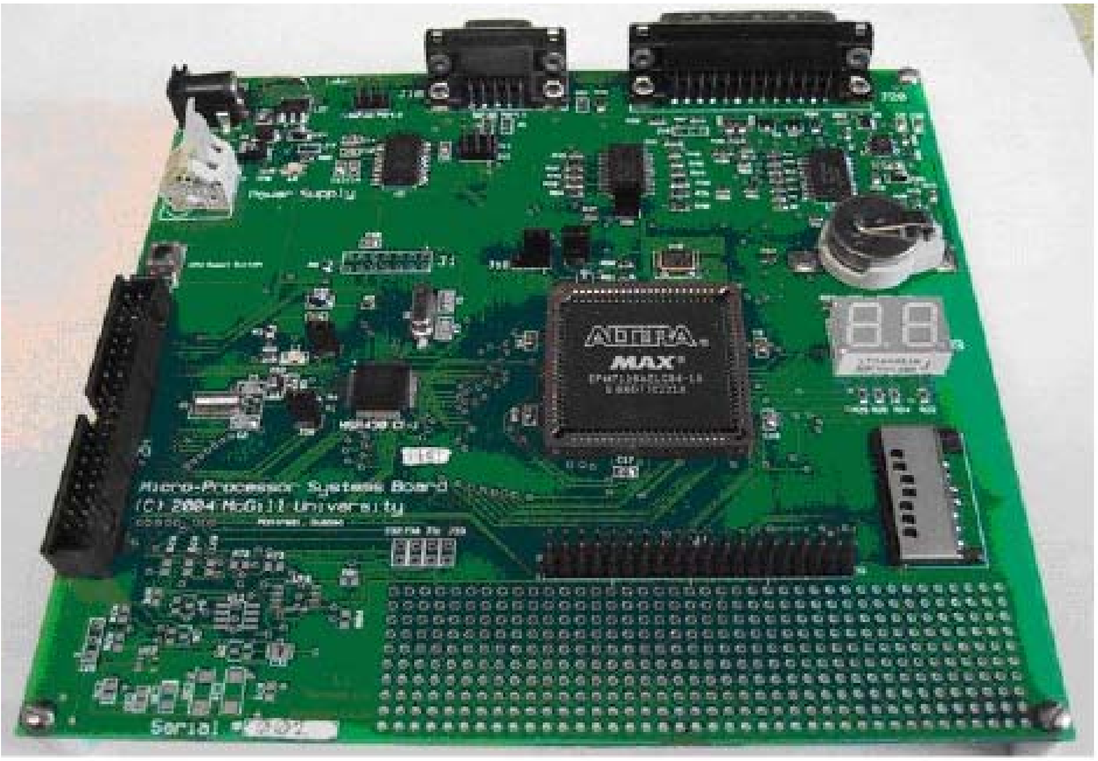
ECSE 426
ISA, Assemblers and Labs

Zeljko Zilic

Room 536
McConnell Building
zeljko@ece.mcgill.ca
www.macs.ece.mcgill.ca/~zeljko



Course Organization

- **Prerequisites:** ECSE-323 and EDEC 206
- **Instructor:** Prof. Zeljko Zilic, Rm. 546, McConnell
  398-1834, Fax: 398-4470, MyCourses
  e-mail: zeljko@ece.mcgill.ca
- **Office Hours**: Wed. 12:30-13:30; by appointment; online
- **TAs:** A. Suyyagh, M. Janidarmian, S. Ding, C. Dong
- **Tutorials**: As announced throughout semester (Thu slot)
- **Lab Demoing**: Thu and Fri
- **Manned Lab (TAs)**: Tue-Fri, as per calendar in MyCourses

Course Content

- Top-down approach to microprocessor programming and design
- Lectures focus on structured computer organization, and progress through “layers” :
  - Problem-oriented language (embedded C) + assembly language
  - Instruction set architecture
  - Microarchitecture
  - Hardware
- Introduction of design principles and impact on real-world architectures
  - ARM Cortex M4 Architecture and ARM Cortex M family in general

Acknowledgements: Hamacher, Vranesic, Zaky, Manjikian for “Computer Organization and Emb. Systems”. [HVZM11]

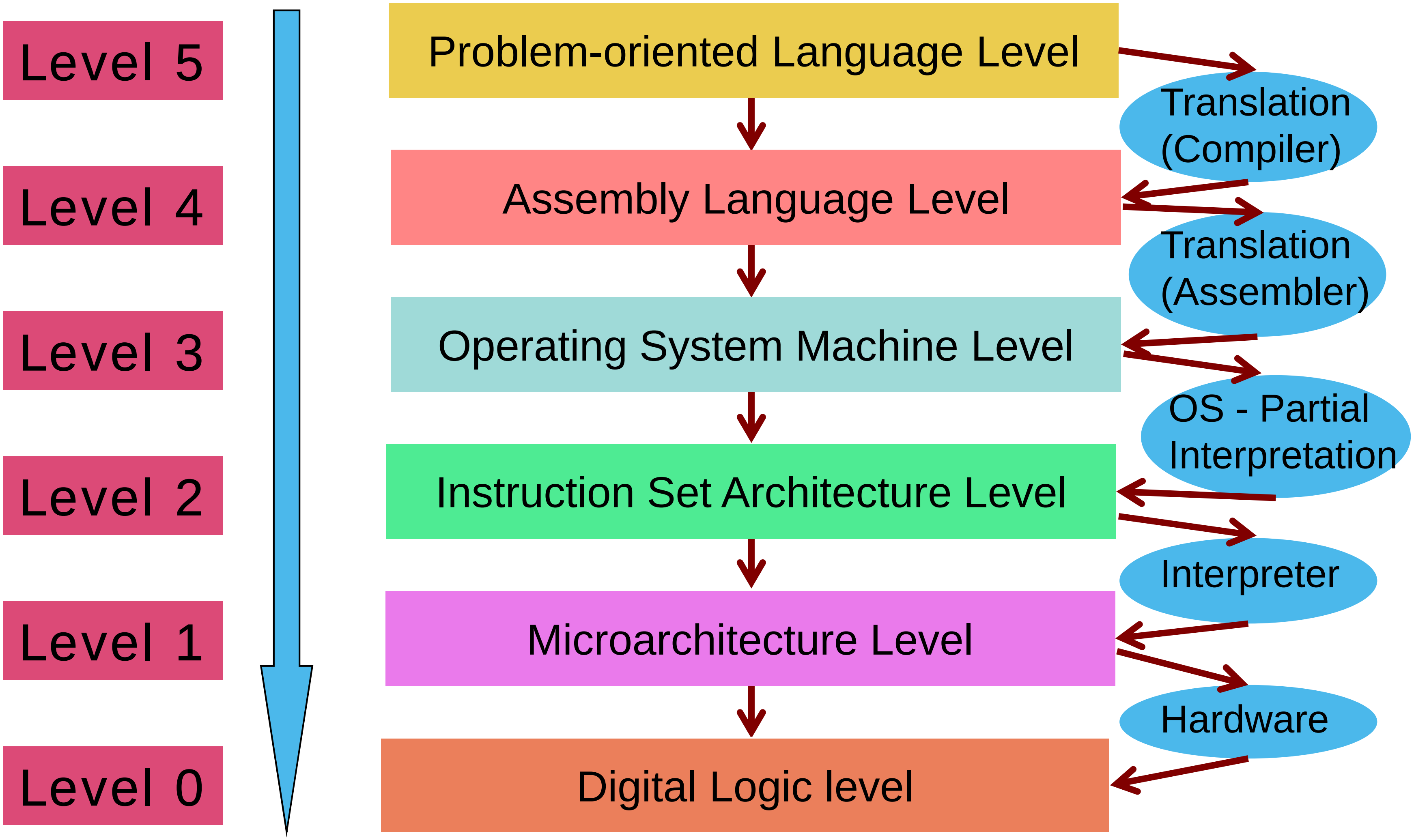
Course Objectives Rephrased

- Understand microprocessor-based systems
- Get familiar with basic tools
- Skills in machine interfacing, assembler and embedded C programming
- Design a sizeable embedded system
  - Previous projects: Music player, file swapping system, PDAs (with handwriting recognition), wireless data collection systems
- Build teamwork skills

Today’s Lecture

- ISA
  - Processor Resources
  - Instruction Types
  - Addressing Modes
- Assembly Language and Tutorial Complement
  - Introduction to the ARM Assembler
  - Lab 1 Overview and Hints

Contemporary Multilevel Machines



Level 5

Level 4

Level 3

Level 2

Level 1

Level 0

Problem-oriented Language Level

Translation (Compiler)

Assembly Language Level

Translation (Assembler)

Operating System Machine Level

OS - Partial Interpretation

Instruction Set Architecture Level

Interpreter

Microarchitecture Level

Hardware

Digital Logic level

Assembly versus C

- Q: Why use assembly if programmers thing in abstract ways similar to problem oriented languages like C, and further programmers may not know the a structure of processors to take most advantage from assembly and machine languages?
- A: Assembly languages provide:
  - Efficiency of compiled code
  - Source code portability
  - Program maintainability
  - Typical bug rates (say, per thousand lines of code)
  - 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

