ECSE 426 – Fall 2015
Microprocessor Systems

Dept. Electrical and Computer Engineering

McGill University

Problem-oriented Language layer

- Compiled to assembly or instruction set level
- You will be using embedded C
- How does this differ from usual use of C?
  - Directly write to registers to control processor operation
  - All of the registers have been mapped to macros
  - Important bit combinations have macros – use these, please !
  - Registers are 32 bits, so int type is 4 bytes
  - Register values may change without your specific instructions
  - Limited output system
  - Floating point operations somewhat inefficient, divide & square-root should be avoided if possible.

Assembly versus C

- Efficiency of compiled code
- Source code portability
- Program maintainability
- Typical bug rates (say, per thousand lines of code)
- The amount of time it will take to develop the solution
- Availability and cost of compilers and other development tools
- Your personal experience (or that of the developers on your team) with specific languages or tools

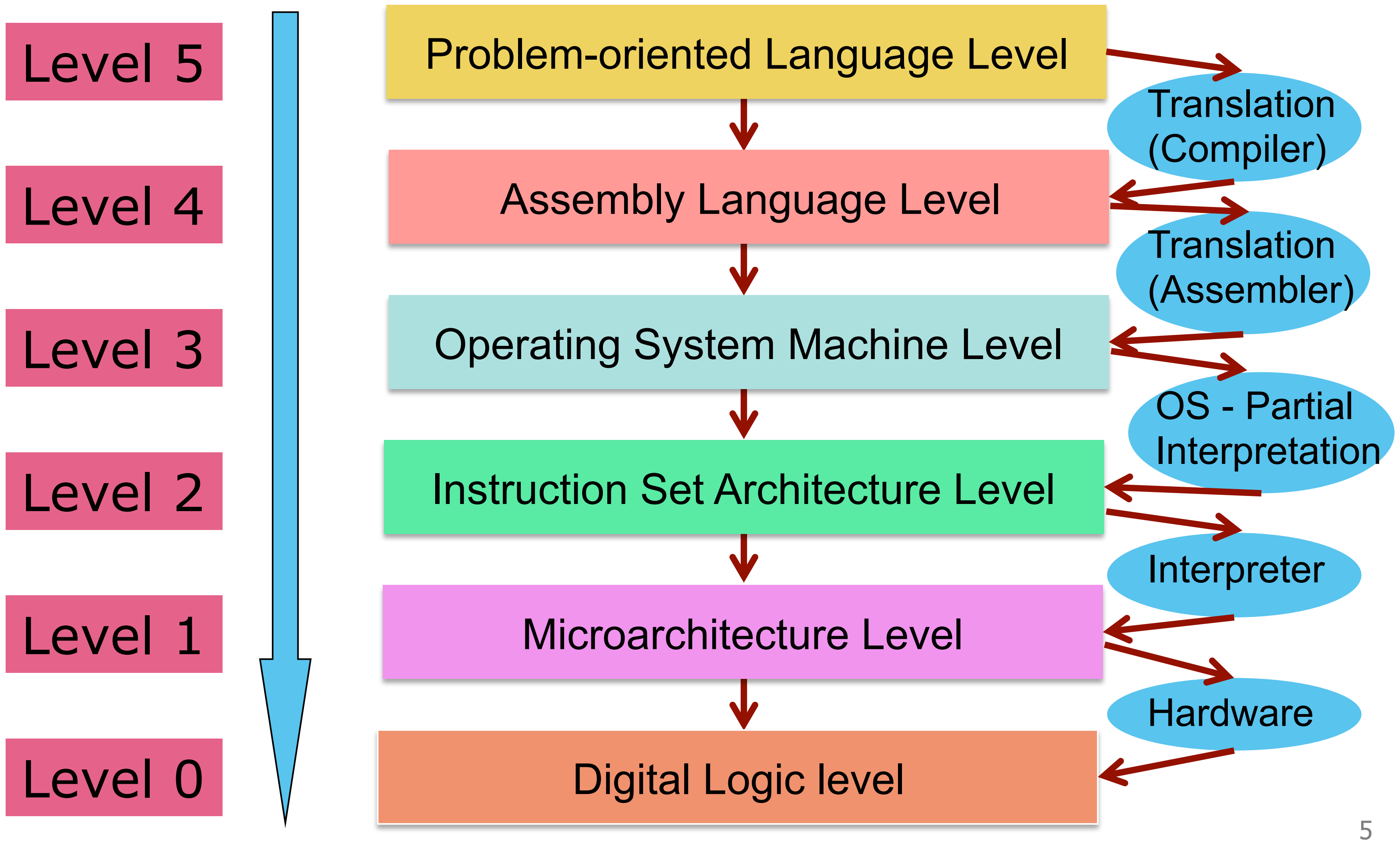
- Don’t rule out Java or C++ if you have the memory to play with.

Problem

- Company “Ostrich” has recently re-developed their embedded software for flagship products
  - Developed in assembly, 80 percent working, 2000 lines of code
  - Suddenly realized that the product isn’t shippable
  - Bugs: system lock-ups indicative of major design flaws or implementation errors & major product performance issues
  - Designer has left the company and provided few notes or comments

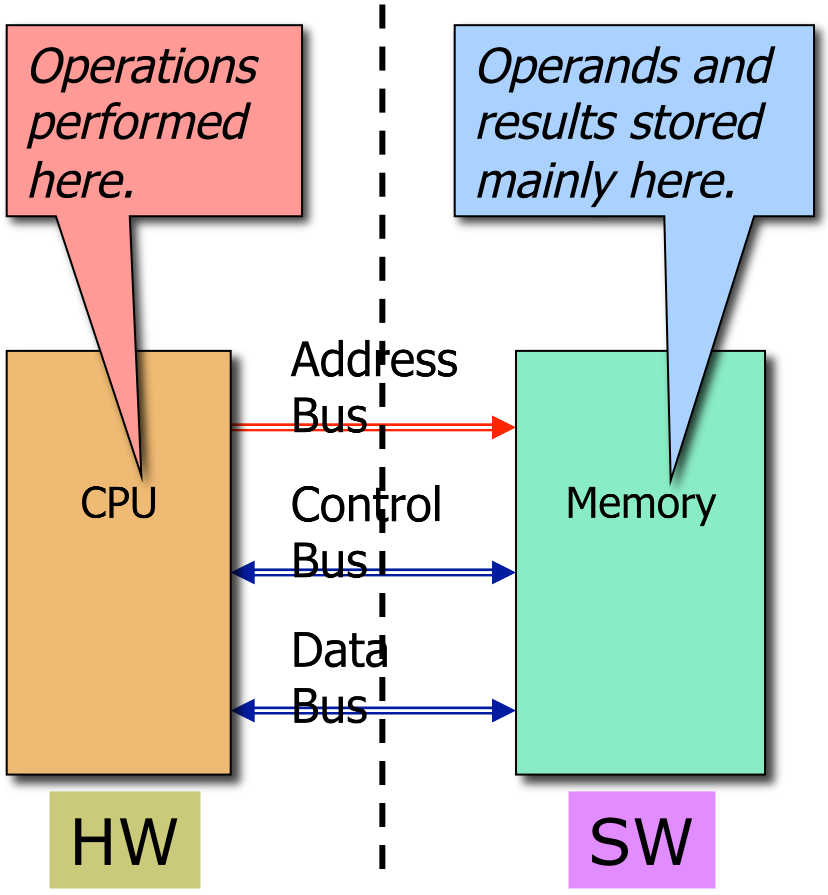
- You are hired as a consultant. Do you:
  - Fix existing code?
  - Perform complete software redesign and implementation? In this case, which language?

Computer Organization



Instruction Set Architecture

- Interface between HW and SW
  - Virtual Machine
    - Many possible implementations of ISA
- Given by
  - Resources
    - Processor Registers
    - Execution Units
  - Operations
    - Instruction Types
    - Data Types
    - Addressing Modes



Operations performed here.
Operands and results stored mainly here.
CPU
Address Bus
Control Bus
Data Bus
Memory
HW
SW

i.e., where and how to address operands

ARM: A Brief History

- Acorn Computers Ltd., Cambridge, England (1983-1985)

- RISC processor concept was introduced in 1980s
- Advanced RISC Machines
- ARM Ltd. – 1990
    - joint venture between Acorn, Apple, and VLSI Technology

ARM Processor: Architecture

- A 32-bit Enhanced RISC processor
  - RISC: Reduced Instruction Set Computing
    - Simplified instructions can provide higher performance
    - Small, highly-optimized set of instructions
- Register-to-register, three operand instruction set
  - All operands are 32-bits wide
- Employs Load Store Architecture
  - Operations operate on registers and not on memory locations

Cortex-M3 and Cortex-M4

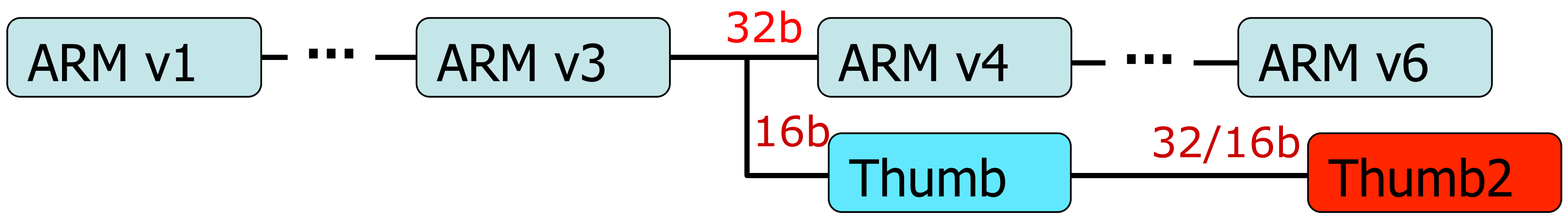
- 32-bit: internal registers, data path and bus interfaces

- Three-stage pipeline
- Harvard bus architecture with unified memory space
  - instructions and data use same address space

