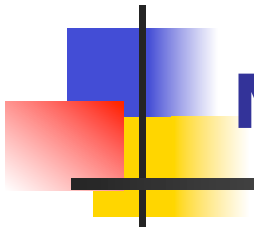


Microprocessor Systems

Fall 2024

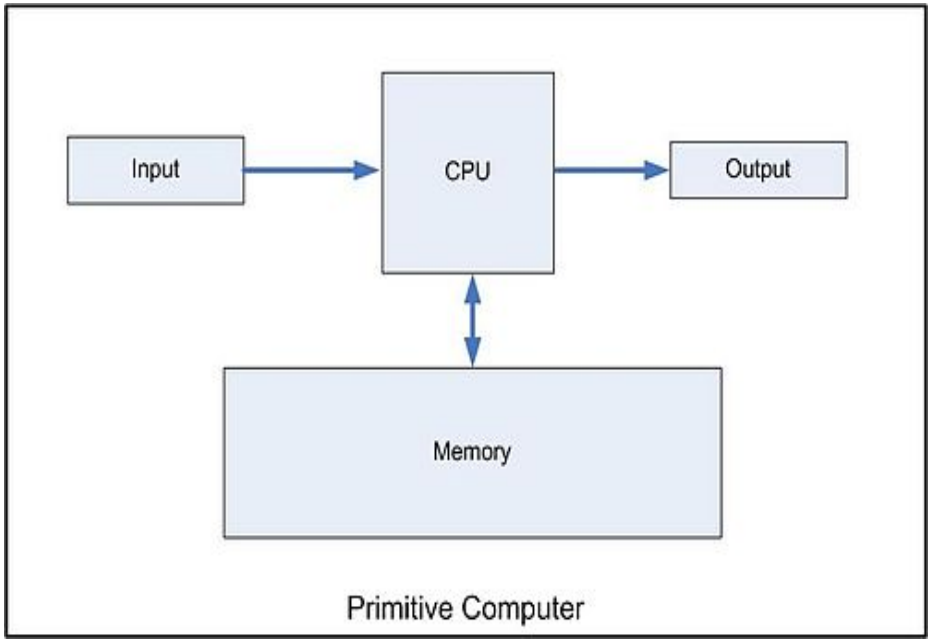
Overview

- CPU overview
- Cortex-M0+ Processor Core
  - Cortex-M0+ Processor Core Registers
  - Memory System and Addressing
  - Thumb Instruction Set

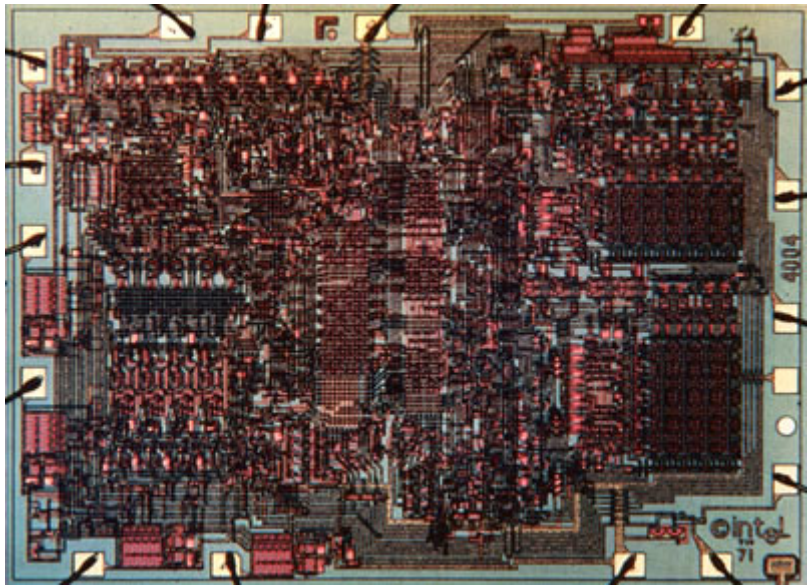
CPU OVERVIEW

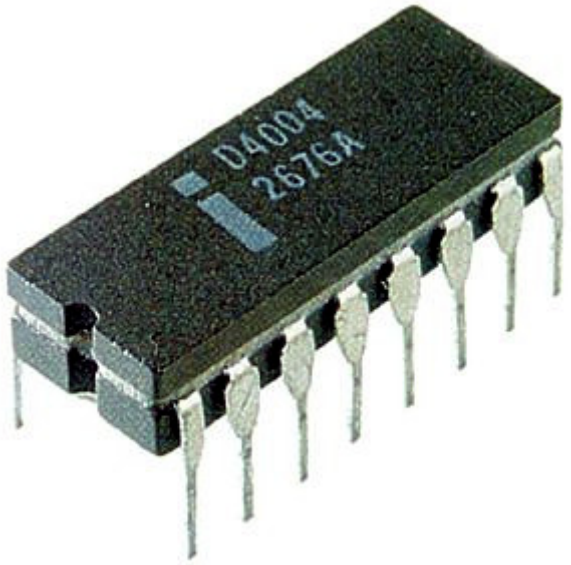
Central Processing Unit (CPU)

- CPU is the fundamental execution/processing unit of the computer
- CPU consists of ALU, Control Unit, and Registers
- CPU is characterized by:
  - Clock frequency
  - Speed
  - Data bus width
  - Instruction set
  - Addressing capability
  - Addressing capacity

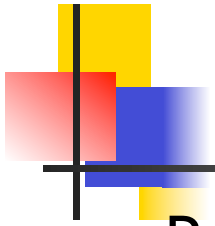


Internal Structure of CPU





- Intel 4004
  - in 1971, commercially available single-chip microprocessor
  - 12 bit address bus
  - 4 bit data bus



