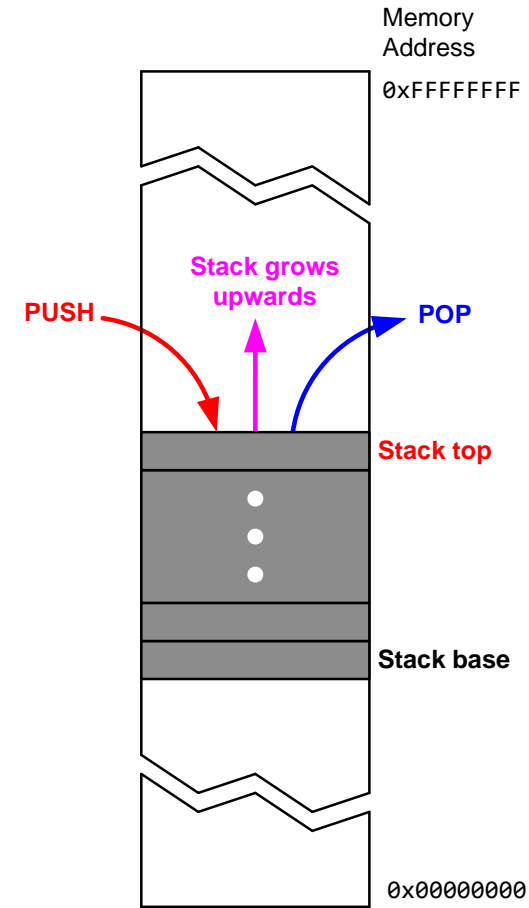
- The stack is a data structure, known as last in first out (LIFO). In a stack, items entered at one end and leave in the reversed order. Stack can be used for
 - Storing original data in registers in subroutine so that the values can be restored at the end of the subroutine
 - Passing information to subroutine
 - Storing local variables
 - Holding processor status and registers when an exception occurs

Stack Growth Convention: Ascending vs Descending



Descending stack:

When items are pushed on to the stack, the stack pointer is decreasing. Stack grows towards low memory address

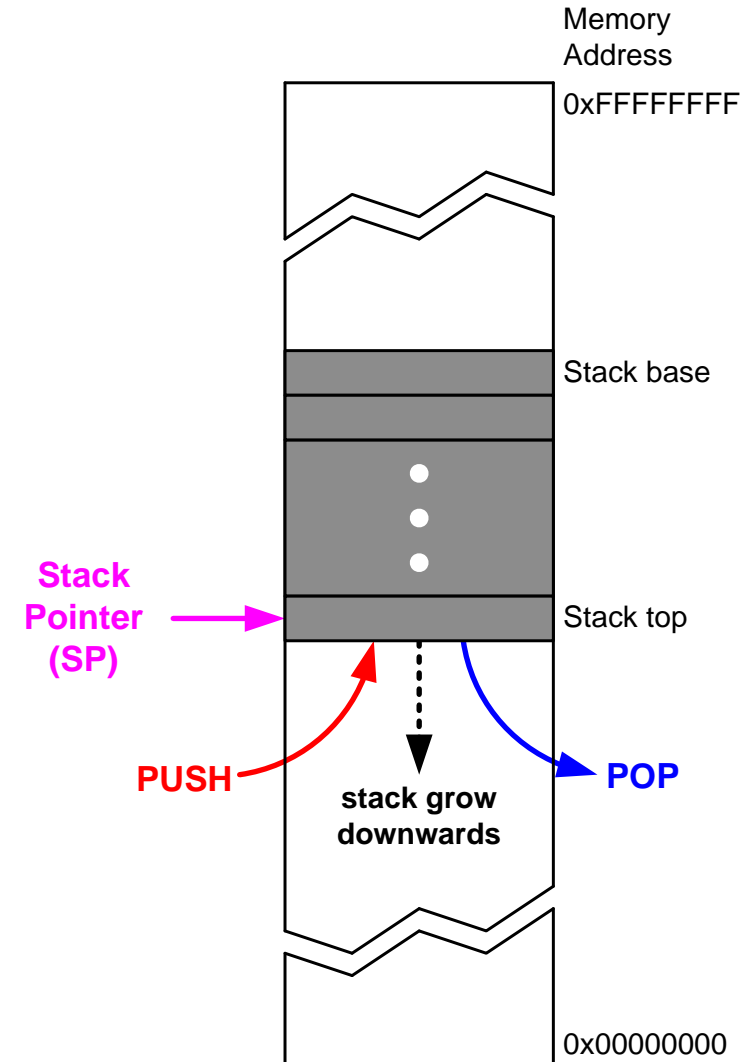


Ascending stack:

When items are pushed on to the stack, the stack pointer is increasing. Stack grows towards high memory address

Cortex-M Stack

- Cortex-M0 uses full descending stack!
- stack pointer (SP) = R13
 - MSP
 - PSP
- stack pointer
 - decremented on **PUSH**
 - incremented on **POP**
 - SP starts at SP = 0x1FFFF100 for





Stack

PUSH {*Rd*}

- $SP = SP - 4 \rightarrow$ descending stack
- $(*SP) = Rd \rightarrow$ full stack

Push multiple registers

They are equivalent.

PUSH {r6, r7, r8}  **PUSH {r8, r7, r6}**  **PUSH {r8}**
PUSH {r7}
PUSH {r6}

- The order in which registers listed in the register list does not matter.
- When pushing multiple registers, these registers are automatically **sorted by name** and **the lowest-numbered register** is stored to the lowest memory address, *i.e.* **is stored last**.

