The ARM Cortex-M0 Processor Architecture Part-2

Copyright © ARM Ltd 2013

Module Syllabus

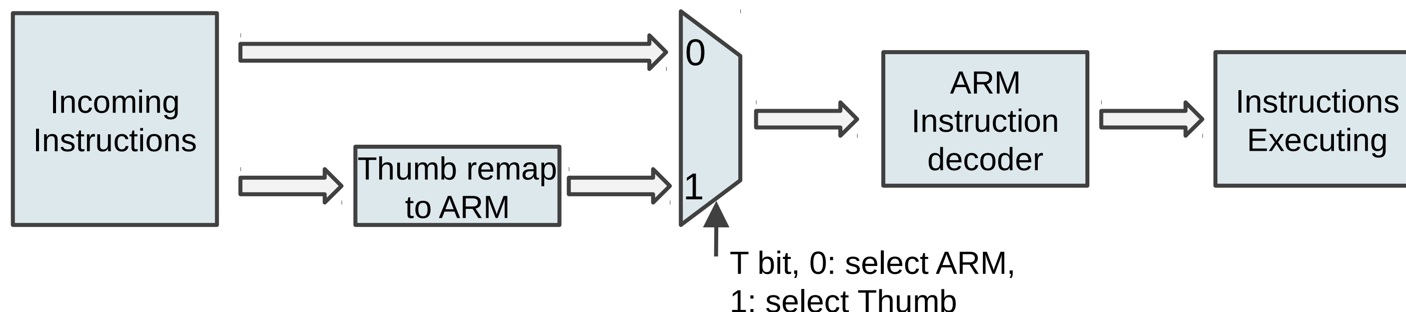
- ARM Cortex-M0 Processor Instruction Set
  - ARM and Thumb Instruction Set
  - Cortex-M0 Instruction Set
  - Data Accessing Instructions
  - Arithmetic Instructions
  - Program Flow Control
- Low-Power Features
  - Low-Power Requirements
  - Cortex-M0 Low-Power Features
  - Cortex-M0 Sleep Mode
  - Developing Low-Power Applications

Copyright © ARM Ltd 2013

ARM Cortex-M0 Processor Instruction Set

ARM and Thumb Instruction Set

- Early ARM processors
  - 32-bit instruction set, called the ARM instructions;
  - Powerful and good performance;
  - Larger program memory compared to 8-bit and 16-bit processors;
  - Larger power consumption.
- Thumb-1 instruction set
  - 16-bit instruction set, first used in ARM7TDMI processor in 1995;
  - Provides a subset of the ARM instructions, giving better code density compared to 32-bit RISC architecture;
  - Code size is reduced by ~30%, but performance is also reduced by ~20%;
  - Can be used together with ARM instructions using a multiplexer.



ARM and Thumb Instruction Set

- Thumb-2 instruction set
  - Consists of both 32-bit Thumb and original 16-bit Thumb-1 instruction sets;
  - Compared to 32-bit ARM instructions set, code size is reduced by ~26%, while keeping a similar performance;
- Cortex-M0 processor
  - Uses ARMv6-M architecture
  - Further reduces circuit size to a minimum;
  - Superset of 16-bit Thumb-1 instructions + minimum subset of 32-bit Thumb-2 instructions;

Copyright © ARM Ltd 2013

The Architecture for the Digital World®
ARM®

Cortex-M0 Instruction Set

16-bit Thumb Instructions Supported on Cortex-M0									
ADCS	ADDS	ADR	ANDS	ASRS	B	BIC	BLX	BKPT	BX
CMN	CMP	CPS	EORS	LDM	LDR	LDRH	LDRSH	LDRB	LDRSB
LSLS	LSRS	MOV	MVN	MULS	NOP	ORRS	POP	PUSH	REV
REV16	REVSH	ROR	RSB	SBCS	SEV	STM	STR	STRH	STRB
SUBS	SVC	SXTB	SXTH	TST	UXTB	UXTH	WFE	WFI	YIELD

32-bit Thumb Instructions Supported on Cortex-M0									
BL	DSB	DMB	ISB	MRS	MSR				

Note: some instructions have a “S” suffix, which is an updated ARM syntax used to support for the Unified Assembler Language (UAL).

