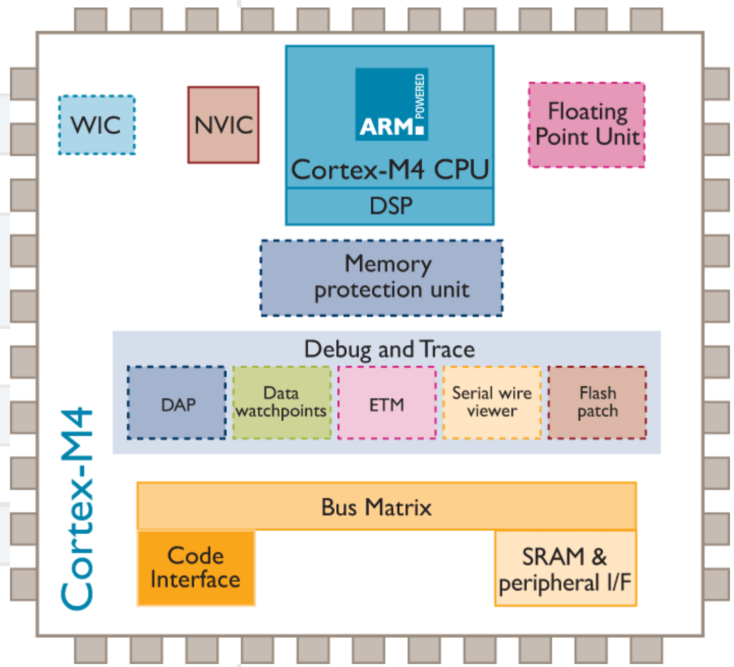
ARM Cortex-M3/M4 Instruction Set & Architecture

Cortex M4 block diagram

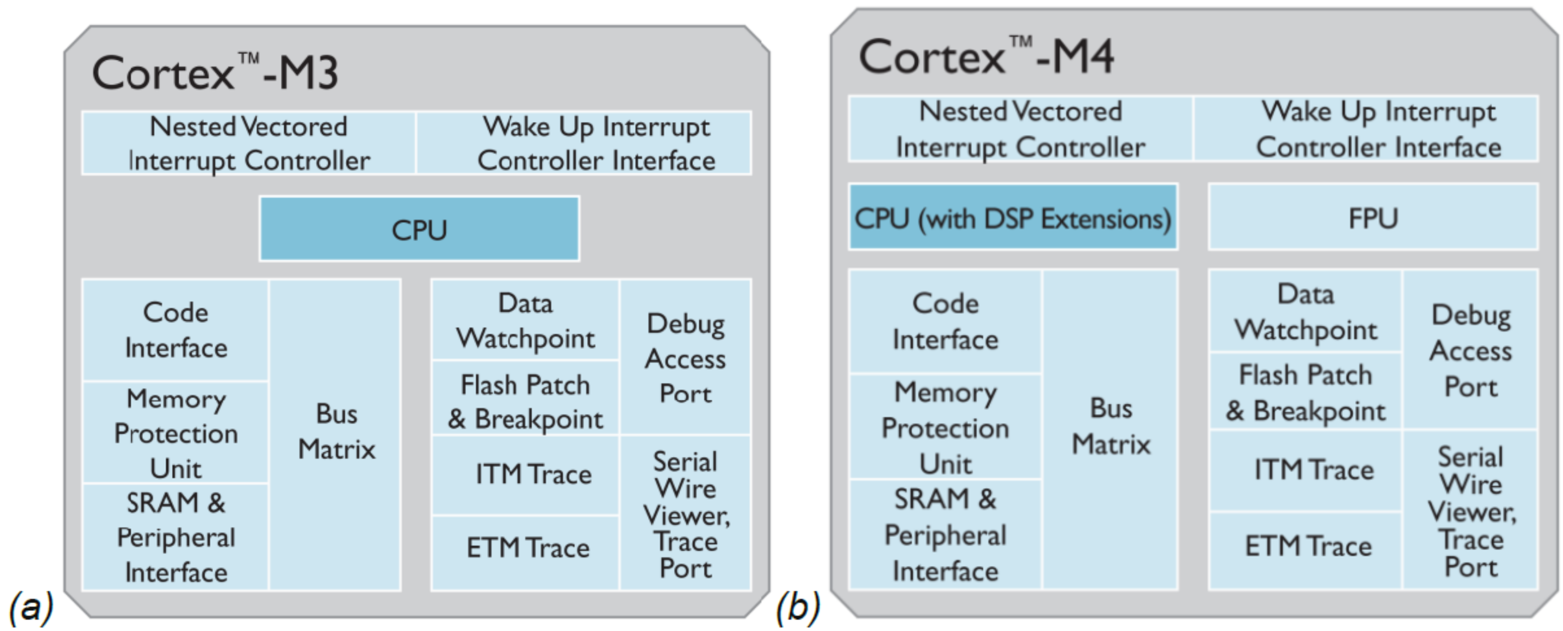
ISA Support	Thumb® / Thumb-2
DSP Extensions	Single cycle 16/32-bit MAC Single cycle dual 16-bit MAC 8/16-bit SIMD arithmetic Hardware Divide (2-12 Cycles)
Floating Point Unit	Single precision floating point unit IEEE 754 compliant
Pipeline	3-stage + branch speculation
Performance Efficiency	3.40 CoreMark/MHz*
Performance Efficiency	Without FPU: 1.25 / 1.52 / 1.91 DMIPS/MHz** With FPU: 1.27 / 1.55 / 1.95 DMIPS/MHz**
Memory Protection	Optional 8 region MPU with sub regions and background region
Interrupts	Non-maskable Interrupt (NMI) + 1 to 240 physical interrupts
Interrupt Priority Levels	8 to 256 priority levels
Wake-up Interrupt Controller	Up to 240 Wake-up Interrupts
Sleep Modes	Integrated WFI and WFE Instructions and Sleep On Exit capability. Sleep & Deep Sleep Signals. Optional Retention Mode with ARM Power Management Kit
Bit Manipulation	Integrated Instructions & Bit Banding
Debug	Optional JTAG & Serial-Wire Debug Ports. Up to 8 Breakpoints and 4 Watchpoints.

Cortex M4

Embedded processor for DSP with FPU



Cortex M4 vs. M3



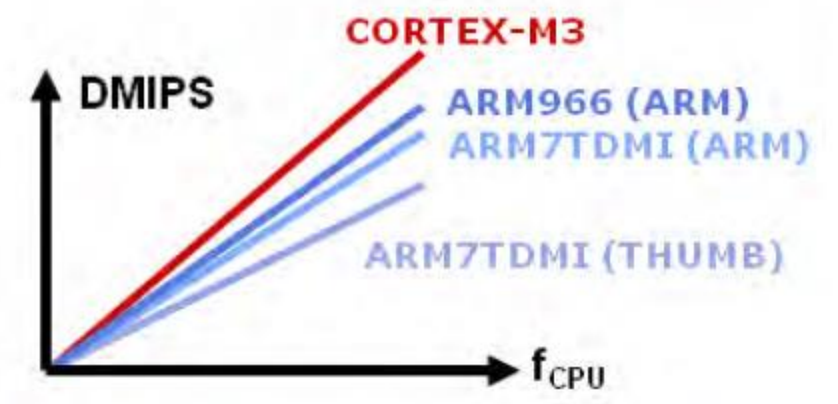
(b) The Cortex-M4 ISA is enhanced efficient DSP features including extended single-cycle cycle 16/32-bit multiply-accumulate (MAC), dual 16-bit MAC instructions, optimized 8/16-bit SIMD arithmetic and saturating arithmetic instructions

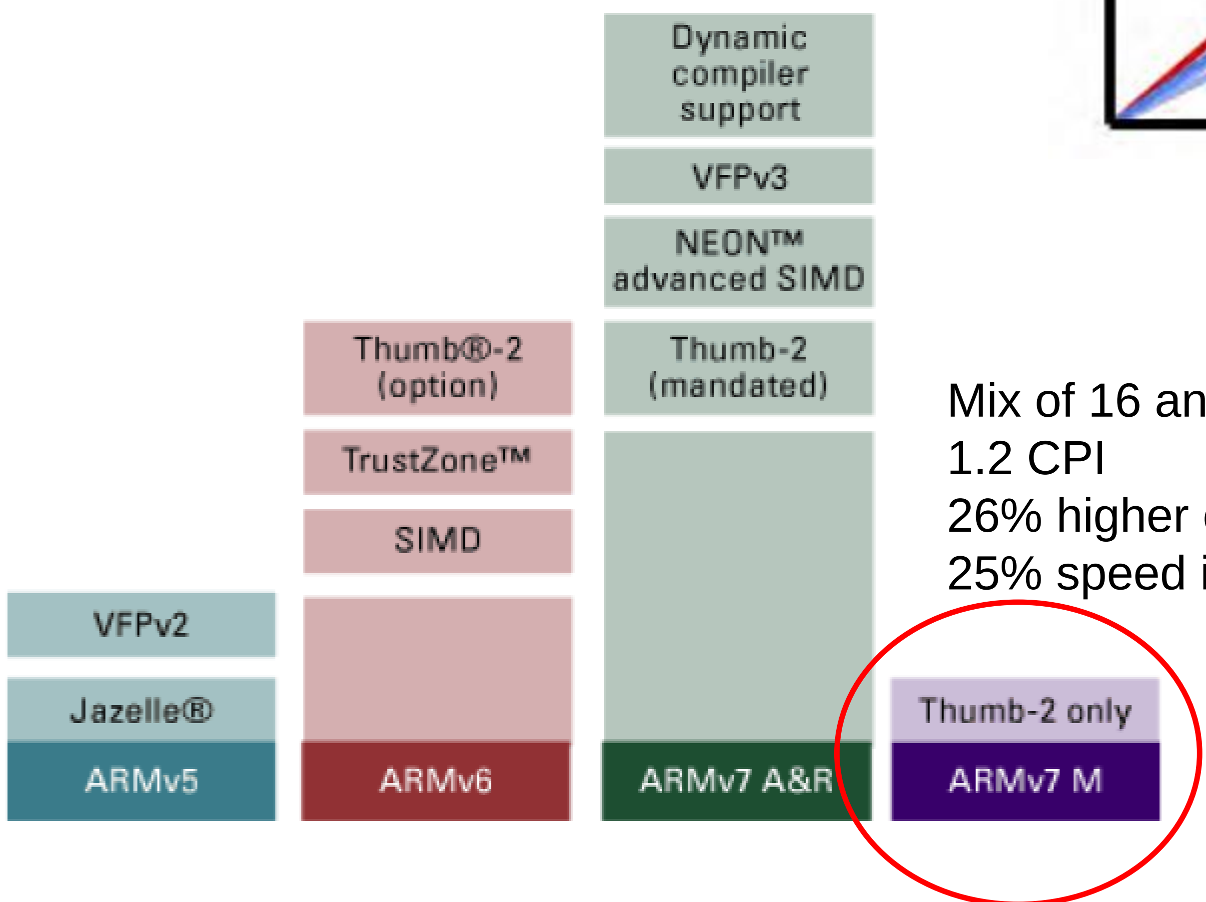
ARM Cortex-M Series Family

Processor	ARM Architecture	Core Architecture	Thumb®	Thumb®-2	Hardware Multiply	Hardware Divide	Saturated Math	DSP Extensions	Floating Point
Cortex-M0	ARMv6-M	Von Neumann	Most	Subset	1 or 32 cycle	No	No	No	No
Cortex-M0+	ARMv6-M	Von Neumann	Most	Subset	1 or 32 cycle	No	No	No	No
Cortex-M1	ARMv6-M	Von Neumann	Most	Subset	3 or 33 cycle	No	No	No	No
Cortex-M3	ARMv7-M	Harvard	Entire	Entire	1 cycle	Yes	Yes	No	No
Cortex-M4	ARMv7E-M	Harvard	Entire	Entire	1 cycle	Yes	Yes	Yes	Optional

ARMv7 M (Thumb-2) features

Source	Destination	Cycles
16b x 16b	32b	1
32b x 16b	32b	1
32b x 32b	32b	1
32b x 32b	64b	3-7*

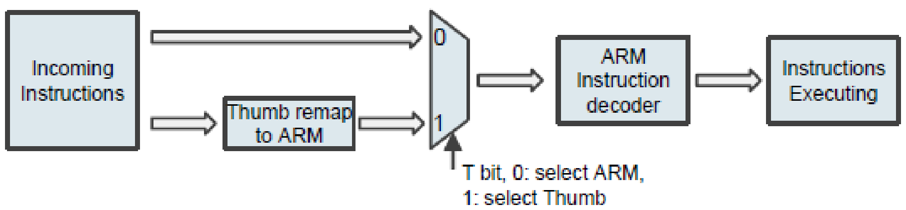


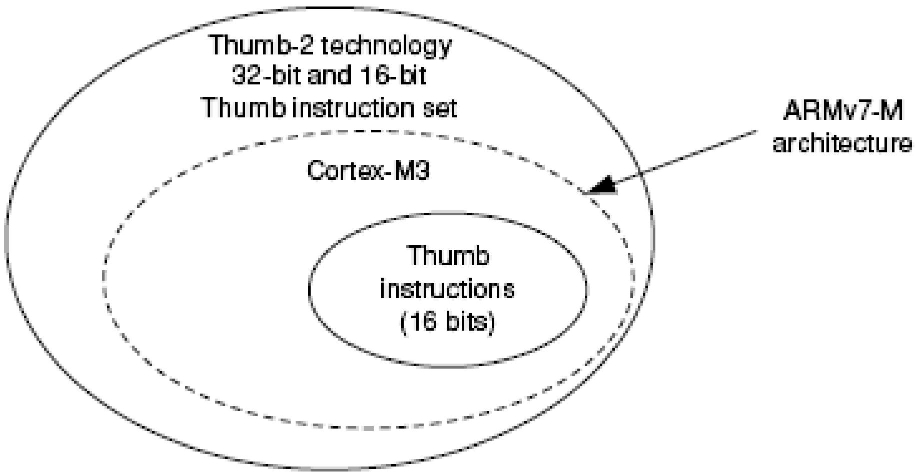


Mix of 16 and 32b instructions
1.2 CPI
26% higher code density ARM32
25% speed improvement over Thumb16

Thumb-2

- Mixes 16 and 32 bits instructions
  - Enhancements: eg. UDIV, SDIF division, bit-field operators UFBX, BFC, BFE, wrt traditional ARMv4T
  - No need to mode switch, can be mixed freely
- **Not** backwards binary compatible
  - But porting is «easy»





Cortex-M4 Processor Overview

- Cortex-M4 Processor
  - Introduced in 2010
  - Designed with a large variety of highly efficient signal processing features
  - Features extended single-cycle multiply accumulate instructions, optimized SIMD arithmetic, saturating arithmetic and an optional Floating Point Unit.
- High Performance Efficiency
  - 1.25 DMIPS/MHz (Dhrystone Million Instructions Per Second / MHz) at the order of μWatts / MHz
- Low Power Consumption
  - Longer battery life – especially critical in mobile products
- Enhanced Determinism
  - The critical tasks and interrupt routines can be served quickly in a known number of cycles