Stack

POP {Rd}

- $Rd = (*SP) \rightarrow$ full stack
- $SP = SP + 4 \rightarrow$ Stack shrinks

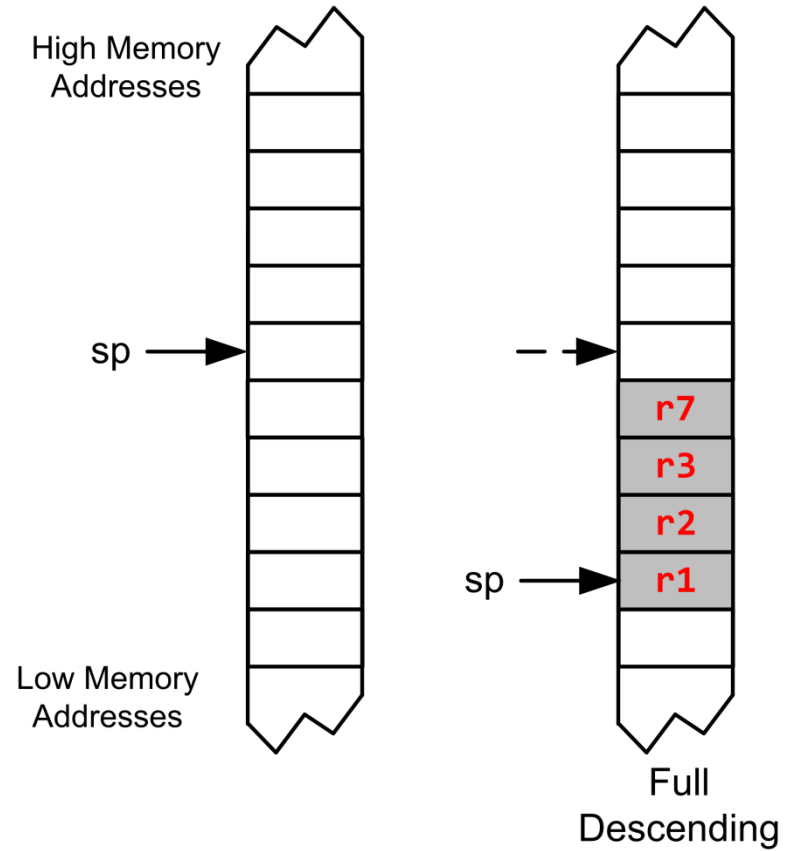
Push multiple registers

**POP {r6, r7, r8} \longleftrightarrow POP {r8, r7, r6} \longleftrightarrow POP {r6}
POP {r7}
POP {r8}**

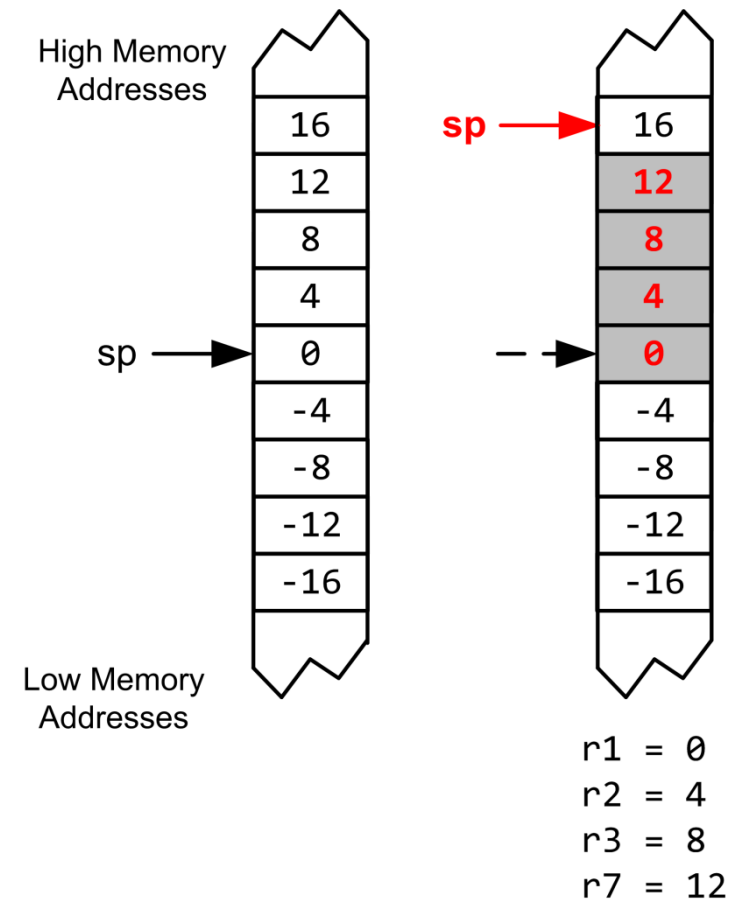
- The order in which registers listed in the register list does not matter.
- When popping multiple registers, these registers are automatically **sorted by name** and **the lowest-numbered register** is loaded from the lowest memory address, *i.e.* **is loaded first**.

Stack

PUSH {r3, r1, r7, r2}



POP {r3, r1, r7, r2}



Changing Program Flow - Subroutines

■ Call

- BL <label> - branch with link
 - Call subroutine at <label>
 - PC-relative, range limited to PC+/-16MB
- Save return address in LR

■ Return

- BX LR - Return from subroutine

Subroutines

- Code sequences that can be called from the main program to perform specific tasks.
- To call a subroutine, the Branch and Link (BL) instruction or Branch and Link with eXchange (BX) instruction can be used. These instructions will save the return address to the Link Register (LR or R14) and execute the unconditional branch to the subroutine label.
- A subroutine name is identified by the label at the first instruction in the subroutine. The last instruction should be the BX LR instruction which retrieves the return address from LR and return to the instruction immediately after the instruction that performed the subroutine call.

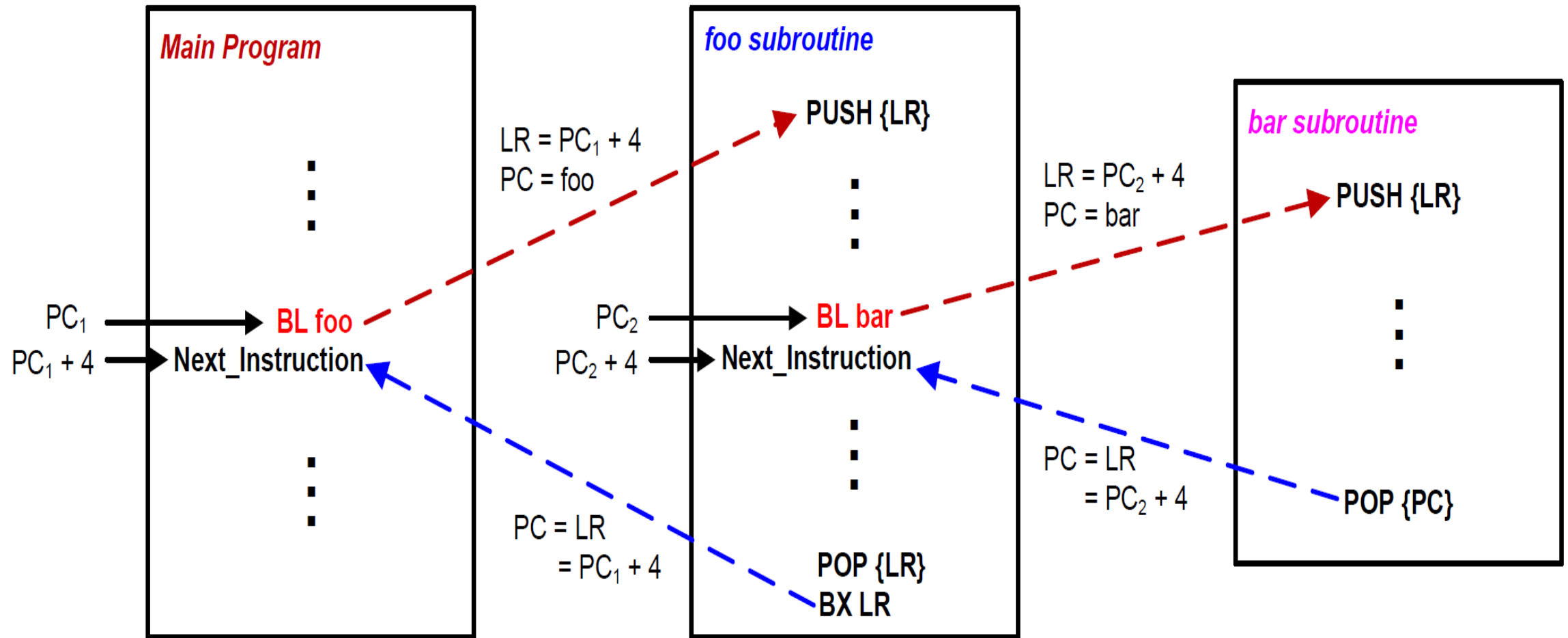
Call a Subroutine

Caller Program	Subroutine/Callee
<pre>MOV r4, #100 ... BL foo ... ADD r4, r4, #1 ;</pre>	<pre>foo PROC ... MOV r4, #10 ; foo changes r4 ... BX LR ENDP</pre>