Copyright © ARM Ltd 2013

Cortex-M0 Instruction Set

Instruction Type	Instructions
Move	MOV, MOVS, MRS, MSR
Load/Store	LDR, LDRH, LDRB, LDRSH, LDRSB, LDM, LDMIA, STR, STRH, STRB, STMIA
Stack	PUSH, POP
Add, Subtract, Multiply	ADDS, ADCS, ADR, SUBS, SBCS, RSBS, MULS
Compare	CMP, CMN
Logical	ANDS, ORRS, EORS, MVNS, BICS, TST
Shift and Rotate	ASRS, LSRS, LSLS, RORS
Reverse	REV, REV16, REVSH
Extend	SXTB, SXTH, UXTB, UXTH
Conditional branch	B, B <cond>, BL, BX, BLX
Memory barrier	DMB, DSB, ISB
Exception	SVC, CPS
Sleep mode	WFI, WFE, SEV
Other	NOP, BKPT, YIELD

Copyright © ARM Ltd 2013

Cortex-M0 Instruction Set

- ARM assembly syntax:

  label

  mnemonic operand1, operand2, … ; Comments

  - Label is used as a reference to an address location;
  - Mnemonic is the name of the instruction;
  - Operand1 is the destination of the operation;
  - Operand2 is normally the source of the operation;
  - Comments are written after " ; ", which does not affect the program;
  - For example

    MOVS R3, #0x11 ;Set register R3 to 0x11

  - Note that the assembly code can be assembled by either ARM assembler (armasm) or assembly tools from a variety of vendors (e.g. GNU tool chain). When using GNU tool chain, the syntax for label and comments is slightly different

Cortex-M0 Instruction Set

- Cortex-M0 Suffix
  - Some instructions can be followed by suffixes to update processor flags or execute the instruction on a certain condition.

Suffix	Description	Example	Example explanation
S	Update APSR (flags)	ADDS R1, #0x21	Add 0x21 to R1 and update APSR
EQ, NE, CS, CC, MI, PL, VS, VC, HI, LS, GE, LT, GT, LE	Condition execution e.g. EQ= equal, NE= not equal, LT= less than	BNE label	Branch to the label if not equal

Copyright © ARM Ltd 2013

Register Access: MOVE

Syntax	Description	Example	Example explanation
MOV <Rd>, <Rm>	Move register into register	MOV R0, R1	Copy value in R1 to R0
MOVS <Rd>, #immed8	Move immediate data (0-255) into register	MOV R0, #0x31	Copy number 0x31 (hex) to R0
MOVS <Rd>, <Rm>	Move register into register and update flags	MOVS R0, R1	Copy value in R1 to R0 and update flags
MOVS <Rd>, #immed8	Move immediate data (0-255) into register and update flags	MOVS R0, #0x31	Copy number 0x31 (hex) to R0 and update flags
MRS <Rd>, <SpecialReg>	Move special register into register	MRS R1, CONTROL	Copy CONTROL register into R1
MSR <SpecialReg>, <Rd>	Move register to special register	MSR PRIMASK, R0	Copy R0 to PRIMASK register

Memory Access: LOAD

Syntax	Description	Example	Example explanation
LDR <Rt>, [<Rn>,<Rm>]	Load word from memory Rt= memory [<Rn>+<Rm>]	LDR R0, [R1, R2]	R0= memory [R1+R2]
LDRH <Rt>, [<Rn>,<Rm>]	Load half word from memory Rt= memory [<Rn>+<Rm>]	LDRH R0, [R1, R2]	R0= memory [R1+R2]
LDRB <Rt>, [<Rn>,<Rm>]	Load a byte from memory Rt= memory [<Rn>+<Rm>]	LDRB R0, [R1, R2]	R0= memory [R1+R2]
LDR <Rt>, [<Rn>,#immed5]	Load word from memory Rt= memory [<Rn> + ZeroExtend (#immed5<<2)]	LDR R0, [R1, #0x4]	R0= memory [R1+0x4]
LDRH <Rt>, [<Rn>,#immed5]	Load word from memory Rt= memory [<Rn> + ZeroExtend (#immed5<<1)]	LDRH R0, [R1, #0x2]	R0= memory [R1+0x2]
LDRB <Rt>, [<Rn>,#immed5]	Load word from memory Rt= memory [<Rn> + ZeroExtend (#immed5)]	LDRB R0, [R1, #0x1]	R0= memory [R1+0x1]

Copyright © ARM Ltd 2013

The Architecture for the Digital World®
ARM®

Memory Access: LOAD

Syntax	Description	Example	Example explanation
LDR <Rt>, =direct number	Load an immediate data into a register	LDR R0, =0x12345678	R0= 0x12345678
LDR <Rt>, [PC (SP), #immed8]	Load word from memory Rt= memory [PC (SP) + #immed8<<2]	LDR R0, [PC, #0x04]	R0= memory [PC+ 0x4]
LDRSH <Rt>, [<Rn>,<Rm>]	Load signed half word from memory Rt= SignExtend (memory [Rn+Rm])	LDRSH R0, [R1, R2]	R0= memory [R1+R2]
LDRSB <Rt>, [<Rn>,<Rm>]	Load signed byte from memory Rt= SignExtend (memory [Rn+Rm])	LDRSB R0, [R1, R2]	R0= memory [R1+R2]