Cortex-M4 Processor Features

- 32-bit Reduced Instruction Set Computing (RISC) processor
- Harvard architecture
  - Separated data bus and instruction bus
- Instruction set
  - Include the entire Thumb®-1 (16-bit) and Thumb®-2 (16/ 32-bit) instruction sets
- 3-stage + branch speculation pipeline
- Performance efficiency
  - 1.25 – 1.95 DMIPS/MHz (Dhrystone Million Instructions Per Second / MHz)
- Supported Interrupts
  - Non-maskable Interrupt (NMI) + 1 to 240 physical interrupts
  - 8 to 256 interrupt priority levels

Cortex-M4 Processor Features

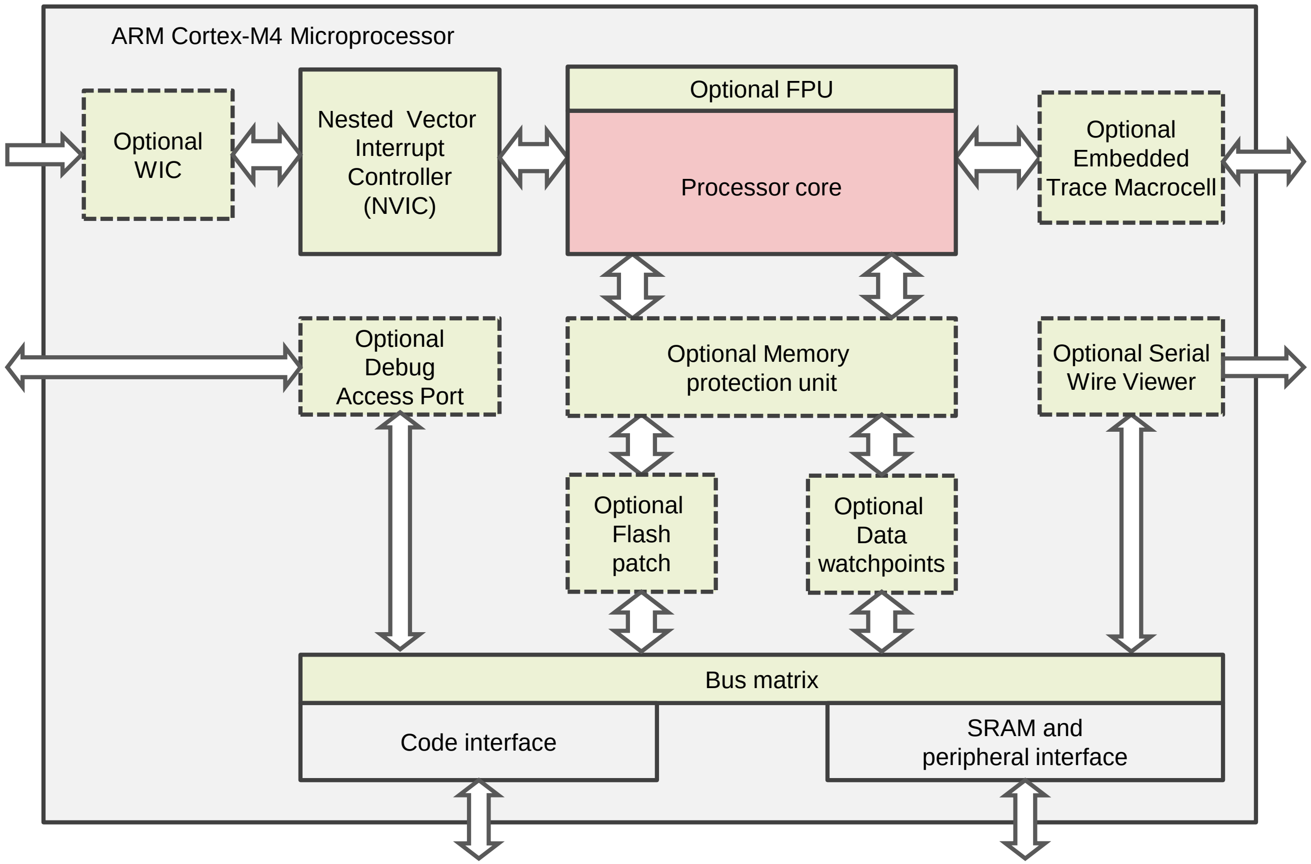
- Supports Sleep Modes
  - Up to 240 Wake-up Interrupts
  - Integrated WFI (Wait For Interrupt) and WFE (Wait For Event) Instructions and Sleep On Exit capability (to be covered in more detail later)
  - Sleep & Deep Sleep Signals
  - Optional Retention Mode with ARM Power Management Kit
- Enhanced Instructions
  - Hardware Divide (2-12 Cycles)
  - Single-Cycle 16, 32-bit MAC, Single-cycle dual 16-bit MAC
  - 8, 16-bit SIMD arithmetic
- Debug
  - Optional JTAG & Serial-Wire Debug (SWD) Ports
  - Up to 8 Breakpoints and 4 Watchpoints
- Memory Protection Unit (MPU)
  - Optional 8 region MPU with sub regions and background region

Cortex-M4 Processor Features

- Cortex-M4 processor is designed to meet the challenges of low dynamic power constraints while retaining light footprints
  - 180 nm ultra low power process –157 µW/MHz
  - 90 nm low power process – 33 µW/MHz
  - 40 nm G process – 8 µW/MHz

ARM Cortex-M4 Implementation Data			
Process	180ULL (7-track, typical 1.8v, 25C)	90LP (7-track, typical 1.2v, 25C)	40G 9-track, typical 0.9v, 25C)
Dynamic Power	157 µW/MHz	33 µW/MHz	8 µW/MHz
Floorplanned Area	0.56 mm^2	0.17 mm^2	0.04 mm^2

Cortex-M4 Block Diagram



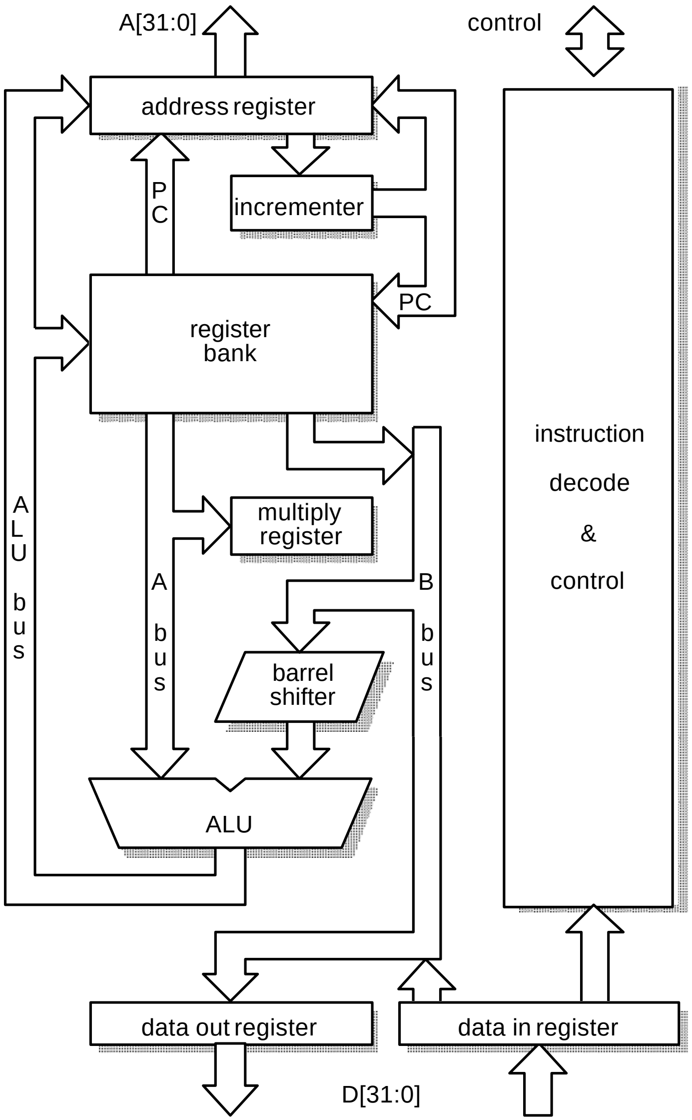
Cortex-M4 Block Diagram

- Bus interconnect
  - Allows data transfer to take place on different buses simultaneously
  - Provides data transfer management, e.g. a write buffer, bit-oriented operations (bit-band)
  - May include bus bridges (e.g. AHB-to-APB bus bridge) to connect different buses into a network using a single global memory space
  - Includes the internal bus system, the data path in the processor core, and the AHB LITE interface unit
- Debug subsystem
  - Handles debug control, program breakpoints, and data watchpoints
  - When a debug event occurs, it can put the processor core in a halted state, where developers can analyse the status of the processor at that point, such as register values and flags

Cortex-M4 Block Diagram

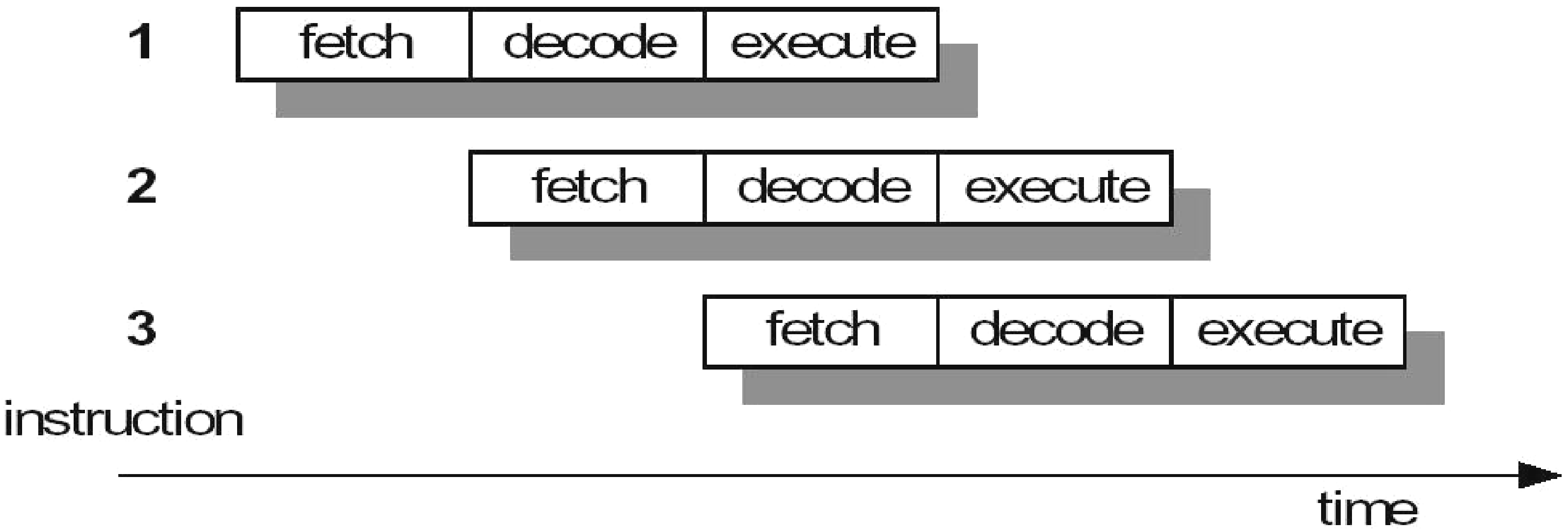
- Nested Vectored Interrupt Controller (NVIC)
  - Up to 240 interrupt request signals and a non-maskable interrupt (NMI)
  - Automatically handles nested interrupts, such as comparing priorities between interrupt requests and the current priority level
- Wakeup Interrupt Controller (WIC)
  - For low-power applications, the microcontroller can enter sleep mode by shutting down most of the components.
  - When an interrupt request is detected, the WIC can inform the power management unit to power up the system.
- Memory Protection Unit (optional)
  - Used to protect memory content, e.g. make some memory regions read-only or preventing user applications from accessing privileged application data

3-Stage Pipeline ARM Organization



- Register Bank
  - 2 read ports, 1 write ports, access any register
  - 1 additional read port, 1 additional write port for r15 (PC)
- Barrel Shifter
  - Shift or rotate the operand by any number of bits
- ALU
- Address register and incrementer
- Data Registers
  - Hold data passing to and from memory
- Instruction Decoder and Control

3-Stage Pipeline (1/2)

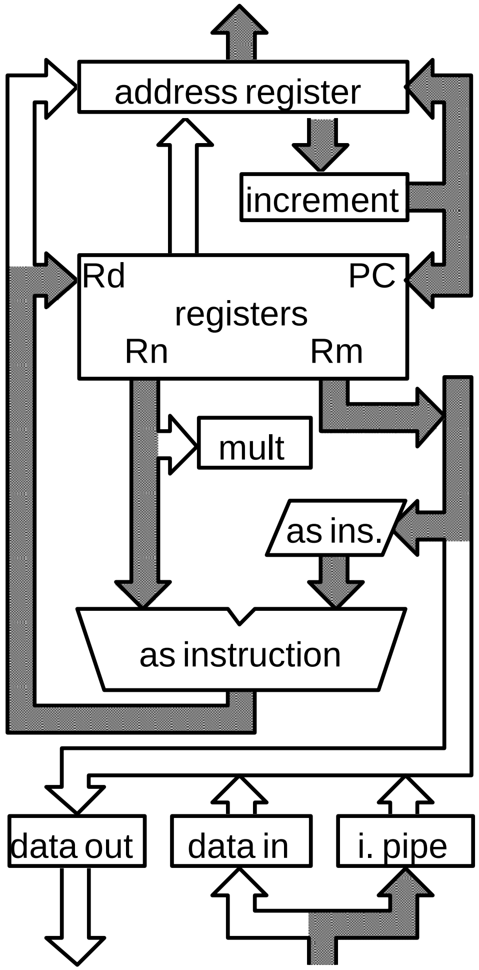


- **Fetch**
  - The instruction is fetched from memory and placed in the instruction pipeline
- **Decode**
  - The instruction is decoded and the datapath control signals prepared for the next cycle
- **Execute**
  - The register bank is read, an operand shifted, the ALU result generated and written back into destination register

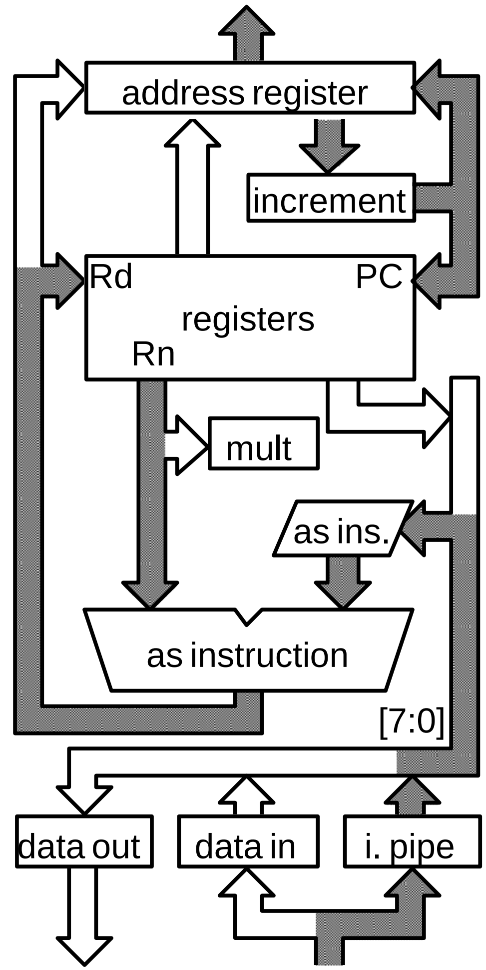
3-Stage Pipeline (2/2)