Problems with Low-level Languages

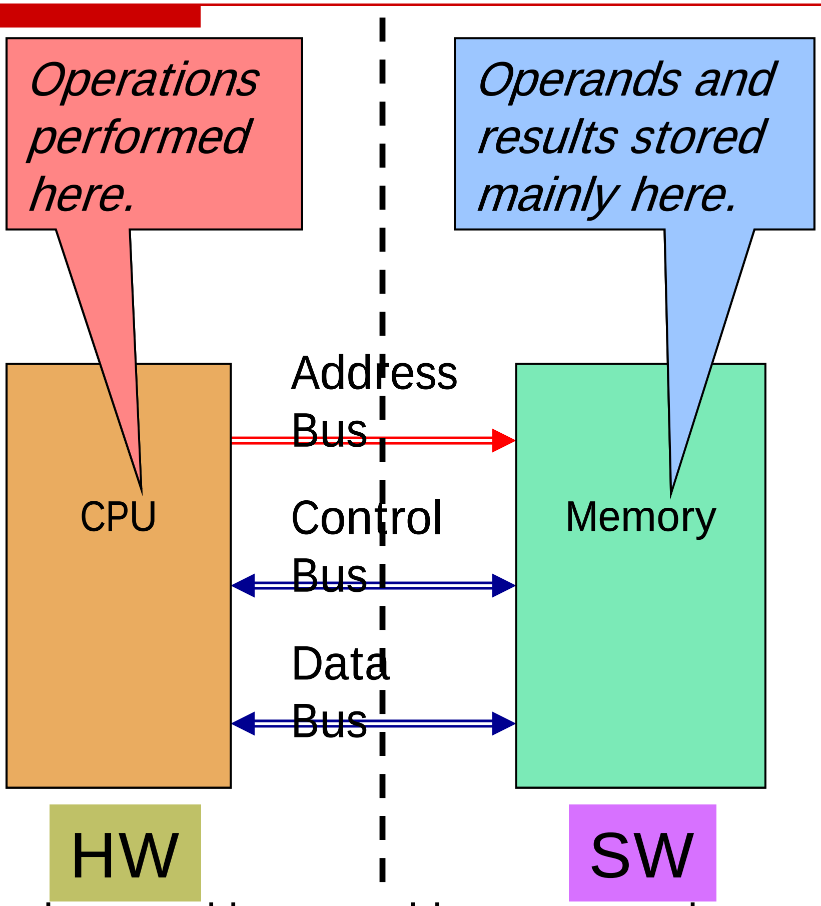
- Programs in low-level languages require detailed documentation, as otherwise they hard to read for people who were not involved in the process of the program creation
- Example: Company “Ostrich” has recently re-developed their embedded software for flagship products
  - Developed in assembly, 80 percent working, 2000 lines of code
- Suddenly it has been realized that the product is not 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?

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 the operation of the processor
  - 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 very inefficient, divide + square-root to be avoided

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 Processor – Brief History

- ARM was developed as Acorn Computers Ltd., Cambridge, England (1983-1985)

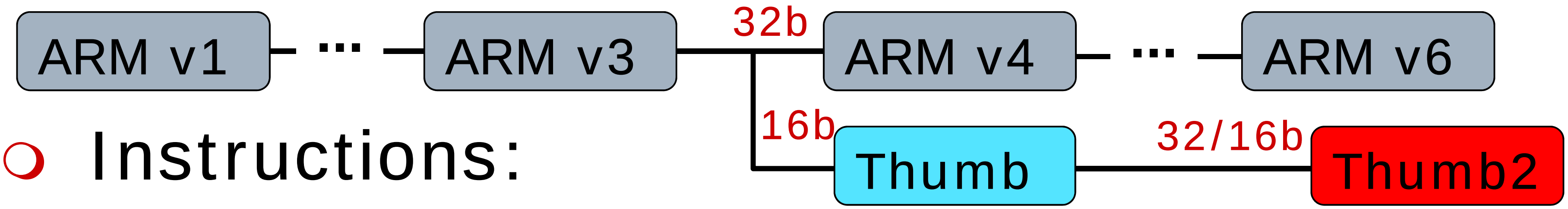
- RISC processor concept was introduced in 1980s
- Advanced RISC Machines
- ARM Ltd. - 1990

ARM Processor - Architecture

- A 32-bit Enhanced RISC processor with register-to-register, three operand instruction set
  - All operands are 32-bit wide
- Employs Load Store Architecture
  - Operations operate on registers and not in memory locations

ARM Cortex ISA: ARMv7-M

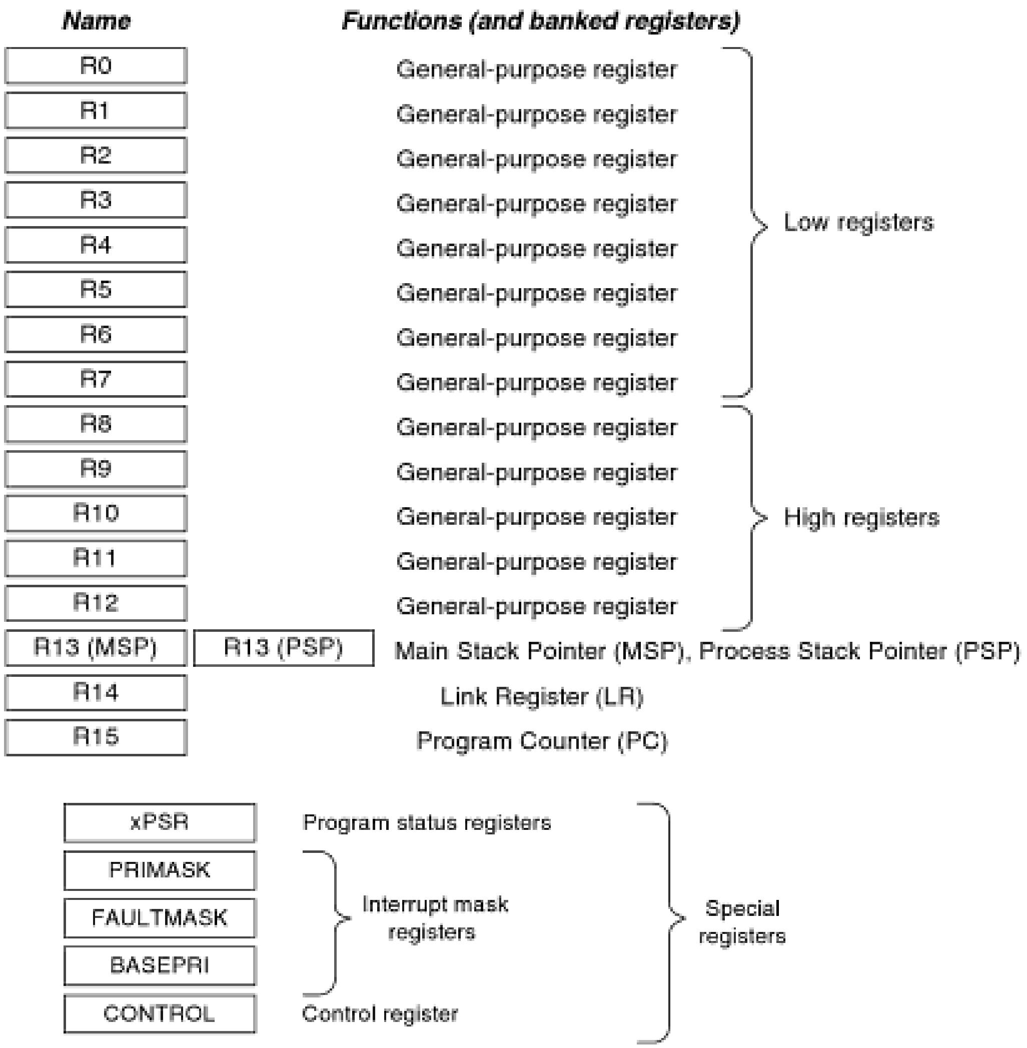
- Implements Thumb2 specification



- Instructions:
  - Data movement: single and multiple word
  - Data processing: arithmetic, logic, shift, rotate
  - Branches: conditional, subroutine, table (case)
  - State change: int. enable, spec. reg., ITE, SVC
  - Semaphores, Barrier, Hint, Prefetch
  - Coprocessor, floating-point, misc.

ARM Registers

- Large register set
  - 16 general purpose registers
  - Program Counter (R15)
  - Link Register (R14)
  - Stack Pointer (R13)
- Registers R0-R3 hold first 4 words of incoming argument



Name		Functions (and banked registers)	
R0		General-purpose register	Low registers
R1		General-purpose register	
R2		General-purpose register	
R3		General-purpose register	
R4		General-purpose register	
R5		General-purpose register	
R6		General-purpose register	
R7		General-purpose register	
R8		General-purpose register	High registers
R9		General-purpose register	
R10		General-purpose register	
R11		General-purpose register	
R12		General-purpose register	
R13 (MSP)	R13 (PSP)	Main Stack Pointer (MSP), Process Stack Pointer (PSP)	
R14		Link Register (LR)	
R15		Program Counter (PC)	

xPSR	Program status registers	Special registers
PRIMASK	Interrupt mask registers	
FAULTMASK		
BASEPRI		
CONTROL	Control register	

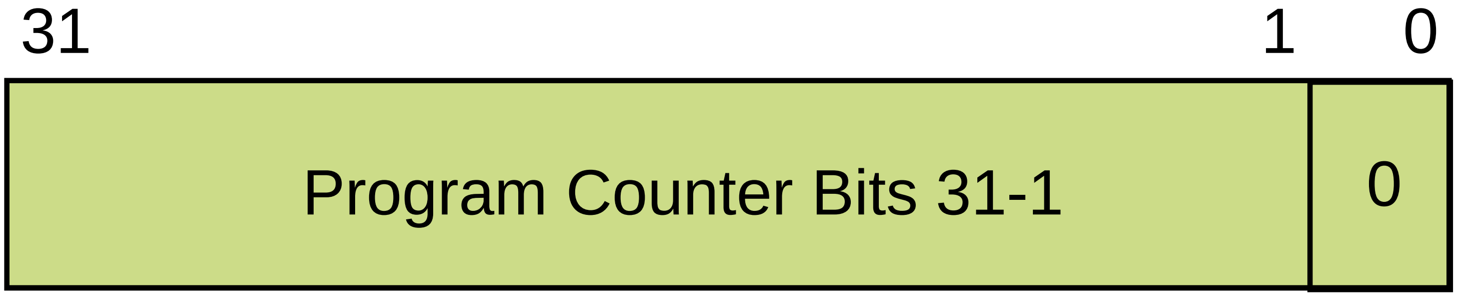
Summary of ARM Register Set

- 16 accessible 32-bit registers (R0-R15) at any time
  - R15 acts as Program Counter (PC)
  - R14 (Link Register) stores subroutine return address
- You can write in ARM assembly language:
  - PC for R15,
  - lr for R14,
  - sp for R13
- Current Program Status register (CPSR) that holds conditional codes
- Some registers are not unique as processor exceptions create new instances of R13 and R14
- As return address is not necessarily saved on the stack by a subroutine call, ARM is very fast at implementing subroutine return calls