ARM University Program
Copyright © ARM Ltd 2013

Memory Access: STORE

Syntax	Description	Example	Example explanation
STR <Rt>, [<Rn>,<Rm>]	Write word to memory memory [<Rn>+<Rm>] = Rt	STR R0, [R1, R2]	memory [R1+R2] = R0
STRH <Rt>, [<Rn>,<Rm>]	Write half word to memory memory [<Rn>+<Rm>] = Rt	STRH R0, [R1, R2]	memory [R1+R2] = R0
STRB <Rt>, [<Rn>,<Rm>]	Write byte to memory memory [<Rn>+<Rm>] = Rt	STRB R0, [R1, R2]	memory [R1+R2] = R0
STR <Rt>, [<Rn>,#immed5]	Write word to memory memory [<Rn> + ZeroExtend (#immed5<<2)] = Rt	STR R0, [R1, #0x4]	memory [R1+0x4] = R0
STRH <Rt>, [<Rn>,#immed5]	Write half word to memory memory [<Rn> + ZeroExtend (#immed5<<1)] = Rt	STRH R0, [R1, #0x2]	memory [R1+0x2] = R0
STRB <Rt>, [<Rn>,#immed5]	Write byte to memory memory [<Rn> + ZeroExtend (#immed5)] = Rt	STRB R0, [R1, #0x1]	memory [R1+0x1] = R0
STR <Rt>, [SP, #immed8]	Write word to memory memory [SP + ZeroExtend (#immed5<<2)] = Rt	STRB R0, [SP, #0x4]	memory [SP+0x4] = R0

Copyright © ARM Ltd 2013

The Architecture for the Digital World® ARM®

Multiple Data Access

Syntax	Description	Example	Example explanation
LDM <Rn>, {<Ra>, <Rb>, …}	Load multiple registers from memory	LDM R0, {R1, R2 – R7}	R1= memory [R0], R2= memory [R0+4], …
LDMIA <Rn>!, {<Ra>, <Rb>, …}	Load multiple registers from memory, and then increment Rn to the last address + 4	LDMIA R0, {R1, R2 – R7}	R1= memory [R0], R2= memory [R0+4], … R0= R0 + 4×7
STMIA <Rn>!, {<Ra>, <Rb>, …}	Store multiple registers to memory, and then increment Rn to the last address + 4	STMIA R0, {R1, R2 – R7}	memory [R0] = R1, memory [R0+4] = R2, … R0= R0 + 4×7

Copyright © ARM Ltd 2013

Stack Access: PUSH and POP

Syntax	Description	Example	Example explanation
PUSH {<Ra>, <Rb>, …}	Write single or multiple registers into memory and update base register	PUSH { R0, R1, R2 }	Memory [SP- 4]= R0, Memory [SP- 8]= R1, Memory [SP- 12]= R2, SP = SP-12
POP {<Ra>, <Rb>, …}	Read single or multiple registers from memory and update base register	POP { R0, R1, R2 }	R2 = Memory [SP], R1 = Memory [SP+ 4], R0 = Memory [SP+ 8], SP = SP+12

Copyright © ARM Ltd 2013

Arithmetic ADD

Syntax	Description	Example	Example explanation
ADDS <Rd>, <Rn>, <Rm>	Add two registers Rd = Rn + Rm Update APSR	ADDS R0, R1, R2	R0 = R1 + R2 Update APSR
ADDS <Rd>, <Rn>, #immed3	Add an immediate constant into a register Rd = Rn + ZeroExtend(#immed3), update APSR	ADDS R0, R1, #0x01	R0 = R1 + 0x01 Update APSR
ADDS <Rd>, #immed8	Add an immediate constant into a register Rd = Rd + ZeroExtend(#immed8), update APSR	ADDS R0, #0x01	R0 = R0 + 0x01 Update APSR
ADD <Rd>, <Rn>	Add two registers without updating APSR Rd = Rd + Rm	ADD R0, R1	R0 = R0 + R1
ADC <Rd>, <Rm>	Add with carry and update APSR Rd = Rd + Rm + Carry Update APSR	ADCS R0, R1	R0 = R0 + R1 + Carry Update APSR
ADD <Rd>, PC, #immed8	Add an immediate constant with PC to a register without updating APSR, alternative to ADR	ADD R0, PC, #0x04	R0 = PC + 0x04