- At any time slice, 3 different instructions may occupy each of these stages, so the hardware in each stage has to be capable of independent operations
- When the processor is executing data processing instructions , the latency = **3** cycles and the throughput = **1** instruction/cycle
- There are exceptions: multiycle instructions and branches

Data Processing Instruction



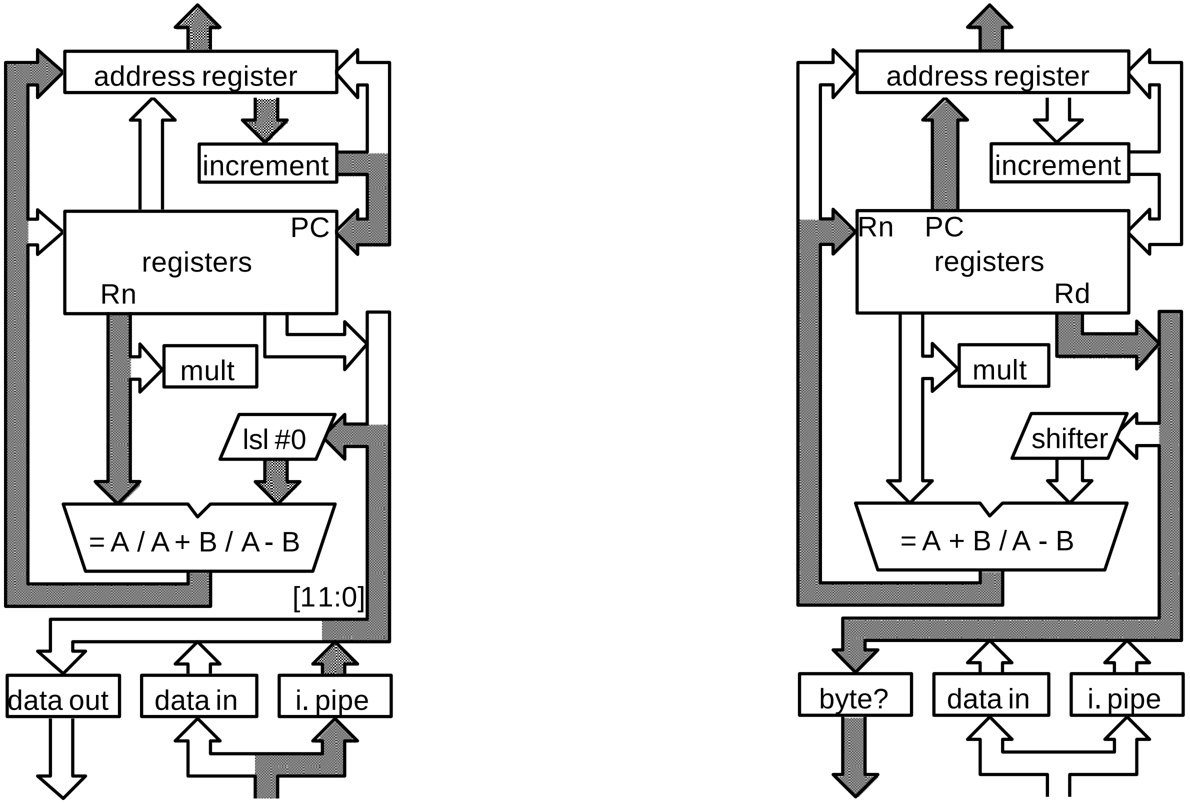
(a) register - register operations



(b) register - immediate operations

All operations take place in a single clock cycle

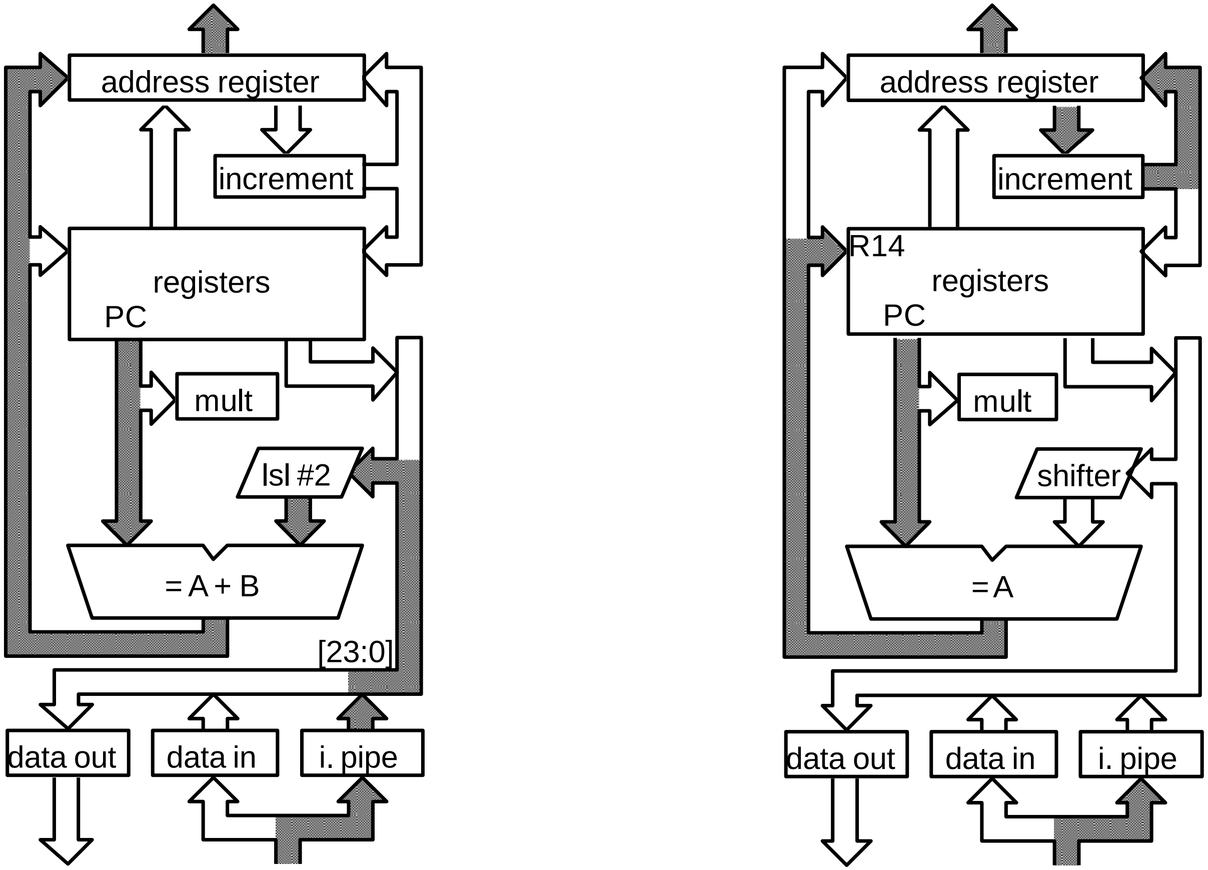
Data Transfer Instructions



(a) 1st cycle - compute address   *(b) 2nd cycle - store data & auto-index*

- Computes a memory address similar to a data processing instruction
- Load instruction follows a similar pattern except that the data from memory only gets as far as the ‘data in’ register on the 2nd cycle and a 3rd cycle is needed to transfer the data from there to the destination register

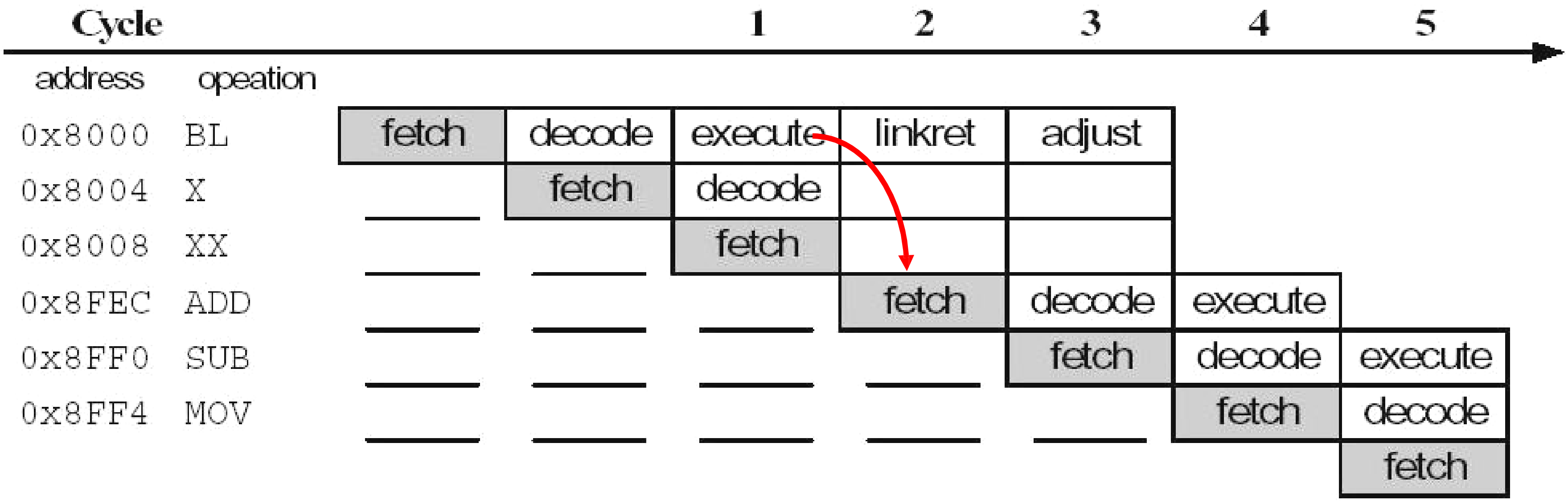
Branch Instructions



(a) 1st cycle - compute branch target *(b) 2nd cycle - save return address*

- The third cycle, which is required to complete the pipeline refilling, is also used to mark the small correction to the value stored in the link register in order that is points directly at the instruction which follows the branch

Branch Pipeline Example



- Breaking the pipeline
- Two clock stalls → IPC goes down

Pipeline summary

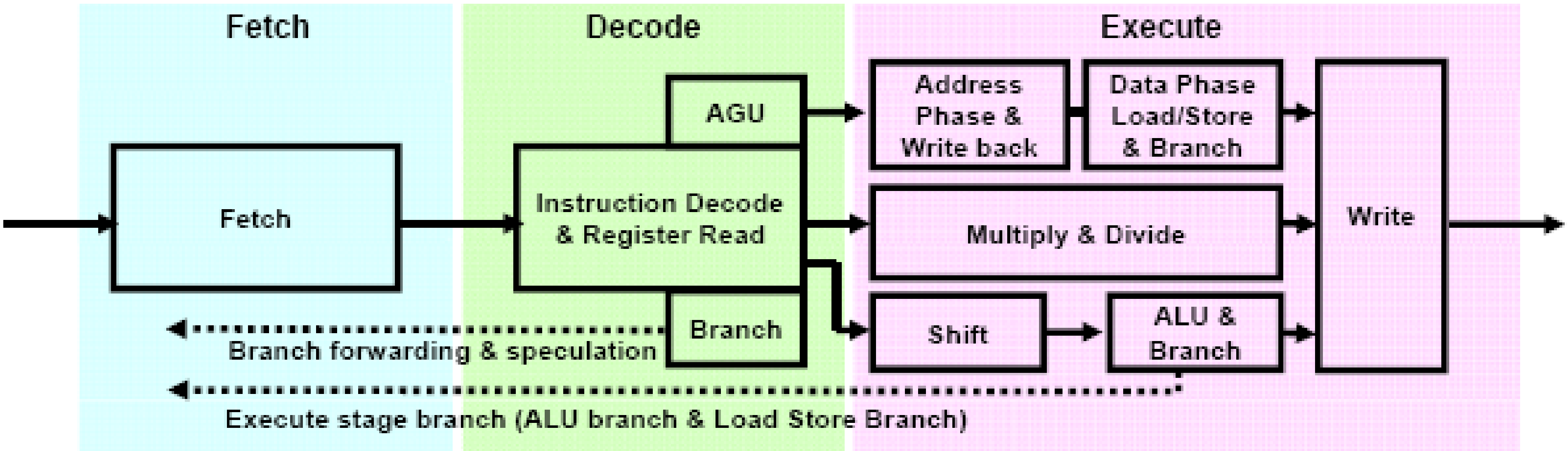
Harvard architecture

- Separate Instruction & Data buses enable parallel fetch & store

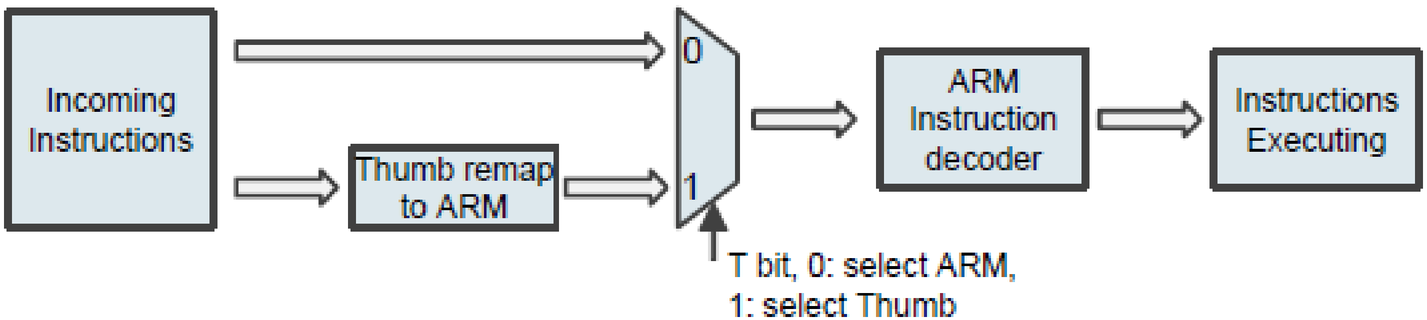
Advanced 3-Stage Pipeline

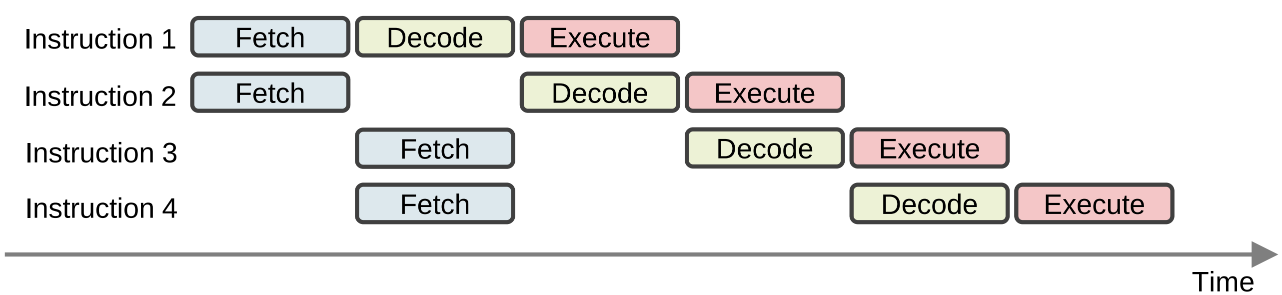
- Includes Branch Forwarding & Speculation