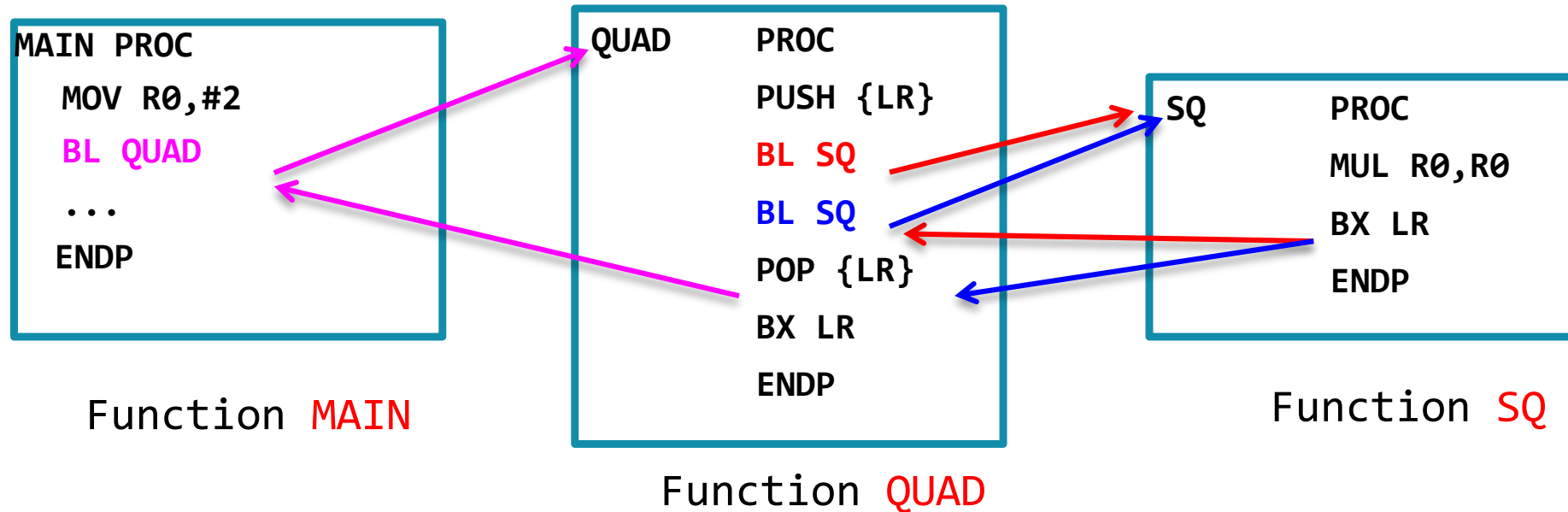
Preserve Runtime Environment via Stack

Caller Program	Subroutine/Callee
<pre>MOV r4, #100 ... BL foo ... ADD r4, r4, #1 ; r4 = 101, not 11</pre>	<pre>foo PROC PUSH {r4} ; preserve r4 ... MOV r4, #10 ; foo changes r4 ... POP {r4} ; Recover r4 BX LR ENDP</pre>

Stacks and Subroutines



Subroutine Calling Another Subroutine



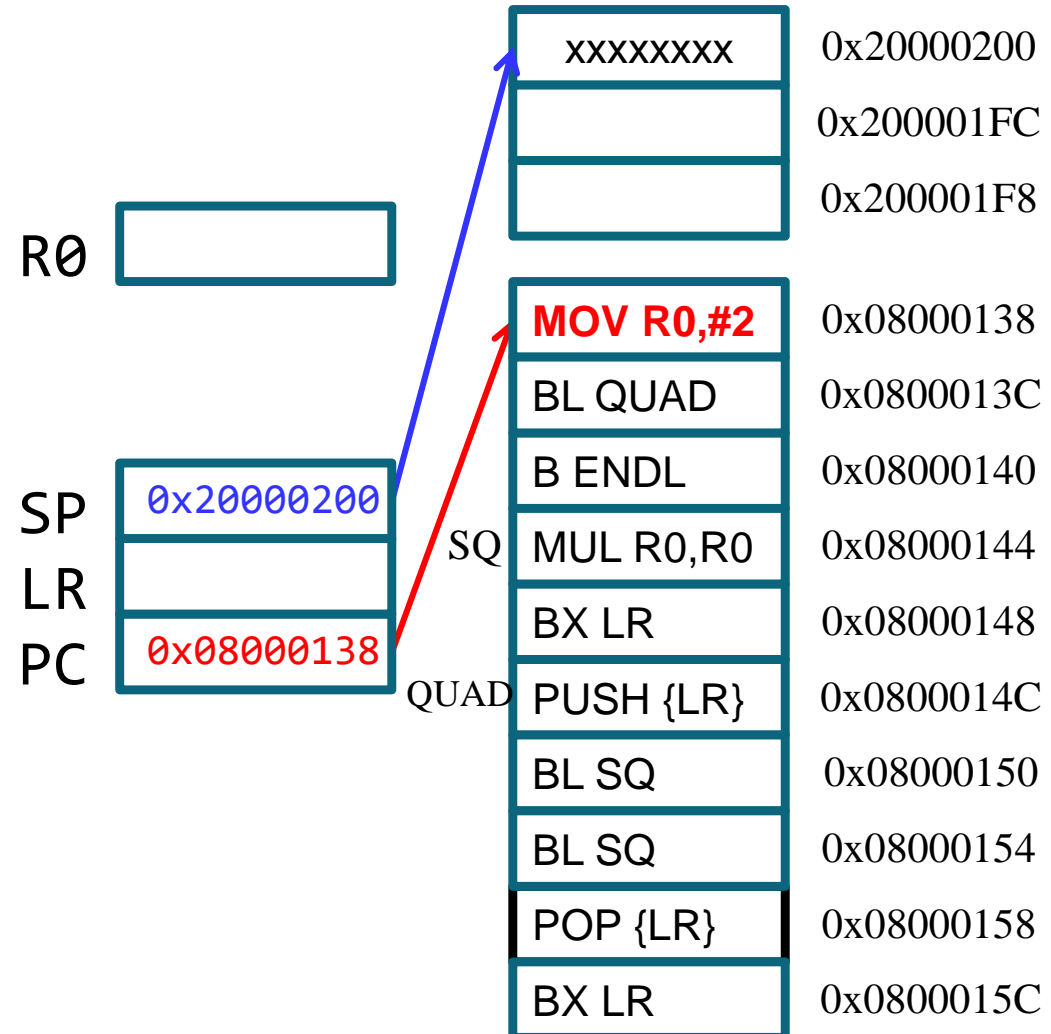
Example: $R0 = R0^4$

```
MOV R0, #2
BL QUAD
B ENDL

SQ    MUL R0, R0
      BX LR

QUAD  PUSH {LR}
      BL SQ
      BL SQ
      POP {LR}
      BX LR

ENDL  ...
```



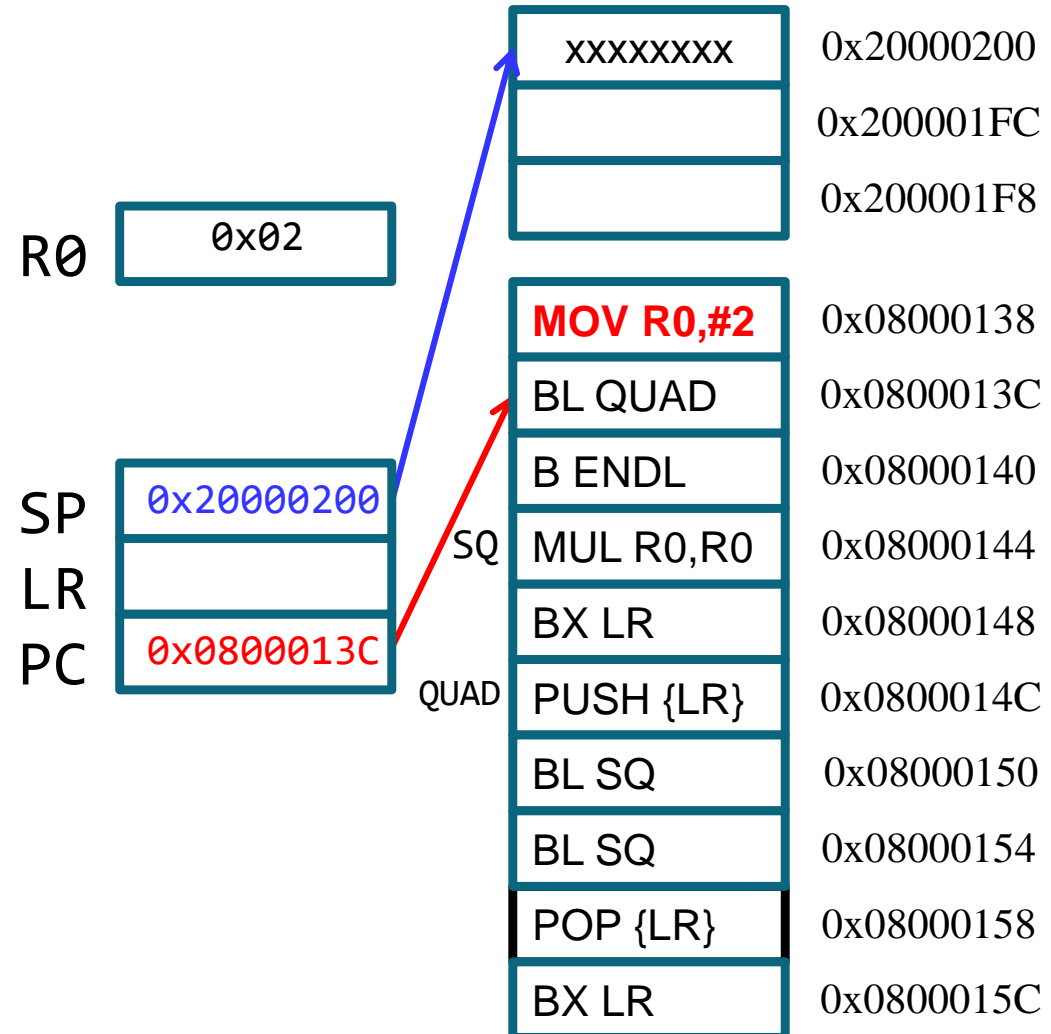
Example: $R0 = R0^4$

```
MOV R0,#2
BL QUAD
B ENDL

SQ    MUL R0,R0
      BX LR

QUAD  PUSH {LR}
      BL SQ
      BL SQ
      POP {LR}
      BX LR

ENDL  ...
```