Arithmetic SUB, MUL

Syntax	Description	Example	Example explanation
SUBS <Rd>, <Rn>, <Rm>	Subtract two registers Rd = Rn - Rm Update APSR	SUBS R0, R1, R2	R0 = R1 - R2 Update APSR
SUBS <Rd>, <Rn>, #immed3	Subtract a register with an immediate constant Rd = Rn - ZeroExtend(#immed3), update APSR	SUBS R0, R1, #0x01	R0 = R1 - 0x01 Update APSR
SUBS <Rd>, #immed8	Subtract a register with an immediate constant Rd = Rd - ZeroExtend(#immed8), update APSR	SUBS R0, #0x01	R0 = R0 - 0x01 Update APSR
SBCS <Rd>, <Rd>, <Rm>	Subtract with carry (borrow) Rd = Rd - Rm - Borrow Update APSR	SBCS R0, R0, R1	R0 = R0 - R1 - Borrow Update APSR
RSBS <Rd>, <Rm>, #0	Reverse subtract (negative) Rd = 0 - Rm, update APSR	RSBS R0, R0, #0	R0 = - R0
MULS <Rd>, <Rm>, <Rd>	Multiply two registers Rd = Rd × Rm, update APSR	MULS R0, R1, R0	R0 = R0 × R1

Arithmetic CMP

Syntax	Description	Example	Example explanation
CMP <Rn>, <Rm>	Compare two registers Calculate Rn - Rm, Update APSR but subtract result is not stored	CMP R0, R1	Calculate R0-R1, Update APSR
CMP <Rn>, #immed8	Compare a register with an immediate constant Calculate Rd – ZeroExtend (#immed8), update APSR but subtract result is not stored	CMP R0, #0x01	Calculate R0-1, Update APSR
CMN <Rn>, <Rm>	Compare negative Calculate Rn – NEG(Rm), update APSR but subtract result is not stored	CMN R0, R1	Calculate R0+R1, Update APSR

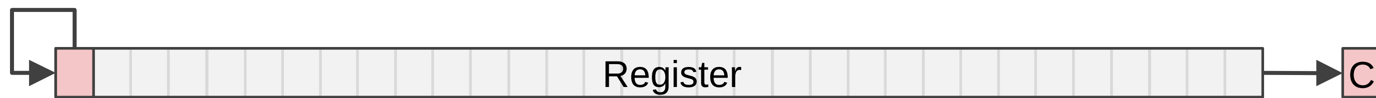
Logic Operation

Syntax	Description	Example	Example explanation
ANDS <Rd>, <Rm>	Logical AND Rd = AND (Rd, Rm), update APSR	ANDS R0, R1	AND (R0, R1), update APSR
ORRS <Rd>, <Rm>	Logical OR Rd = OR (Rd, Rm), update APSR	ORRS R0, R1	OR (R0, R1), update APSR
EORS <Rd>, <Rm>	Logical exclusive OR Rd = XOR (Rd, Rm), update APSR	EORS R0, R1	XOR (R0, R1), update APSR
MVNS <Rd>, <Rm>	Logical bitwise NOT Rd= NOT(Rm), update APSR	MVNS R0, R1	R0 = NOT (R1), update APSR
BICS <Rd>, <Rm>	Logical bitwise clear Rd = AND (Rd, NOT(Rm)), update APSR	BICS R0, R1	AND (R0, NOT(R1)), update APSR
TST <Rd>, <Rm>	Test (bitwise AND) Calculate AND (Rd, Rm), update APSR but the AND result is not stored	TST R0, R1	Calculate AND (R0, R1), update APSR

Copyright © ARM Ltd 2013

Shift Operation

Syntax	Description	Example	Example explanation
ASRS <Rd>, <Rm>	Arithmetic shift right Rd = Rd >> Rm, last bit shift out is copy to APSR.C, APSR.N, APSR.Z are also updated	ASRS R0, R1	R0 = R0 >> R1, update APSR
ASRS <Rd>, <Rm>, #immed5	Rd = Rm >> immed5, , last bit shift out is copy to APSR.C, APSR.N, APSR.Z are also updated	ASRS R0, R1, #0x01	R0 = R1 >> 1, update APSR
LSRS <Rd>, <Rm>	Logic shift right Rd = Rd >> Rm, last bit shift out is copy to APSR.C, APSR.N, APSR.Z are also updated	LSRS R0, R1	R0 = R0 >> R1, update APSR
LSRS <Rd>, <Rm>, #immed5	Logic shift right Rd = Rm >> #immed5, last bit shift out is copy to APSR.C, APSR.N, APSR.Z are also updated	LSRS R0, R1, #0x01	R0 = R1 >> 1, update APSR



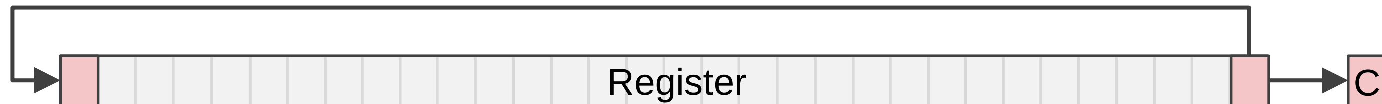
Arithmetic shift right (ASR)

Logic shift right (LSR)