Additional Write-Back via Bus Matrix

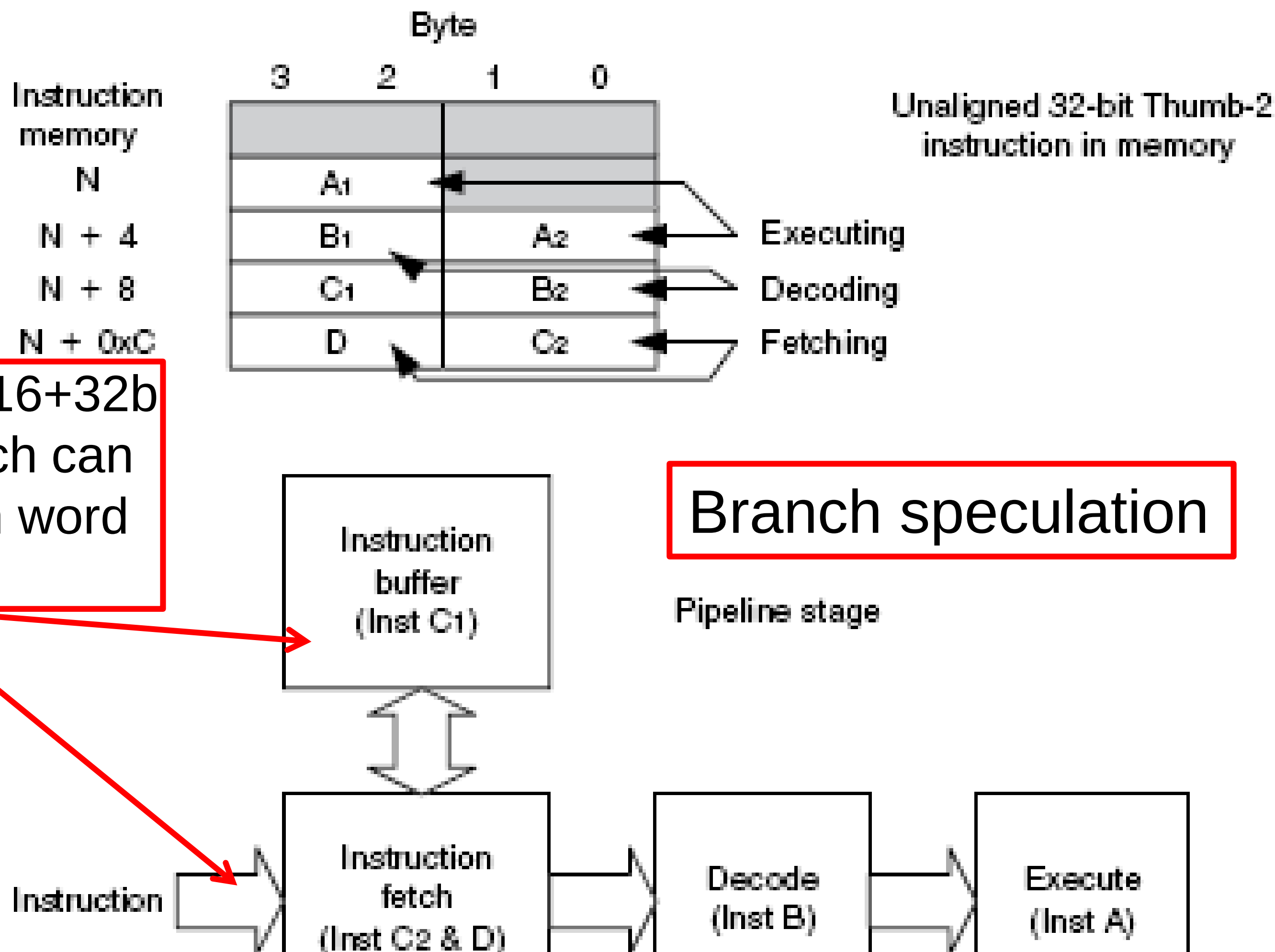


Decoding Thumb





Instruction Prefetch & Execution



Handles mix of 16+32b instructions which can be misaligned in word address

Branch speculation

Processor Modes

- The ARM has seven basic operating modes:
  - Each mode has access to:
    - Its own stack space and a different subset of registers
  - Some operations can only be carried out in a privileged mode

Exception modes (Supervisor, FIQ, IRQ, Abort, Undef)

Mode	Description	
Supervisor (SVC)	Entered on reset and when a Software Interrupt instruction (SWI) is executed	**Privileged modes**
FIQ	Entered when a high priority (fast) interrupt is raised	
IRQ	Entered when a low priority (normal) interrupt is raised	
Abort	Used to handle memory access violations	
Undef	Used to handle undefined instructions	
System	Privileged mode using the same registers as User mode	
User	Mode under which most Applications / OS tasks run	**Unprivileged mode**

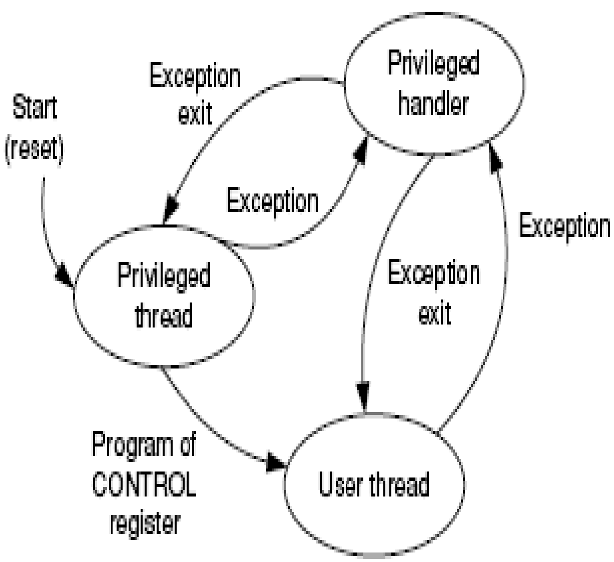
Operating Modes

User mode:

– Normal program execution mode
– System resources unavailable
– Mode changed by exception only

Exception modes:

– Entered upon exception
– Full access to system resources
– Mode changed freely



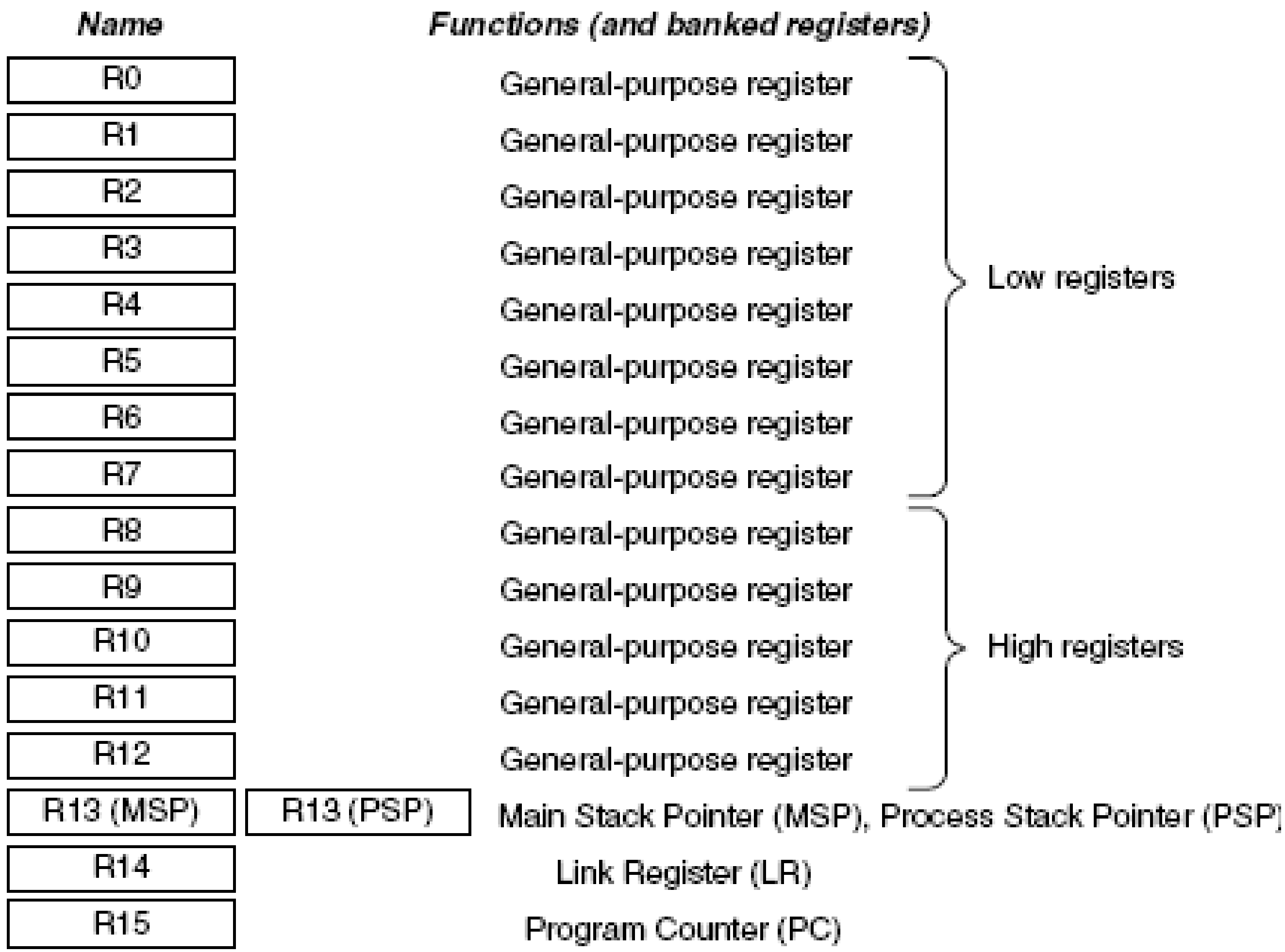
		Operations (privilege out of reset)	Stacks (Main out of reset)
Modes (Thread out of reset)	**Handler** - An exception is being processed	Privileged execution Full control	Main Stack Used by OS and Exceptions
	Thread - No exception is being processed - Normal code is executing	Privileged/Unprivileged	Main/Process

Exceptions

Exception	Mode	Priority	IV Address
Reset	Supervisor	1	0x00000000
Undefined instruction	Undefined	6	0x00000004
Software interrupt	Supervisor	6	0x00000008
Prefetch Abort	Abort	5	0x0000000C
Data Abort	Abort	2	0x00000010
Interrupt	IRQ	4	0x00000018
Fast interrupt	FIQ	3	0x0000001C

Table 1 - Exception types, sorted by Interrupt Vector addresses

Registers


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

ARM Registers

- 31 general-purpose 32-bit registers
- 16 visible, R0 – R15
- Others speed up the exception process

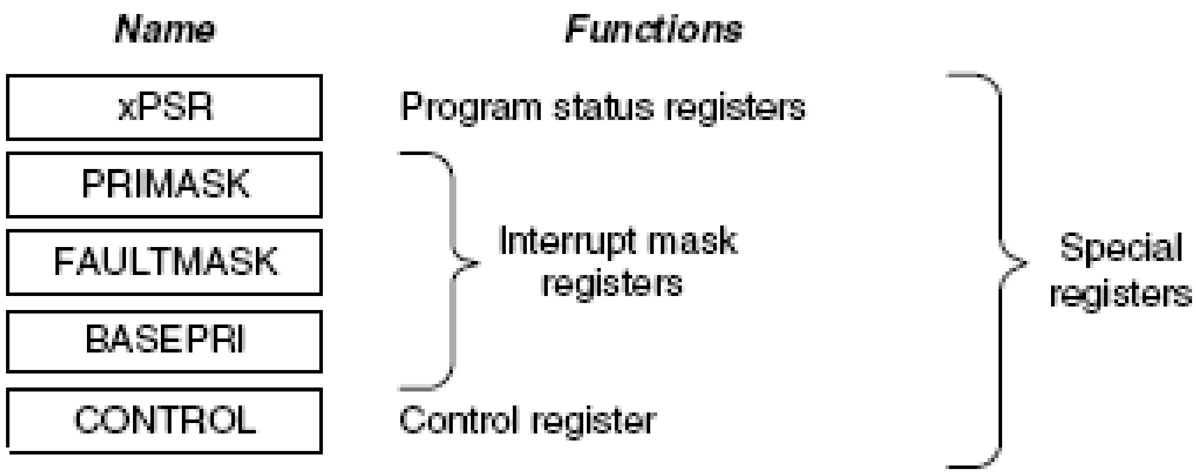
ARM Registers (2)

- Special roles:
  - Hardware
    - R14 – Link Register (LR):
      optionally holds return address
      for branch instructions
    - R15 – Program Counter (PC)
  - Software
    - R13 - Stack Pointer (SP)

ARM Registers (3)

- Current Program Status Register (CPSR)
- Saved Program Status Register (SPSR)
- On exception, entering *mod* mode:
  - (PC + 4) → LR
  - CPSR → SPSR_mod
  - PC ← IV address
  - R13, R14 replaced by R13_mod, R14_mod
  - In case of FIQ mode R7 – R12 also replaced