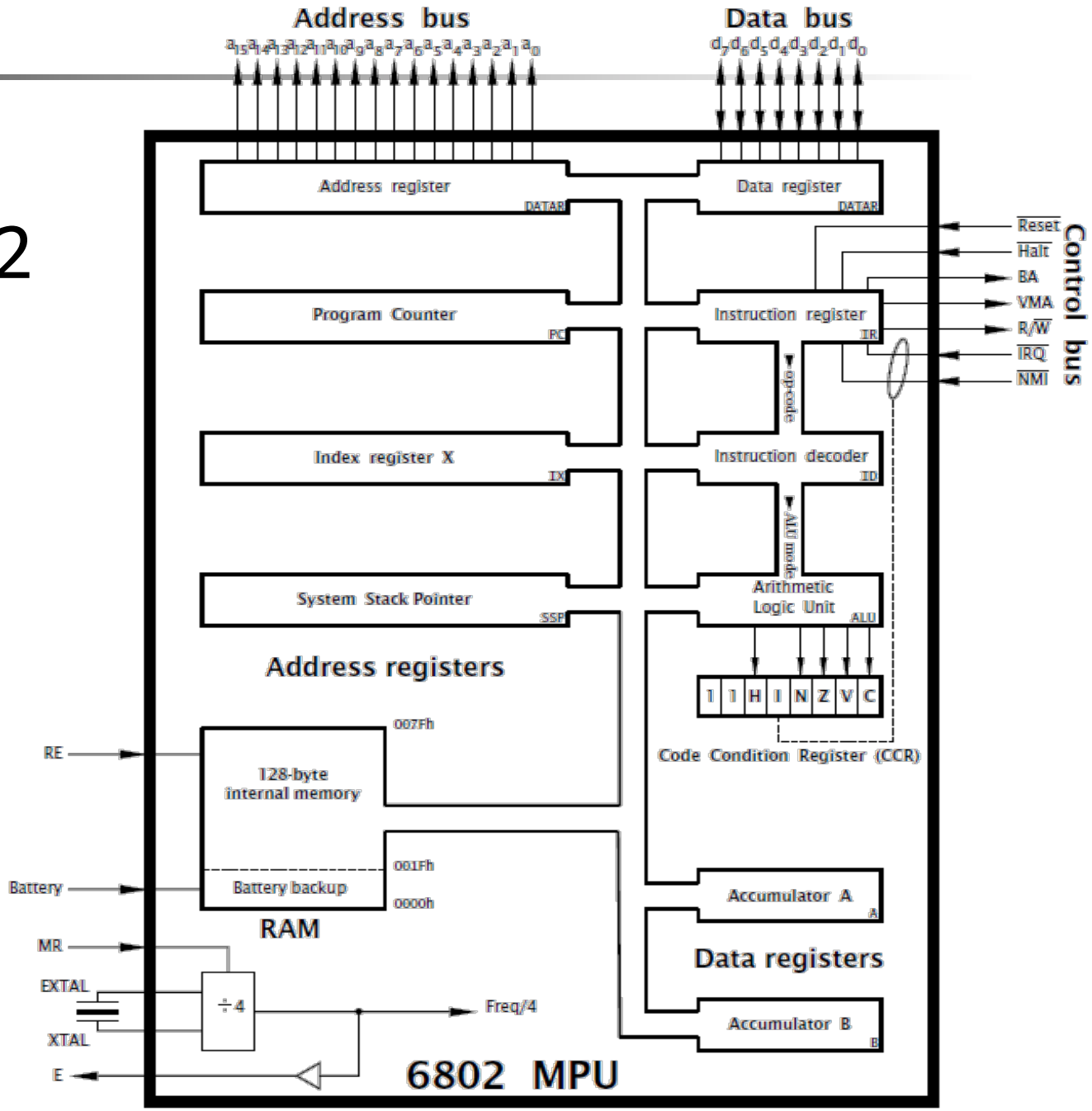
CPU Elements

- Program Counter (PC)
- Instruction Register (IR)
- Instruction Decoder
- Arithmetic and Logic Unit (ALU)
- General Purpose Registers
- Special Purpose Registers (SP, BP, IX, CCR, etc.)
- Control Unit (CU)

Internal Structure of CPU

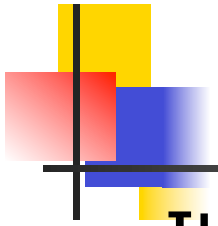
Example:

Motorola 6802



Registers in the Fetch Unit

- **Program Counter:** holds the memory location of the next instruction.
- **Instruction Register:** holds the current instruction being executed

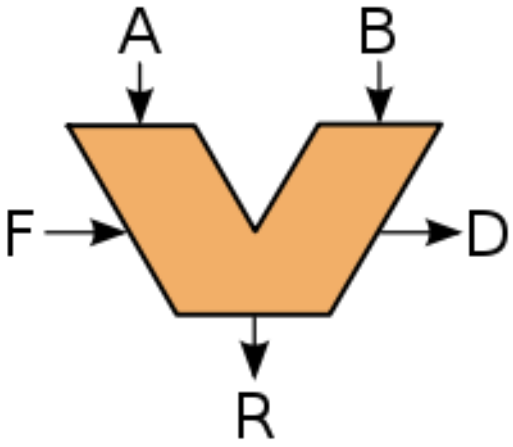


Instruction Decoder

- It decodes the instructions and generates the control signals

Arithmetic Logic Unit (ALU)

- ALU performs all arithmetic and logic operations in a microprocessor
- ALU has two inputs (A, B) for the operands and one input for a control signal that selects the operation
- Operation and Shift control bits determine, which type of operation to perform (F)
- Output is the result of operation (R) and status information (D)
- Status information is used to indicate cases
  - Zero: if all result lines have value 0
  - Overflow: integer overflow of add and subtract functions
    - For unsigned integers, it does not provide any useful information



Registers

- A register is a storage location in the CPU
- It is used to hold data or a memory address during the execution of an instruction
- Because the register file is small and close to the ALU, accessing data in registers is much faster than accessing data in memory outside the CPU
- The register file makes program execution more efficient
- The number of registers varies from computer to computer

Condition Code Register or Flag Register

- Depending on the outcomes of Arithmetic or Logical operations, we can *branch* and *jump*
- The eight-bit Condition Code Register (CCR) provides a status report on the ALU's **activity**
  - **C**arry/Borrow
  - **H**alf carry from bit 3 to bit 4
  - o**V**erflow
- CCR also provides a status report **after loading** ACC
  - **Z**ero
  - **N**egative

			V	Z	N	H	C

Condition Code Register (CCR)

- They flag certain conditions resulting from the ALU outcomes
- Example:

A= 01001000 B= 01111001

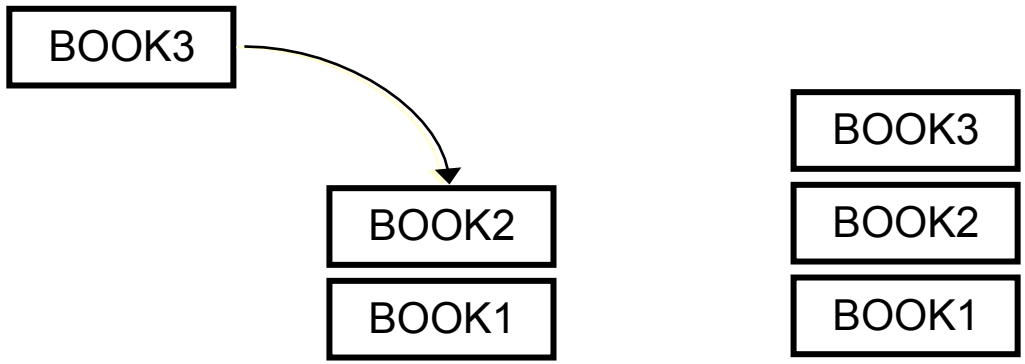
A+B:

```
A      01001000
B     +01111001
      ----------
      11000001      V=1  Z=0  N=1  H=1  C=0
```

- Depending on the outcomes of Arithmetic or Logical operations, we can branch and jump

The Stack

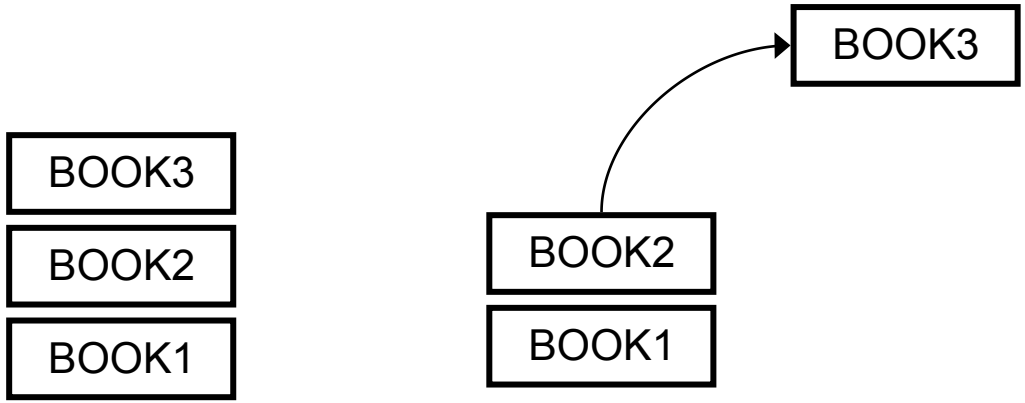
- A stack is a last-in-first-out data structure
- A stack of a computer works just like a real stack, e.g., of books. If you have a stack of books, you can put another book on top:



- This is called a **push**
- All that happens is the stack gets one book deeper, and the last book you added is on top