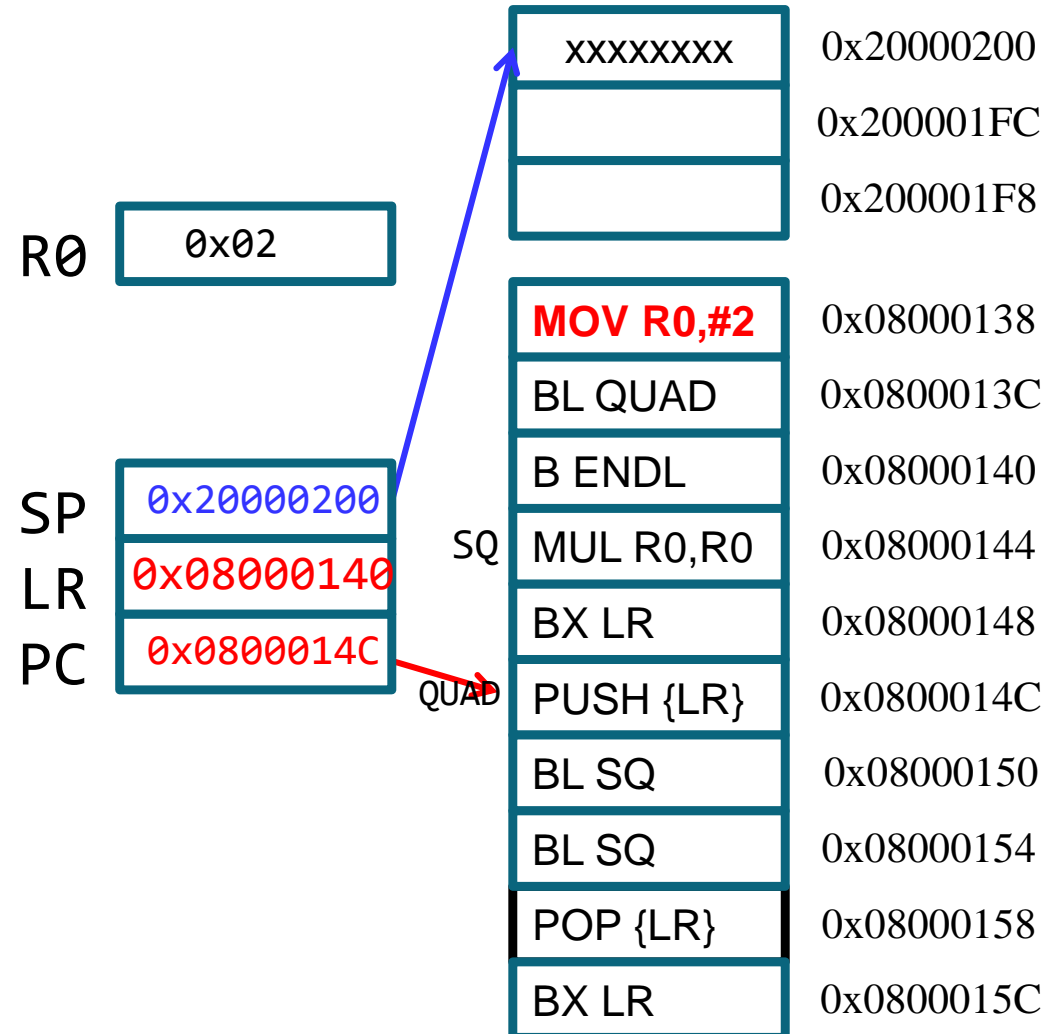
Example: $R0 = R0^4$

```
MOV R0,#2
BL QUAD
B ENDL

SQ    MUL R0,R0
      BX LR

QUAD  PUSH {LR}
      BL SQ
      BL SQ
      POP {LR}
      BX LR

ENDL  ...
```



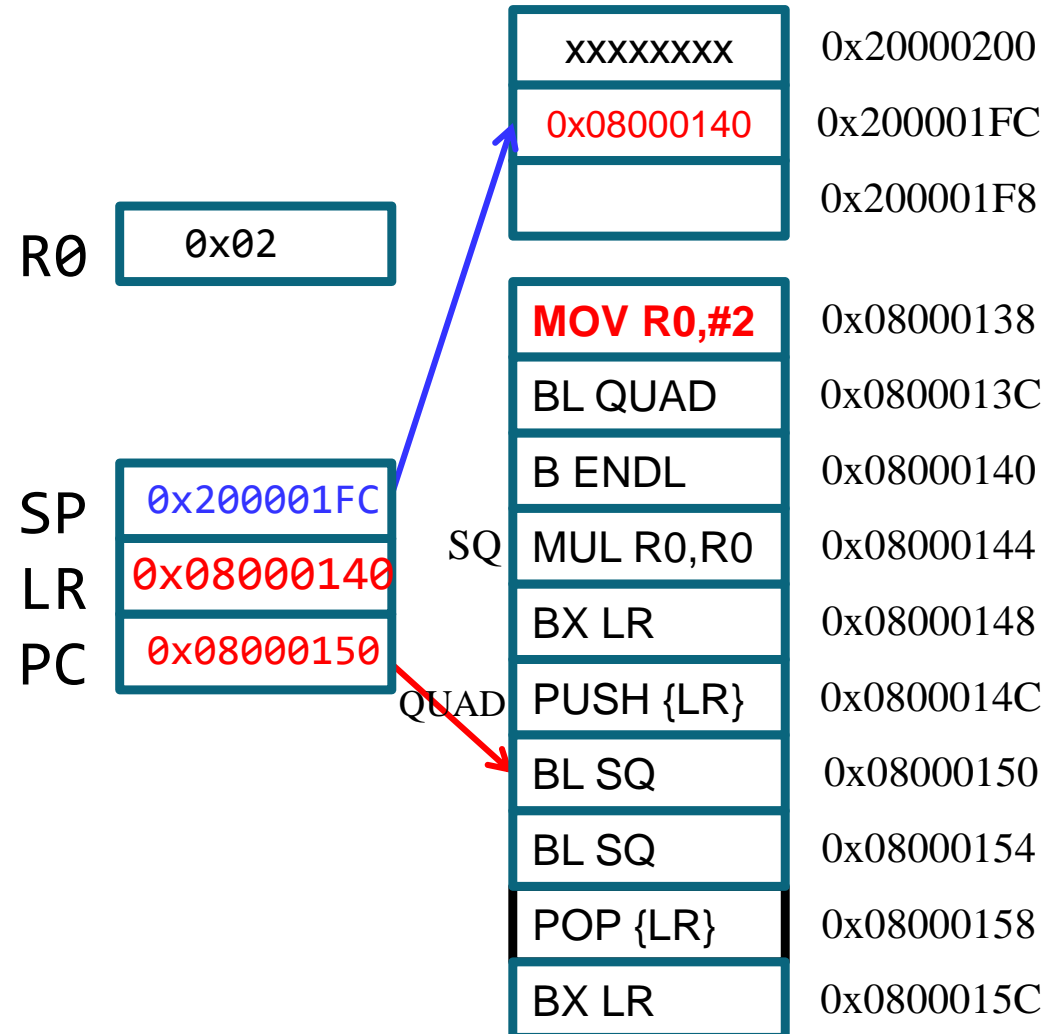
Example: $R0 = R0^4$

```
MOV R0, #2
BL QUAD
B ENDL

SQ    MUL R0, R0
      BX LR

QUAD  PUSH {LR}
      BL SQ
      BL SQ
      POP {LR}
      BX LR

ENDL  ...
```