Program Counter – R15

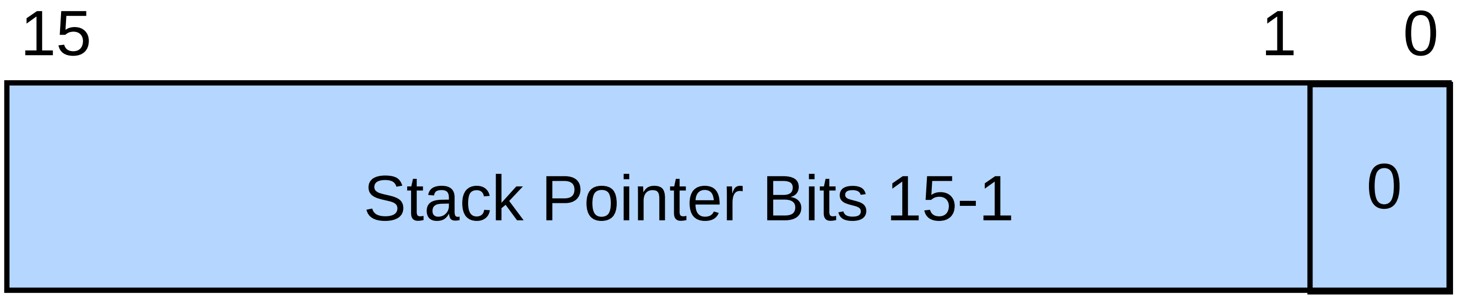
- Tells you where you are in the program
  - Next instruction address (pipelining blurs that)
- 32 bits, aligned to even addresses

LDR PC, =LABEL	Branch to address in LABEL
MOV PC, R4	Branch to address contained in R4
MOV R0, PC	Not recommended, processor-dependent



Stack Pointer

- Used by CPU to store return addresses of calls and interrupts
- Don’t push and pop the SP and PC !
- SP can also be used by software.
- Initialized into RAM by user, aligned to even addresses



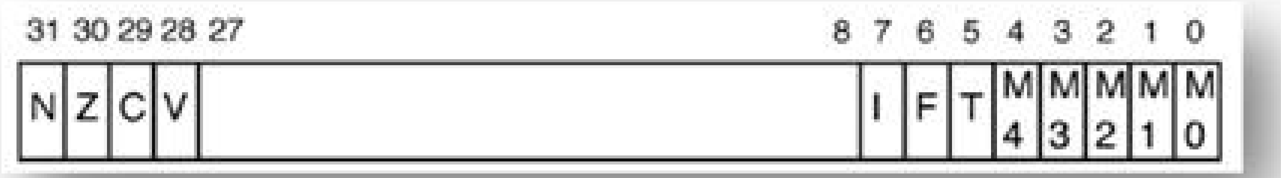
15 1 0

Stack Pointer Bits 15-1 | 0

Current Program Status Register (CPSR)

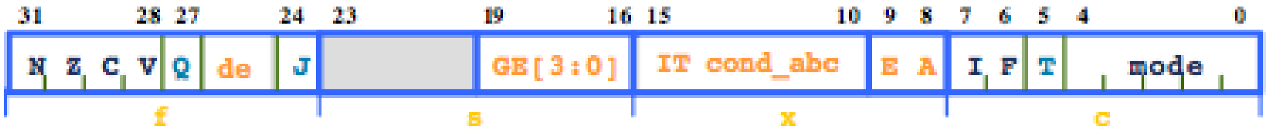
- CPSR stores the status of current execution
- ARM has more than one CPSR
- In normal mode of operation CPSR contains current values of conditional code bits N, Z, C, and V, and 8 system status bits
- When interrupt occurs, the ARM saves pre-exception values of CPSR in a stored Saved Program Status Register (SPSR)
  - There is one SPSR register for each of the interrupt mode
- SPSR keeps the status of program and processor

CPSR



- CPSR contains number of flags that report and control operation of CPU
- Condition code flags:
  - N – Negative Result from ALU
  - Z – Zero result from ALU
  - C – ALU operation carried out (carry from ALU operation)
  - V – ALU operation overflown
- 8 lower bits I, F, T anf M[4:0]
  - Interrupt Enable Bits
    - I = 1 - IRQ, interrupt prohibition
    - F =1 - FIQ interrupt prohibition
- T flag reflects the processor running
  - If T = 0 then processor in ARM Mode
  - If T = 1 then processor in THUMB Mode
- M[4:0] – operating mode bits
  - Determine the processor operating mode
- Bits 27-8 are reserved and when conditions change in the code PSR flag or control bits reserve bits do not change

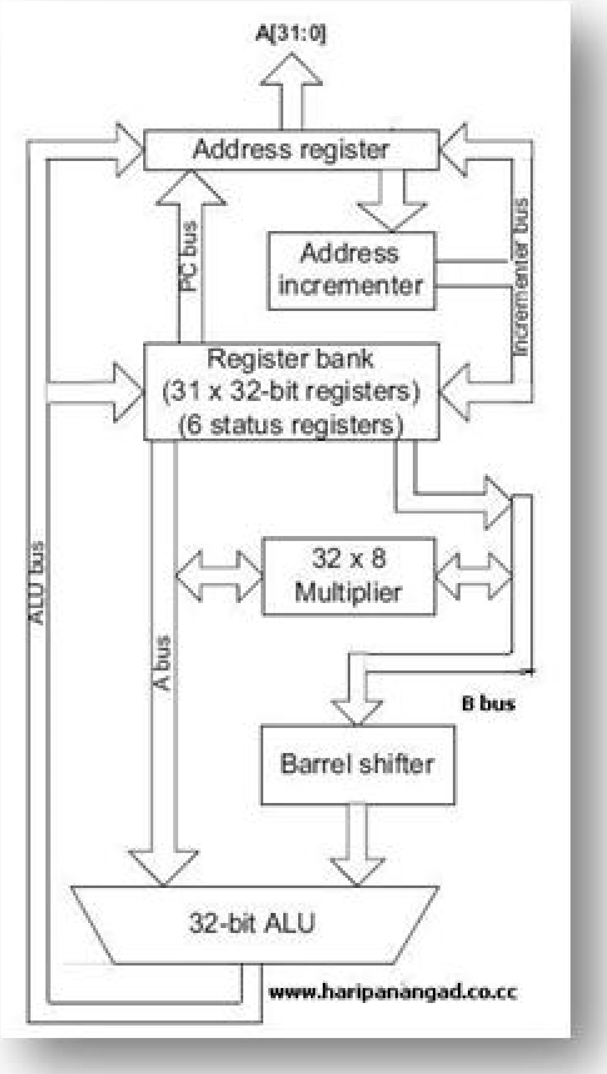
Program Status Register - Summary



- Condition code flags
  - N = Negative result from ALU
  - Z = Zero result from ALU
  - C = ALU operation Carried out
  - V = ALU operation oVerflowed
- Sticky Overflow flag - Q flag
  - Architecture 5TE and later only
  - Indicates if saturation has occurred
- J bit
  - Architecture 5TEJ and later only
  - J = 1: Processor in Jazelle state
- Interrupt Disable bits
  - I = 1: Disables IRQ
  - F = 1: Disables FIQ
- T Bit
  - T = 0: Processor in ARM state
  - T = 1: Processor in Thumb state
  - Introduced in Architecture 4T
- Mode bits
  - Specify the processor mode
- New bits in V6
  - GE[3:0] used by some SIMD instructions
  - E bit controls load/store endianness
  - A bit disables imprecise data aborts
  - IT [abcde] IF THEN conditional execution of Thumb2 instruction groups

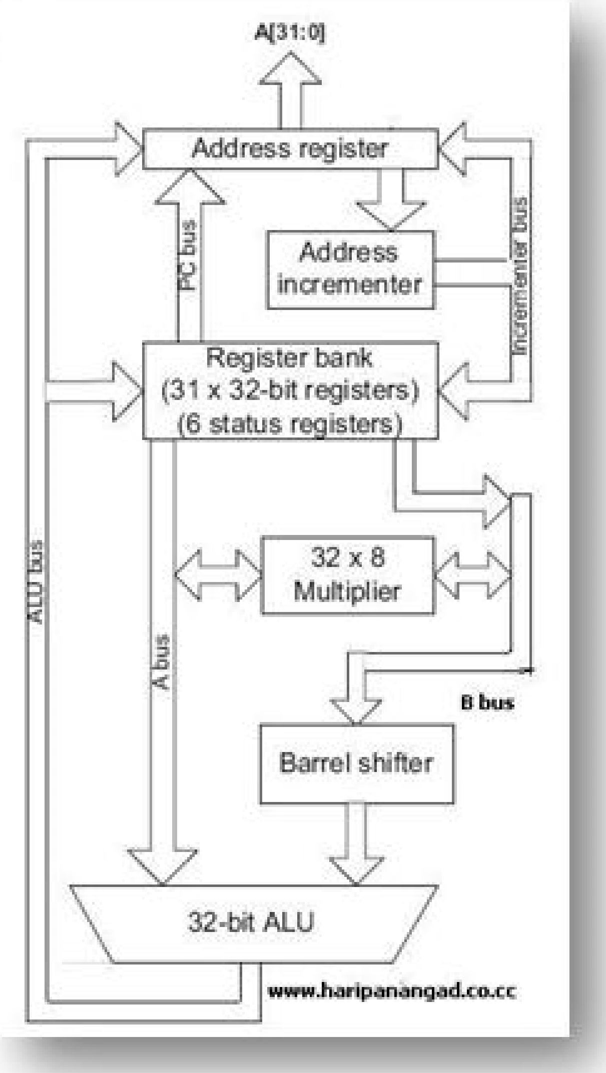
ARM – Core Datapath

- 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 and 1 destination register
- Barrel shifter preprocesses data before it enters ALU