The Stack

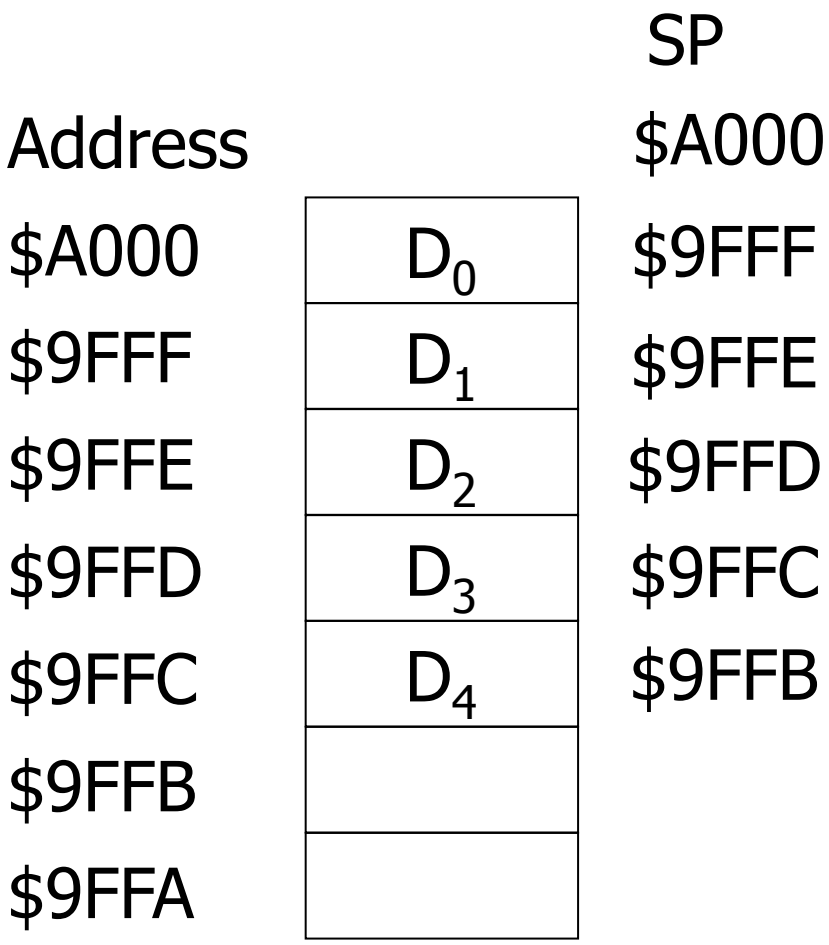
- You can also take a book off the top of the stack:



- This is called a **pop**.
- The stack gets one book shorter, and the book you get from the top is the one you added, or pushed, most recently
- Because a pop gives you back the item you most recently pushed, a stack is called a **last-in-first-out**, or **LIFO**, structure

Stack Pointer

- The stack is a way of using the memory.
- All that's needed is some **unused memory and an index register, called the Stack Pointer** (SP), that always points to the next available (empty) location above the current top of the stack
- The stack grows toward lower addresses



Control Unit

- The control unit is a **synchronous sequential** logic circuit that sends control signals to the data processing unit, memory and other parts of the system
- The signals from the control unit tells the data processing unit to manipulate data according to the algorithm built into the sequential logic circuit
- The control unit is **instruction controlled**; therefore it can do more than one algorithm based on its design (programmable)
- Typical control units recognize several hundred different instruction codes

System Clock

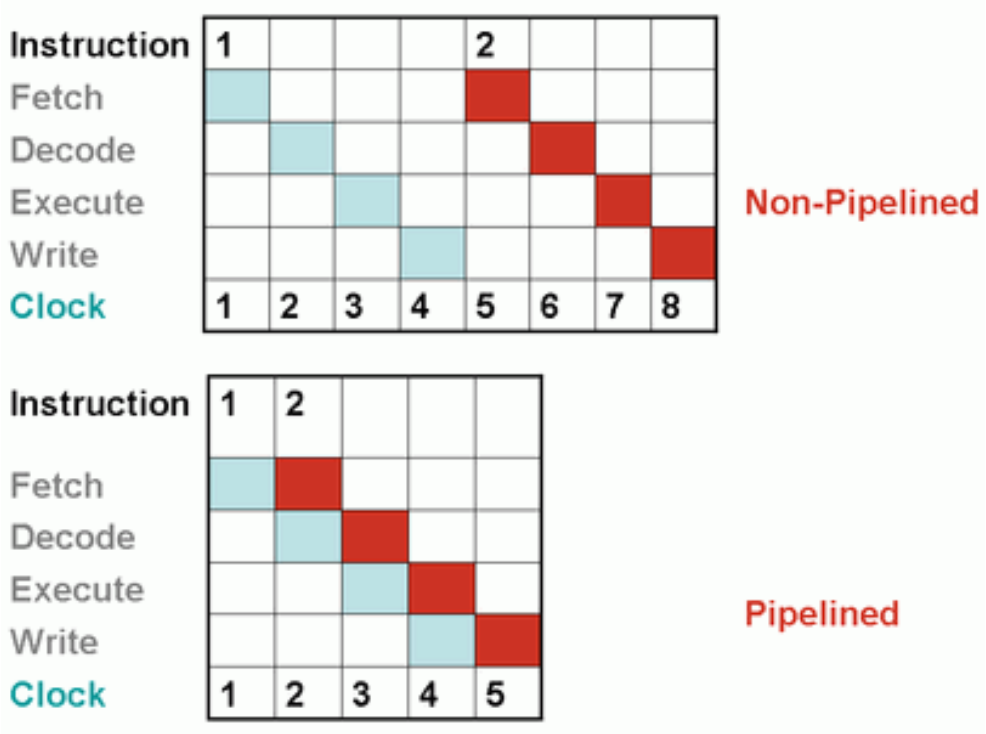
- In order to regulate when the control unit issues its control signals, computers use a system clock
- System clock generates **regular pulses** to synchronize all system events and determine the speed at which processing can occur
- Each fetch-execute instruction cycle is divided into states, which are one clock pulse long
- Most instructions require multiple steps, and so require **several clock pulses** to complete (multi-cycle processor design)
- Some individual steps (e.g. a **memory access**) take longer & may require additional clock pulses to complete – these clock cycles spent waiting are called **wait states**

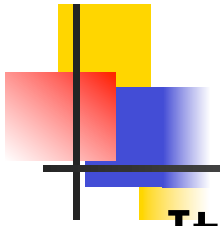
System Clock

- The clock speed of a CPU determines how often a new instruction is executed, and is measured in MHz or GHz
  - For example: 1.7GHz means that a computer could executes 1,700,000,000 instructions per second! (if it executes 1 instruction at a cycle)

System Clock

- However, all recent microprocessors overlap the fetching, decoding and execution of a number of instructions at the same time – this is called **pipelining**
- Therefore, clock speed is not necessarily an accurate measure of performance, and other measurements are required





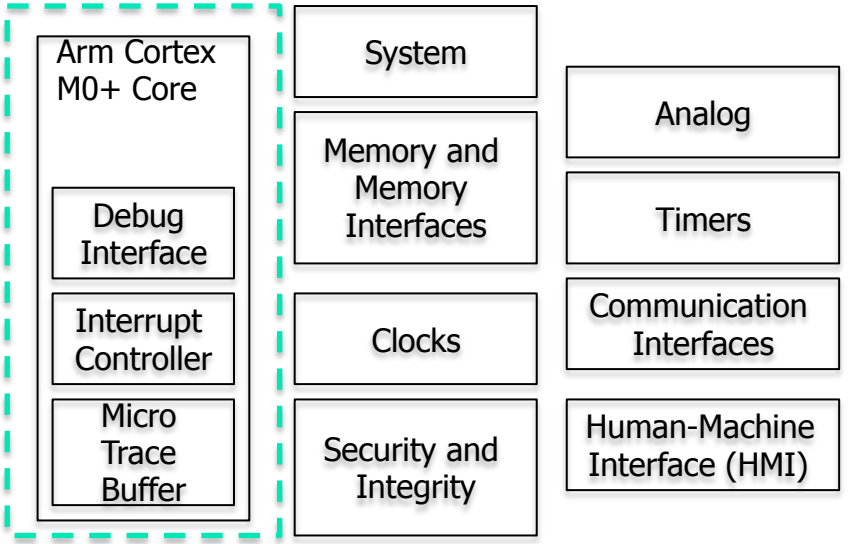
Comparing Clocks

- It is difficult to compute CPU performance just by comparing clock speed
  - You must also consider how many clock cycles it takes to execute 1 instruction
  - How fast memory is
  - How fast the bus is
  - etc.
- In addition, there are different clocks in the computer, the Control Unit and the whole CPU are governed by the system clock
- There is usually a bus clock as well to regulate the usage of the slower buses