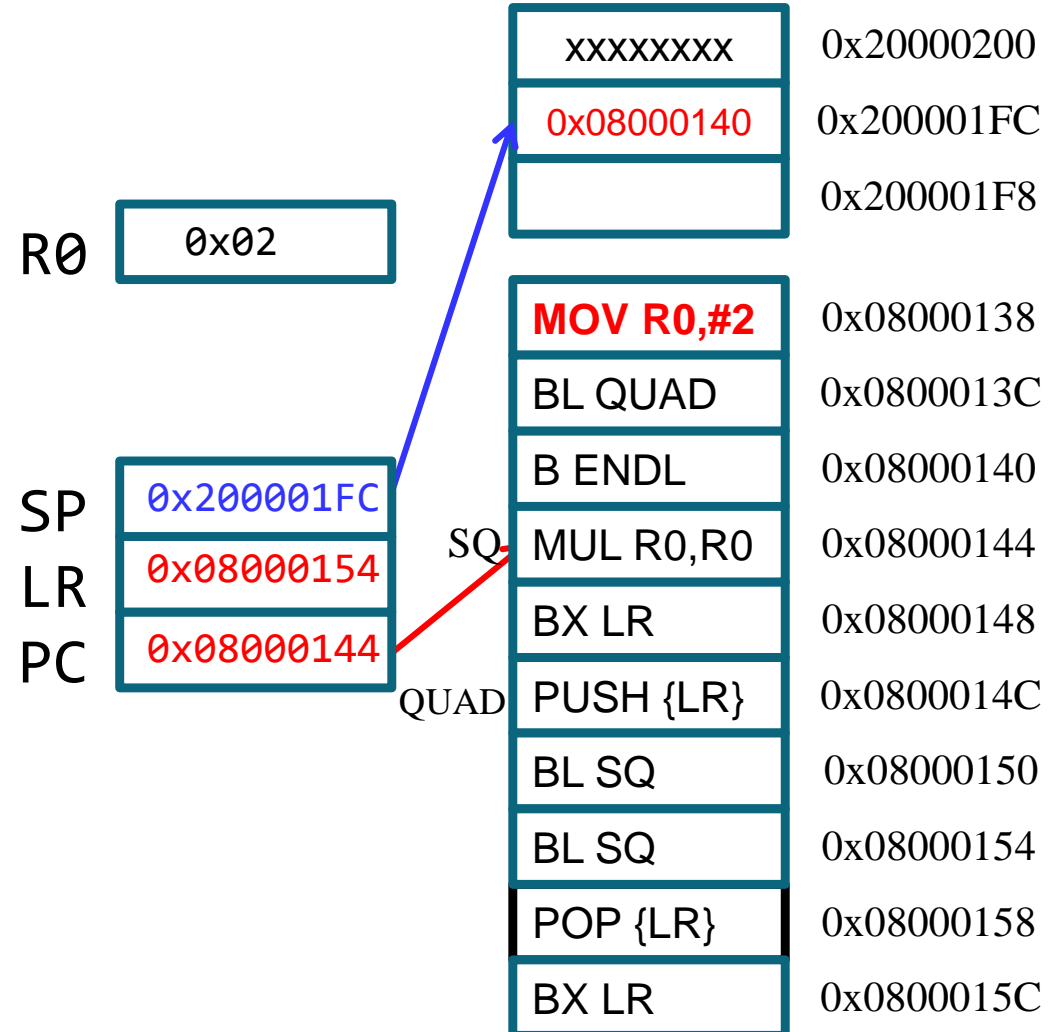
Example: $R0 = R0^4$

```
MOV R0, #2
BL QUAD
B ENDL

SQ    MUL R0, R0
      BX LR

QUAD  PUSH {LR}
      BL SQ
      BL SQ
      POP {LR}
      BX LR

ENDL  ...
```



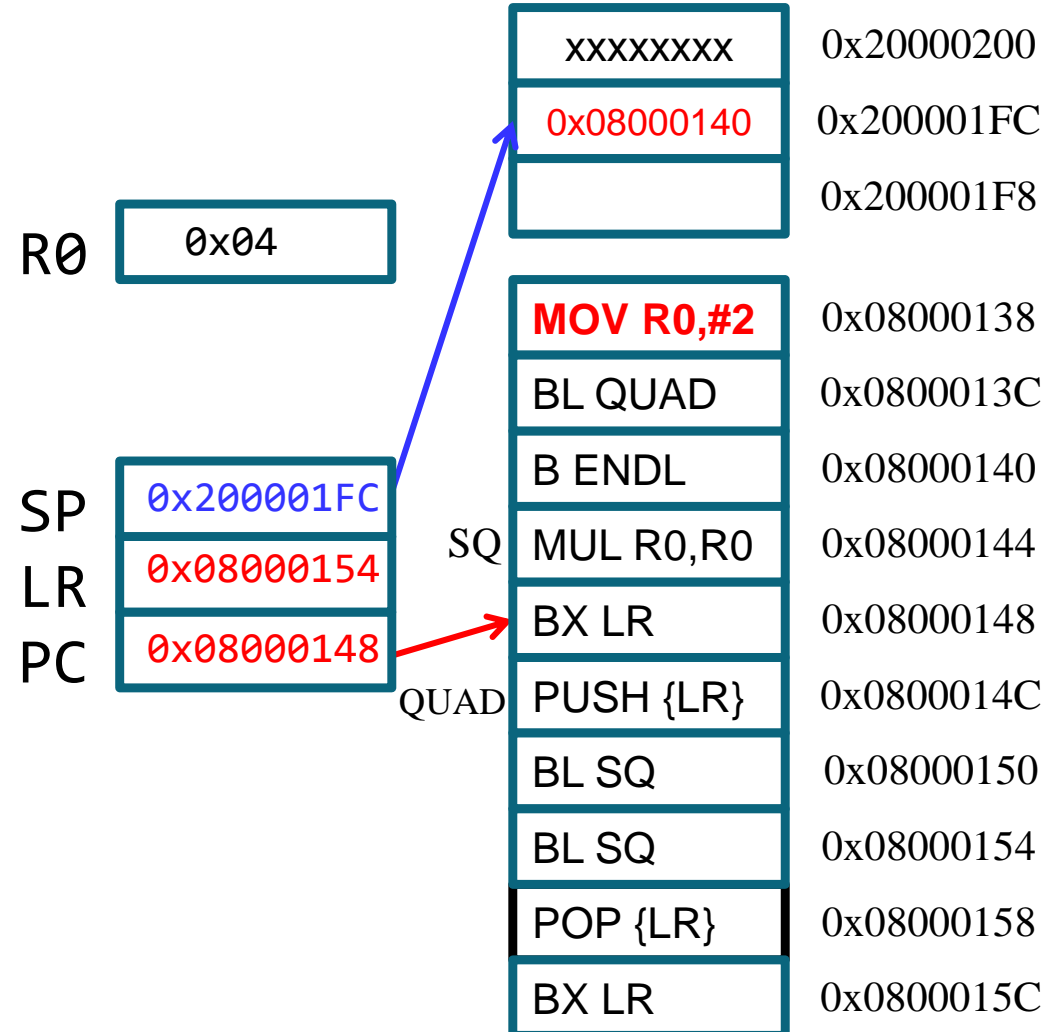
Example: $R0 = R0^4$

```
MOV R0, #2
BL QUAD
B ENDL

SQ    MUL R0, R0
      BX LR

QUAD  PUSH {LR}
      BL SQ
      BL SQ
      POP {LR}
      BX LR

ENDL  ...
```