Shift Operation

Syntax	Description	Example	Example explanation
LSLS <Rd>, <Rm>	Logic shift left Rd = Rd << Rm, last bit shift out is copy to APSR.C, APSR.N, APSR.Z are also updated	LSLS R0, R1	R0 = R0 << R1, update APSR
LSLS <Rd>, <Rm>, #immed5	Logic shift left Rd = Rm << #immed5, last bit shift out is copy to APSR.C, APSR.N, APSR.Z are also updated	LSLS R0, R1, #0x01	R0 = R1 << 1, update APSR
RORS <Rd>, <Rm>	Rotate right Rd = Rd rotate right by Rm bits, last bit shift out is copy to APSR.C, APSR.N, APSR.Z are also updated	RORS R0, R1	R0 = R0 >> R1 (rotate) Update APSR

Logic shift left (LSL)

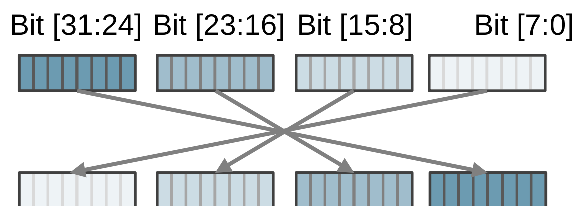


Rotate right (ROR)

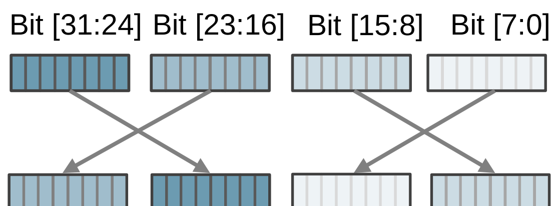
Copyright © ARM Ltd 2013

Reverse Ordering Operation

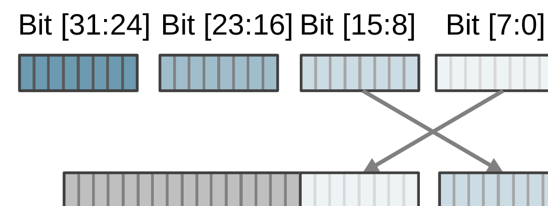
Syntax	Description	Example	Example explanation
REV <Rd>, <Rm>	Byte order reverse Rd = {Rm [7:0], Rm [15:8], Rm [23:16], Rm [31:24] }	REV R0, R1	R0 = {R1 [7:0], R1 [15:8], R1 [23:16], R1 [31:24] }
REV16 <Rd>, <Rm>	Byte order reverse within half word Rd = {Rm [23:16], Rm [31:24], Rm [7:0], Rm [15:8] }	REV R0, R1	R0 = {R1 [23:16], R1 [31:24], R1 [7:0], R1 [15:8] }
REVSH <Rd>, <Rm>	Byte order reverse within lower half word, then sign extend result Rd = SignExtend { Rm [7:0], Rm [15:8] }	REVSH R0, R1	R0 = SignExtend { R1 [7:0], R1 [15:8] }



REV operation



REV16 operation



REVSH operation

Extend Operation

Syntax	Description	Example	Example explanation
SXTB <Rd>, <Rm>	SignExtend lowest byte in a word of data Rd = SignExtend (Rm [7:0])	SXTB R0, R1	R0 = SignExtend (R1 [7:0])
SXTH <Rd>, <Rm>	SignExtend lower half word in a word of data Rd = SignExtend (Rm [15:0])	SXTH R0, R1	R0 = SignExtend (R1 [15:0])
UXTB <Rd>, <Rm>	UnsignExtend lowest byte in a word of data Rd = ZeroExtend (Rm [7:0])	UXTB R0, R1	R0 = ZeroExtend (R1 [7:0])
UXTH <Rd>, <Rm>	UnsignExtend lower half word in a word of data Rd = ZeroExtend (Rm [15:0])	UXTH R0, R1	R0 = ZeroExtend (R1 [15:0])

Copyright © ARM Ltd 2013

Program Flow Control

Syntax	Description	Example	Example explanation
B <label>	Branch to an address (unconditional) Branch range is +/- 2046 bytes of current PC	B loop	Change PC to the address with label of “loop”
B <cond> <label>	Conditional branch, branch to an address depending of APSR	BEQ loop	If APSR.Z =1, then change PC to the address with label of “loop”
BL <label>	Branch and link, branch to an address and store return address to LR. Branch range is +/- 16MB of current PC. Usually used for calling a subroutine or function. Once the function is completed, it can be return by executing “BX LR”	BL functionA	Change PC to the address with label of “functionA”, LR = PC +4
BX <Rm>	Branch and exchange. Branch to an address specified by a register, and change processor state (ARM or Thumb) depending on bit[0] of the register (0 for ARM; 1 for Thumb)	BX R0	PC = R0
BLX	Branch and link with exchange. Branch to an address specified by a register, save return address to LR, and change processor state depending on bit[0] of the register.	BLX R0	PC = R0 LR = PC +4