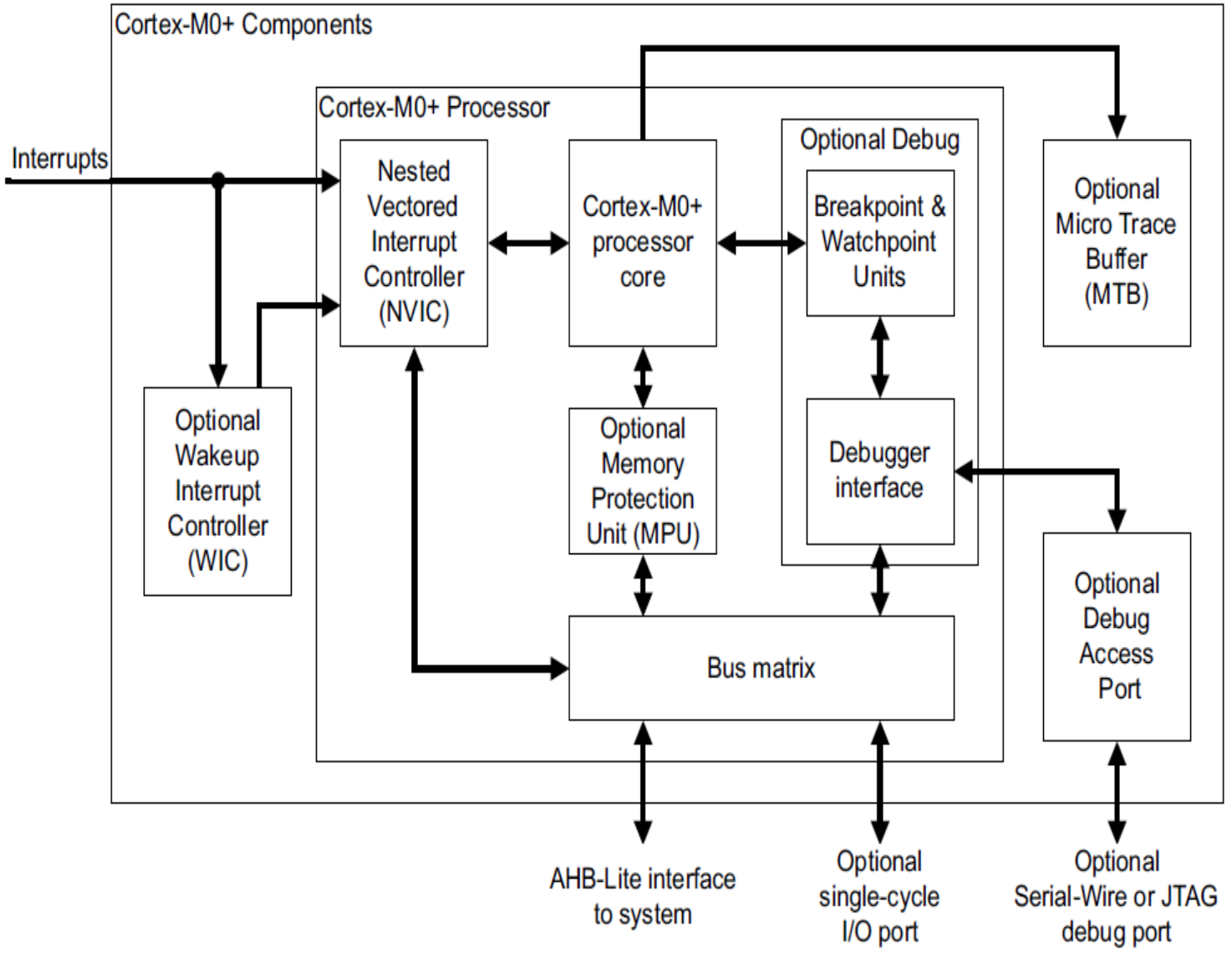
CORTEX-M0+ CPU CORE

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



An ISA defines the hardware/software interface

- A “contract” between architects and programmers
- Register set
- Instruction set
  - Addressing modes
  - Word size
  - Data formats
  - Operating modes
  - Condition codes
- *Calling conventions*
  - Really not part of the ISA (usually)
  - Rather part of the ABI
  - But the ISA often provides meaningful support.

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
    - Load data into the registers
    - Process the data
    - 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)

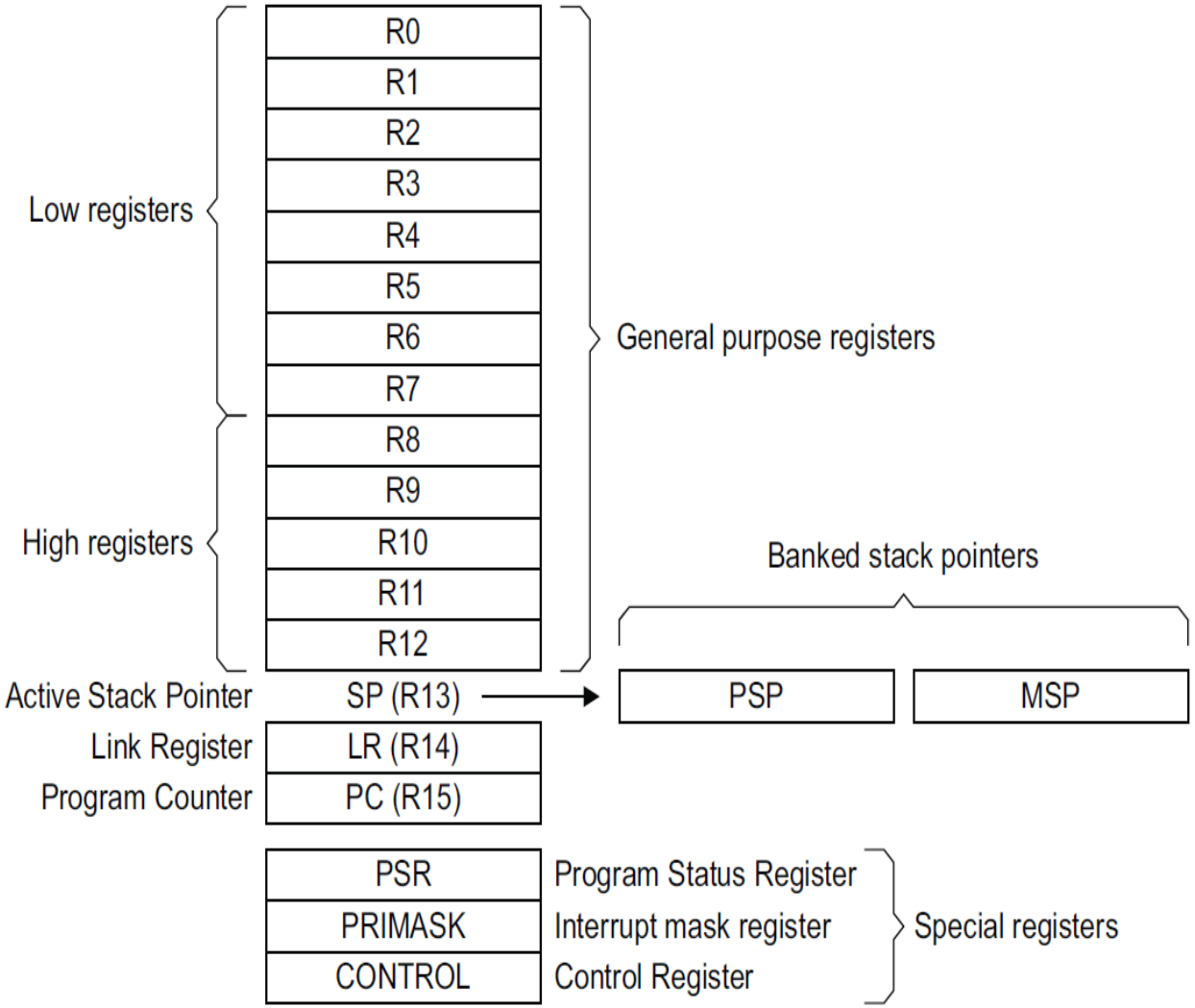
Arm Architecture

- The ARM is a *Reduced Instruction Set Computer* (RISC), as it incorporates these typical RISC architecture features:
  - a large uniform register file
  - a *load/store* architecture, where data-processing operations only operate on register contents, not directly on memory contents
  - simple addressing modes, with all load/store addresses being determined from register contents and instruction fields only
  - uniform and fixed-length instruction fields, to simplify instruction decode