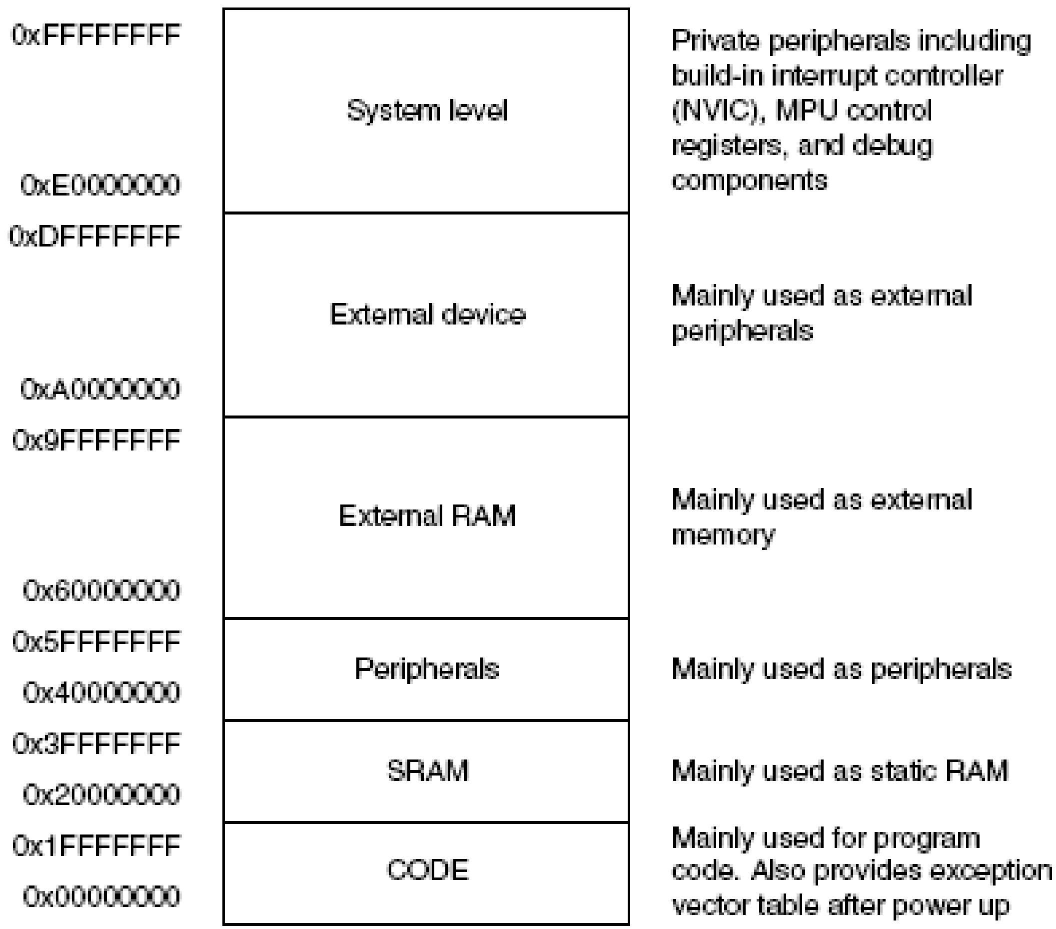
Special Registers



Register	Function
xPSR	Provide arithmetic and logic processing flags (zero flag and carry flag), execution status, and current executing interrupt number
PRIMASK	Disable all interrupts except the nonmaskable interrupt (NMI) and hard fault
FAULTMASK	Disable all interrupts except the NMI
BASEPRI	Disable all interrupts of specific priority level or lower priority level
CONTROL	Define privileged status and stack pointer selection

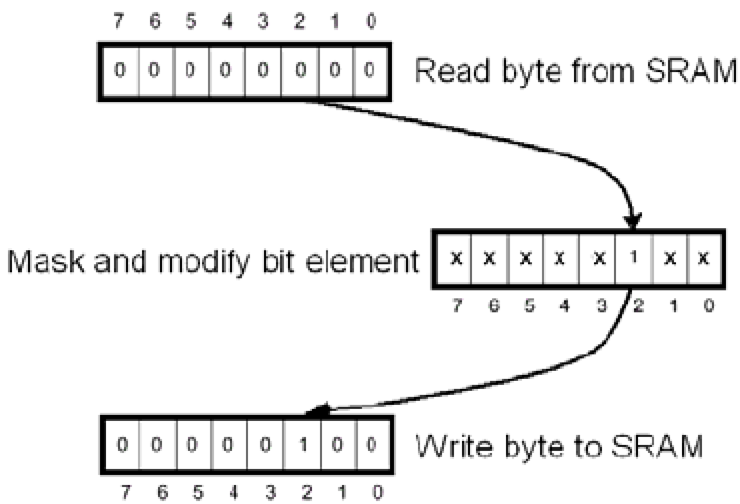
Memory map

- Statically defined memory map (faster addr decoding) 4GB of address psace



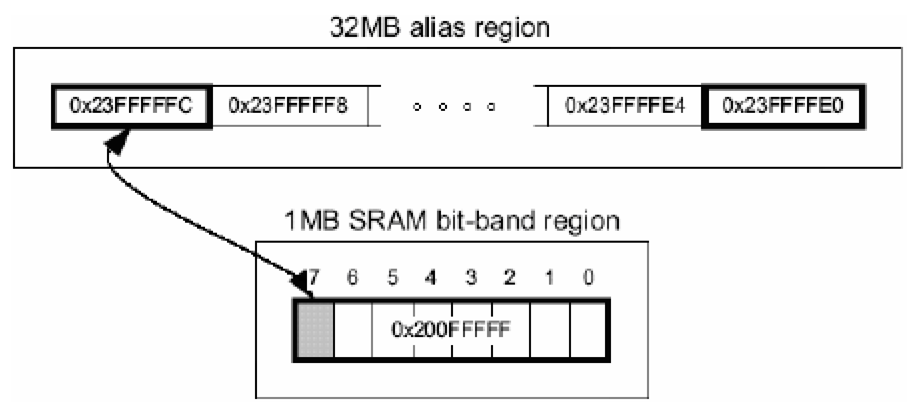
Bit Banding

- Fast single-bit manipulation: 1MB → 32MB aliased regions in SRAM & Peripheral space




```
LDR R0,=0x200FFFFF ; Setup address
MOV R2, #0x4       ; Setup data
LDR R1, [R0]       ; Read
ORR R1, R2         ; Modify bit
STR R1, [R0]       ; Write back result
```

Traditional bit manipulation method



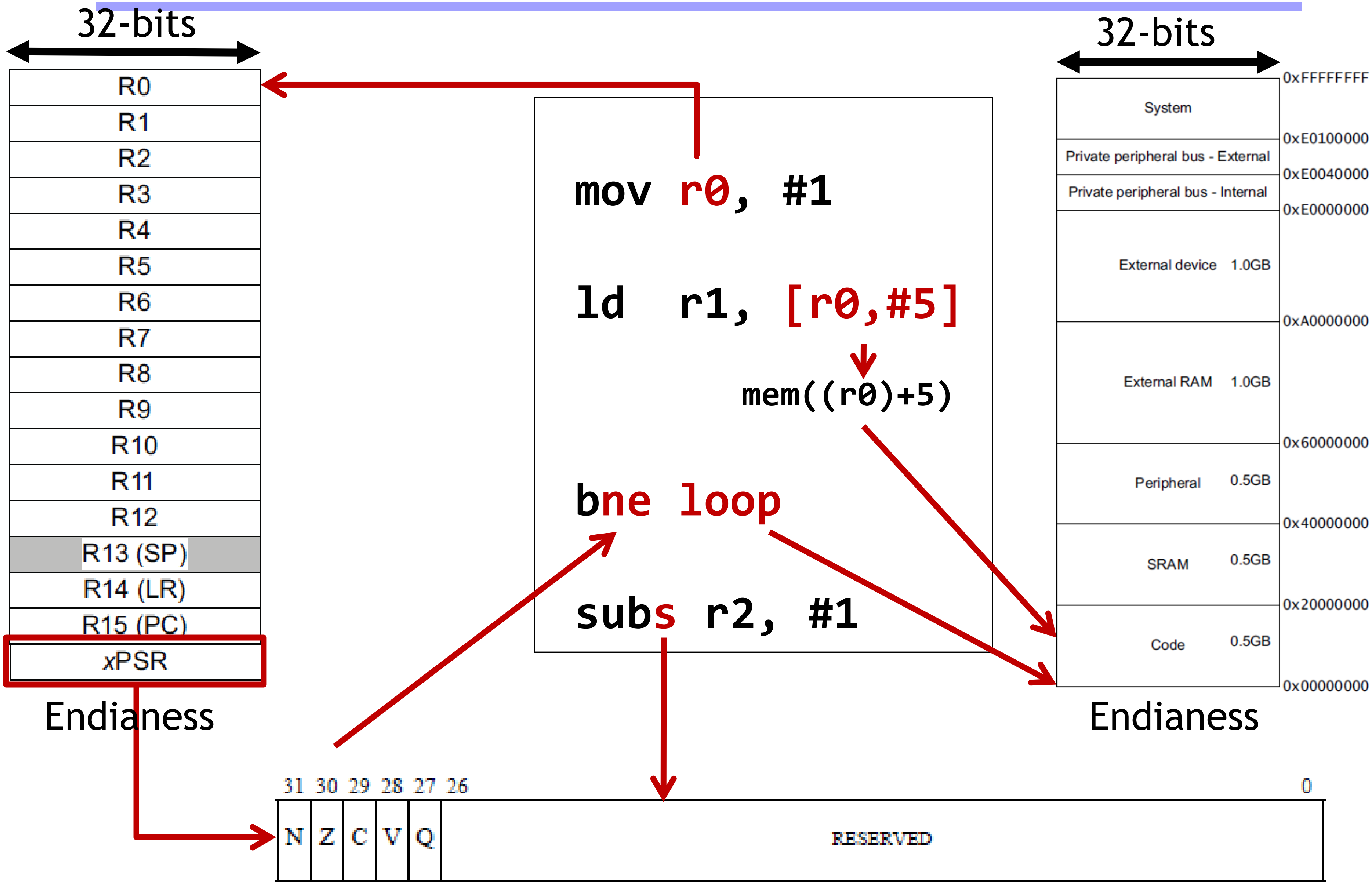

```
LDR R0,=0x23FFFFFC ; Setup address
MOV R1, #0x1       ; Setup data
STR R1, [R0]       ; Write
```

Direct, single cycle access with bit banding

Cortex M3/M4 Instruction Set

Major Elements of ISA

(registers, memory, word size, endianess, conditions, instructions, addressing modes)



Traditional ARM instructions

- Fixed length of 32 bits
- Commonly take two or three operands
- Process data held in registers
- Shift & ALU operation in single clock cycle
- Access memory with load and store instructions only
  - Load/Store multiple register
- Can be extended to execute conditionally by adding the appropriate suffix
- Affect the CPSR status flags by adding the ‘S’ suffix to the instruction

Thumb-2

- Original 16-bit Thumb instruction set
  - a subset of the full ARM instructions
  - performs similar functions to selective 32-bit ARM instructions but in 16-bit code size
- For ARM instructions that are not available
  - more 16-bit Thumb instructions are needed to execute the same function compared to using ARM instructions
  - but performance may be degraded
- Hence the introduction of the Thumb-2 instruction set
  - enhances the 16-bit Thumb instructions with additional 32-bit instructions
- All ARMv7 chips support the Thumb-2 (& ARM) instruction set
  - but Cortex-M3 supports only the 16-bit/32-bit Thumb-2 instruction set

16bit Thumb-2

Some of the changes used to reduce the length of the instructions from 32 bits to 16 bits:

- reduce the number of bits used to identify the register
  - less number of registers can be used
- reduce the number of bits used for the immediate value
  - smaller number range
- remove options such as ‘S’
  - make it default for some instructions
- remove conditional fields (N, Z, V, C)
- no conditional executions (except branch)
- remove the optional shift (and no barrel shifter operation
  - introduce dedicated shift instructions
- remove some of the instructions
  - more restricted coding

Thumb-2 Implementation

- The 32-bit ARM Thumb-2 instructions are added through the space occupied by the Thumb BL and BLX instructions

31 16	15 0
Hw1	Hw2

32-bit Thumb-2 Instruction format

- The first Halfword (Hw1)
  - determines the instruction length and functionality
- If the processor decodes the instruction as 32-bit long
  - the processor fetches the second halfword (hw2) of the instruction from the instruction address plus two

Unified Assembly Language

- UAL supports generation of either Thumb-2 or ARM instructions from the same source code
  - same syntax for both the Thumb code and ARM code
  - enable portability of code for different ARM processor families
- Interpretation of code type is based on the directive listed in the assembly file
- Example:
  - For GNU GAS, the directive for UAL is

.syntax unified

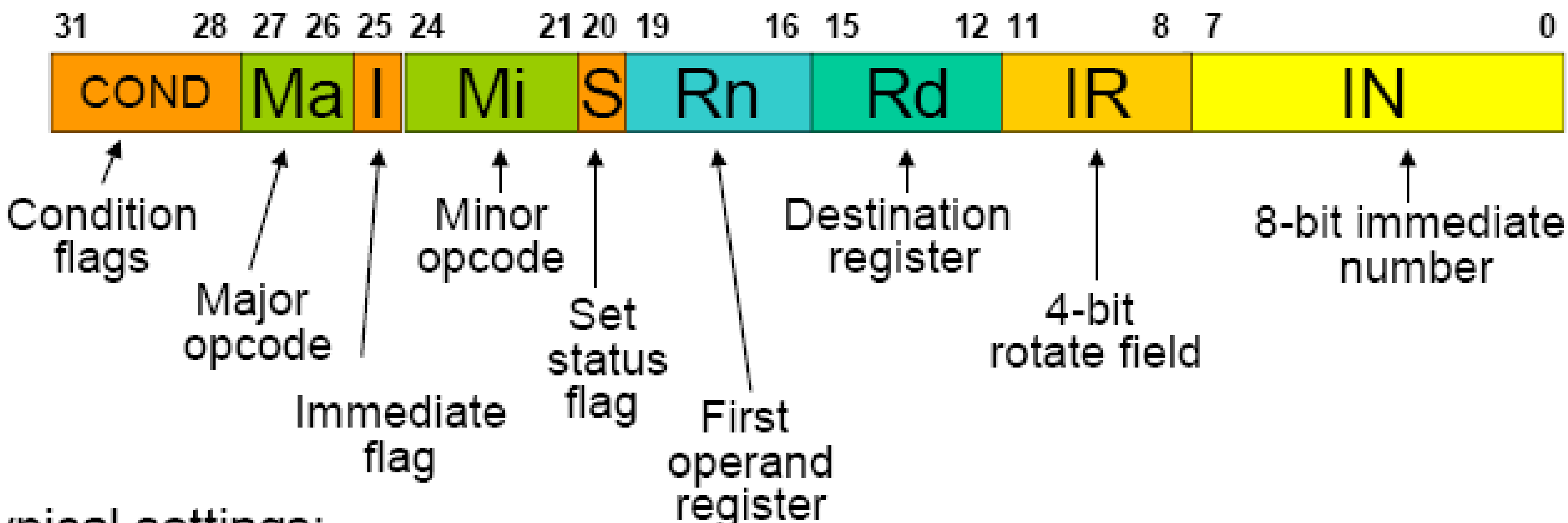
- For ARM assembler, the directive for UAL is

THUMB

32bit Instruction Encoding

Example: ADD instruction format

- ARM 32-bit encoding for ADD with immediate field



Typical settings:

Major opcode = 00 (this indicates data operation instructions)