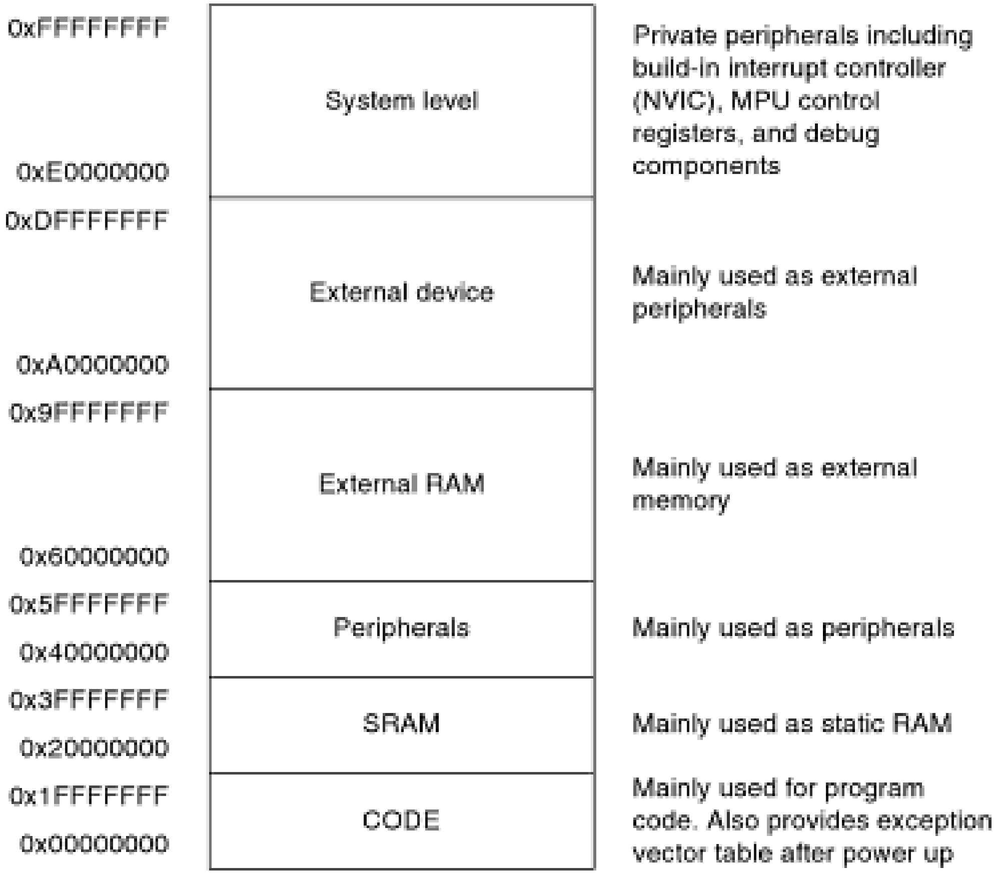
ARM Organization

- In order to be processed data has to be moved from memory location to central set of registers
  - After processing data is stored back in memory
- Register bank connected to ALU via two datapath busses 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 register bank responsible for generating addresses for next operation
  - Registers can handle both data and address
- Address Incrementer block increments or decrements register values independent of ALU



ARM Cortex M Address Space

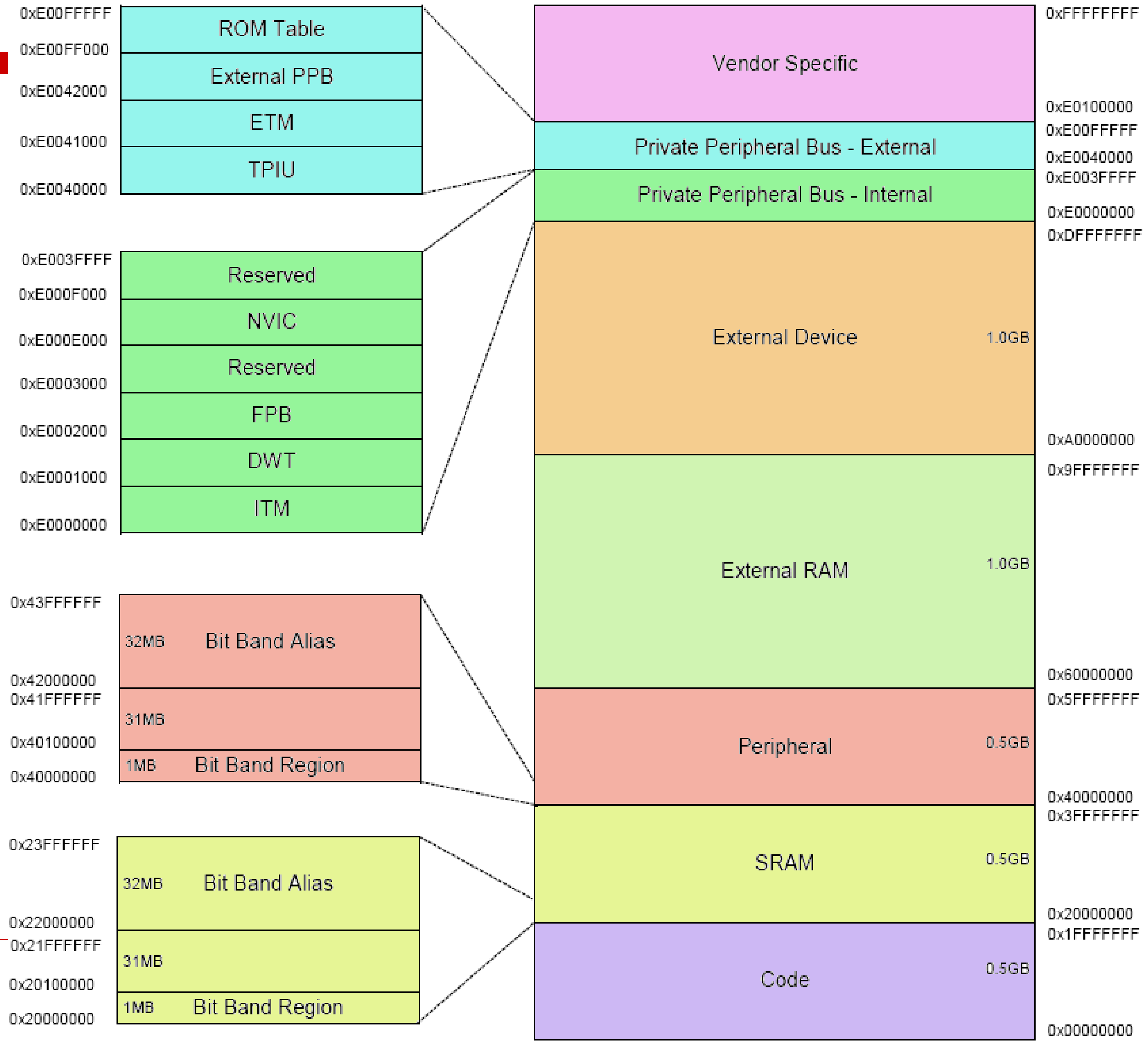
- 4 GB address space (ROM/Flash for Code)
- Built-in and external RAM
- Built-in and external IO
  - Memory-mapped IO
  - Special Function Registers (SFRs) accessed by processor



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

Map -More

- Fixed 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

0xFFFFFFFF
Vendor Specific
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

Memory Regions

- Program code located in code, SRAM or external RAM regions
  - Best to put program code in code region as then the instruction fetches and data accesses are carried out simultaneously on two separate bus interfaces
- SRAM memory (0x20000000 – 0x3FFFFFFF) is for connecting internal SRAM
  - Regions is executable, hence a program can be copied here 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

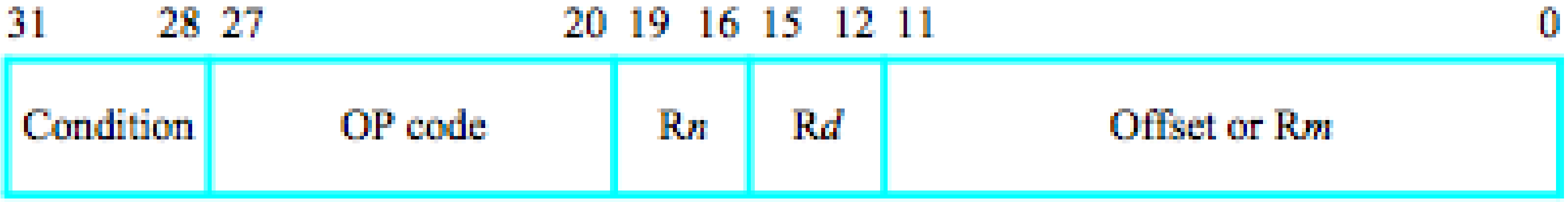
Memory Regions

- Two slots of 1 GB memory space are allocated for external RAM and external devices
  - External RAM regions (0x60000000 – 0x7FFFFFFF) and (0x80000000 – 0x9FFFFFFF) intended for either on-chip or off-chip memory. Code can be executed in this region
  - External devices (0xA0000000 – 0xBFFFFFFF) and (0xC0000000 – 0xDFFFFFFF) intended for external devices and/or shared memory. This is a non-executable region
- System region (0xE0000000 – 0xFFFFFFFF) – a 0.5 GB memory for system-level components, internal peripheral buses, external peripheral bus and vendor-specific system peripherals
  - It is a non-executable region

ISA – Machine Code

- Machine code vs. assembly
  - Machine code is a string of bits that processors “read”. It is practically unreadable for humans
  - Assembly is intended for human reading
- In assembler the following instruction formatting is commonly used:

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



31 28	27 20	19 16	15 12	11 0
Condition	OP code	R*n*	R*d*	Offset or R*m*

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

ARM Cortex ISA: ARMv7-M

- Instructions:
  - Data movement: single and multiple word
  - Data processing: arithmetic, logic, shift, rotate
  - Branches: conditional, subroutine, table (case)
  - State change: int. enable, spec. reg., ITE, SVC
  - Semaphores, Barrier, Hint, Prefetch
  - Coprocessor, floating-point, misc.