Arm Architecture

- In addition, the ARM architecture provides:
  - control over both the *Arithmetic Logic Unit* (ALU) and shifter in most data-processing instructions to maximize the use of an ALU and a shifter
  - auto-increment and auto-decrement addressing modes to optimize program loops
  - Load and Store Multiple instructions to maximize data throughput
  - conditional execution of almost all instructions to maximize execution throughput
- These enhancements to a basic RISC architecture allow ARM processors to achieve a good balance of high performance, small code size, low power consumption, and small silicon area

ARM Processor Core Registers



Stack Pointer

Note: there are two stack pointers!

SP_process (PSP) used by:

- Base app code (when not running an exception handler)

SP_main (MSP) used by:

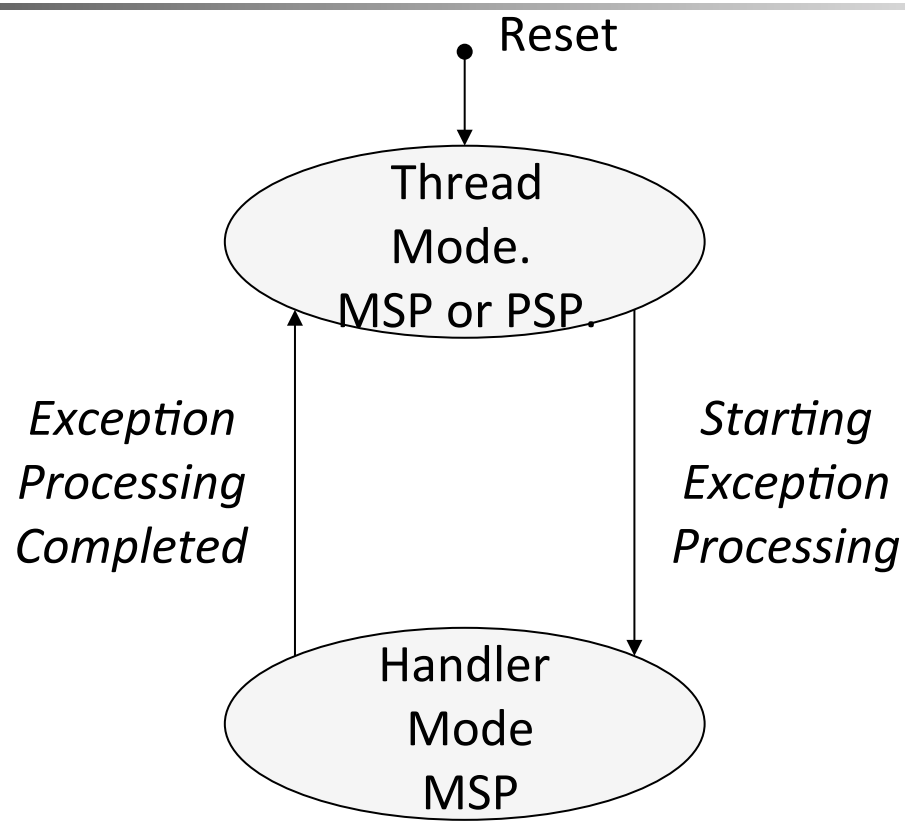
- OS kernel
- Exception handlers
- App code w/ priviledged access

Mode dependent

ARM Processor Core Registers (32 bits each)

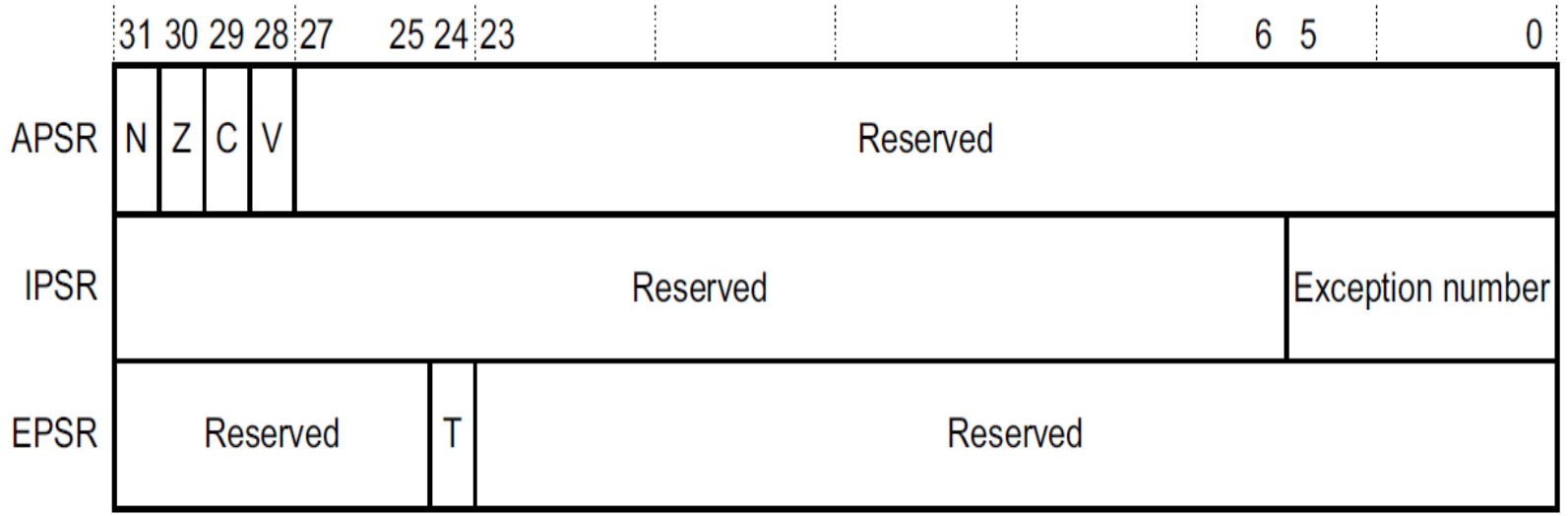
- R0-R12 - General purpose registers for data processing
- SP - Stack pointer (R13)
  - Can refer to one of two SPs
    - Main Stack Pointer (MSP)
    - Process Stack Pointer (PSP)
  - Uses MSP initially, and whenever in Handler mode
  - When in Thread mode, can select either MSP or PSP using SPSEL flag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