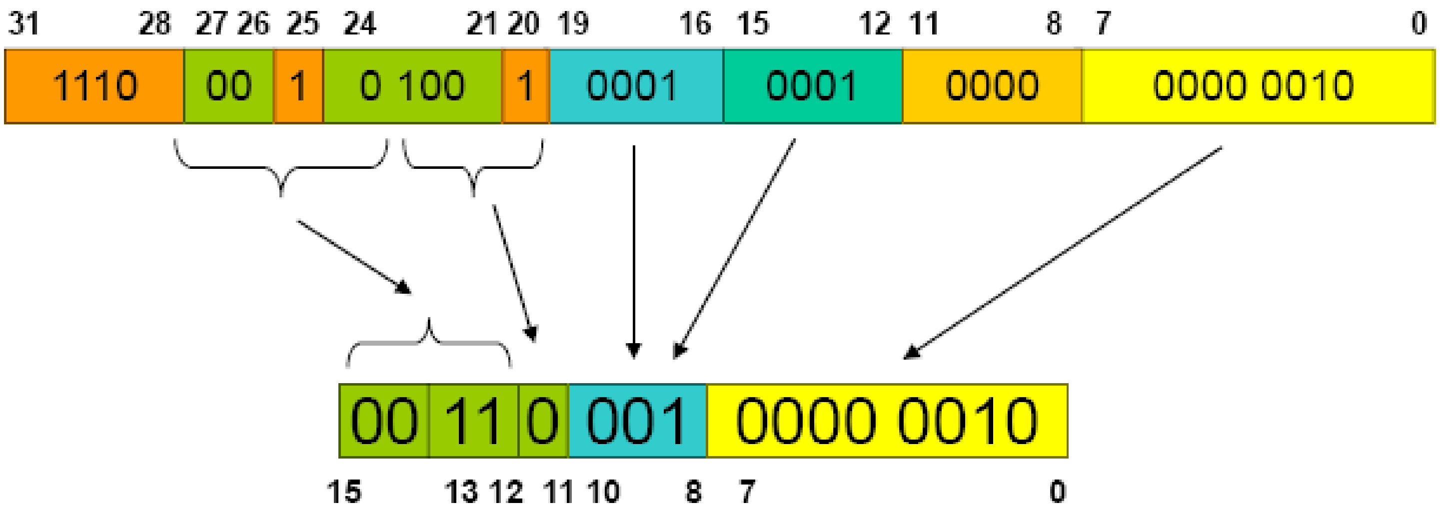
Minor opcode = 0100 (specifically, 100 ⇨ ADD instruction)

Immediate flag = 1 (immediate field in operand 2)

Set status flag = 1 (set carry flag after operation)

ARM and 16-bit Instruction Encoding

ARM 32-bit encoding: `ADDS r1, r1, #2`



- Equivalent 16-bit Thumb instruction: `ADD r1, #2`
  - No condition flag
  - No rotate field for the immediate number
  - Use 3-bit encoding for the register
  - Shorter opcode with implicit flag settings (e.g. the set status flag is always set)

Application Program Status Register (APSR)

31	30	29	28	27	26 … 0
N	Z	C	V	Q	RESERVED

APSR bit fields are in the following two categories:

- Reserved bits are allocated to system features or are available for future expansion. Further information on currently allocated reserved bits is available in *The special-purpose program status registers (xPSR)* on page B1-8. Application level software must ignore values read from reserved bits, and preserve their value on a write. The bits are defined as UNK/SBZP.

- Flags that can be set by many instructions:

  N, bit [31] Negative condition code flag. Set to bit [31] of the result of the instruction. If the result is regarded as a two's complement signed integer, then N == 1 if the result is negative and N = 0 if it is positive or zero.

  Z, bit [30] Zero condition code flag. Set to 1 if the result of the instruction is zero, and to 0 otherwise. A result of zero often indicates an equal result from a comparison.

  C, bit [29] Carry condition code flag. Set to 1 if the instruction results in a carry condition, for example an unsigned overflow on an addition.

  V, bit [28] Overflow condition code flag. Set to 1 if the instruction results in an overflow condition, for example a signed overflow on an addition.

  Q, bit [27] Set to 1 if an SSAT or USAT instruction changes (saturates) the input value for the signed or unsigned range of the result.

Updating the APSR

- SUB Rx, Ry
  - Rx = Rx - Ry
  - APSR unchanged
- SUBS
  - Rx = Rx - Ry
  - APSR N or Z bits might be set
- ADD Rx, Ry
  - Rx = Rx + Ry
  - APSR unchanged
- ADDS
  - Rx = Rx + Ry
  - APSR C or V bits might be set

Conditional Execution

- Each data processing instruction prefixed by condition code
- Result – smooth flow of instructions through pipeline
- 16 condition codes:

EQ	equal	MI	negative	HI	unsigned higher	GT	signed greater than
NE	not equal	PL	positive or zero	LS	unsigned lower or same	LE	signed less than or equal
CS	unsigned higher or same	VS	overflow	GE	signed greater than or equal	AL	always
CC	unsigned lower	VC	no overflow	LT	signed less than	NV	special purpose

Conditional Execution

- Every ARM (32 bit) instruction is conditionally executed.
- The top four bits are ANDed with the CPSR condition codes, If they do not matched the instruction is executed as NOP
- The AL condition is used to execute the instruction irrespective of the value of the condition code flags.
- By default, data processing instructions do not affect the condition code flags but the flags can be optionally set by using "S". Ex: SUBS r1,r1,#1
- Conditional Execution improves code density and performance by reducing the number of forward branch instructions.

Normal	**Conditional**
CMP r3,#0	CMP r3,#0
BEQ skip	ADDNE r0,r1,r2
ADD r0,r1,r2	
skip	

Conditional Execution and Flags

- ARM instructions can be made to execute conditionally by post-fixing them with the appropriate condition code
  - This can increase code density and increase performance by reducing the number of forward branches

```
CMP      r0, r1
ADDGT    r2, r2, #1
ADDLE    r3, r3, #1
```

r0 - r1, compare r0 with r1 and set flags

if > r2=r2+1 flags remain unchanged

if <= r3=r3+1 flags remain unchanged

- By default, data processing instructions do not affect the condition flags but this can be achieved by post fixing the instruction (and any condition code) with an "S"

```
loop
    ADD      r2, r2, r3
    SUBS     r1, r1, #0x01
    BNE      loop
```

r2=r2+r3

decrement r1 and set flags

if Z flag clear then branch

Conditional execution examples

C source code

```
if (r0 == 0)
{
  r1 = r1 + 1;
}
else
{
  r2 = r2 + 1;
}
```

ARM instructions

unconditional

```
  CMP r0, #0
  BNE else
  ADD r1, r1, #1
  B end
else
  ADD r2, r2, #1
end
  ...
```

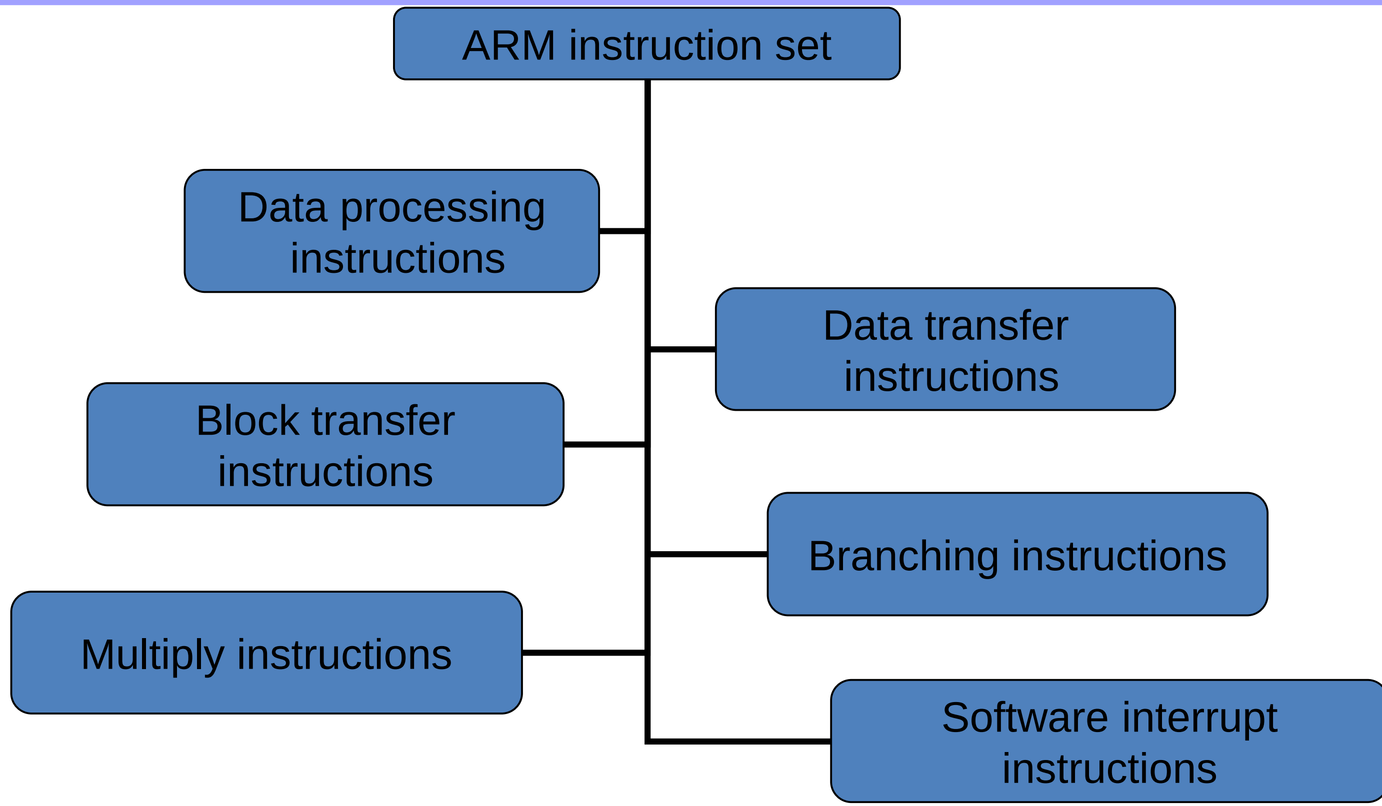
- 5 instructions
- 5 words
- 5 or 6 cycles

conditional

```
  CMP r0, #0
  ADDEQ r1, r1,
#1
  ADDNE r2, r2,
#1
  ...
```

- 3 instructions
- 3 words
- 3 cycles

ARM Instruction Set (3)



Data Movement	45.28%	Logical	3.91%
Flow Control	28.73%	Shift	2.92%
Arithmetic	10.75%	Bit Manipulation	2.05%
Compare	5.92%	I/O & Others	0.44%

Structural view of ARM ISA

	31–28	27	26	25	24	23	22	21	20	19–16	15–12	11–8	7	6–5	4	3–0
Data processing immediate shift	cond [1]	0	0	0	opcode				S	Rn	Rd	shift amount		shift	0	Rm
Miscellaneous instructions: See Figure 3-3	cond [1]	0	0	0	1	0	x	x	0	x x x x	x x x x	x x x x	x	x x	0	x x x x
Data processing register shift [2]	cond [1]	0	0	0	opcode				S	Rn	Rd	Rs	0	shift	1	Rm
Miscellaneous instructions: See Figure 3-3	cond [1]	0	0	0	1	0	x	x	0	x x x x	x x x x	x x x x	0	x x	1	x x x x
Multiplies, extra load/stores: See Figure 3-2	cond [1]	0	0	0	x	x	x	x	x	x x x x	x x x x	x x x x	1	x x	1	x x x x
Data processing immediate [2]	cond [1]	0	0	1	opcode				S	Rn	Rd	rotate	immediate			
Undefined instruction [3]	cond [1]	0	0	1	1	0	x	0	0	x x x x	x x x x	x x x x	x	x x	x	x x x x
Move immediate to status register	cond [1]	0	0	1	1	0	R	1	0	Mask	SBO	rotate	immediate			
Load/store immediate offset	cond [1]	0	1	0	P	U	B	W	L	Rn	Rd	immediate				
Load/store register offset	cond [1]	0	1	1	P	U	B	W	L	Rn	Rd	shift amount		shift	0	Rm
Undefined instruction	cond [1]	0	1	1	x	x	x	x	x	x x x x	x x x x	x x x x	x	x x	1	x x x x
Undefined instruction [4,7]	1 1 1 1	0	x	x	x	x	x	x	x	x x x x	x x x x	x x x x	x	x x	x	x x x x
Load/store multiple	cond [1]	1	0	0	P	U	S	W	L	Rn	register list					
Undefined instruction [4]	1 1 1 1	1	0	0	x	x	x	x	x	x x x x	x x x x	x x x x	x	x x	x	x x x x
Branch and branch with link	cond [1]	1	0	1	L	24-bit offset										
Branch and branch with link and change to Thumb [4]	1 1 1 1	1	0	1	H	24-bit offset										
Coprocessor load/store and double register transfers [6]	cond [5]	1	1	0	P	U	N	W	L	Rn	CRd	cp_num	8-bit offset			
Coprocessor data processing	cond [5]	1	1	1	0	opcode1				CRn	CRd	cp_num	opcode2		0	CRm
Coprocessor register transfers	cond [5]	1	1	1	0	opcode1			L	CRn	Rd	cp_num	opcode2		1	CRm
Software interrupt	cond [1]	1	1	1	1	swi number										
Undefined instruction [4]	1 1 1 1	1	1	1	1	x	x	x	x	x x x x	x x x x	x x x x	x	x x	x	x x x x

Data Processing Instructions

- Arithmetic and logical operations
- 3-address format:
  - Two 32-bit operands
    (op1 is register, op2 is register or immediate)
  - 32-bit result placed in a register
- Barrel shifter for op2 allows full 32-bit shift within instruction cycle

Data Processing Instructions (2)

- Arithmetic operations:
  - ADD, ADDC, SUB, SUBC, RSB, RSC
- Bit-wise logical operations:
  - AND, EOR, ORR, BIC
- Register movement operations:
  - MOV, MVN
- Comparison operations:
  - TST, TEQ, CMP, CMN

Data Processing Instructions (3)

Conditional codes

+

Data processing instructions

+

Barrel shifter

=

Powerful tools for efficient coded programs

Data Processing Instructions (4)

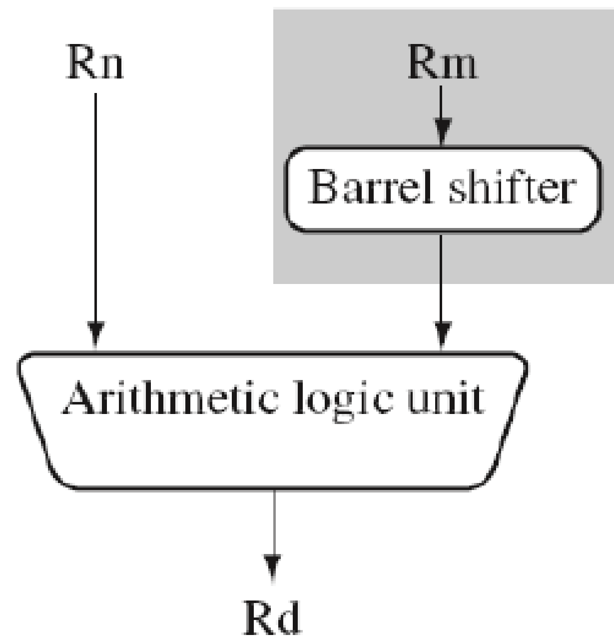
e.g.:

if (z==1) R1=R2+(R3*4)

compiles to

EQADDS R1,R2,R3, LSL #2

(SINGLE INSTRUCTION !)