- Based on ARMv7-M architecture
  - high performance processors designed for microcontrollers
  - low power

ARM Cortex ISA: ARMv7-M

- Implements Thumb2 specification



- Mixture of 32-bit and 16-bit instructions

ARM Processor Modes

- Processor Mode determines
  - which registers are active (visible)
  - the rights to modify the Program Status Registers (PSRs)

- Each mode can be either privileged or non-privileged
  - Privileged mode
    - full read/write access to PSR
    - The software can use all instructions
  - Unprivileged mode
    - Only read access to control fields of PSRs,
    - Read/write access to condition flags
    - Might have restricted access to memory or peripherals

ARM Modes

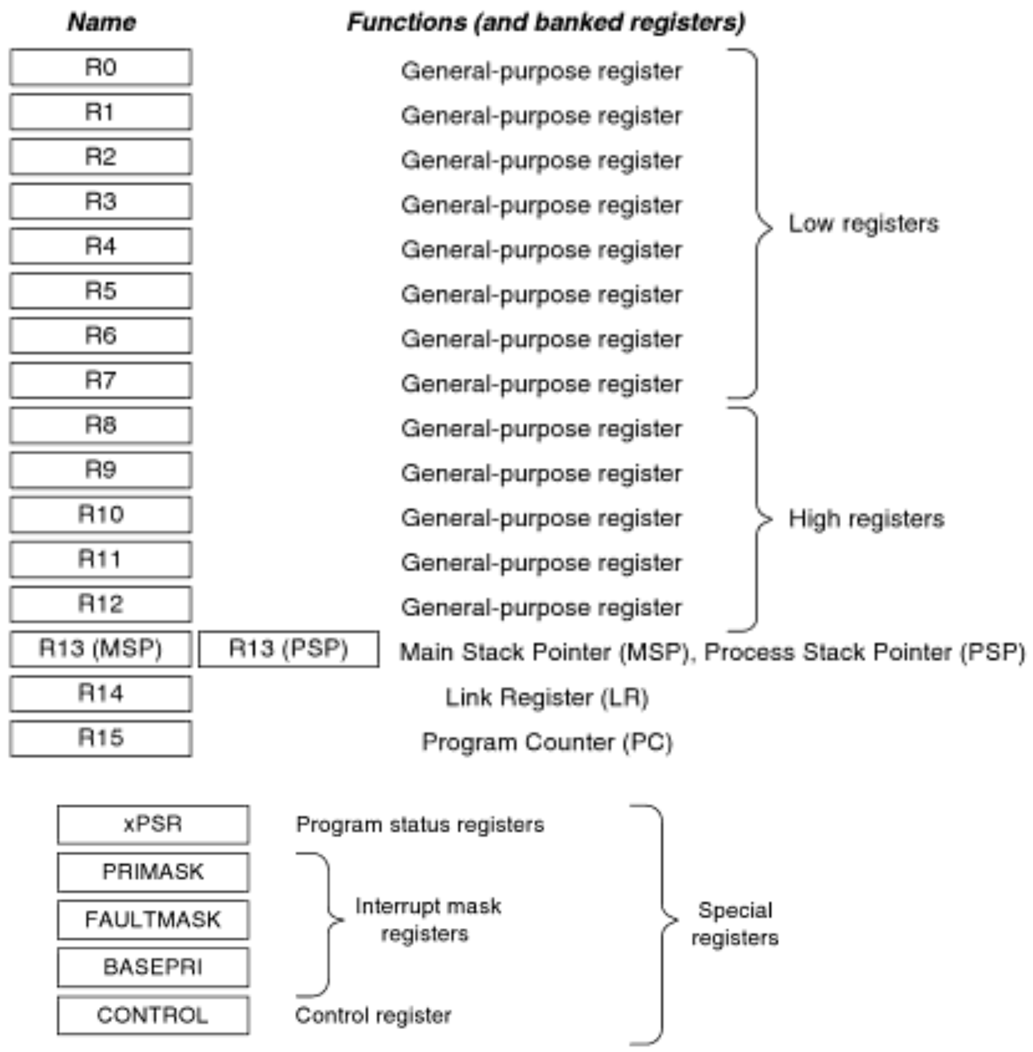
- Operation states
  - Thumb state: processor running program code
  - Debug state: halted by debugger; stops executing instructions

- Operation modes
  - Thread mode: default state (privileged to begin with)
  - Handler mode: exceptions and interrupts

Registers

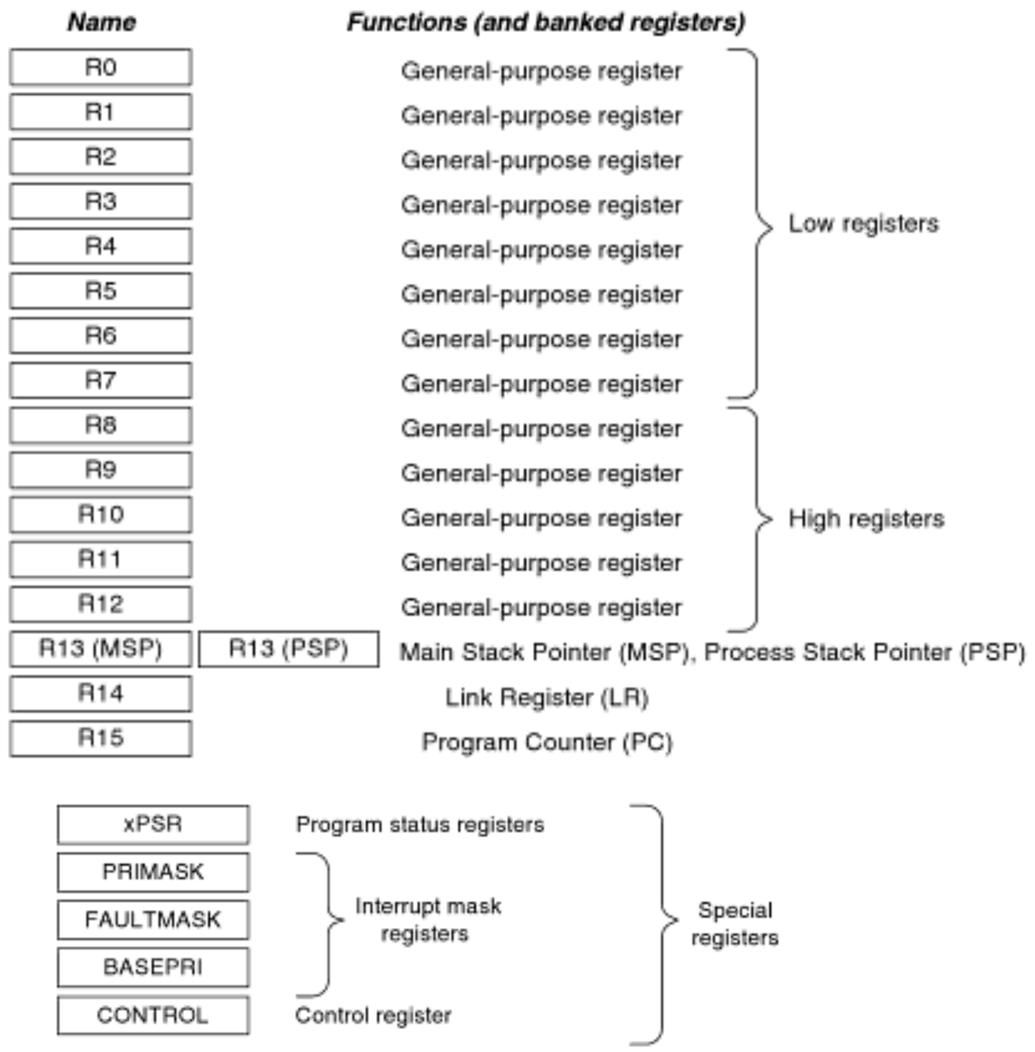
- Large register set
  - R0-R12: General purpose
  - Program Counter (R15)
  - Link Register (R14)
  - Stack Pointer (R13)

- Registers R0-R3 hold first 4 words of incoming argument



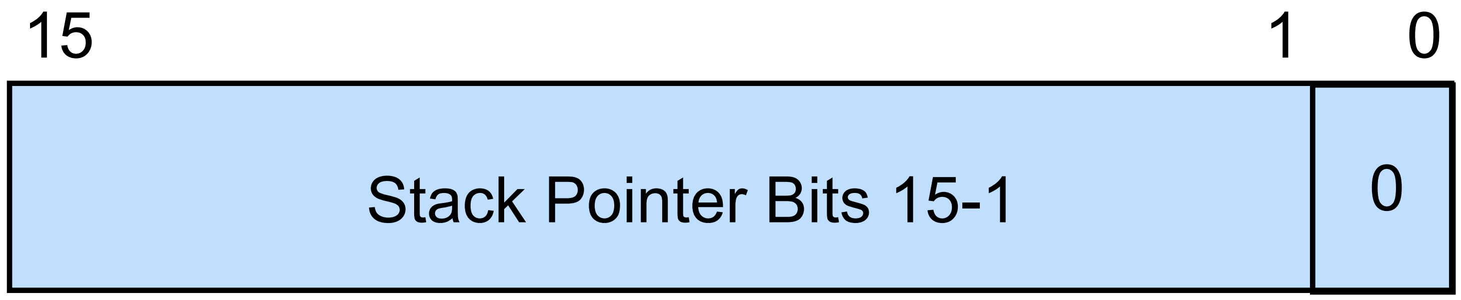
Registers

- R0-R7
  - low registers
  - Many 16-bit instructions can only access low registers
- R8-R12
  - High registers
  - 32-bit instructions



R13: Stack Pointer (SP)

- Access stack memory using PUSH and POP operations
- Two different stack pointers
  - Main Stack Pointer (MSP): default, selected after reset
  - Process Stack Pointer (PSP): only in thread mode
- Lowest two bits always zero
- PUSH and POP are 32-bit (addresses must be aligned to 32-bit word boundaries)



Stack

- PUSH and POP
  - access stack memory
  - Usually used to save (and restore) registers before other data processing

- `PUSH {R0} ; R13 = R13-4, then Memory[R13] = R0`
- `POP {R0}  ; R0 = Memory[R13], then R13 = R13 + 4`

- Cortex-M4 uses a full-descending stack arrangement.
- Stack pointer automatically decrements when new data is stored in the stack.