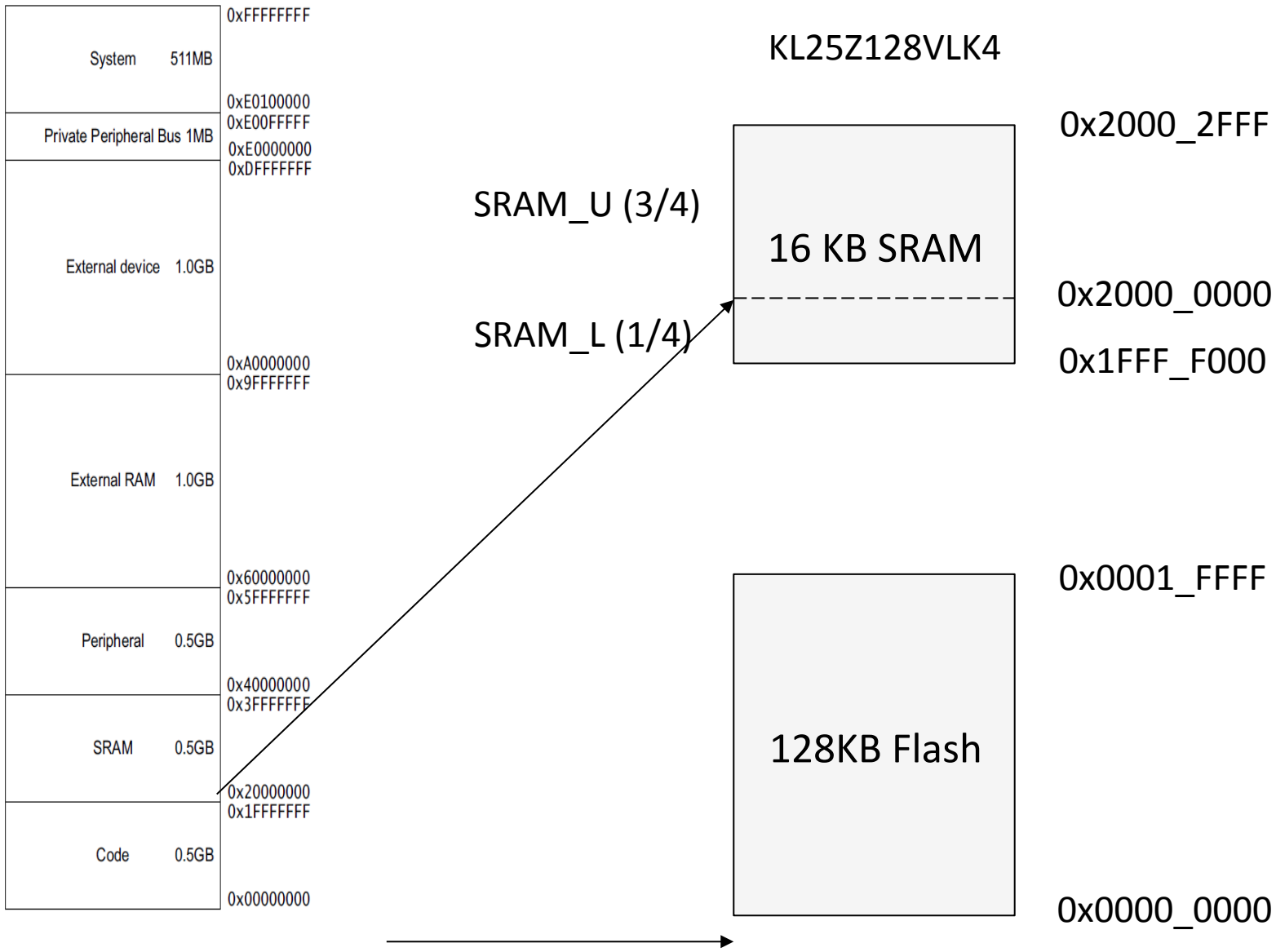


- Program Status Register (PSR) is three views of same register
  - Application PSR (APSR)
    - Condition code flag bits Negative, Zero, oVerflow, Carry
  - Interrupt PSR (IPSR)
    - Holds exception number of currently executing ISR
  - Execution PSR (EPSR)
    - Thumb state

ARM Processor Core Registers

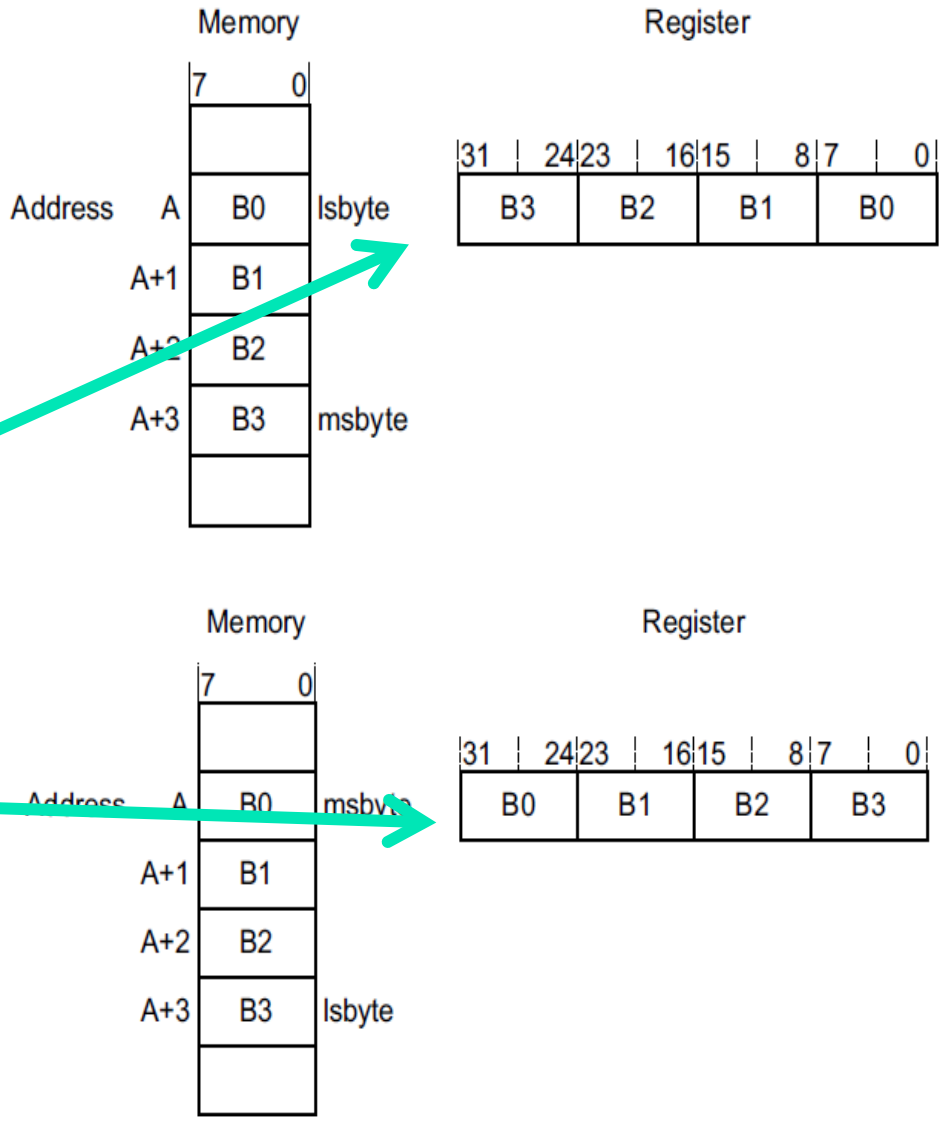
- PRIMASK - Exception mask register
  - Bit 0: PM Flag (Priority Mask Flag)
    - Set to 1 to prevent activation of all exceptions with configurable priority
  - Access using CPS, MSR and MRS instructions
  - Use to prevent data race conditions with code needing atomicity
- CONTROL
  - Bit 1: SPSEL flag
    - 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 (operating system threads)
    - 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

ARM Cortex-M3 Memory Formats (Endian)

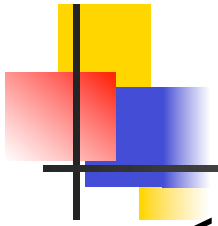
- Default memory format for ARM CPUs: LITTLE ENDIAN
- Bytes 0-3 hold the first stored word
- Bytes 4-7 hold the second stored word
- Processor contains a configuration pin BIGEND
  - Enables hardware system developer to select format:
    - Little Endian
    - Big Endian (BE-8)
  - Pin is sampled on reset
  - Cannot change endianness when out of reset

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)
- Thumb state indicated by program counter being odd (LSB = 1)
  - Branching to an even address will cause an exception, since switching back to ARM state is not allowed
- 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)