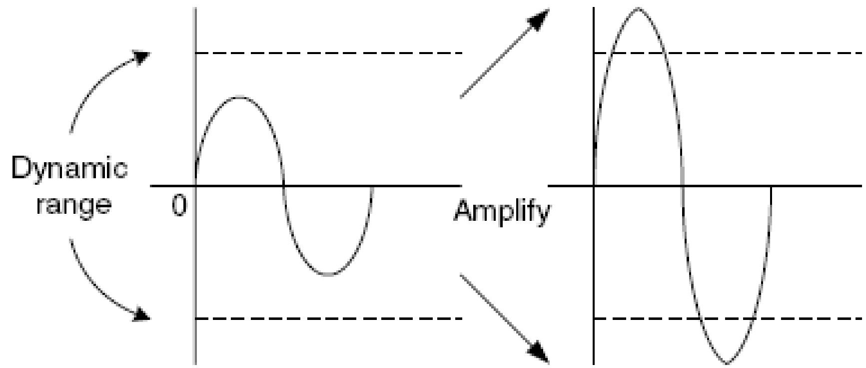


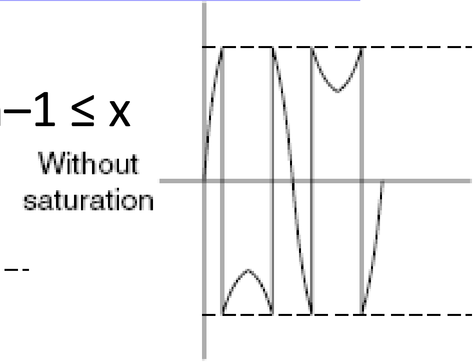
Multiply Instructions

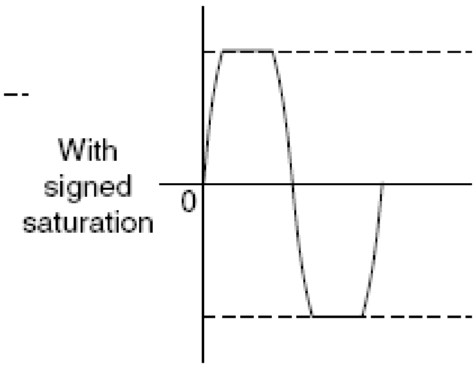
- Integer multiplication (32-bit result)
- Long integer multiplication (64-bit result)
- Built in Multiply Accumulate Unit (MAC)
- Multiply and accumulate instructions add product to running total

Saturated Arithmetic

The QADD and QSUB instructions apply the specified add or subtract, and then saturate the result to the signed range $-2n–1 \le x \le 2n–1-1$,







For signed *n*-bit saturation, this means that:

- if the value to be saturated is less than -2^{n-1}, the result returned is -2^{n-1}
- if the value to be saturated is greater than $2^{n-1}-1$, the result returned is $2^{n-1}-1$
- otherwise, the result returned is the same as the value to be saturated.

For unsigned *n*-bit saturation, this means that:

- if the value to be saturated is less than 0, the result returned is 0
- if the value to be saturated is greater than 2^{n-1}, the result returned is 2^{n-1}
- otherwise, the result returned is the same as the value to be saturated.

If the returned result is different from the value to be saturated, it is called saturation. If saturation occurs, the instruction sets the Q flag to 1 in the APSR. Otherwise, it leaves the Q flag unchanged. To clear the Q flag to 0, you must use the MSR instruction, see *MSR on page 186*.

To read the state of the Q flag, use the MRS instruction, see *MRS on page 185*.

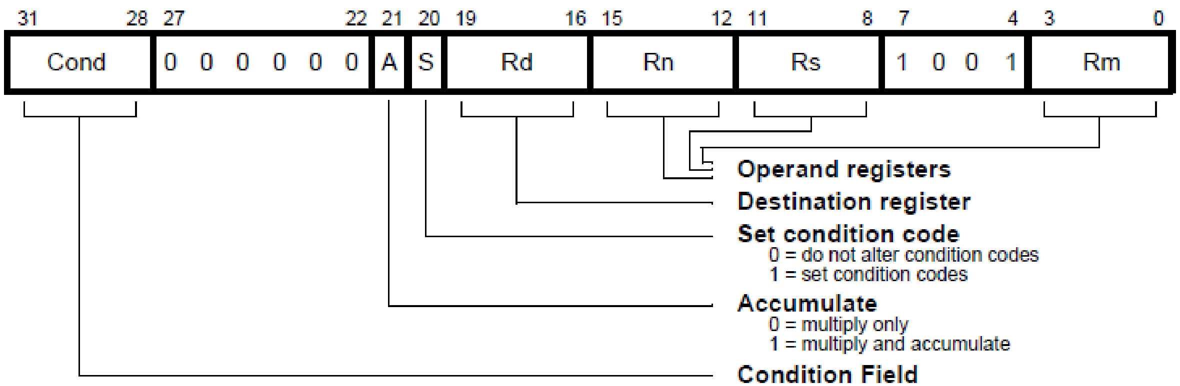
Multiply Instructions

- Instructions:

MUL	Multiply	32-bit result
MULA	Multiply accumulate	32-bit result
UMULL	Unsigned multiply	64-bit result
UMLAL	Unsigned multiply accumulate	64-bit result
SMULL	Signed multiply	64-bit result
SMLAL	Signed multiply accumulate	64-bit result

MUL, MULA

- Multiply, multiply accumulate




```
MUL{cond}{S} Rd,Rm,Rs
MLA{cond}{S} Rd,Rm,Rs,Rn
```

{cond}	two-character condition mnemonic. See ***Table 4-2: Condition code summary*** on page 4-5.
{S}	set condition codes if S present
Rd, Rm, Rs and Rn	are expressions evaluating to a register number other than R15.

```
MUL      R1,R2,R3      ; R1:=R2*R3
MLAEQS   R1,R2,R3,R4   ; Conditionally R1:=R2*R3+R4,
                       ; setting condition codes.
```

Data Transfer Instructions

- Load/store instructions
- Used to move signed and unsigned
  Word, Half Word and Byte to and from registers
- Can be used to load PC
  (if target address is beyond branch instruction range)

LDR	Load Word	STR	Store Word
LDRH	Load Half Word	STRH	Store Half Word
LDRSH	Load Signed Half Word	STRSH	Store Signed Half Word
LDRB	Load Byte	STRB	Store Byte
LDRSB	Load Signed Byte	STRSB	Store Signed Byte

(op2): *Memory Addressing Mode*

Used when accessing memory

Reading (Loading) data from memory: DESTINATION ← M(SOURCE) DESTINATION must be a register SOURCE is any (*op2*) value

```
LDR   r1, [r12]
```
R1 ← M(R12)

Writing (Storing) data into memory M(DESTINATION) ← SOURCE SOURCE must be a register DESTINATION is any (*op2*) value

```
STR   r1, [r12]
```
M(R12) ← R1

Store is the *only* ARM instruction to place the SOURCE before the DESTINATION

Memory Addressing (Syntax)

- Offset Addressing

`[Rn, #(value)]`	Offset Immediate
`[Rn, Rm]`	Offset Register
`[Rn, Rm, (shift) #(value)]`	Offset scaled

- Pre-Index Addressing

`[Rn, #(value)]!`	Pre-Index Immediate
`[Rn, Rm]!`	Pre-Index Register
`[Rn, Rm, (shift) #(value)]!`	Pre-Index scaled

- Post-Index Addressing

`[Rn], #(value)`	Post-Index Immediate
`[Rn], Rm`	Post-Index Register
`[Rn], Rm, (shift) #(value)`	Post-Index scaled

Memory Addressing (RTL)

- Offset Addressing: `LDR  R0, [R1, R2]`

  $(op2) \leftarrow R1 + R2$
  $MBR \leftarrow M((op2))$
  $R0 \leftarrow MBR$

- Pre-Index Addressing: `LDR  R0, [R1, R2]!`

  $(op2) \leftarrow R1 + R2$
  $R1 \leftarrow (op2)$
  $MBR \leftarrow M((op2))$
  $R0 \leftarrow MBR$

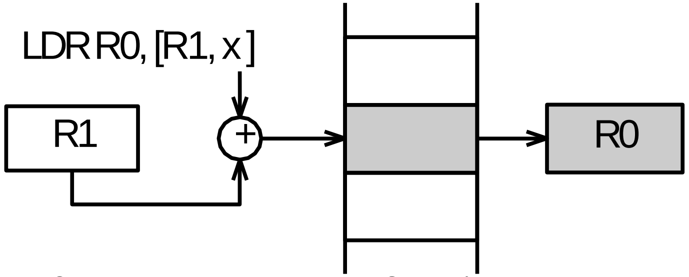
- Post-Index Addressing: `LDR  R0, [R1], R2`

  $(op2) \leftarrow R1$
  $R1 \leftarrow R1 + R2$
  $MBR \leftarrow M((op2))$
  $R0 \leftarrow MBR$

Memory Addressing (RTL)

- Offset Addressing: `LDR   R0, [R1, R2]`

(*op2*) ← R1 + R2
MBR ← M((*op2*))
R0 ← MBR



- Pre-Index Addressing: `LDR   R0, [R1, R2]!`

(*op2*) ← R1 + R2
R1 ← (*op2*)
MBR ← M((*op2*))
R0 ← MBR

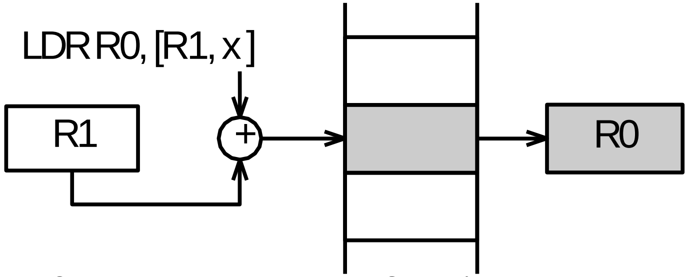
- Post-Index Addressing: `LDR   R0, [R1], R2`

(*op2*) ← R1
R1 ← R1 + R2
MBR ← M((*op2*))
R0 ← MBR

Memory Addressing (RTL)

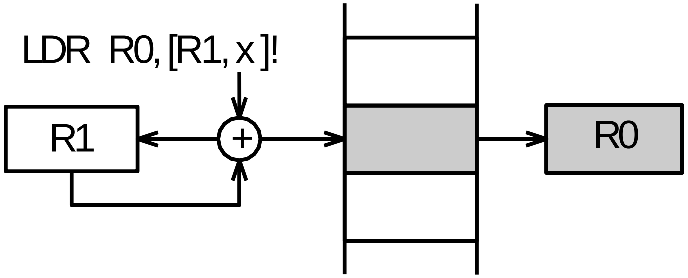
- Offset Addressing: `LDR  R0, [R1, R2]`

(*op2*)	←	R1 + R2
MBR	←	M((*op2*))
R0	←	MBR



- Pre-Index Addressing: `LDR  R0, [R1, R2]!`

(*op2*)	←	R1 + R2
R1	←	(*op2*)
MBR	←	M((*op2*))
R0	←	MBR



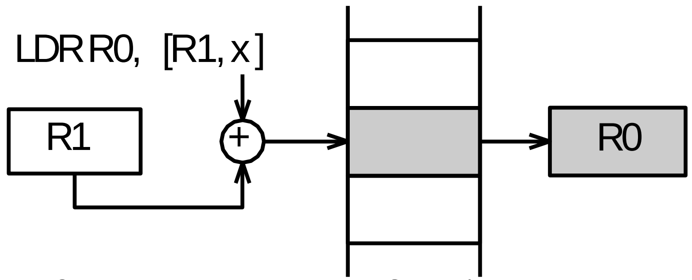
- Post-Index Addressing: `LDR  R0, [R1], R2`

(*op2*)	←	R1
R1	←	R1 + R2
MBR	←	M((*op2*))
R0	←	MBR

Memory Addressing (RTL)

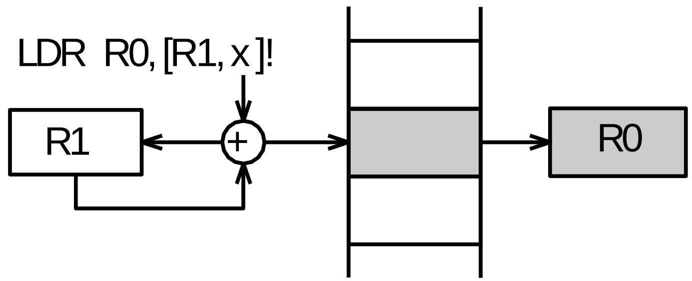
- Offset Addressing: `LDR R0, [R1, R2]`

  (*op2*) ← R1 + R2
  MBR ← M((*op2*))
  R0 ← MBR



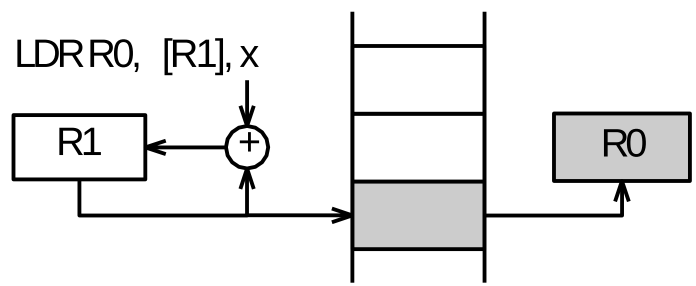
- Pre-Index Addressing: `LDR R0, [R1, R2]!`

  (*op2*) ← R1 + R2
  R1 ← (*op2*)
  MBR ← M((*op2*))
  R0 ← MBR



- Post-Index Addressing: `LDR R0, [R1], R2`

  (*op2*) ← R1
  R1 ← R1 + R2
  MBR ← M((*op2*))
  R0 ← MBR

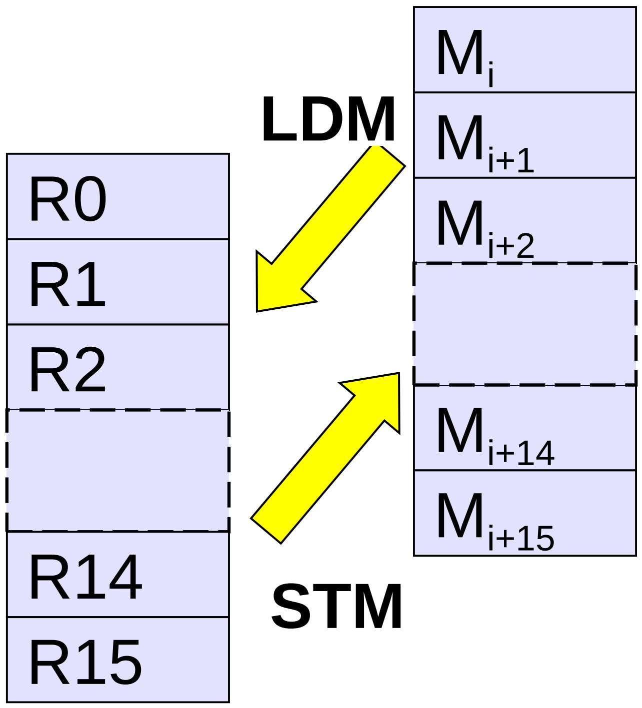


<offset> options

- An immediate constant
  - #10
- An index register
  - <Rm>
- A shifted index register
  - <Rm>, LSL #<shift>

Block Transfer Instructions

- Load/Store Multiple instructions (*LDM*/*STM*)
- Whole register bank or a subset copied to memory or restored with single instruction