Stack

- `subroutine_1`
  - `PUSH {R0-R7, R12, R14} ; Save registers`
  - `… ; Do your processing`
  - `POP {R0-R7, R12, R14} ; Restore registers`
  - `BX R14 ; Return to calling function`

- Can push/pop multiple registers at the same time
- Usually POPs will mirror PUSHes (but there are exceptions)

- PUSH and POP operations are always word aligned
  - addresses must be 0x0, 0x4, 0x8, …
  - R13 bits 0 and 1 are hardwired to zero and always read as zero

R14: Link Register (LR)

- Holds return address when calling a function or subroutine
- Program control return to calling program by loading LR into PC

- Function or subroutine call: LR updated automatically
- If a function needs to call another function, needs to save LR on stack

- During exception handling, LR updated automatically to a special EXC_RETURN value

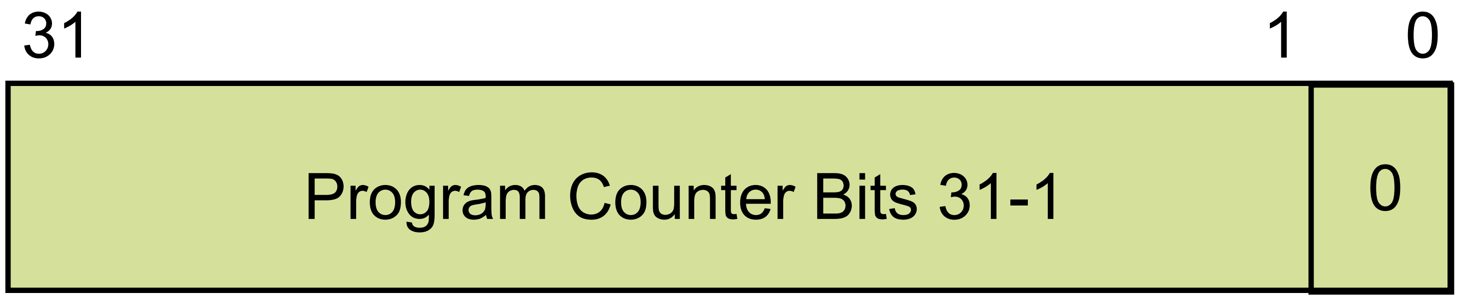
R14: Link Register (lr)

```
main ; Main program
…
BL function1     ; Call function1 using Branch with Link
                  ; instruction. PC = function 1 and
                  ; LR = the next instruction in main
…
function1
…                ; Program code for function 1
BX LR            ; Return
```

- Bit 0 of the program counter is always 0
- But LR bit 0 is readable and writable
  - Some branch/call instructions require that bit zero is set to one

R15: Program Counter

- Tells you where you are in the program
  - Next instruction address (pipelining blurs that)
- Readable and writeable
  - read: returns current instruction + 4
  - write: branch operation
- Instructions must be half-word or word-aligned
  - LSB is zero



R15: Program Counter

- 32 bits, aligned to even addresses
  - LDR PC, =LABEL Branch to address in LABEL
  - MOV PC, R4 Branch to address contained in R4
  - MOV R0, PC Not recommended, processor-dependent
- Need to set LSB to one to indicate Thumb state for some of the branch & read memory instructions
  - handled by compiler automatically in C

- Branches and calls usually have dedicated instructions
- Value of PC useful for accessing literal data stored in program memory

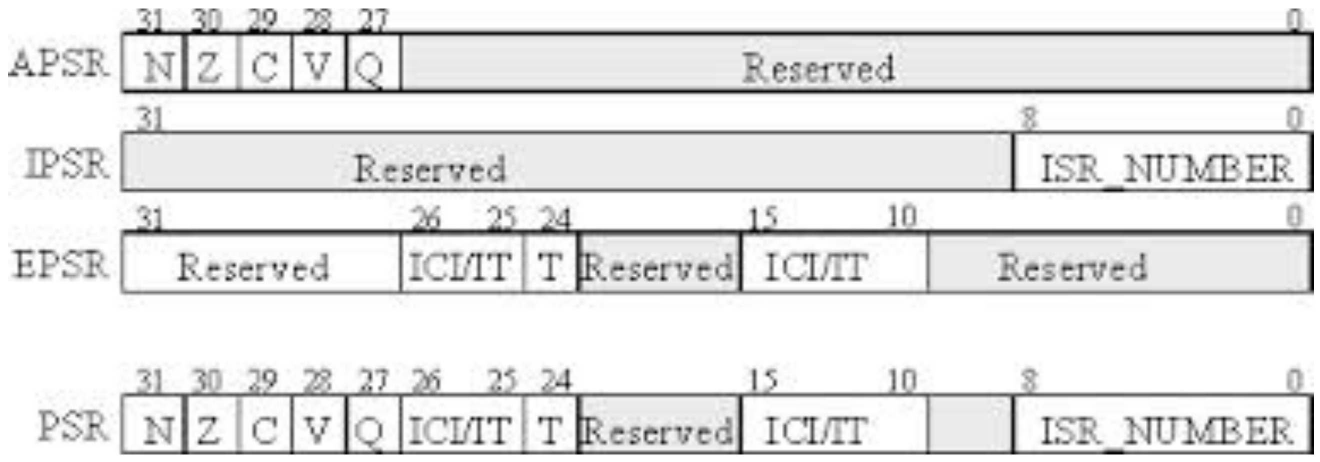
ARM: Register Summary

- 16 accessible 32-bit registers (R0-R15) at any time
  - R15 acts as the Program Counter (PC)
  - R14 (Link Register) stores subroutine return address
  - R13: stack pointer

- You can (and should!) write in ARM assembly language:
  - PC for R15,
  - lr for R14,
  - sp for R13

Program Status Register (PSR)

- Three status registers
  - Application PSR (APSR)
  - Execution PSR (EPSR)
  - Interrupt PSR (IPSR)
- Accessed as one combined register (xPSR)



Reproduced from Yiu

Program Status Register (PSR)

- Access combined PSR
- `MRS r0, PSR; read combined program status word`
- `MSR PSR, r0; write combined program status word`

- Or access an individual PSR
- `MRS r0, APSR; Read flag state into r0`
- `MRS r0, IPSR; Read exception/interrupt state`
- `MSR APSR, r0; write flag state`

- EPSR cannot be accessed by code directly using MSR/MRS
- IPSR is read only

Program Status Register (PSR)

- APSR contains current values of conditional code bits
  - N – Negative Result from ALU
  - Z – Zero result from ALU
  - C – Carry from ALU operation
  - V – ALU operation overflow
- IPSR: Exception number
  - indicates which exception processor is handling
- EPSR:
  - ICI (Interrupt-Continuable Instruction) bits
  - IT: IF-THEN instruction status bit for conditional execution

Control Register

- Access level in thread mode (privileged/unprivileged): bit 0
- Selection of stack pointer (MSP/PSP): bit 1
- Current context uses floating point unit: bit 2

- All zeros: privileged, MSP, FPU unused

- Can only be modified in privileged access level

- Switch back to privileged access through an exception
  - exception handler can clear the nPRIV bit

Floating Point Registers

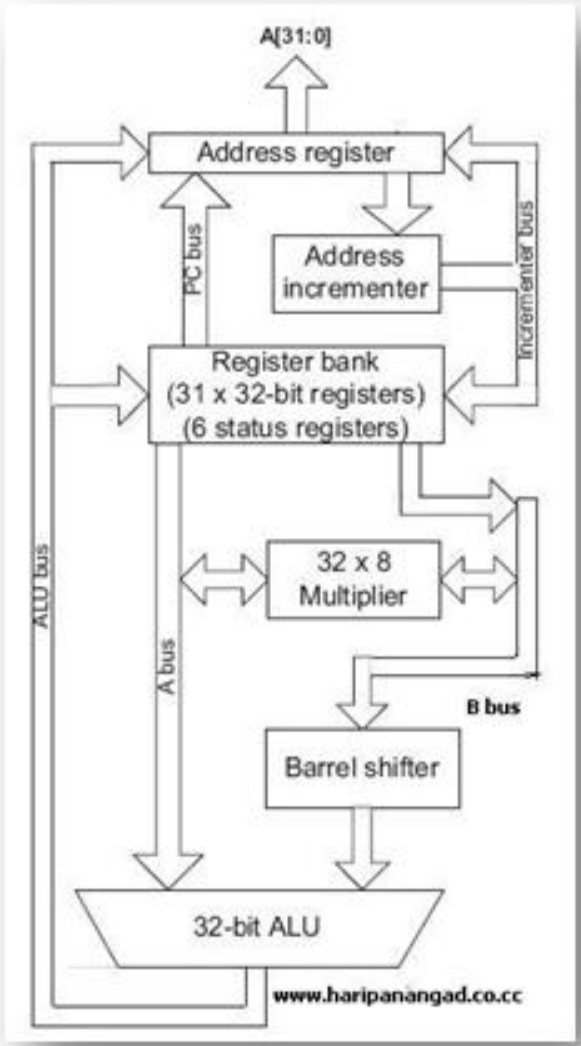
- Cortex M4 has additional registers for floating point operations
- S0:S31 can be accessed using floating point instructions
- or accessed as pairs (D0:D15)
- Does not support double precision floating point ops

  but can transfer double precision data

ARM: Core Datapath

- Separate control and datapath
- In datapath operands are not fetched directly from memory locations
  - Data is placed in register files
  - No data processing takes place in memory