Assembler Instruction Format

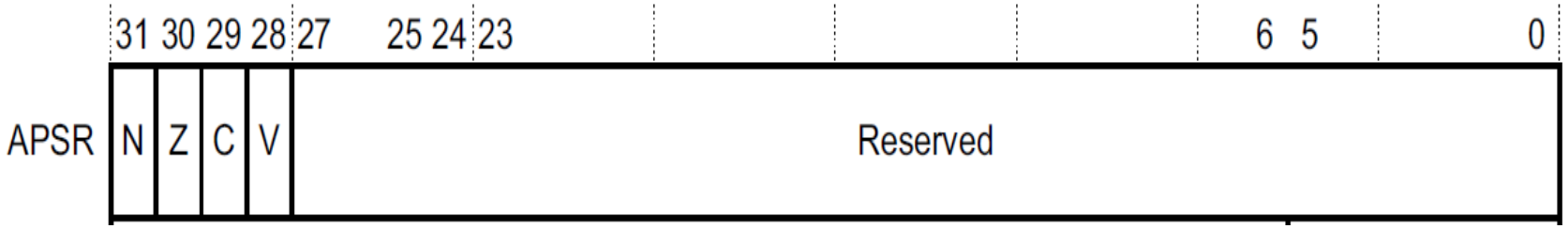
- <operation> <operand1> <operand2> <operand3>
  - There may be fewer operands
  - First operand is typically destination (<Rd>)
  - Other operands are sources (<Rn>, <Rm>)

- 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>

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

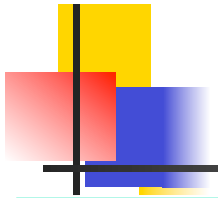
Update Condition Codes in APSR?



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

Updating the APSR

- SUB Rx, Ry
  - Rx = Rx - Ry
  - APSR unchanged
- SUBS
  - Rx = Rx - Ry
  - APSR N, Z, C, V updated
- ADD Rx, Ry
  - Rx = Rx + Ry
  - APSR unchanged
- ADDS
  - Rx = Rx + Ry
  - APSR N, Z, C, V updated



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

Modes for Addressing Memory

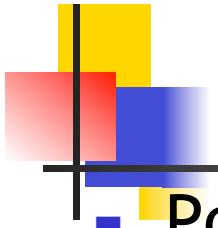
- Offset Addressing mode: [<Rn>, <offset>] accesses address <Rn>+<offset>
- Base Register <Rn>
  - Can be register R0-R7, SP or PC
- <offset> is added or subtracted from base register to create effective address
  - Can be an immediate constant
  - Can be another register, used as index <Rm>
- Auto-update: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

Addressing Modes

- Offset Addressing
  - Offset is added or subtracted from base register
  - Result used as effective address for memory access
  - [<Rn>, <offset>]
  - Examples:
    - LDR R2, [R0] ; Load R2 with the word pointed by R0
    - STR R2, [R3] ; Store the word in R2 in the location pointed by R3
    - LDR R0, [R1, #20] ; loads R0 with the word pointed at by R1+20

Addressing Modes (continues)

- Pre-indexed Addressing
  - Offset is applied to base register
  - Result used as effective address for memory access
  - Result written back into base register
  - [<Rn>, <offset>]!
  - Example:
    - LDR R0, [R1, #4]! ; loads R0 with the word pointed at by R1+4
      ; then update the pointer by adding 4 to R1

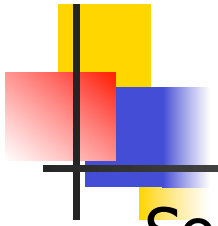


Addressing Modes (continues)

- Post-indexed Addressing
  - The address from the base register is used as the EA
  - The offset is applied to the base and then written back
  - [<Rn>], <offset>
  - Example:
    - LDR R0, [R1], #4; loads R0 with the word pointed at by R1
      ; then update the pointer by adding 4 to R1

Summary of ARM's Indexed Addressing Modes

Addressing Mode	Assembly Mnemonic	Effective address	FinalValue in R1
Pre-indexed, base unchanged	LDR R0, [R1, #d]	R1 + d	R1
Pre-indexed, base updated	LDR R0, [R1, #d]!	R1 + d	R1 + d
Post-indexed, base updated	LDR R0, [R1], #d	R1	R1 + d



Loading/Storing Smaller Data Sizes

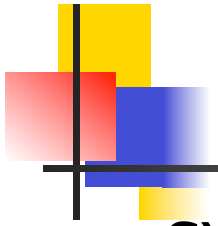
- Some load and store instructions can handle half-word (16 bits) and byte (8 bits)
- Store just writes to half-word or byte
  - STRH, STRB
- Loading a byte or half-word requires padding or extension: What do we put in the upper bits of the register?
  - Example: How do we extend 0x80 into a full word?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

In-Register Size Extension

- Can also extend byte or half-word that is 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

	Signed	Unsigned
Byte	SXTB	UXTB
Half-word	SXTH	UXTH



Example

- SXTB R0, R1; R0 = Sign Extend (R1[7:0])
- SXTH R0, R1; R0 = Sign Extend (R1 [15:0])

- UXTB R0, R1; R0 = Zero Extend (R1 [7:0])
- UXTH R0, R1; R0 = Zero Extend (R1 [15:0])

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
- The accesses happens in order of increasing register numbers, with the lowest numbered register using the lowest memory address and the highest number register using the highest memory address.

Example for LDM/STM

```
; Load the address of the source data array
ADR R0, SRC_ADR
; Load 6 consecutive addresses starting at SRC_ADDR
; R0 is NOT modified
; All three instructions below do exactly the same thing
LDM R0, {R1-R6}
LDM R0, {R1, R2, R3, R4, R5, R6}
LDM R0, {R6, R1, R2, R4, R5, R3}
; Load the destination address
ADR R7, DEST_ADDR
STM R7, {R1-R6}
```

Example for LDM/STM

- This example demonstrates how to use the LDM/STM instructions that do modify the base registers. The only functional difference between the previous example and this example is that R0 and R7 both get modified.

```
; Load Src and Dest Addresses
ADR R0, SRC_ADR
ADR R7, DEST_ADDR
; Use '!' to write back to the base address
; The base address is incremented based on the
; number of WORDs being accessed
LDM R0!, {R1-R3} ; R0 <- R0 + 12
STM R7!, {R1-R3} ; R7 <- R7 + 12
LDM R0!, {R1-R3} ; R0 <- R0 + 12
STM R7!, {R1-R3} ; R7 <- R7 + 12
```

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??"

Example for LDR

```
LDR r1,=0xfff        ; loads 0xfff into r1

LDR r2,= place       ; loads the address of place into r2
                     ; (place is a label)
```

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>

Examples

- MOV r3, #0
- MOV r0, r12 ; does not update flags

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)

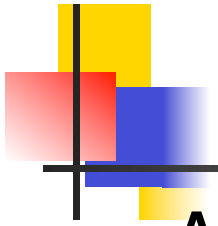
Examples

- PUSH { R0, R1, R2 }; Memory [SP- 4]= R2,
  ; Memory [SP- 8]= R1,
  ; Memory [SP- 12]= R0,
  ; SP = SP-12

POP { R0, R1, R2 }; R0 = Memory [SP],
; R1 = Memory [SP+ 4],
; R2 = Memory [SP+ 8],
; SP = SP+12

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>

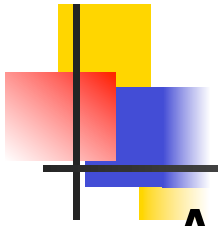


Add Instructions with Stack Pointer

- 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>

Examples

- ADDS R0, R1, R2 ; R0 = R1 + R2
  ; Update APSR
- ADDS R0, R1, #0x01 ; R0 = R1 + Zero Extend (0x01)
  ; Update APSR
- ADDS R0, #0x01 ; R0 = R0 +Zero Extend (0x01)
  ; Update APSR
- ADD R0, R1 ; R0 = R0 + R1
- ADCS R0, R1 ; R0 = R0 + R1 + Carry
  ; Update APSR
- ADD R0, PC, #0x04 ; R0 = PC + 0x04



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>

Examples

- 64-bit addition

```
ADDS    R0, R0, R2    ; add the least significant words
ADCS    R1, R1, R3    ; add the most significant words with carry
```

  - Multiword values do not have to use consecutive registers.
  - Example shows instructions that subtract a 96-bit integer contained in R1, R2, and R3 from another contained in R4, R5, and R6.
  - The example stores the result in R4, R5, and R6.