Addressing Modes

- Addressing mode describes manner in which data in CPU registers and memory is accessed
  - During program execution data can reside either in:
    - Machine instruction itself
      - Displacement
    - CPU registers
    - Main memory
    - Secondary storage
      - Access to this type of data not direct
- Three addressing modes are generally supported by assembly language:
  - Data register direct addressing
  - Address register direct addressing
  - Address register indirect with displacement addressing
- Accessing data
  - When data ***is*** part of instruction then it can be accessed immediately
    - Immediate data (addressing scheme)
  - When data ***is not*** part of instruction then we have three classes of data access:
    - Register addressing scheme
    - Memory addressing scheme

Instruction Operands

- Instruction operand can be:
  - ARM register
  - Constant
  - Instruction-specific parameter
- Many instructions are two-operand
  - Many data-processing instructions have flexible second operand Operand2, which can be
    - Constant
    - Register

Operand2 as a Constant

- Constant Operand2 is specified in form:
  - #constant
  - Where #constant can be
    - Any constant that can be produced by shifting 8-bit value left by any number of bits within a 32-bit word
    - Any constant of the form 0x00XY00XY
    - Any constant of the form 0xXY00XY00
    - Any constant of the form 0xXYXYXYXY
      - X and Y are hex numbers

Load (LDR) and Store (STR)

- LDR instruction loads register with value from memory
- STR instruction store register value into memory
- Memory address to load from or to store to is an offset from register Rn
  - Offset is specified by register Rm
- Value to load or store can be a byte, halfword or word

ISA: Data Movement Instructions

Load, Store and Move to register

```
LDR Rd, EA ; EA=effective address
STR Rd, EA
MOV Rd, Rm
```

Major Variations:

- MVN: Move negative
- LDM, STM: Load and Store multiple
- PUSH, POP: stack (shortcut for LDM, STM via SP)
- Conditional; doubleword; exclusive

EA: Basic Addressing Mode

- Pre-indexed – effective address (EA) of operand: sum of base register and signed offset
  - Operator "[]" denotes indirection in reference (to memory)
- Example:

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

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

```
LDR Rd, [Rn, Rm]
```

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

Store Timing

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

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

- Execution takes always one cycle
  - Address generation is performed in the initial cycle, and data store is performed at the same time as the next instruction is executing

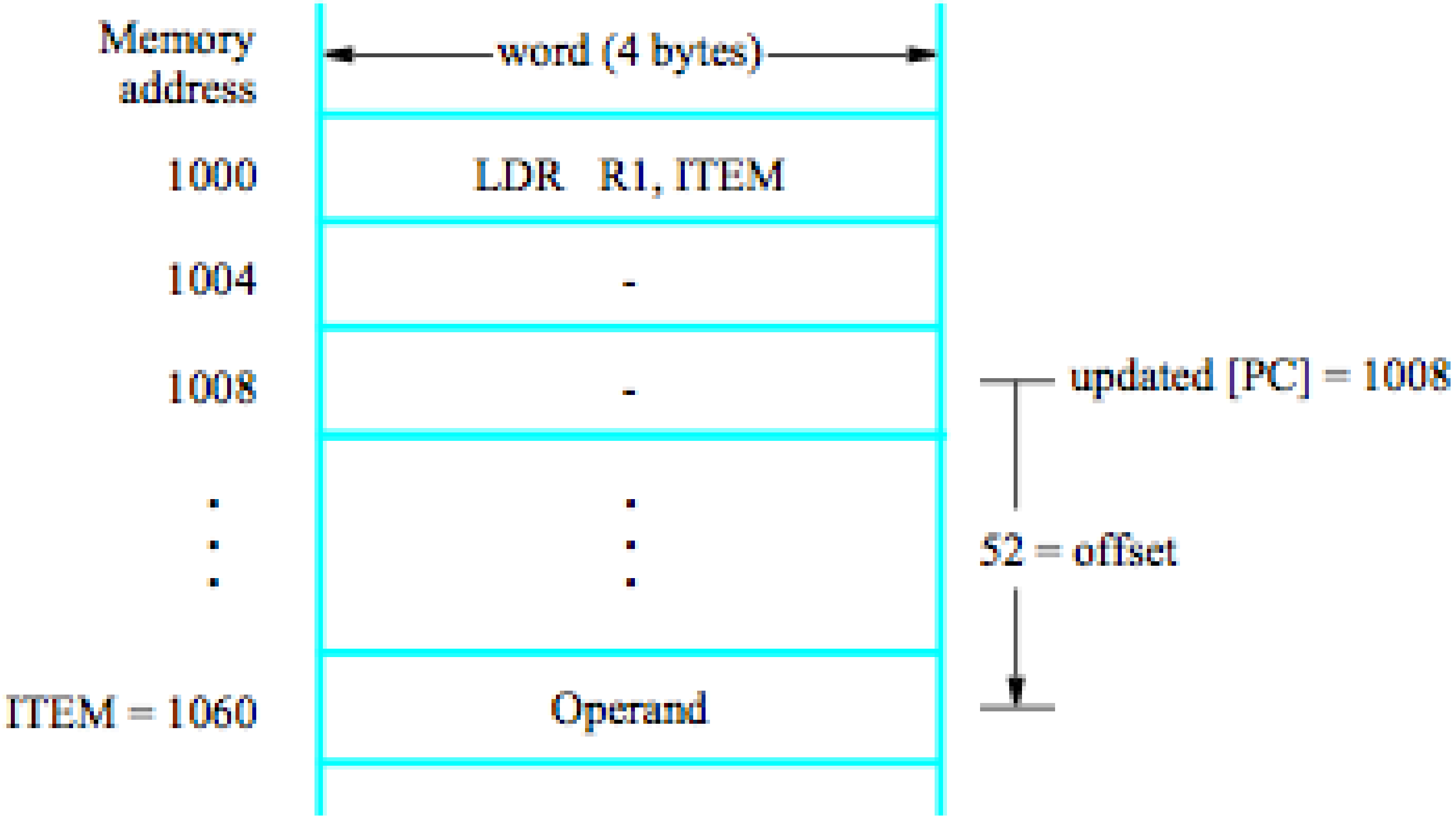
Rationale for Addressing Modes

- Instructions have fixed length, and they have to accommodate different addressing modes with different length of addresses
- EA derived from register Rn and offset offs
- Offset: register Rm or immediate value #imm
- Indexed:
  - Pre: offset added to index reg. before op [Rn, offs]
  - Post: after op, reg. augmented for offset [Rn], offs
  - Pre with writeback: pre-ind. and update register [Rn, offs]**!**
- Relative to PC: EA=[PC]+#imm
- Also: inline shifting offset register - barrel shifter
- Example (ARM, no Thumb/Thumb2): `LDR R4, [R2, -R3, LSL #4]`**!**

  Result: R4 <- [[R2]-16x[R3]]; main operation

  R2 <- R2-16xR3 ; index reg. update
- Summary of [HVZM11] textbook

Example: LDR Instruction Relative to PC



Memory address

word (4 bytes)