Copyright © ARM Ltd 2013

Suffixes for Conditional Branch

Suffix	Branch condition	Flags (APSR)
EQ	Equal	APSR.Z = 1
NE	Not equal	APSR.Z = 0
CS	Carry set	APSR.C = 1
CC	Carry clear	APSR.C = 0
MI	Negative	APSR.N = 1
PL	Positive	APSR.N = 0
VS	Overflow	APSR.V = 1
VC	No overflow	APSR.V = 0
HI	Unsigned higher	APSR.C = 1 and APSR.Z = 0
LS	Unsigned lower or same	APSR.C = 0 or APSR.Z = 1
GE	Signed greater than or equal	APSR.N = APSR.V
LT	Singed less than	APSR.N != APSR.V
GT	Signed greater than	(APSR.Z = 0) and (APSR.N = APSR.V)
LE	Signed less than or equal	(APSR.Z = 1) or (APSR.N != APSR.V)

Copyright © ARM Ltd 2013

Memory Barrier Instructions

Syntax	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 the next instruction is executed
ISB	Instruction synchronization barrier, Flushes the pipeline and ensures that all previous instructions are completed before executing new instructions

Copyright © ARM Ltd 2013

Exception-Related Instructions

Syntax	Description	Example	Example explanation
SVC <immed8>	Supervisor call Trigger the SVC exception	SVC #3	Trigger SVC exception , with parameter = 3
CPS	Change processor state Enable or disable interrupt Does not block NMI and hard fault handler	CPSIE I CPSID I	Enable interrupt (clearing PRIMASK) Disable interrupt (Setting PRIMASK)

Copyright © ARM Ltd 2013

Sleep Mode Related Instructions

Syntax	Description
WFI	Wait for interrupt. Stops program execution until an interrupt arrives or until the processor enters a debug state
WFE	Wait for event. Stop the program execution until an event arrives (internal event register is set) or until the processor enters a debug state
SEV	Send event to all processors in multiprocessing environment (including itself)

Copyright © ARM Ltd 2013

Other Instructions

Syntax	Description	Example	Example explanation
NOP	No operation Used to produce instruction alignment or to introduce delay	NOP	No operation
BKPT <immed8>	Breakpoint Put processor into halting stage, during which users carry out debug tasks through the debugger. Usually BKPT is inserted by a debugger to replace the original instruction. An 8-bit immediate data can be used as an identifier for the debugger.	BKPT #0	Breakpoint, with identifier of 0.
YIELD	Indicate task is stalled Used in multithread system, indicates that the current thread is delayed (e.g. waiting for hardware) and can be swapped out. Execute as NOP on the CotextM0 processor	YIELD	Same as NOP in Cortex-M0 processor

Copyright © ARM Ltd 2013

Data Insertion and Alignment

- Insert data inside programs
  - DCD: insert a word-size data;
  - DCB: insert a byte-size data;
  - ALIGN:
    - used before inserting a word-size data;
    - Uses a number to determine the alignment size.

- For example

```
...
ALIGN       4                   ; Align to a word boundary
MY_DATA         DCD0x12345678   ; Insert a word-size data
MY_STRING   DCB"Hello"  ,   0   ; Null terminated string
...
```


Copyright © ARM Ltd 2013

Low-Power Features

Copyright © ARM Ltd 2013

Low-Power Requirements

- In some circumstances, the system has to meet different low-power requirements, for example:
  - Low operating power
    - During system operating time, the power is mainly consumed by the transistor switching current;
  - Low standby power
    - In the standby mode, the power consumption mainly resides in the leakage circuits, clock circuits, active peripherals, analog systems, and RAM retention.
  - High energy efficiency
    - Measured by the ratio between the processing capability and the power consumption, such as Dhrystone Million Instructions Per Second / micro Walt (DMIPS/ mW)
  - Wakeup latency
    - The elapsed time from an interrupt request to the time the processor starts ISR. The less time consumed, the more time the processor can stay in sleep mode, and hence improve battery life.