- 96-bit subtraction

```
SUBS    R4, R4, R1    ; subtract the least significant words
SBCS    R5, R5, R2    ; subtract the middle words with carry
SBCS    R6, R6, R3    ; subtract the most significant words with carry
```

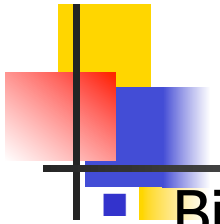
  - Example shows the RSBS instruction used to perform a 1's complement of a single register

- Arithmetic negation

```
RSBS    R7, R7, #0    ; subtract R7 from zero
```

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: upper word of result is truncated
- MULS R0, R1, R0 ; R0 = R0 × R1
  ; Update APSR

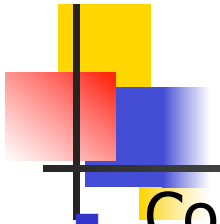


Logical Operations

- 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>

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

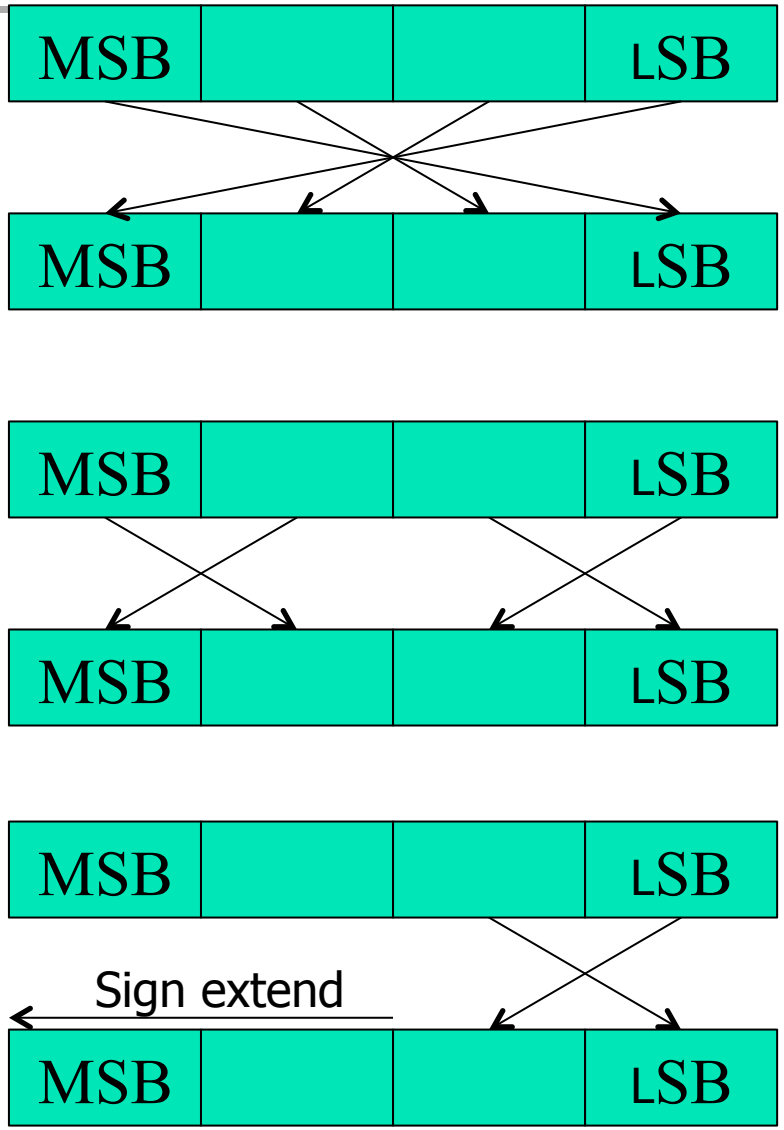
- Common features
  - All of these instructions update APSR condition flags
  - Shift/rotate amount (in number of bits) specified by last operand
- Logical shift left - shifts in zeroes on right
  - LSLS <Rd>,<Rm>,#<imm5>
  - LSLS <Rdn>,<Rm>
- Logical shift right - shifts in zeroes on left
  - LSRS 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

Examples

- ASRS R7, R5, #9 ; Arithmetic shift right by 9 bits
- LSLS R1, R2, #3 ; Logical shift left by 3 bits with flag update
- LSRS R4, R5, #6 ; Logical shift right by 6 bits
- RORS R4, R4, R6 ; Rotate right by the value in the bottom byte of R6

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>



Examples

- REV R3, R7 ; Reverse byte order of value in R7 and write it to R3
- REV16 R0, R0 ; Reverse byte order of each 16-bit half-word in R0
- REVSH R0, R5 ; Reverse signed half-word

Changing Program Flow - Branches

- Branches (conditional and unconditional)
  - Branch without link (i.e. no possibility of return) to `target`
  - The PC is not saved!
- 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 != 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

Changing Program Flow - Subroutines

- Call
  - BL <label> - branch with link
    - Store the return address in the link register (`lr`)
    - Call subroutine at <label>
      - PC-relative, range limited to PC+/-16MB
    - Save return address in LR
  - BLX <Rd> - branch with link and exchange
    - Call subroutine at address in register Rd (exchange Rd with PC)
      - Supports full 4GB address range
      - LSB of target address must be set to 1 to ensure continued execution in Thumb state
    - Save return address in LR
- Return
  - BX <Rd> branch and exchange
    - Branch to address specified by <Rd>
    - LSB of target address must be set to 1 to ensure continued execution in Thumb state
      - Supports full 4 GB address space
    - BX LR - Return from subroutine

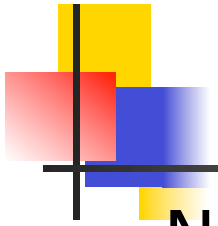
Examples

```
B      loopA  ; Branch to loopA
BL     funC   ; Branch with link (Call) to function funC, return address
              ; stored in LR
BX     LR     ; Return from function call
BLX    R0     ; Branch with link and exchange (Call) to a address stored
              ; in R0
BEQ    labelD ; Conditionally branch to labelD if last flag setting
               ; instruction set the Z flag, else do not branch.
```

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>

Exercise: What is the value of r2 at done?

```
...
start:
      movs r0, #1
      movs r1, #1
      movs r2, #1
      sub  r0, r1
      bne  done
      movs r2, #2
done:
      b    done
...
```

Solution: What is the value of r2 at done?

```
...
start:
        movs r0, #1         // r0 ← 1, Z=0
        movs r1, #1         // r1 ← 1, Z=0
        movs r2, #1         // r2 ← 1, Z=0
        sub  r0, r1         // r0 ← r0-r1
                            // but Z flag untouched
                            // since sub vs subs
        bne  done           // NE true when Z==0
                            // So, take the branch
        movs r2, #2         // not executed
done:
        b    done           // r2 is still 1
...
```

An example ARM assembly language program for GNU

```
        .equ    STACK_TOP, 0x20000800
        .text
        .syntax unified
        .thumb
        .global _start
        .type   start, %function

_start:
        .word   STACK_TOP, start
start:
        movs r0, #10
        movs r1, #0
loop:
        adds r1, r0
        subs r0, #1
        bne  loop
deadloop:
        b    deadloop
        .end
```

What's it all mean?

```
_start:
        .word   STACK_TOP, start       /* Inserts word 0x20000800 */
                                       /* Inserts word (start) */
start:
        movs r0, #10                   /* We've seen the rest ... */
        movs r1, #0
loop:
        adds r1, r0
        subs r0, #1
        bne  loop
deadloop:
        b    deadloop
        .end
```