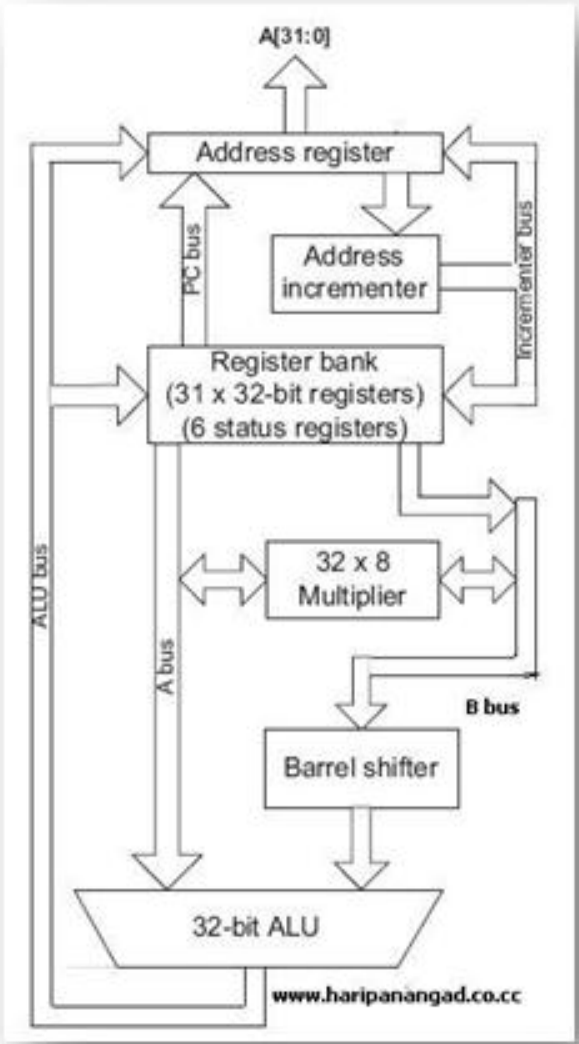
- Instructions typically use 3 registers
  - 2 source registers
  - 1 destination register
- Barrel shifter preprocesses data before it enters ALU



ARM Organization

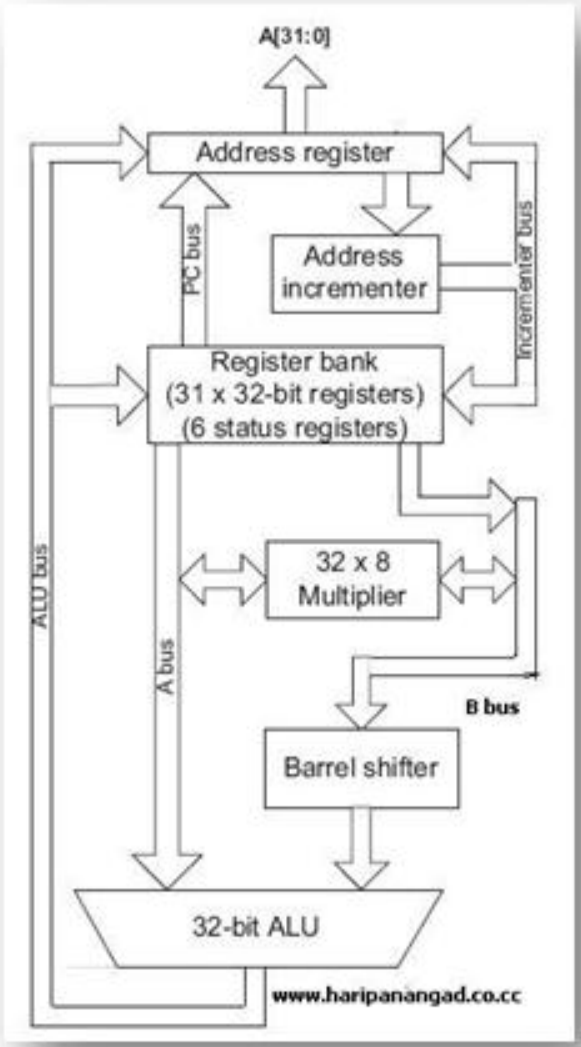
- Data has to be moved from the memory location to the central set of registers
  - After processing it is stored back in memory

- Register bank connected to ALU via two datapath buses A and B
  - Bus B goes through Barrel shifter
    - It pre-processes data from source register by shl, shr or rotation
  - PC is part of the register bank
    - responsible for generating addresses for next operation
  - Registers can handle both data and address



ARM Organization

- Address Incrementer
  - block increments or decrements register values independent of ALU



Cortex-M feature set comparison

	Cortex-M0	Cortex-M3	Cortex-M4
Architecture Version	V6M	v7M	v7ME
Instruction set architecture	Thumb, Thumb-2 System Instructions	Thumb + Thumb-2	Thumb + Thumb-2, DSP, SIMD, FP
DMIPS/MHz	0.9	1.25	1.25
Bus interfaces	1	3	3
Integrated NVIC	Yes	Yes	Yes
Number interrupts	1-32 + NMI	1-240 + NMI	1-240 + NMI
Interrupt priorities	4	8-256	8-256
Breakpoints, Watchpoints	4/2/0, 2/1/0	8/4/0, 2/1/0	8/4/0, 2/1/0
Memory Protection Unit (MPU)	No	Yes (Option)	Yes (Option)
Integrated trace option (ETM)	No	Yes (Option)	Yes (Option)
Fault Robust Interface	No	Yes (Option)	No
Single Cycle Multiply	Yes (Option)	Yes	Yes
Hardware Divide	No	Yes	Yes
WIC Support	Yes	Yes	Yes
Bit banding support	No	Yes	Yes
Single cycle DSP/SIMD	No	No	**Yes**
Floating point hardware	No	No	**Yes**
Bus protocol	AHB Lite	AHB Lite, APB	AHB Lite, APB
CMSIS Support	Yes	Yes	Yes

Cortex
Low-Power Leadership from ARM

ARM Cortex-M3/4 Processor

- Complex processor core providing standard CPU and system architecture
- Architecture v7-M
  - Only 2 processor modes (thread/handler)
  - Processor automatically saves/restores state in exceptions
    - Improved interrupt speeds
  - Enhanced Flash access

ARM Cortex-M3/4 Processor

- Banked Stack Pointer (SP) (banked: only one visible at a time)
  - Main Stack Pointer (MSP): The default stack pointer, used by the operating system (OS) kernel and exception handlers
  - Process Stack Pointer (PSP): Used by user application code

  - Stack usage by OS kernel can be separated from application usage: better reliability and optimized stack space usage
  - No OS: Can use the MSP all the time

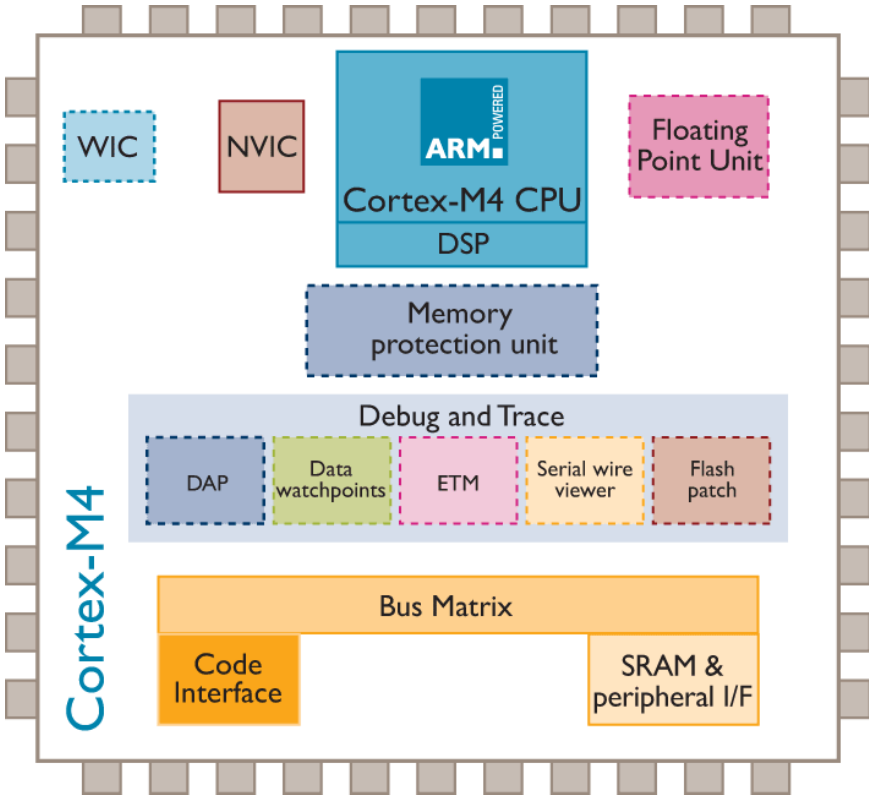
Cortex-M

- Basic peripherals are built into Cortex-M, e.g.
  - interrupt controller
  - 24-bit SysTick timer
  - debug system
  - memory map and sleep mode
- Based on Harvard architecture (code and data bus separated)
  - Enables simultaneous instruction fetch with data load/store
- Multiple buses for performing operations in parallel
- Bus matrix connects processor and debug interface to external busses

Cortex-M4

- ARMv7ME Architecture
  - Thumb-2 Technology
  - Compatible with Cortex-M3
  - Single precision FPU
  - DSP and SIMD extensions
  - Single cycle MAC (32 x 32 + 64 -> 64)