1000 LDR R1, ITEM

1004 -

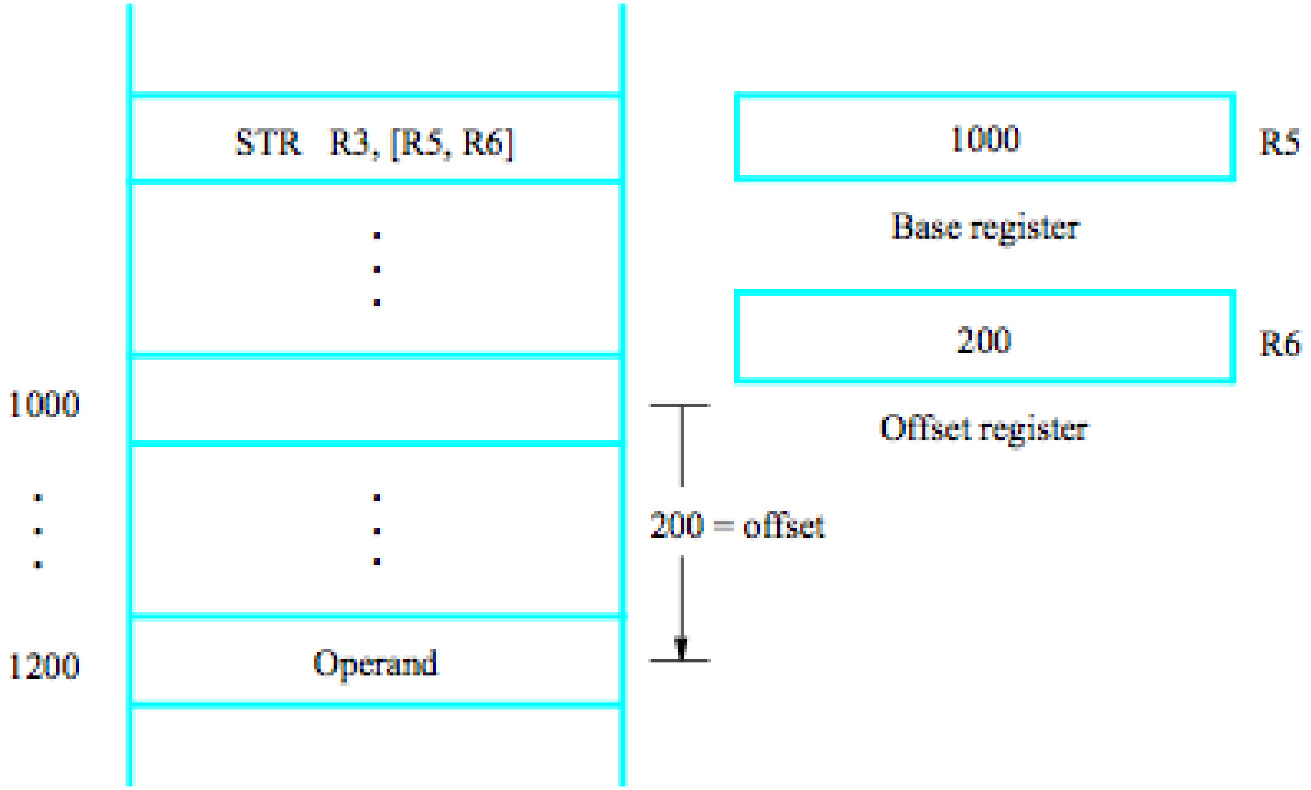
1008 -

updated [PC] = 1008

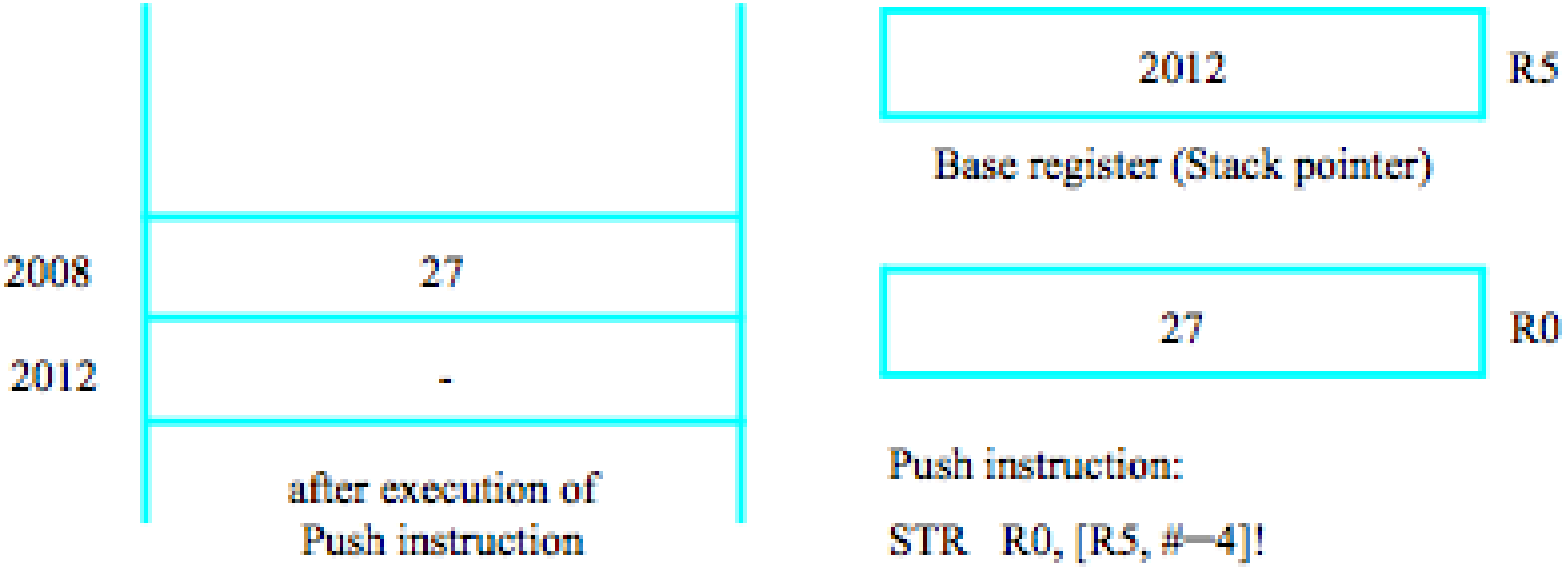
52 = offset

ITEM = 1060 Operand

Example: STR Instruction with Offset Addressing Mode



Example: STR Instruction with Post-decrement Addressing



Push instruction:

STR R0, [R5, #—4]!

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 – brach
  - BL – branch with link
  - BEQ – conditional branch

- Exampl

```
B   loopA  ;          branch to loopA
BEQ        target  ;          conditional branch to target
BX         LR      ;          return from function call
```

Conditional Execution Example

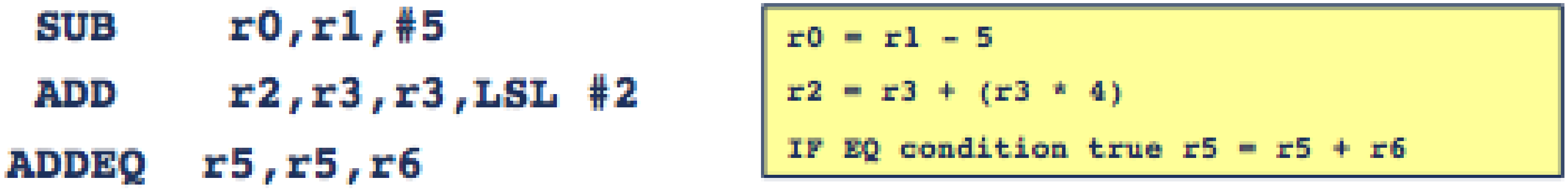
```
int gcd(int a, int b){
    while (a != b)
        if(a>b) a= a-b
            else b = b-a;
    return a;}
```

```
gcd     CMP     R0, R;
        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



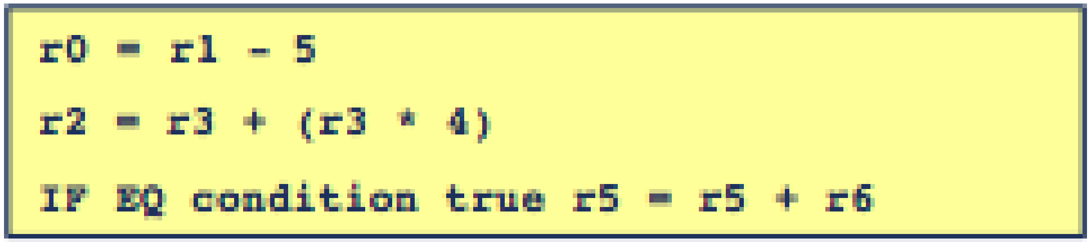
```
SUB     r0,r1,#5
ADD     r2,r3,r3,LSL #2
ADDEQ  r5,r5,r6
```

```
r0 = r1 - 5
r2 = r3 + (r3 * 4)
IF EQ condition true r5 = r5 + r6
```

ARM Instruction Set - Summary

- Example: branching instruction

```
B    <Label>
```

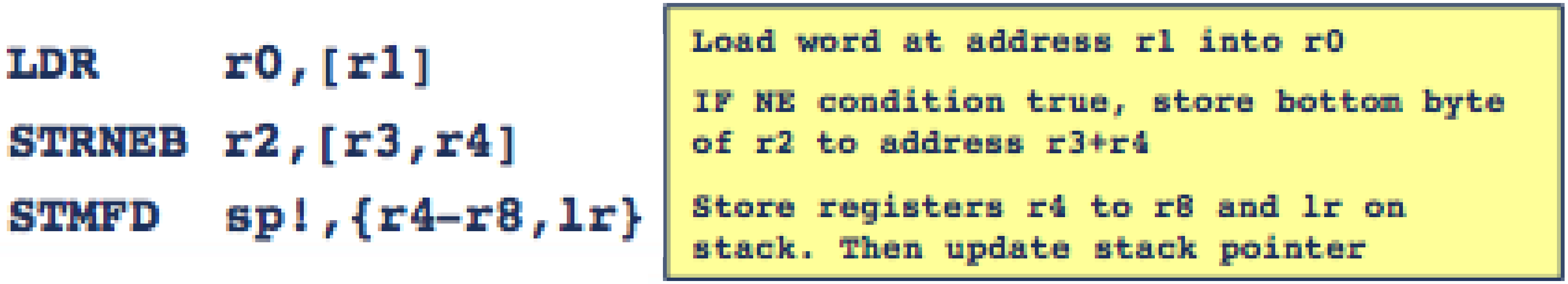


r0 = r1 - 5

r2 = r3 + (r3 * 4)

IF EQ condition true r5 = r5 + r6

- Example: memory access instructions