Copyright © ARM Ltd 2013

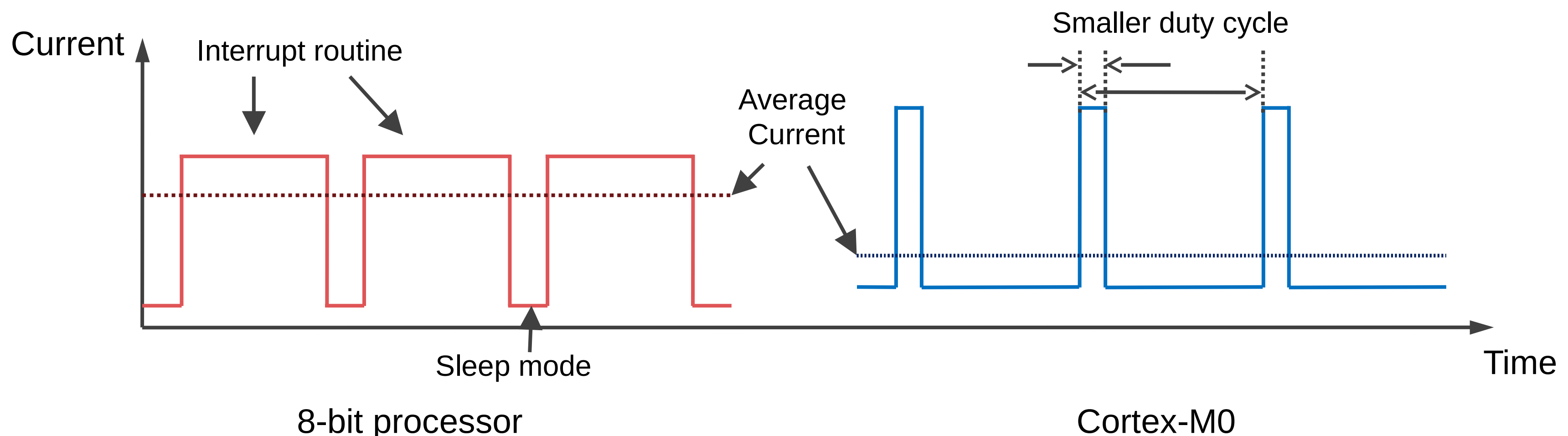
ARM and Cortex-M0 Low-Power Advantages

- The ARM processors are designed with low-power applications in mind, with 20 years of low-power processor design experience;
- Different ARM processor are optimized for different groups of applications based on different low-power requirements;
- Cortex-M0 processor is designed to target smaller embedded systems to meet ultra-low-power and high energy efficiency requirements;
  - Cortex-M0 is 32-bit processor, but only requires 12K gates, making it smaller than many 16-bit processor and much smaller than other 32-bit processors;
  - On the other hand, the 32-bit architecture gives Cortex-M0 higher processing capability, which significantly reduces the operation duty cycle.

Copyright © ARM Ltd 2013

Cortex-M0 Low-Power Advantages

- 8-bit processor
  - Might requires lower current in both working and sleep modes;
  - But slower processing speed, hence longer processing time;
- Cortex-M0 (32-bit)
  - Might requires higher current in both working and sleep modes;
  - But faster processing speed, therefore shorter processing time;
- In an interrupt-driven application, Cortex-M0 requires less energy consumption by reducing its duty cycle;



Copyright © ARM Ltd 2013

Cortex-M0 Low-Power Features

- Two architectural sleep modes:
  - Normal sleep and deep sleep;
- Two instructions for entering sleep modes:
  - WFE (Wait for Event) and WFI (Wait for Interrupt);
- Sleep-on-Exit feature
  - Allow processor to stay in sleep mode as much as possible;
- Wakeup Interrupt Controller (WIC)
  - An optional feature allows the clock of the processor to be completely removed during deep sleep;
- Low-power design implementation
  - Since the gate count is very low, the static leakage power is tiny compared to other 32-bit processors.

Cortex-M0 Sleep Mode

- The Cortex-M0 processor supports two sleep modes: normal sleep mode and deep sleep mode;
- The exact meaning and behaviours of the two modes depends on the implementation of the microcontroller, e.g.
  - Normal sleep – switch off some of the clock signals;
  - Deep sleep – reduce voltage supplies to the memory blocks, switch off additional components;
- To enter a sleep mode, we can use WFE (wait for event) or WFI (wait for interrupt) instructions;
- The processor can exit the sleep mode, if an event (including interrupts) occurs;

Copyright © ARM Ltd 2013

System Control Register

- The sleep features can be programed by accessing the System Control Register (SCR) in the in the System Control Block (at address 0xE000ED10).
- System control register (0xE000ED10)

Bits	Field	Description
0	Reserved	-
1	SleepOnExit	Sleep-on-Exit enable bit
2	SleepDeep	Sleep mode type bit, 0: normal sleep; 1: deep sleep
3	Reserved	-
4	SeVOnPend	Send Event on Pend bit, enable generation of event by a new interrupt pending status
[31:5]	Reserved	-

- SCR can be accessed by register symbol “SCB->SCR”, for example:
  - SCB -> SCR = 1<1; //Enable sleep-on-exit bit

Copyright © ARM Ltd 2013

Processor Wakeup Conditions

- The processor can exit the sleep mode on different conditions

Type	Priority	SeVOnPend	PRIMASK	Wake up	Execute ISR
WFE	IRQ priority > current level	--	0	Yes	Yes
	IRQ priority > current level	0	1	No	No
	IRQ priority ≤ current level	0	--	No	No
	IRQ priority ≤ current level	1	--	Yes	No
WFI	IRQ priority > current level	--	0	Yes	Yes
	IRQ priority > current level	--	1	Yes	No
	IRQ priority ≤ current level	--	--	No	No

Copyright © ARM Ltd 2013

Send-Event-on-Pending

- The Send-Event-on-Pend feature allows any interrupts (including disabled ones) to wake up the processor that stays in the sleep mode by executing WFE instruction.