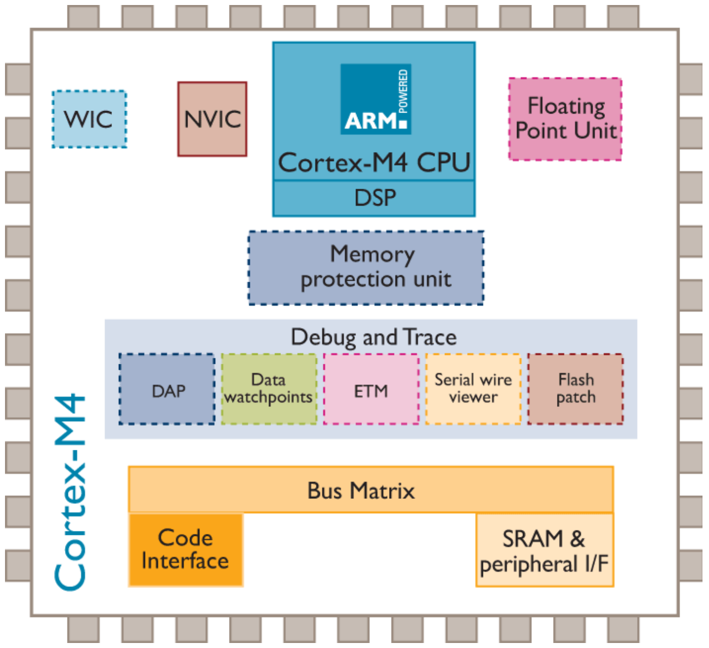
- Microarchitecture
  - 3-stage pipeline with branch speculation
  - 3x AHB-Lite Bus Interfaces



Cortex-M4

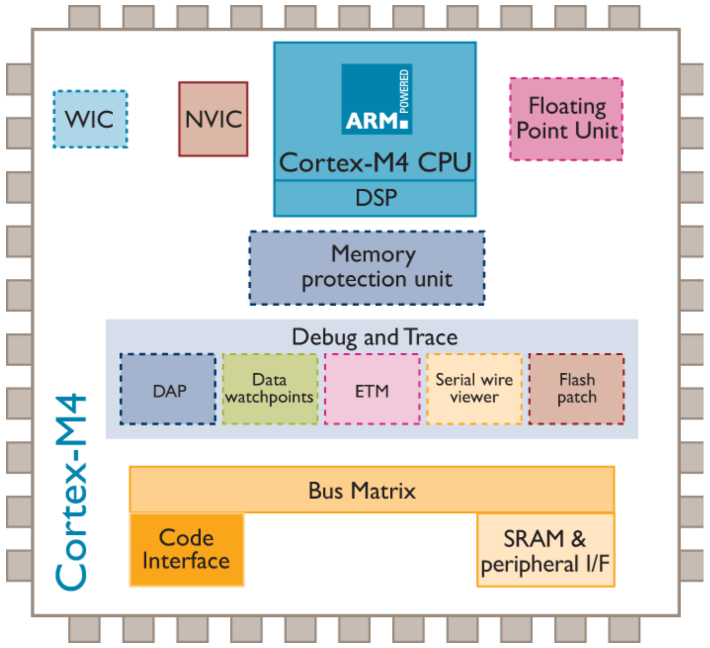
- Configurable for ultra low power
  - Deep Sleep Mode, Wakeup Interrupt Controller
  - Power down features for Floating Point Unit

- Flexible configurations for wider applicability
  - Configurable Interrupt Controller (1-240 Interrupts and Priorities)
  - Debug & Trace
  - Optional Memory Protection Unit



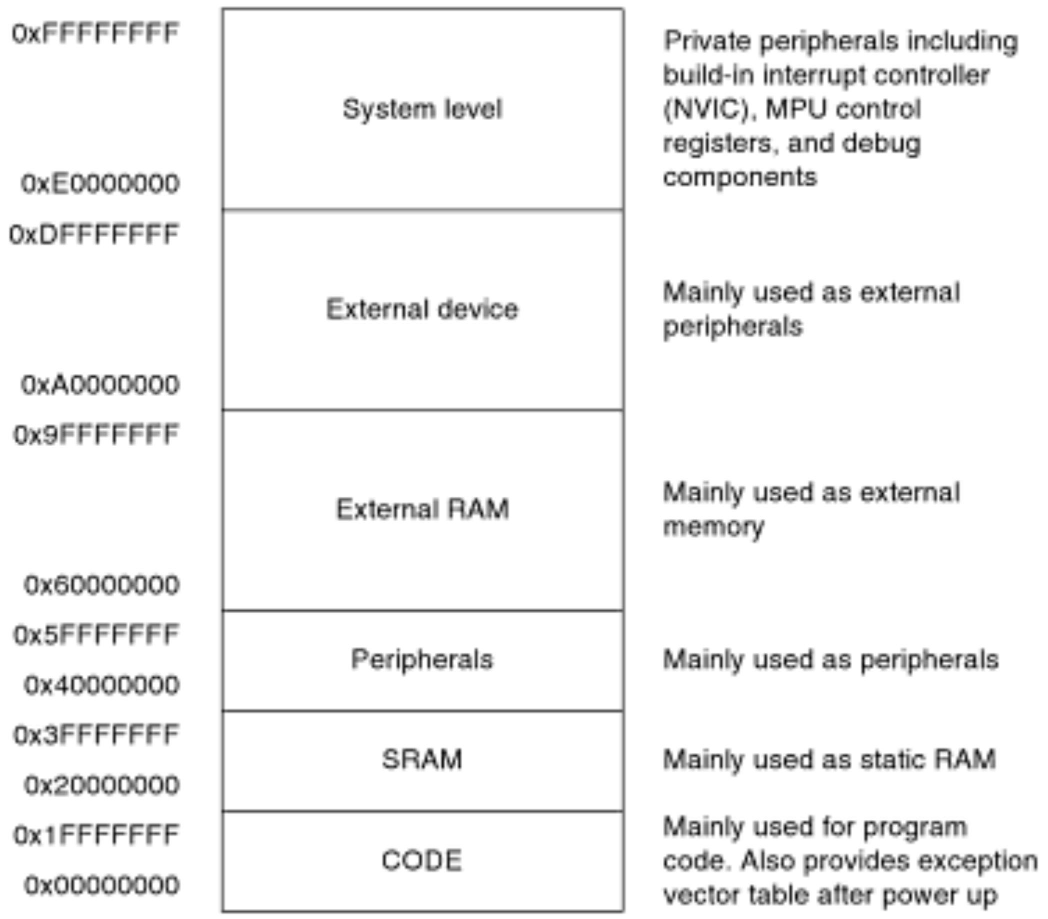
Cortex-M4

- CPU core is closely coupled to the interrupt controller (NVIC) and various debug logic blocks
- CM4Core: The Cortex-M4 core contains the registers, ALU, data path, and bus interface.
- Nested Vectored Interrupt Controller (NVIC)
  - built-in interrupt controller.
  - Number of interrupts is customized by chip manufacturers (up to 240)
- Bus interconnect



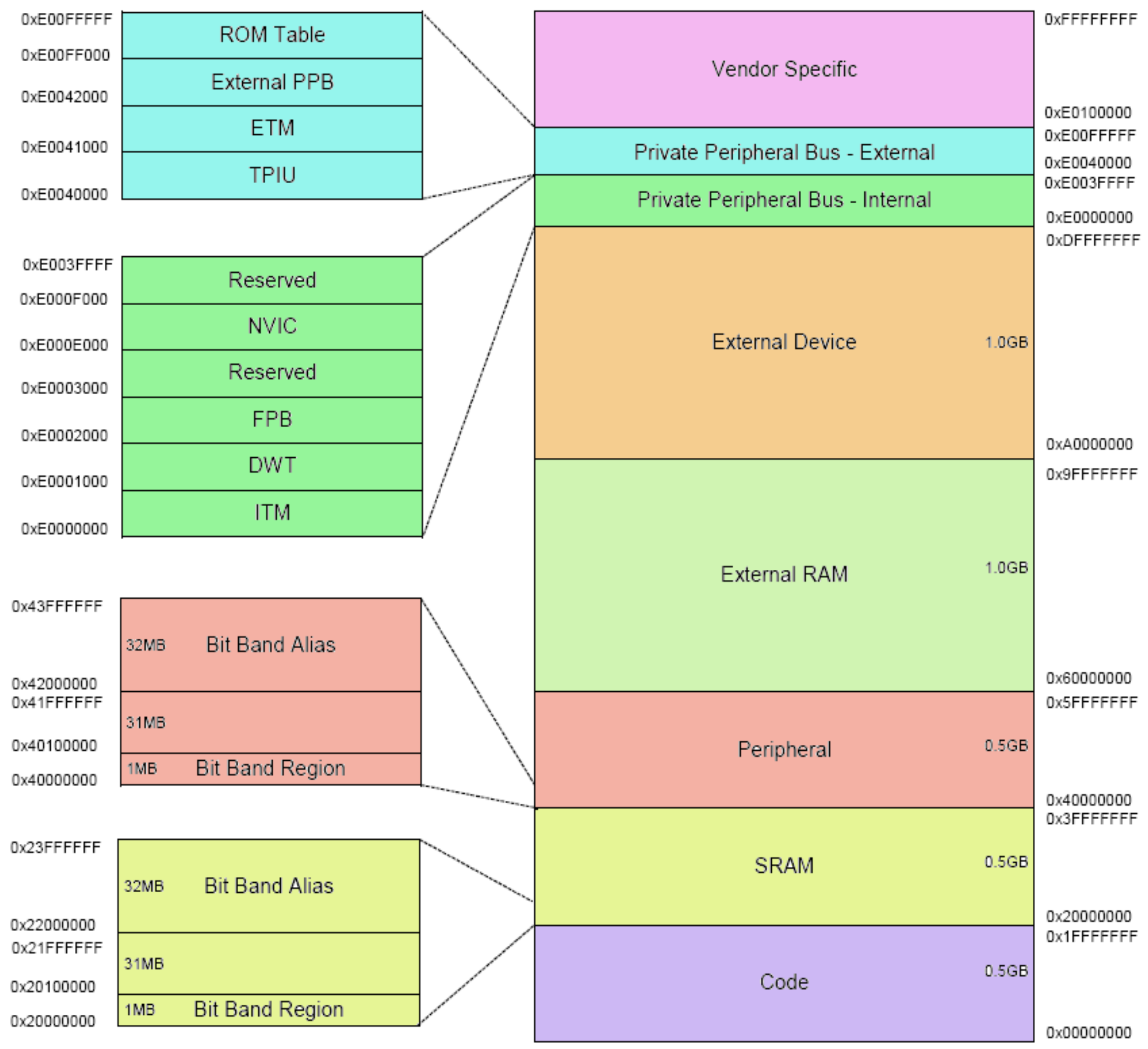
Cortex M Address Space

- 4 GB address space
  - (ROM/Flash for Code)
- Built-in and external RAM
- Built-in and external IO
  - Memory-mapped IO
  - Special Function Registers (SFRs) accessed by processor