```
LDR     r0,[r1]
STRNEB  r2,[r3,r4]
STMFD   sp!,{r4-r8,lr}
```

Load word at address r1 into r0

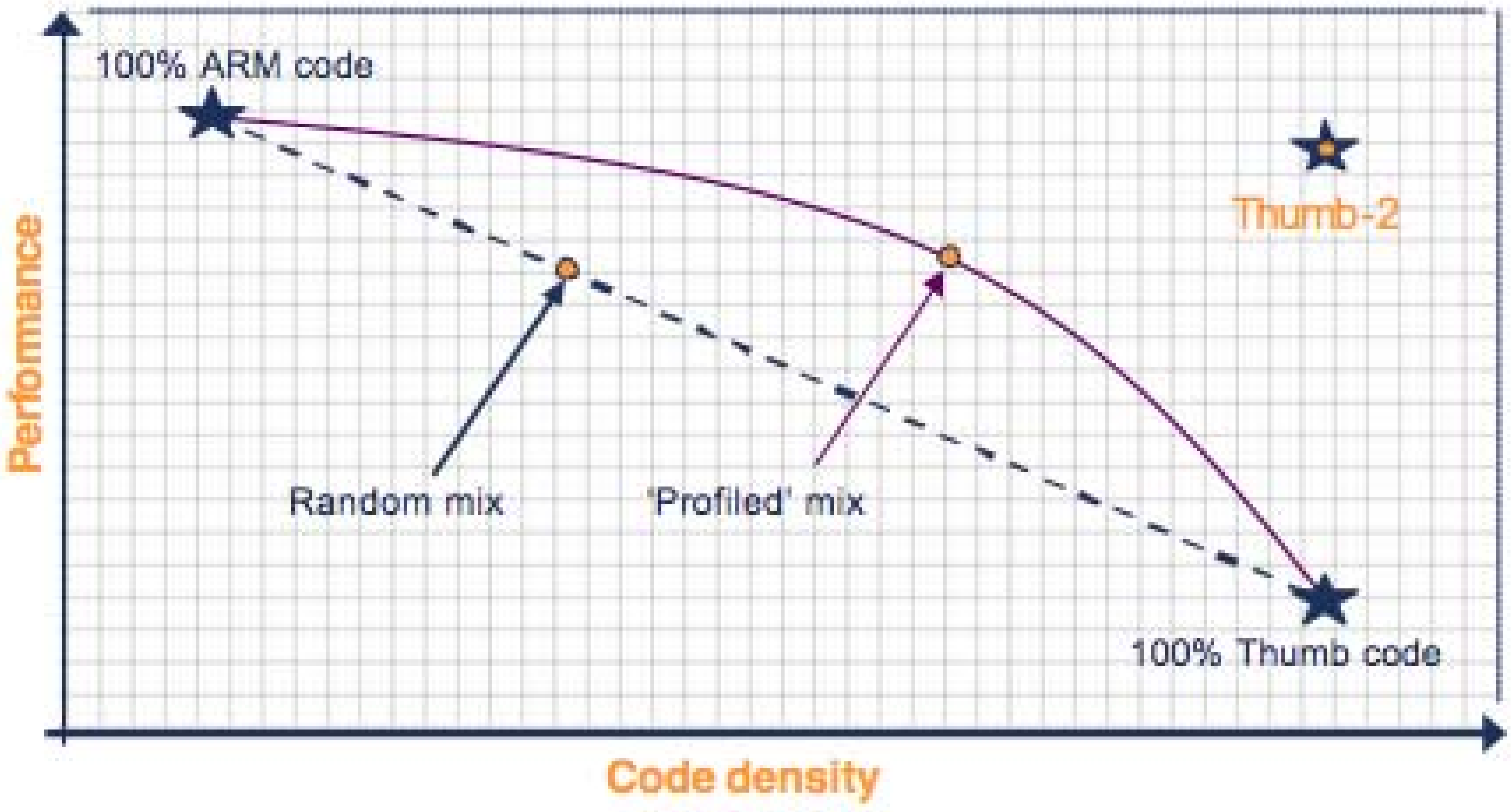
IF NE condition true, store bottom byte of r2 to address r3+r4

Store registers r4 to r8 and lr on stack. Then update stack pointer

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

Thumb Performance vs. Density



Assembler

- Assembler is a utility program used to translate assembly language into machine code
  - Translation is nearly isomorphic (one-to-one) from assembly mnemonic statements into machine instructions and data
    - Note that the translation of a single high-level language instruction into machine code generally results in many machine instructions
    - In some cases assembler may provide pseudoinstructions expanding into several machine language instructions upon translation
      - Example: If assembly language is not supporting “branch if greater or equal” instruction then assembler can provide a pseudoinstrution expanding to “set if les than” and “branch id zero”

Assembler - Pass 1

```
public static void pass_one( ) {                                        // This procedure is an outline of pass one of a simple assembler
boolean more_input = true;                                              // flag that stops pass one
String line, symbol, literal, opcode;                                   // fields of the instruction
int location_counter, length, value, type;                              // misc. variables
final int END_STATEMENT = -2;                                           // signals end of input
location_counter = 0;                                                   // assemble first instruction at 0
initialize_tables( );                                                   // general initialization
while (more_input) {                                                    // more_input set to false by END
        line = read_next_line( );                                       // get a line of input
        length = 0;                                                     // # bytes in the instruction
        type = 0;                                                       // which type (format) is the instruction
        if (line_is_not_comment(line)) { symbol = check_for_symbol(line);   // is this line labeled?
                if (symbol != null) enter_new_symbol(symbol, location_counter);   // if it is, record symbol and value
                literal = check_for_literal(line);                      // does line contain a literal?
                if (literal != null)  enter_new_literal(literal);       // if it does, enter it in table
                // Now determine the opcode type. -1 means illegal opcode.
                opcode = extract_opcode(line);                          // locate opcode mnemonic
                type = search_opcode_table(opcode);                     // find format, e.g. OP REG1,REG2
                if (type < 0)  type = search_pseudo_table(opcode);      // if not an opcode, is it a pseudoinstruction?
                switch(type) {                                          // determine the length of this instruction
                        case 1: length = get_length_of_type1(line); break;
                        case 2: length = get_length_of_type2(line); break;   // other cases here
                 }
        }
write_temp_file(type, opcode, length, line);                            // useful info for pass two
location_counter = location_counter + length;// update loc_ctr
if (type == END_STATEMENT) {                                            // are we done with input?
        more_input = false;                                             // if so, perform housekeeping tasks
        rewind_temp_for_pass_two( );                                    // like rewinding the temp file
        sort_literal_table( );                                          // and sorting the literal table
        remove_redundant_literals( );                                   // and removing duplicates from it
        }
}
```

Assembler - Pass 2

```
public static void pass_two( )                    // This is an outline of pass two of a simple assembler
boolean more_input = true;                        // flag that stops pass one of assembler
String line, opcode;                              // fields of the instruction
int location_counter, length, type;               // misc. variables
final int END_STATEMENT = -2;                     // signals end of input
final int MAX_CODE = 16;                          // max bytes of code per instruction
byte code[ ] = new byte[MAX_CODE];                // holds generated code per instruction
location_counter = 0;                             // assemble first instruction at 0
while (more_input) {                              // more_input set to false by END
    type = read_type( );                          // get type field of next line
    opcode = read_opcode( );                      // get opcode field of next line
    length = read_length( );                      // get length field of next line
    line = read_line( );                          // get the actual line of input
    if (type != 0) {                               // type 0 is for comment lines
      switch(type) {                              // generate the output code
          case 1: eval_type1(opcode, length, line, code); break;
          case 2: eval_type2(opcode, length, line, code); break;
                                                  // other cases here
          }
    }
write_output(code);                               // write the binary code
write_listing(code, line);                        // print one line on the listing
location_counter = location_counter + length;     // update loc_ctr
if (type == END_STATEMENT) {                      // are we done with input?
    more_input = false;                           // if so, perform housekeeping tasks
    finish_up( );                                 // odds and ends
    }
}
```

ARM Assembly Language

- Assemblers: nearly as powerful as high-level languages
  - ARM Assembler “feels” somewhat differently
- One (machine) command per line, but with ARM, could expand
- Assembly code: commands or directives (which are commands to assembler tool, rather than 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 multiple lines

Example: Assembly Directives

```
          AREA    PROGRAM, CODE
;subroutine that takes 2 7-digit BCD numbers and places results in the first one
; 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 in R0
;other registers unchanged. Need to be pushed to stack if changed, and pop-ed before return
; no memory used
;return via LR register
;position-independent code, needs to be linked with C routine
          EXPORT bcdadd
          ENTRY
bcdadd    ; label of the entry point to the subroutine, the rest of code comes after
...
          END
```

Example: Assembly Directives, cont.

- AREA directive instructs the assembler to assembly a new code or data section
- Syntax