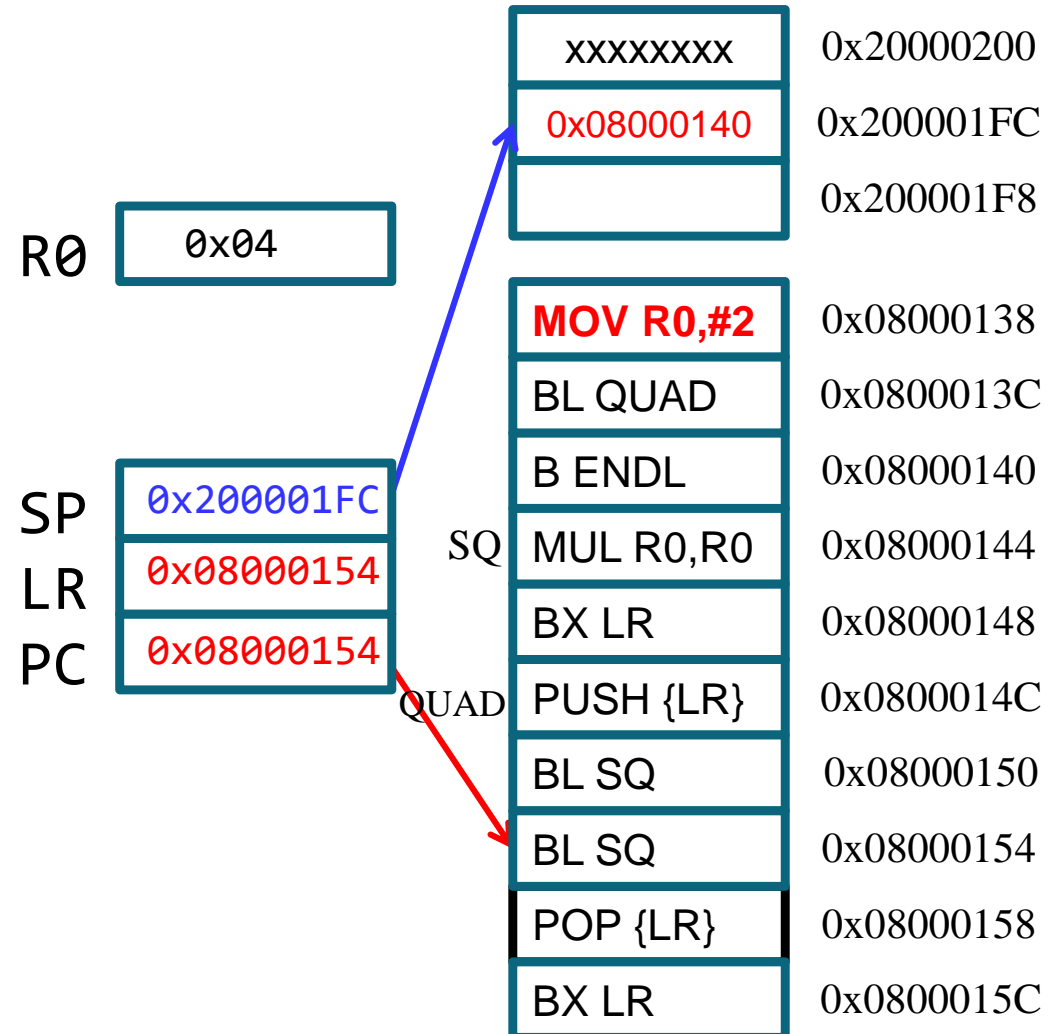
Example: $R0 = R0^4$

```
MOV R0, #2
BL QUAD
B ENDL

SQ    MUL R0, R0
      BX LR

QUAD  PUSH {LR}
      BL SQ
      BL SQ
      POP {LR}
      BX LR

ENDL  ...
```



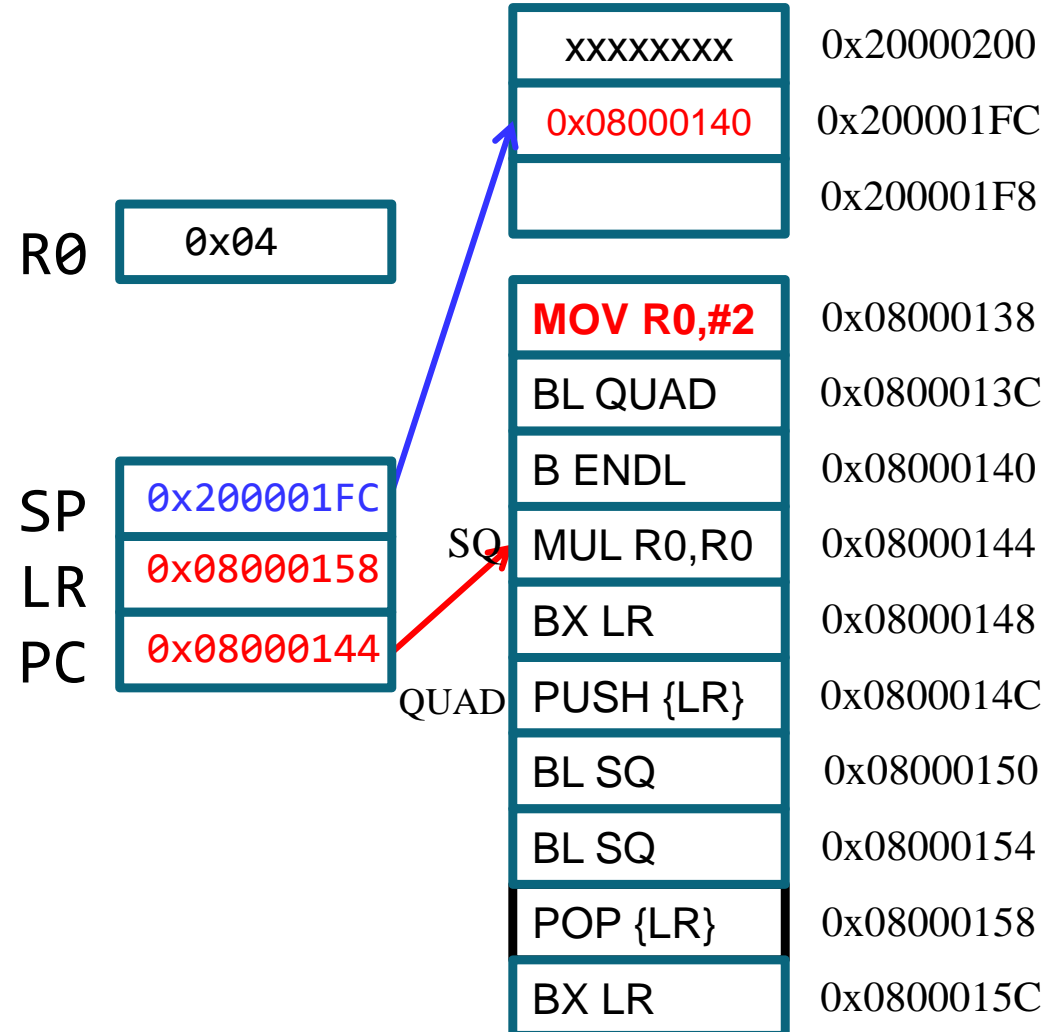
Example: $R0 = R0^4$

```
MOV R0, #2
BL QUAD
B ENDL

SQ    MUL R0, R0
      BX LR

QUAD  PUSH {LR}
      BL SQ
      BL SQ
      POP {LR}
      BX LR

ENDL  ...
```