0xFFFFFFFF
0xE0000000
System level
Private peripherals including build-in interrupt controller (NVIC), MPU control registers, and debug components
0xDFFFFFFF
0xA0000000
External device
Mainly used as external peripherals
0x9FFFFFFF
0x60000000
External RAM
Mainly used as external memory
0x5FFFFFFF
0x40000000
Peripherals
Mainly used as peripherals
0x3FFFFFFF
0x20000000
SRAM
Mainly used as static RAM
0x1FFFFFFF
0x00000000
CODE
Mainly used for program code. Also provides exception vector table after power up

Memory Map



0xE00FFFFF
ROM Table
0xE00FF000
External PPB
0xE0042000
ETM
0xE0041000
TPIU
0xE0040000
0xE003FFFF
Reserved
0xE000F000
NVIC
0xE000E000
Reserved
0xE0003000
FPB
0xE0002000
DWT
0xE0001000
ITM
0xE0000000
0x43FFFFFF
32MB
Bit Band Alias
0x42000000
0x41FFFFFF
31MB
0x40100000
1MB
Bit Band Region
0x40000000
0x23FFFFFF
32MB
Bit Band Alias
0x22000000
0x21FFFFFF
31MB
0x20100000
1MB
Bit Band Region
0x20000000
Vendor Specific
0xFFFFFFFF
0xE0100000
0xE00FFFFF
Private Peripheral Bus - External
0xE0040000
0xE003FFFF
Private Peripheral Bus - Internal
0xE0000000
0xDFFFFFFF
External Device
1.0GB
0xA0000000
0x9FFFFFFF
External RAM
1.0GB
0x60000000
0x5FFFFFFF
Peripheral
0.5GB
0x40000000
0x3FFFFFFF
SRAM
0.5GB
0x20000000
0x1FFFFFFF
Code
0.5GB
0x00000000

Cortex Memory Regions

- Program code located in code, SRAM or external RAM regions
  - Best to put program code in code region
    - instruction fetches and data accesses are carried out simultaneously on two separate bus interfaces

- SRAM memory (0x20000000 – 0x3FFFFFFF) : connecting internal SRAM
  - Executable: program can be copied and executed from this part of memory
  - Access through system interface bus

- On-chip peripherals (0x40000000 – 0x5FFFFFFF) is a 0.5 GB block intended for peripherals
  - Code cannot be executed from this region

Cortex Memory Regions

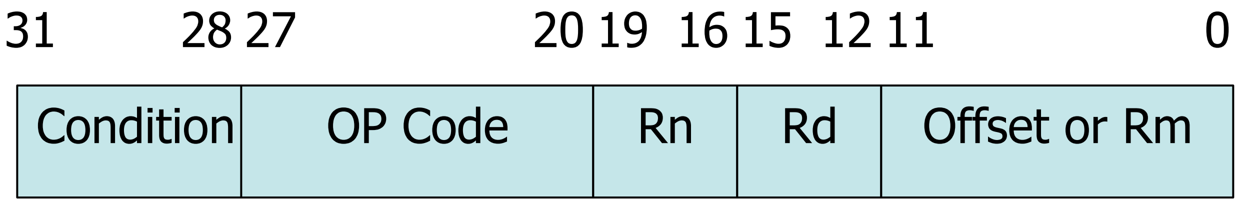
- Two slots of 1 GB memory space allocated for external RAM and devices
  - External RAM regions
    - intended for either on-chip or off-chip memory
    - Code can be executed in this region
  - External devices
    - intended for external devices and/or shared memory.
    - Non-executable region

- System region: 0.5GB memory (0xE0000000 – 0xFFFFFFFF)
  - System-level components
  - Internal peripheral buses & external peripheral bus
  - Vendor-specific system peripherals
  - Non-executable region

Instruction Set Architecture

- Machine code vs. assembly
  - Machine code is a string of bits that processors "read".
    - practically unreadable for humans
  - Assembly is intended for human reading
- In assembly the following instruction formatting is common:

```
  label opcode operand1, operand2, … comments
```



Assembly Language

```
label opcode operand1, operand2, … comments
```

- Label is optional
- Opcode is the actual instruction
- Depending on the instruction there can be two or mode operands
  - Usually, the first operand is the destination of the operation

Reset

- After the processor exits reset, it will read two words from memory:
    - Address 0x00000000: Starting value of R13 (the stack pointer)
    - Address 0x00000004: Reset vector (starting address of program execution)

- Initial value for the MSP is put at the beginning of the memory map
- Followed by the vector table (contains vector address values)

- The first item in the vector table (exception type 1) is the reset vector
    - Second piece of data fetched by the processor after reset.

Moving Data

- Basic function: transfer of data
  - Register to register
  - Memory to register
  - Special register to register
  - Moving an immediate data value into a register

- The command to move data between registers is MOV (move)
  - `MOV R8, R3`
  - `MVN R8, R3   ; (move negative of value in R8)`

- Basic instructions for accessing memory are Load and Store
  - LDR: from memory to registers; STR: from registers to memory
  - Transfers can be in different data sizes (byte, half word, word, double word)

Moving Data

- Multiple Load/Store operations can be combined into single instructions
  - LDM (Load Multiple) and STM (Store Multiple)
  - e.g. Store multiple words to memory location specified by *Rd*
  - `STMIA.W R8!, {R0-R3} ; R8=0x8000 changed to 0x8010 after`
  - `                     ; store (increment by 4 words)`
  - `STMIA.W R8 , {R0-R3} ; R8 unchanged after store`

- Memory accesses with pre-indexing and post-indexing.
  - Pre-indexing : register adjusted & updated address is used
  - `LDR.W R0,[R1, #offset]! ; Read memory[R1+offset], with`
    `                        ; R1 updated to R1+offset`
  - Post-indexing : register adjusted after address is used
  - `LDR.W R0,[R1], #offset ; Read memory[R1], with R1`
    `                       ; updated to R1+offset`

Moving Data

- Moving immediate data into a register is common
  - 8 bits: `MOV R0, #0x12 ; Set R0 to 0x12`
  - Larger values (Thumb 2): `MOVW.W R0,#0x789A ; Set R0 to 0x789A`
  - 32-bit can use two instructions to set the upper and lower halves:
  - `MOVW.W R0,#0x789A ; Set R0 lower half to 0x789A`
  - `MOVT.W R0,#0x3456 ; Set R0 upper half to 0x3456. Now`
    `; R0=0x3456789A`

Store Timing

```
STR Rd, [Rn, #offset]
```

```
Effect: Rd <-[[Rn]+offset]
```

- Execution takes always one cycle
  - Address generation is performed in the initial cycle
  - Data store is performed at the same time as next instruction executes

Data Processing

- Many data operation instructions can have multiple instruction formats.
  - ADD R0, R1 ; R0 = R0+R1
  - ADD R0, #0x12 ; R0 = R0 + 0x12
  - ADD.W R0, R1, R2 ; R0 = R1+R2

Arithmetic & Conditional Execution

- Mostly 3-operand (two inputs and output)

```
XYZ Rd, Rn, <Op2>
```

- Example:
  - ADD – add
  - ADC – add with carry
  - SUB – subtract, etc.

Examples:

`ADD R0, R1, R2;`	R0 = R1+R2, no flag change
`ADDS R0, R1, R2;`	as above, but CNZV affected
`ADDCSS R0, R1, R2;`	if C flag then ADDS

Branch Instructions

- Branch can be conditional or unconditional
- Syntax

```
B [cond] label
```

- Example
  - `B label ; Branch to a labeled address`
  - `BX reg ; Branch to address specified by a register`

- Calling a function

  -
    ```
    BL label ; Branch to a labeled address and save
             ; return address in LR
    ```

  -
    ```
    BLX reg ; Branch to an address specified by a
            ; register and save return address in LR
    ```
  - Function can be terminated using BX LR
    - program control then returns to the calling process

Branch Instructions

- BL instruction destroys the current content of your LR register.
- If program code needs the LR register later, LR must be saved
- The common method is to push the LR to stack

```
main
    ...
    BL functionA
    ...
functionA
    PUSH {LR} ; Save LR content to
              ; stack
    ...
    BL functionB
    ...
    POP {PC} ; Use stacked LR content
             ;  to return to main
functionB
    PUSH {LR}
    ...
    POP {PC} ; Use stacked LR content
             ; to return to functionA
```

Conditional Branch Instructions

- Flags in the Current Program Status Register (CPSR) determine whether a branch should be carried out.