```
    AREA sectionmane { , attr}{,attr}…
```

  - sectionname – name given to the section. Any can be chosen
- CODE is a keyword indicating that the section to follow is a code
- EXPORT is a keyword indicating functions, variables, etc. visible to external segment
- ENTRY is a keyword indicating the entry point of a segment

Assembler Directives

- Important directives are these for data definition and storage
  - Equivalent of high-level language
    - Symbolic constant (to be replaced throughout code): EQU
      - EQU directive gives symbolic name to a numeric constant, a register-relative value or a PC-relative value
    - Variable: DCD, DCB
      - DCD directive allocates one or more words of memory, aligned on four-byte boundaries
      - DCB directive allocated one or more bytes of memory, and defines the initial runtime contents of the memory
    - Register variable: RN
    - Debug support: INFO, ASSERT
- 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

Assembler Directives

- Important directives are these for data definition and storage
  - Equivalent of high-level language
    - Symbolic constant (to be replaced throughout code): EQU
      - EQU directive gives symbolic name to a numeric constant, a register-relative value or a PC-relative value
    - Variable: DCD, DCB
      - DCD directive allocates one or more words of memory, aligned on four-byte boundaries
      - DCB directive allocated one or more bytes of memory, and defines the initial runtime contents of the memory
    - Register variable: RN
    - Debug support: INFO, ASSERT
- 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

- Note that some ARM instructions dropped

Example: no LDR Rt, [Rn, Rm, LSL #imm]~~!~~ but LDR Rt, [Rn, #imm]! is OK

- no writeback with shift[ARMV7-M App. Level Ref. Manual, p. A7-291]

- Use conditional execution -> avoid branches
  - Simpler code
  - Full pipeline
- Useful instructions for bit-field operations:

BIC, BFC - sets some bits to zero

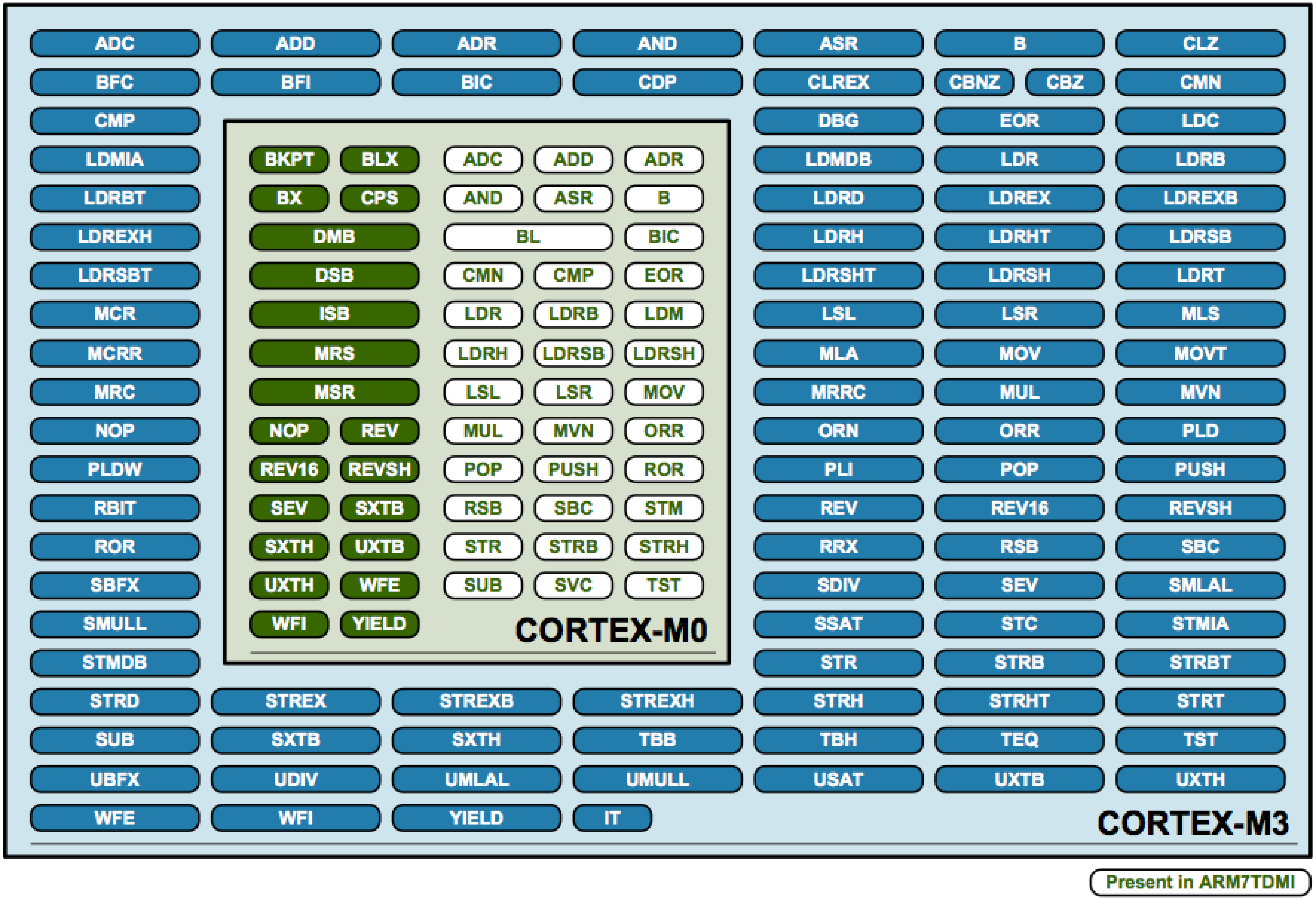
BFI - inserts bit fields to a word

STMDB, LDMIA - push & pop (decrement before, increment after)

AND, ORR, EOR- logical bit manipulations

- Rely on Keil tool and documentation posted on Web
  - Instruction Set Quick Reference Card

Cortex M3 ISA at Glance



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

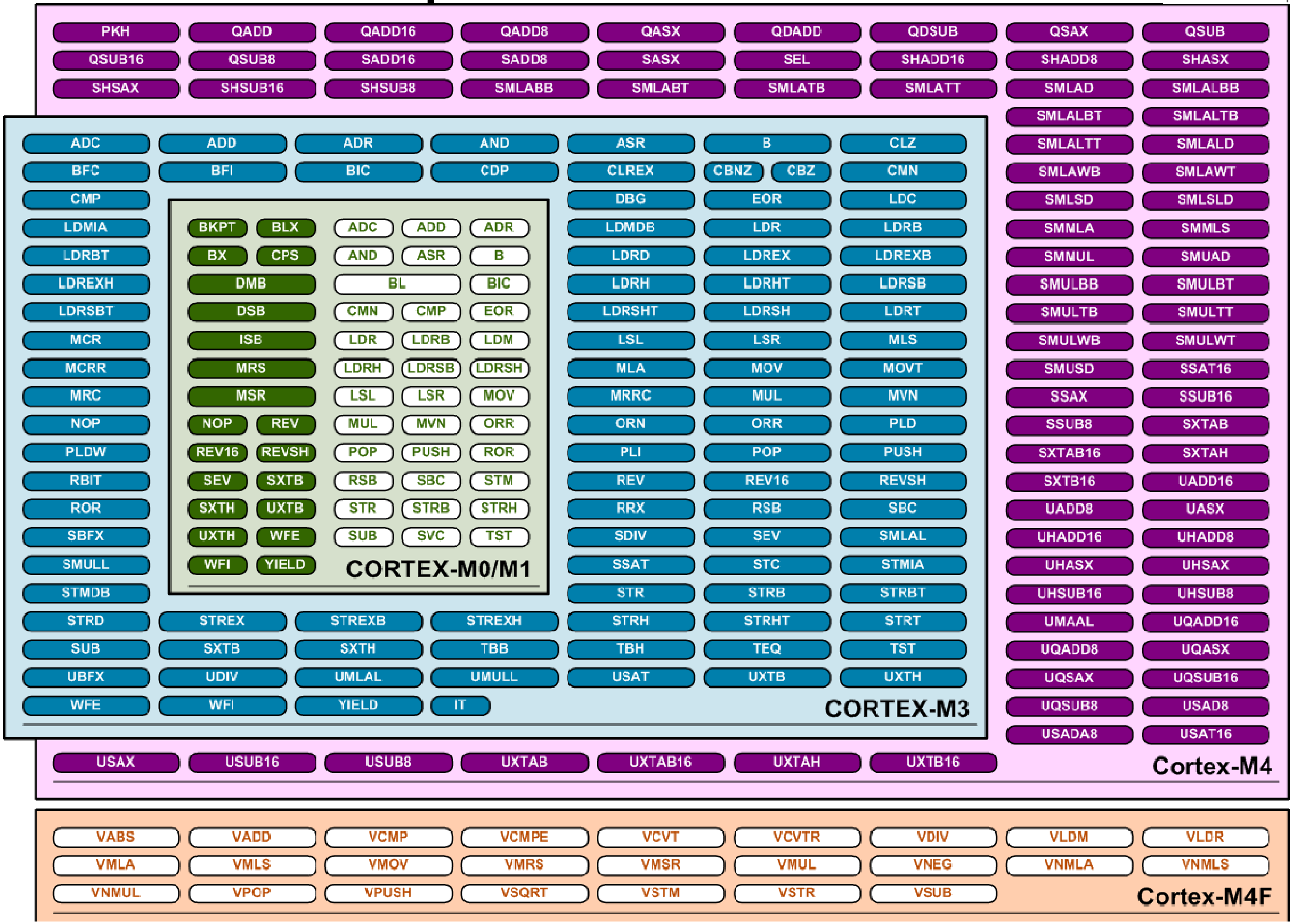
CORTEX-M0

CORTEX-M3

Present in ARM7TDMI

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
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

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

CORTEX-M3

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

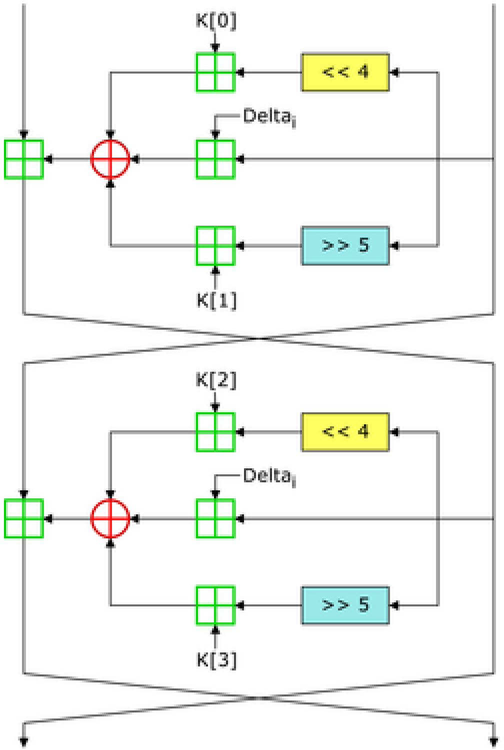
CORTEX-M0/M1

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

Lab 1: Encryption/Decryption

- Tiny Encryption Algoritm (TEA): Cambridge U. from 1994.
- 64 rounds of “mixing” the plaintext with 128-bit key
- Simple key schedule, suitable for embedded systems
    - Reasonably strong encryption
- Operations used:
    - Addition
    - XOR
    - Shift (left, right)



Lab 1: Tools

- Keil IDE: uVIsion
- Last version: Keil ARM MDK 4.7
- Procedure
  - Create project: assembler code only
  - When asked, include startup code
  - Edit startup code to branch to entry point
  - Add assembler code file, finish design
  - Compile/build and simulate