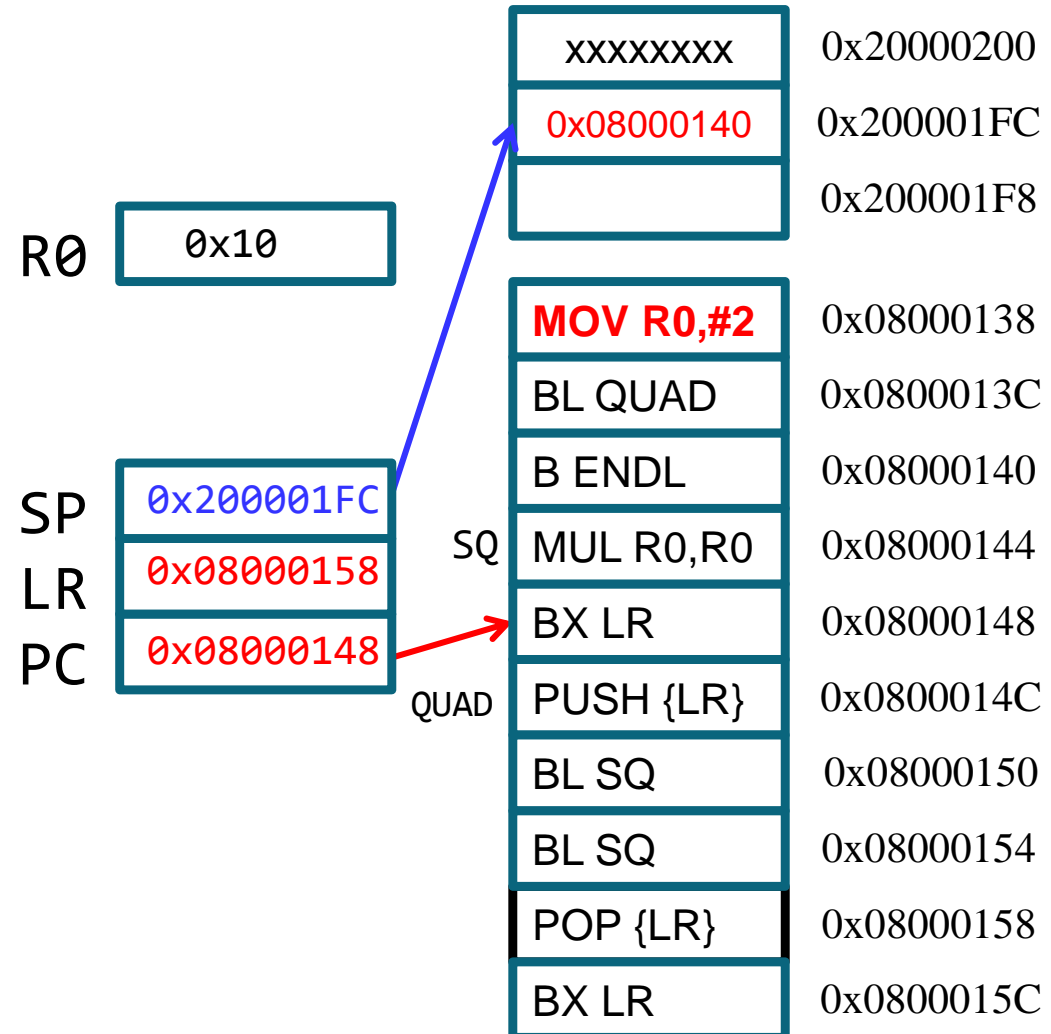
Example: $R0 = R0^4$

```
MOV R0, #2
BL QUAD
B ENDL

SQ    MUL R0, R0
      BX LR

QUAD  PUSH {LR}
      BL SQ
      BL SQ
      POP {LR}
      BX LR

ENDL  ...
```



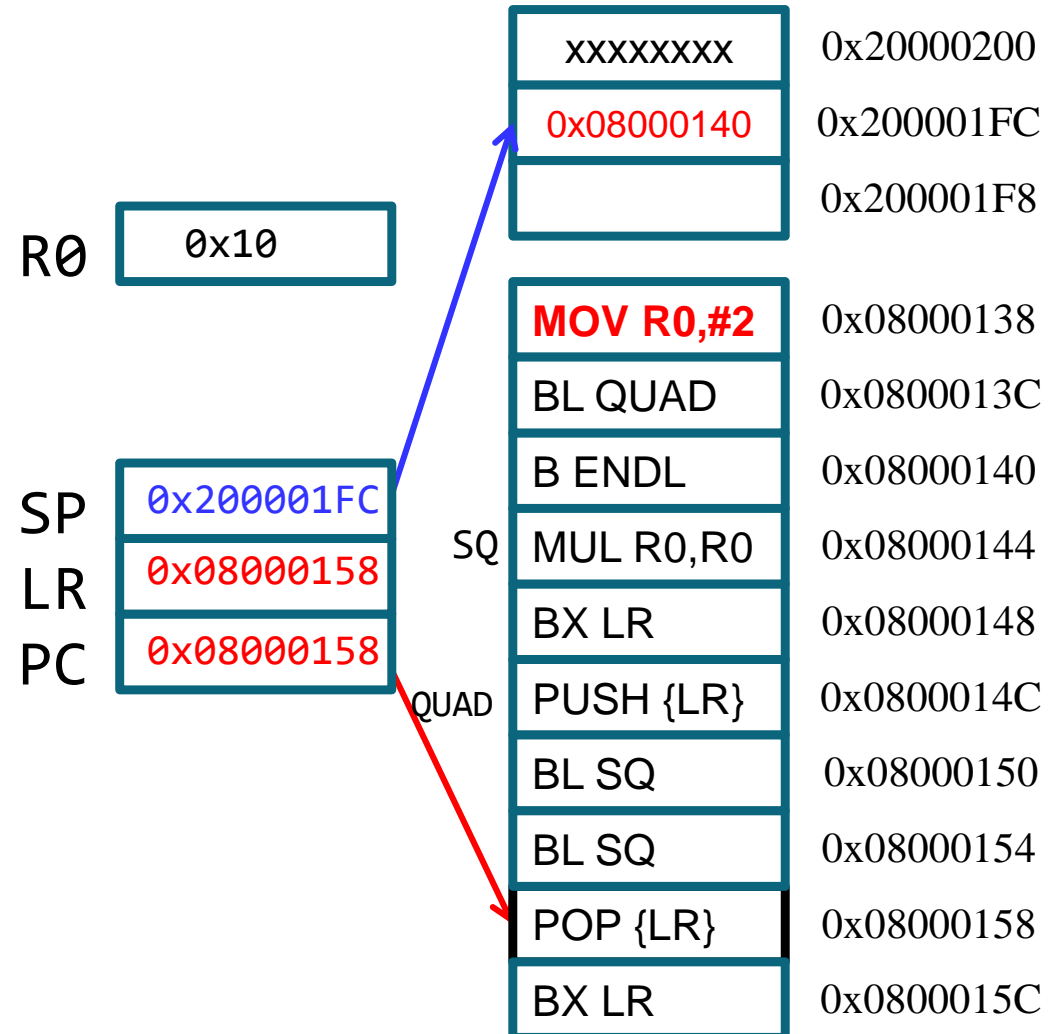
Example: $R0 = R0^4$

```
MOV R0, #2
BL QUAD
B ENDL

SQ    MUL R0, R0
      BX LR

QUAD  PUSH {LR}
      BL SQ
      BL SQ
      POP {LR}
      BX LR

ENDL  ...
```