- Five flag bits: Four used for branch decisions

Flag	PSR bit	Description
N	31	Negative flag (last operation result is a negative value)
Z	30	Zero (last operation result returns a zero value)
C	29	Carry (last operation returns a carry out or borrow)
V	28	Overflow (last operation results in an overflow)

Conditional Execution Example

```
int gcd(int a, int b){
    while (a != b)
        if(a>b) a= a-b
            else b = b-a;
    return a;}
```

```
gcd     CMP     R0, R1;
        BEQ     end
        BLT     less
        SUB     R0, R0, R1
        B       gcd
less    SUB     R1, R1, R0
        B       gcd
end     …
```

```
gcd     CMP         R0, R1;
        SUBGT       R0, R0, R1; SUB if R1 > R0
        SUBLT       R1, R1, R0; SUB if R1 < R0
        BNE         gcd;
```

ARM Instruction Set - Summary

- All instructions are 32 bit long, many execute in one cycle
- Instructions can be conditionally executed
- Example: data processing instruction

```
 if (R0 equal R1) then {
       R3 = R4 + R5
       R3 = R3 / 2
 }
else {
       R3 = R6 + R7
       R3 = R3 / 2
}
```

```
CMP R0, R1 ; Compare R0 and R1
ITTEE EQ ; If R0 equal R1, Then-Then-Else-Else
ADDEQ R3, R4, R5 ; Add if equal
ASREQ R3, R3, #1 ; Arithmetic shift right if equal
ADDNE R3, R6, R7 ; Add if not equal
ASRNE R3, R3, #1 ; Arithmetic shift right if not equal
```

Thumb Instruction Set

- Thumb is a 16-bit instruction set
  - Optimized for code density from C code (~65% of ARM code size)
  - Improved performance from narrow memory
  - Subset of functionality of ARM instruction set

- Thumb is not a “regular” instruction set
  - Constraints are not generally consistent
  - Targeted at compiler generation, not hard coding

Assembler

- Utility program: translate assembly language into machine code
  - Translation is nearly isomorphic (one-to-one)
  - Assembly mnemonic statements $\Rightarrow$ machine instructions and data

  - Translation of a single high-level language instruction into machine code generally results in many machine instructions
  - Assembler may provide pseudo-instructions
    - expand into several machine language instructions
    - Example:
      - Assembly language does not support “branch if greater or equal” instruction
      - Assembler can provide pseudo-instruction that expands to “set if less than” and “branch if zero”

ARM Assembly Language

- Assemblers: nearly as powerful as high-level languages

- Assembly code: commands or directives (commands to the assembler tool rather than the processor)

Format:

```
{label}{instruction | directive | pseudo-instruction} {;
   comment}
```

Notes:

- Everything is optional
- Label must start the line (no spaces or tabs before)
- Spaces and tabs freely used otherwise
- Comments can’t spread over multiple lines

Example: Assembly Directives, cont.

- AREA directive instructs assembler to assemble a new code/data section
- Section: independent, named, indivisible chunks of code (manipulated by linker)
- Syntax `AREA sectionmane { , attr}{,attr}…`
  - sectionname: name given to the section. Any can be chosen

- CODE: keyword indicating that the section to follow is code
- EXPORT: keyword indicating functions, variables, etc. visible to external segment
- ENTRY: keyword indicating the entry point of a segment

Example: Assembly Directives

```
      AREA   PROGRAM, CODE
; takes 2 7-digit BCD numbers and places results in the first
; bcdadd(a, b)-> a' with a'= a+b for encoding below
; svxxD6 D5D4 D3D2 D1D0 where s=1 is negative, v=1 is
; overflow, and 7 4-bit BCD Di
; ARM Calling convention: arg1 and arg2 in R0 and R1,
; respectively. Result is in R0 & other registers unchanged.
; no memory used; return via LR register
; position-independent code, needs to be linked with C routine
      EXPORT bcdadd
      ENTRY
bcdadd ; labels entry point to subroutine & rest of code after
…
      END
```

Assembler Directives

- Important directives for data definition and storage
  - Symbolic constant (to be replaced throughout code): EQU
    - EQU directive gives symbolic name to:
      - numeric constant, a register-relative value or a PC-relative value
  - Variable: DCD, DCB
      - allocate one or more words of memory
      - defines the initial runtime contents of the memory
      - DCD: aligned on four-byte boundaries
  - Register variable: RN (define a name for a specified register)
  - Debug support: INFO, ASSERT
    - INFO supports diagnostic generation
    - ASSERT generates error message during assembly if assertion is false

Assembler Directives

- Examples:
  - Array DCD 1, 2, 3 ; array, not only 1 word
  - FailingGrade DCB “D, C-, C, or C+”, 0
  - INFO 0, “Version 1.0” ;reserves 10 bytes
  - PerfectNumber EQU 10 ; Pythagora ~500 B.C
  - LF EQU 10 ; linefeed in ASCII
  - Area RN R5 ; R5 holds var Area

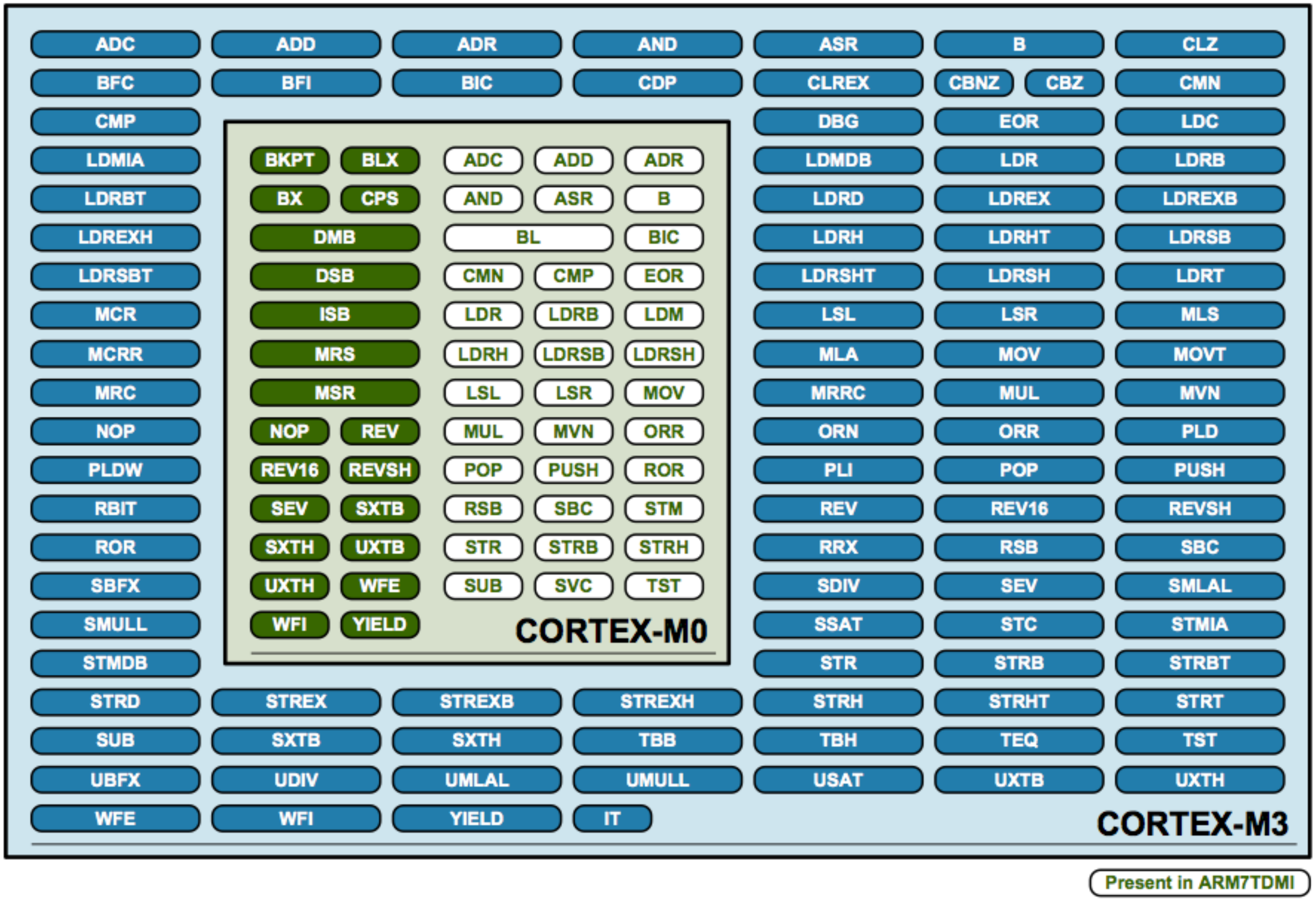
Cortex ISA and Assembler: Hints

- Use conditional execution -> avoid branches
  - Simpler code
  - Full pipeline

- Useful instructions for bit-field operations:

BIC, BFC	- sets some bits to zero
BFI	- inserts bit fields to a word
AND, ORR, EOR	- logical bit manipulations

Cortex M3 ISA at Glance



Cortex-M4 processors

Cortex
Low-Power Leadership from ARM

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

ADC ADD ADR AND ASR B CLZ
BFC BFI BIC CDP CLREX CBNZ CBZ CMN
CMP DBG EOR LDC
LDMIA LDMDB LDR LDRB
LDRBT LDRD LDREX LDREXB
LDREXH LDRH LDRHT LDRSB
LDRSBT LDRSHT LDRSH LDRT
MCR LSL LSR MLS
MCRR MLA MOV MOVT
MRC MRRC MUL MVN
NOP ORN ORR PLD
PLDW PLI POP PUSH
RBIT REV REV16 REVSH
ROR RRX RSB SBC
SBFX SDIV SEV SMLAL
SMULL SSAT STC STMIA
STMDB STR STRB STRBT
STRD STREX STREXB STREXH STRH STRHT STRT
SUB SXTB SXTH TBB TBH TEQ TST
UBFX UDIV UMLAL UMULL USAT UXTB UXTH
WFE WFI YIELD IT

BKPT BLX ADC ADD ADR
BX CPS AND ASR B
DMB BL BIC
DSB CMN CMP EOR
ISB LDR LDRB LDM
MRS LDRH LDRSB LDRSH
MSR LSL LSR MOV
NOP REV MUL MVN ORR
REV16 REVSH POP PUSH ROR
SEV SXTB RSB SBC STM
SXTH UXTB STR STRB STRH
UXTH WFE SUB SVC TST
WFI YIELD

CORTEX-M0/M1

CORTEX-M3

SMLALBT SMLALTB
SMLALTT SMLALD
SMLAWB SMLAWT
SMLSD SMLSLD
SMMLA SMMLS
SMMUL SMUAD
SMULBB SMULBT
SMULTB SMULTT
SMULWB SMULWT
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