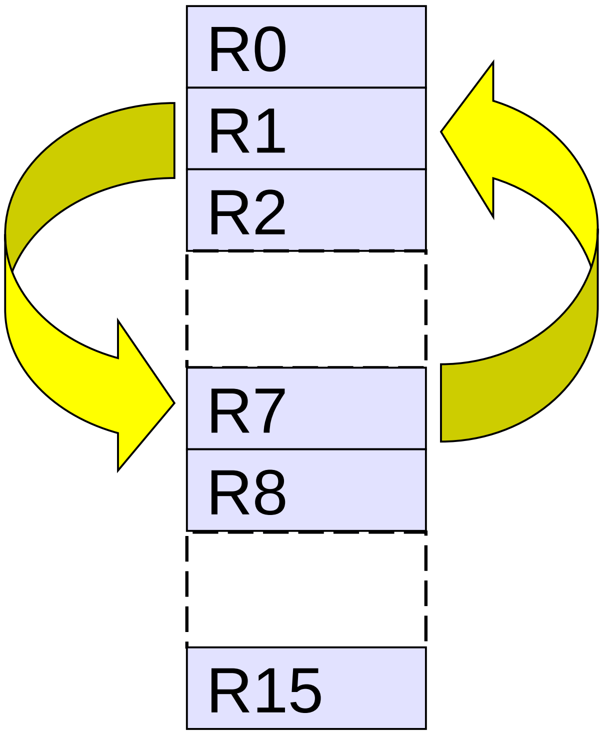


Swap Instruction

- Exchanges a word between registers
  - Two cycles

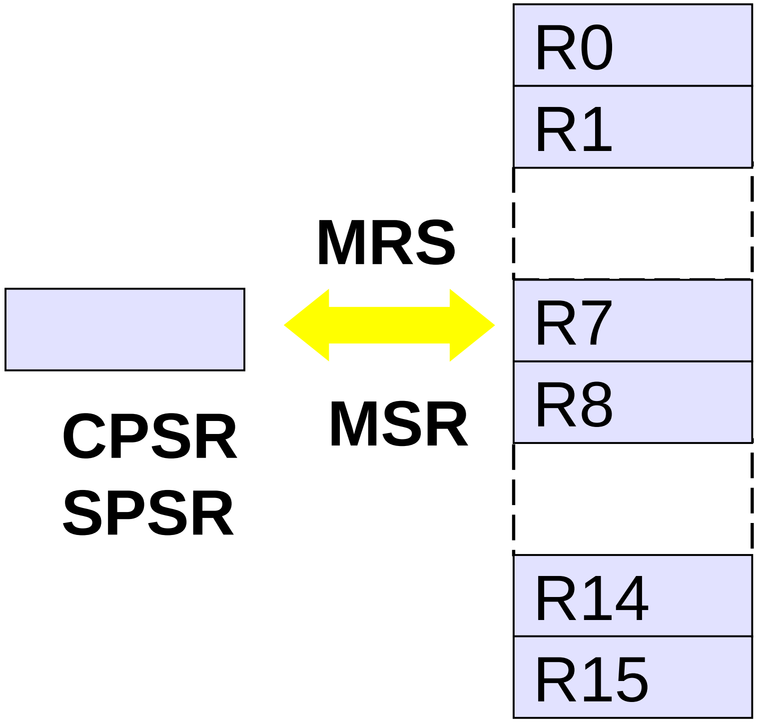
    but

  single atomic action
- Support for RT semaphores



Modifying the Status Registers

- Only indirectly
- *MSR* moves contents from CPSR/SPSR to selected GPR
- *MRS* moves contents from selected GPR to CPSR/SPSR
- Only in privileged modes



Branching Instructions

- *Branch* (B):

  jumps forwards/backwards up to 32 MB

- *Branch link* (BL):

  same + saves (PC+4) in LR

- Suitable for function call/return
- Condition codes for conditional branches

Program: sum16.s

```
Main
            LDR     R0, =Data1  ;load the address of the lookup
                                table
            EOR     R1, R1, R1  ;clear R1 to store sum
            LDR     R2, Length  ;init element count
Loop
            LDR     R3, [R0]    ;get the data
            ADD     R1, R1, R3  ;add it to r1
            ADD     R0, R0, #+4 ;increment pointer
            SUBS    R2, R2, #1  ;decrement count with zero set
            BNE     Loop        ;if zero flag is not set, loop

Table       DCW     &2040       ;table of values to be added
            DCW     &1C22
TablEnd     DCD     0

Length      DCW     (TablEnd - Table) / 4 ;because we’re having to align
```

Program: sum16.s

```
Main
            LDR     R0, =Data1  ;load the address of the lookup table
            EOR     R1, R1, R1  ;clear R1 to store sum
            LDR     R2, Length  ;init element count
Loop
            LDR     R3, [R0]    ;get the data
            ADD     R1, R1, R3  ;add it to r1
            ADD     R0, R0, #+4 ;increment pointer
            SUBS    R2, R2, #1  ;decrement count with zero set
            BNE     Loop        ;if zero flag is not set, loop

Table       DCW     &2040       ;table of values to be added
            DCW     &1C22
TablEnd     DCD     0

Length      DCW     (TablEnd - Table) / 4 ;because we're having to align
```

EOR Quick way of setting R1 to zero

Program: sum16.s

```
Main
        LDR     R0, =Data1  ;load the address of the lookup table
        EOR     R1, R1, R1  ;clear R1 to store sum
        LDR     R2, Length  ;init element count
Loop
        LDR     R3, [R0]    ;get the data
        ADD     R1, R1, R3  ;add it to r1
        ADD     R0, R0, #+4 ;increment pointer
        SUBS    R2, R2, #1  ;decrement count with zero set
        BNE     Loop        ;if zero flag is not set, loop

Table   DCW     &2040       ;table of values to be added
        DCW     &1C22
TablEnd DCD     0

Length  DCW     (TablEnd - Table) / 4 ;because we're having to align
```

Loop — Label the next instruction

Program: sum16.s

```
Main
        LDR     R0, =Data1  ;load the address of the lookup table
        EOR     R1, R1, R1  ;clear R1 to store sum
        LDR     R2, Length  ;init element count
Loop
        LDR     R3, [R0]    ;get the data
        ADD     R1, R1, R3  ;add it to r1
        ADD     R0, R0, #+4 ;increment pointer
        SUBS    R2, R2, #1  ;decrement count with zero set
        BNE     Loop        ;if zero flag is not set, loop

Table   DCW     &2040       ;table of values to be added
        DCW     &1C22
TablEnd DCD     0

Length  DCW     (TablEnd - Table) / 4 ;because we're having to align
```

ADD — Move pointer (R0) to next word

Program: sum16.s

```
Main
            LDR     R0, =Data1   ;load the address of the lookup table
            EOR     R1, R1, R1   ;clear R1 to store sum
            LDR     R2, Length   ;init element count
Loop
            LDR     R3, [R0]     ;get the data
            ADD     R1, R1, R3   ;add it to r1
            ADD     R0, R0, #+4  ;increment pointer
            SUBS    R2, R2, #1   ;decrement count with zero set
            BNE     Loop         ;if zero flag is not set, loop

Table       DCW     &2040        ;table of values to be added
            DCW     &1C22
TablEnd     DCD     0

Length      DCW     (TablEnd - Table) / 4 ;because we’re having to align
```

LDR/ADD — Using Post-index addressing we can remove the ADD:

```
LDR R3, [R0], #4
```

Program: sum16.s

```
Main
          LDR    R0, =Data1  ;load the address of the lookup table
          EOR    R1, R1, R1  ;clear R1 to store sum
          LDR    R2, Length  ;init element count
Loop
          LDR    R3, [R0]    ;get the data
          ADD    R1, R1, R3  ;add it to r1
          ADD    R0, R0, #+4 ;increment pointer
          SUBS   R2, R2, #1  ;decrement count with zero set
          BNE    Loop        ;if zero flag is not set, loop

Table     DCW    &2040       ;table of values to be added
          DCW    &1C22
TablEnd   DCD    0

Length    DCW    (TablEnd - Table) / 4 ;because we're having to align
```

SUBS	Subtract and set flags Decrement loop counter, R2

Program: sum16.s

```
Main
             LDR    R0, =Data1  ;load the address of the lookup table
             EOR    R1, R1, R1  ;clear R1 to store sum
             LDR    R2, Length  ;init element count
Loop
             LDR    R3, [R0]    ;get the data
             ADD    R1, R1, R3  ;add it to r1
             ADD    R0, R0, #+4 ;increment pointer
             SUBS   R2, R2, #1  ;decrement count with zero set
             BNE    Loop        ;if zero flag is not set, loop

Table        DCW    &2040       ;table of values to be added
             DCW    &1C22
TablEnd      DCD    0

Length       DCW    (TablEnd - Table) / 4 ;because we're having to align
```

BNE *Branch* to *Loop* if counter is *n*ot *e*qual to zero

Program: sum16.s

```
Main
        LDR     R0, =Data1  ;load the address of the lookup table
        EOR     R1, R1, R1  ;clear R1 to store sum
        LDR     R2, Length  ;init element count
Loop
        LDR     R3, [R0]    ;get the data
        ADD     R1, R1, R3  ;add it to r1
        ADD     R0, R0, #+4 ;increment pointer
        SUBS    R2, R2, #1  ;decrement count with zero set
        BNE     Loop        ;if zero flag is not set, loop

Table   DCW     &2040       ;table of values to be added
        DCW     &1C22
TablEnd DCD     0

Length  DCW     (TablEnd - Table) / 4 ;because we're having to align
```

DCW Assembler will calculate the length of data table for me

Program: sum16b.s

```
Main
        LDR     R0, =Data1      ;load the address of the lookup
                                table
        EOR     R1, R1, R1      ;clear R1 to store sum
        LDR     R2, Length      ;init element count
        CMP     R2, #0          ;zero length table ?
        BEQ     Done            ;yes => skip over sum loop
Loop
        LDR     R3, [R0]        ;get the data that R0 points to
        ADD     R1, R1, R3      ;add it to R1
        ADD     R0, R0, #+4     ;increment pointer
        SUBS    R2, R2, #0x1    ;decrement count with zero set
        BNE     Loop            ;if zero flag is not set, loop
Done
        STR     R1, Result      ;otherwise done - store result
        SWI     &11
```

Program: sum16b.s

```
Main
        LDR     R0, =Data1      ;load the address of the lookup table
        EOR     R1, R1, R1      ;clear R1 to store sum
        LDR     R2, Length      ;init element count
        CMP     R2, #0          ;zero length table ?
        BEQ     Done            ;yes => skip over sum loop
Loop
        LDR     R3, [R0]        ;get the data that R0 points to
        ADD     R1, R1, R3      ;add it to R1
        ADD     R0, R0, #+4     ;increment pointer
        SUBS    R2, R2, #0x1    ;decrement count with zero set
        BNE     Loop            ;if zero flag is not set, loop
Done
        STR     R1, Result      ;otherwise done - store result
        SWI     &11
```

EOR Quick way of setting R1 to zero

Program: sum16b.s

```
Main
        LDR     R0, =Data1      ;load the address of the lookup table
        EOR     R1, R1, R1      ;clear R1 to store sum
        LDR     R2, Length      ;init element count
        CMP     R2, #0          ;zero length table ?
        BEQ     Done            ;yes => skip over sum loop
Loop
        LDR     R3, [R0]        ;get the data that R0 points to
        ADD     R1, R1, R3      ;add it to R1
        ADD     R0, R0, #+4     ;increment pointer
        SUBS    R2, R2, #0x1    ;decrement count with zero set
        BNE     Loop            ;if zero flag is not set, loop
Done
        STR     R1, Result      ;otherwise done - store result
        SWI     &11
```

CMP	Is table length zero?

Program: sum16b.s

```
Main
        LDR     R0, =Data1      ;load the address of the lookup table
        EOR     R1, R1, R1      ;clear R1 to store sum
        LDR     R2, Length      ;init element count
        CMP     R2, #0          ;zero length table ?
        BEQ     Done            ;yes => skip over sum loop
Loop
        LDR     R3, [R0]        ;get the data that R0 points to
        ADD     R1, R1, R3      ;add it to R1
        ADD     R0, R0, #+4     ;increment pointer
        SUBS    R2, R2, #0x1    ;decrement count with zero set
        BNE     Loop            ;if zero flag is not set, loop
Done
        STR     R1, Result      ;otherwise done - store result
        SWI     &11
```

BEQ — Skip zero length tables
Protects from processing an empty list

Program: sum16b.s

```
Main
        LDR     R0, =Data1      ;load the address of the lookup table
        EOR     R1, R1, R1      ;clear R1 to store sum
        LDR     R2, Length      ;init element count
        CMP     R2, #0          ;zero length table ?
        BEQ     Done            ;yes => skip over sum loop
Loop
        LDR     R3, [R0]        ;get the data that R0 points to
        ADD     R1, R1, R3      ;add it to R1
        ADD     R0, R0, #+4     ;increment pointer
        SUBS    R2, R2, #0x1    ;decrement count with zero set
        BNE     Loop            ;if zero flag is not set, loop
Done
        STR     R1, Result      ;otherwise done - store result
        SWI     &11
```

LDR/ADD — Using Post-index addressing we can remove the ADD:

```
LDR R3, [R0], #4
```

Program: sum16b.s

```
Main
        LDR     R0, =Data1      ;load the address of the lookup table
        EOR     R1, R1, R1      ;clear R1 to store sum
        LDR     R2, Length      ;init element count
        CMP     R2, #0          ;zero length table ?
        BEQ     Done            ;yes => skip over sum loop
Loop
        LDR     R3, [R0]        ;get the data that R0 points to
        ADD     R1, R1, R3      ;add it to R1
        ADD     R0, R0, #+4     ;increment pointer
        SUBS    R2, R2, #0x1    ;decrement count with zero set
        BNE     Loop            ;if zero flag is not set, loop
Done
        STR     R1, Result      ;otherwise done - store result
        SWI     &11
```

SUBS/BNE — Decrement counter and branch to *Loop* if not zero

IF-THEN Instruction

- Another alternative to execute conditional code is the new 16-bit IF-THEN (IT) instruction
  - no change in program flow
  - no branching overhead
- Can use with 32-bit Thumb-2 instructions that do not support the 'S' suffix
- Example:

```
CMP R1, R2          ; If R1 = R2
IT EQ               ; execute next (1st)
                    ; instruction
ADDEQ R2, R1, R0    ; 1st instruction
```

- The conditional codes can be extended up to 4 instructions

Software Interrupt

- *SWI* instruction
  - Forces CPU into supervisor mode
  - Usage: SWI #n

31 28	27 24	23 0
Cond	Opcode	Ordinal

- Maximum 2^{24} calls
- Suitable for running privileged code and making OS calls

Barrier instructions

- Useful for multi-core & Self-modifying code

Instruction	Description
DMB	Data memory barrier; ensures that all memory accesses are completed before new memory access is committed
DSB	Data synchronization barrier; ensures that all memory accesses are completed before next instruction is executed
ISB	Instruction synchronization barrier; flushes the pipeline and ensures that all previous instructions are completed before executing new instructions