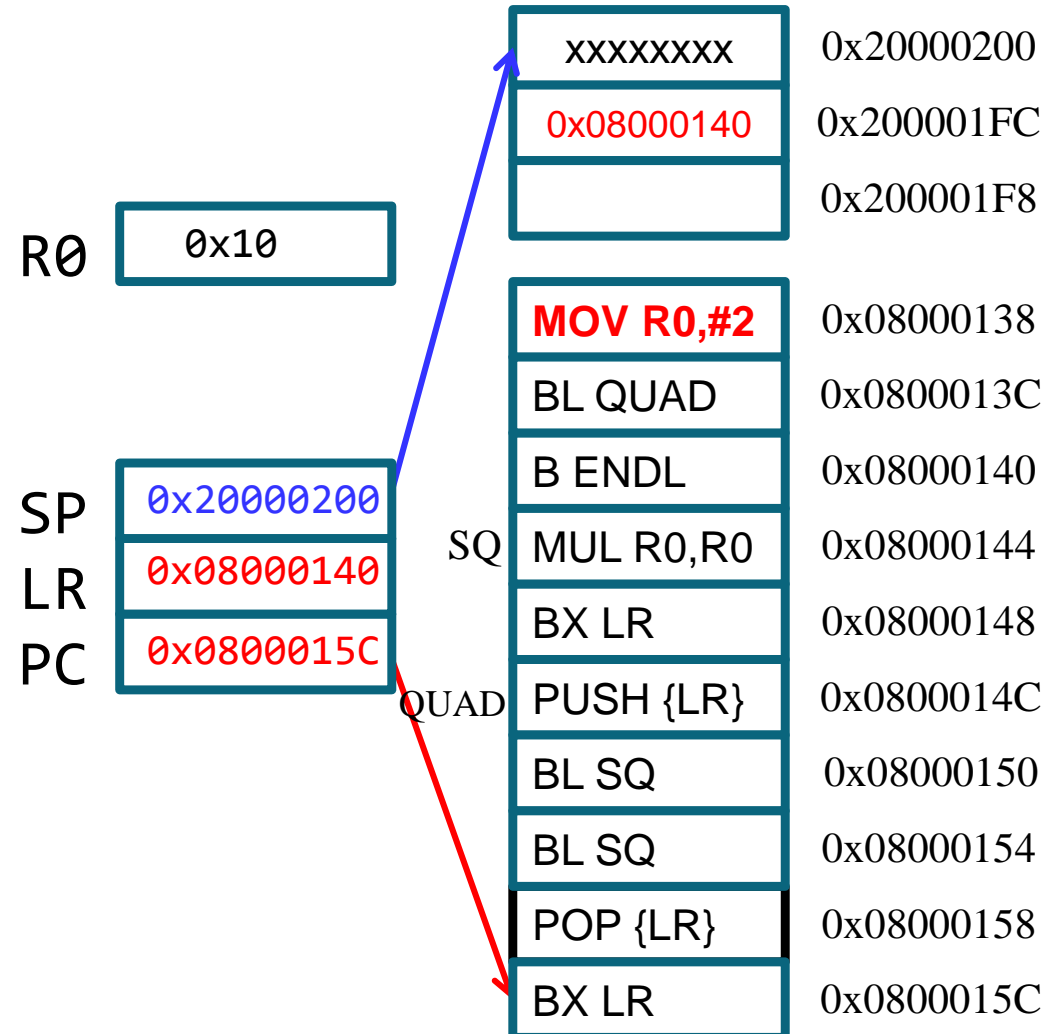
Example: $R0 = R0^4$

```
MOV R0, #2
BL QUAD
B ENDL

SQ    MUL R0, R0
      BX LR

QUAD  PUSH {LR}
      BL SQ
      BL SQ
      POP {LR}
      BX LR

ENDL  ...
```



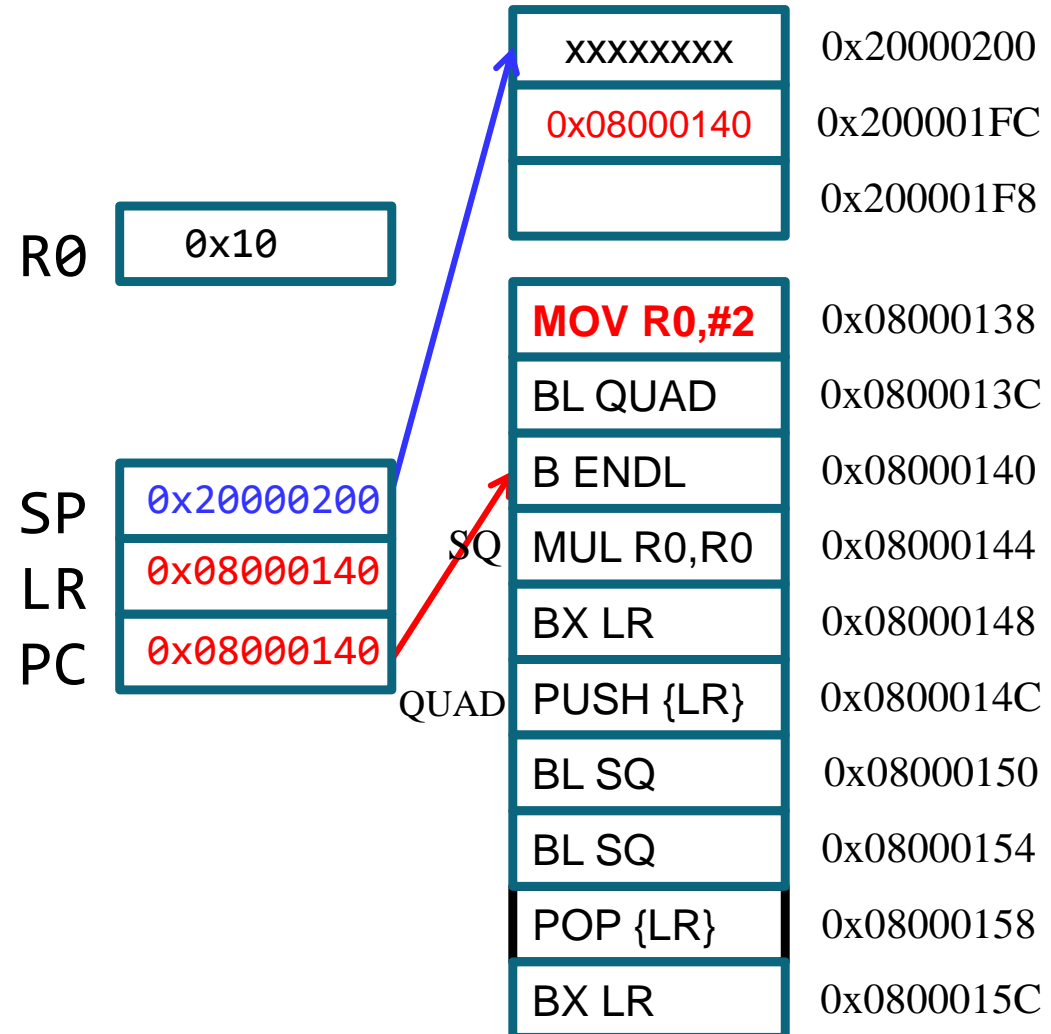
Example: $R0 = R0^4$

```
MOV R0,#2
BL QUAD
B ENDL

SQ    MUL R0,R0
      BX LR

QUAD  PUSH {LR}
      BL SQ
      BL SQ
      POP {LR}
      BX LR

ENDL  ...
```



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 <i>The special-purpose CONTROL register</i> 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>