Cortex-M4

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

Cortex-M4F

Embedded C

- C preprocessor
  - #define, #ifndef, #if, #ifdef, #else …etc.
    - #define specifies flags for conditional compilation
    - All remaining preprocessor statements initiate conditional compilation
      - Example: #ifdef compiles a block of code if some condition is defined in the #define statement
  - #ifndef is used to prevent multiple includes
    - Use #ifndef if you want to include a new definition
  - Widely used in firmware (embedded SW) drivers

C Preprocessor Examples

Inline macro functions:

```
#define MIN(n,m) (((n) < (m)) ? (n) : (m))

#define MAX(n,m) (((n) < (m)) ? (m) : (n))

#define ABS(n) ((n < 0) ? -(n) : (n))
```

Macro used to set LCD control
(## is used to actual arguments during macro expansion)

```
#define SET_VAL(x) LCD_Settings.P##x
```

Nested macro definitions

```
#define SET(x, val) SET_VAL(x) = val
#define DEF_SET(x) SET(x, DS_P##x)
```

Global variables

- Distinguish global variables from local by choosing appropriate naming convention
  - Example: RX_Buffer_Gbl
  - Stick to your convention throughout the program
- Use them as Software flags
  - Example: PACKET_RECEIVED – use capitals
- Have them all in ONE place
- Global variables are easy to observe during debug (“watch variables”)

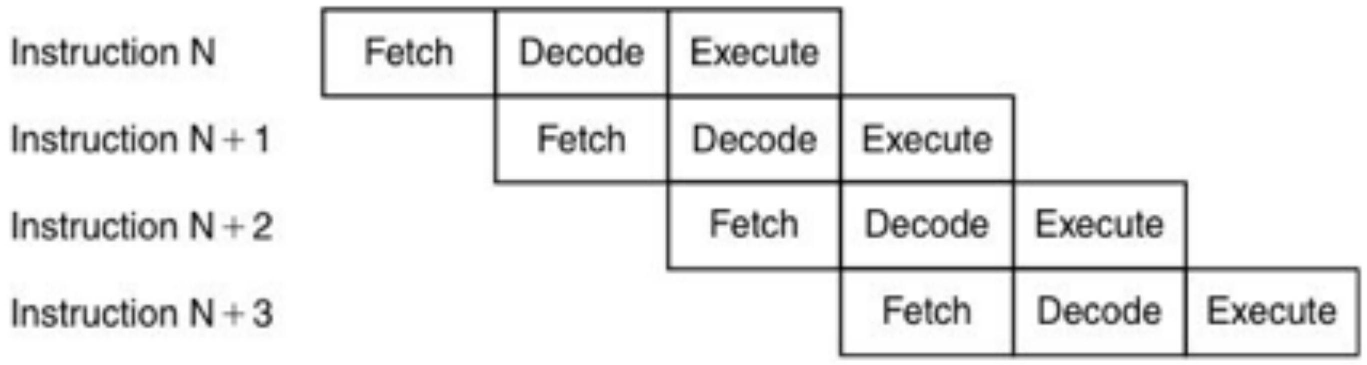
Cortex Pipeline

- CPU can execute most instructions in a single cycle
- Three-stage pipeline to increase flow rate of instructions

- Several operations to be performed simultaneously rather than serially
  - While the first instruction being executed,
  - the next instruction is being decoded
  - and the third is being fetched from memory

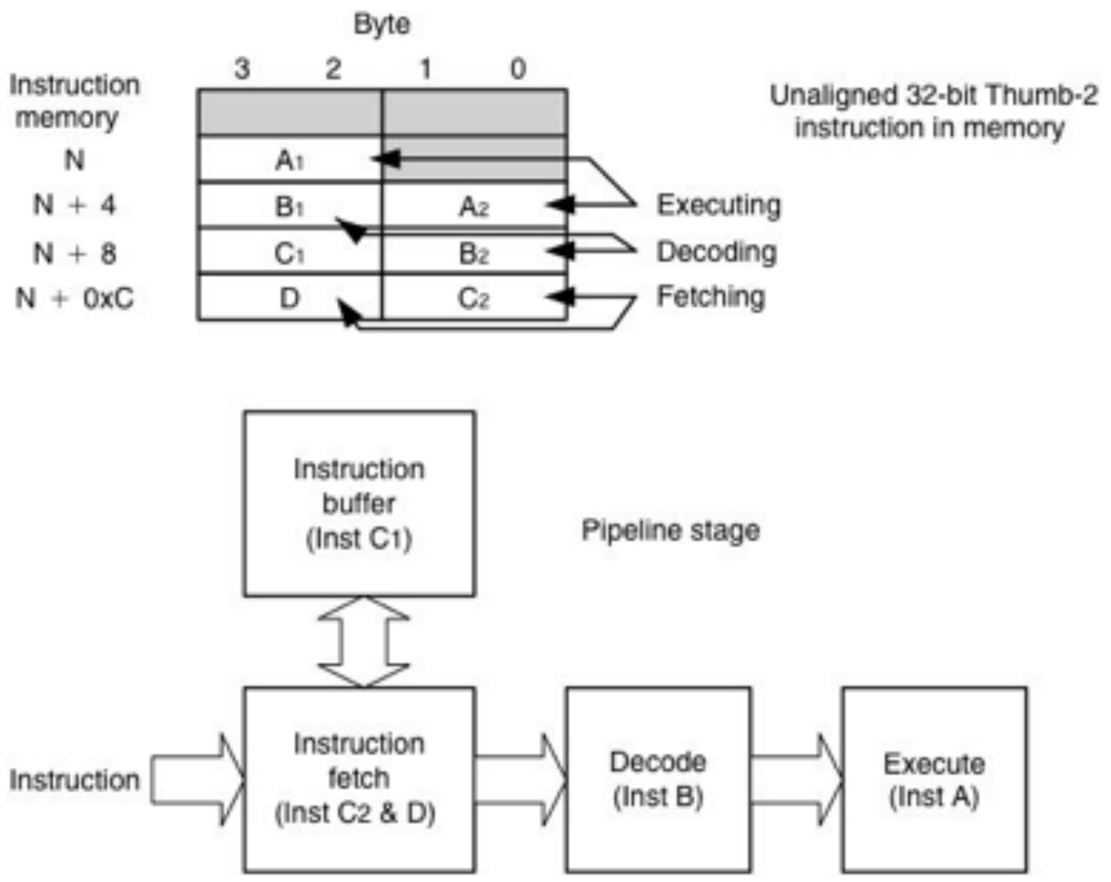
Cortex Pipeline

- PC points to the instruction being fetched, not executed
  - Debug tools will hide that from you
  - This is now part of the ARM Architecture and applies to all processors



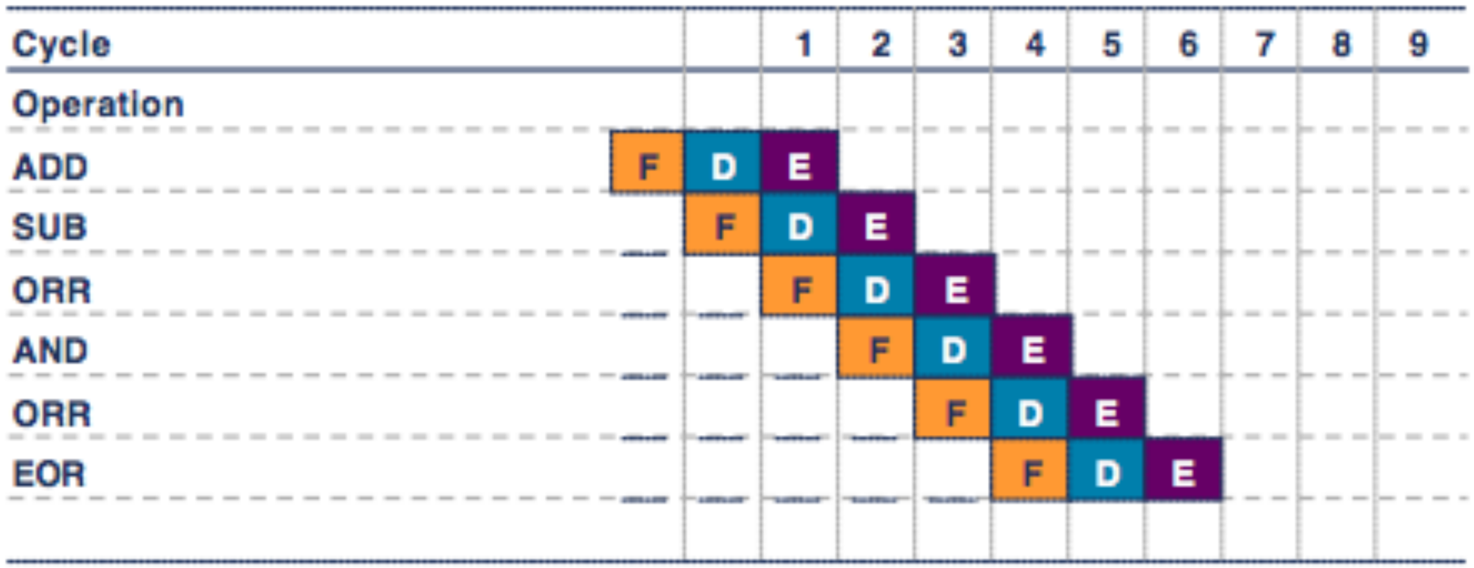
Cortex M Pipelining

- Three stages:
  - Fetch
    - Instruction is fetched
    - Data is returned from instruction memory
  - Decode
    - Generation of address using forward registers, branch forwarding
  - Execution
    - Instruction is executed in a single pipeline stage



Optimal Pipelining

- All operations are on registers (single cycle execution)
- Example: 6 clock cycles to execute 6 instructions
- Clock cycle per instruction (CPI = 1)



Cycle			1	2	3	4	5	6	7	8	9
Operation											
ADD	F	D	E								
SUB		F	D	E							
ORR			F	D	E						
AND				F	D	E					
ORR					F	D	E				
EOR						F	D	E			

F - Fetch D - Decode E - Execute