- When SEVONPEND bit is set, if an interrupt switches from inactive to pending state, an event will be generated, and the processor will be waken up from WFE sleep.

Copyright © ARM Ltd 2013

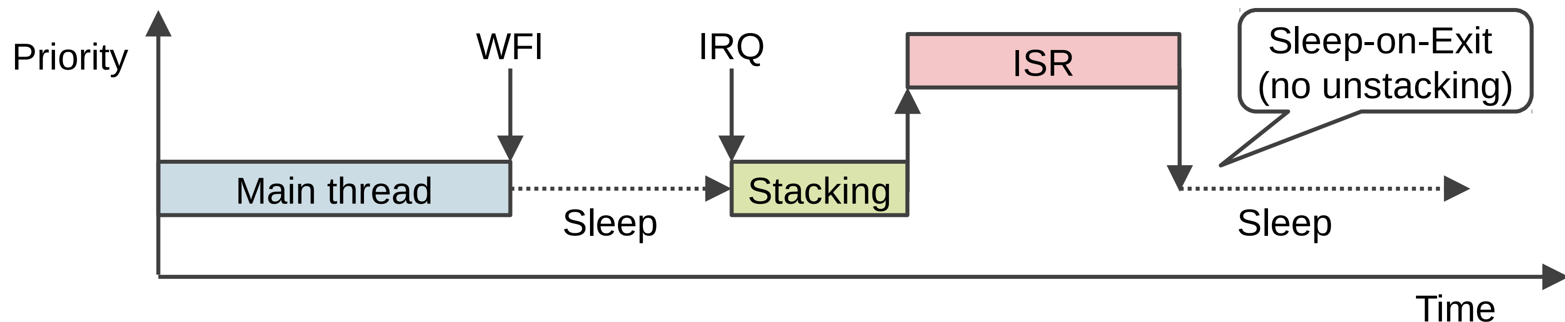
Sleep-on-Exit Feature

- The Sleep-on-Exit feature allows the processor to enter sleep mode as soon as all the exception handlers are completed;
  - Same as executing WFI immediately after finishing the exception handler;
  - The processor does not sleep if the program returns from one exception handler to another exception handler;
  - Hence the processor can stay in the sleep mode as much as possible;
  - Ideal for interrupt-driven applications.

Copyright © ARM Ltd 2013

Sleep-on-Exit Feature

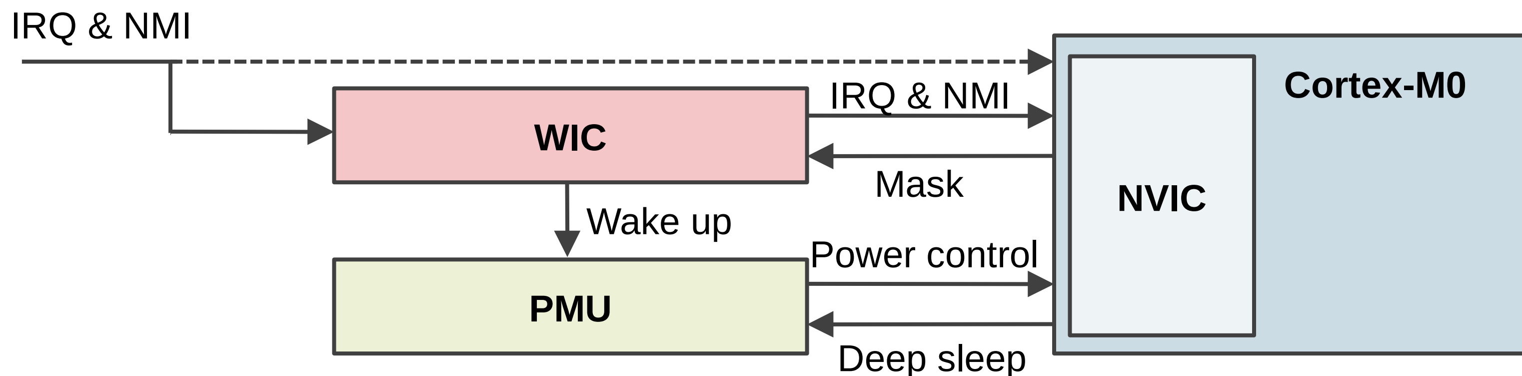
- The Sleep-on-Exit feature does not carry out unstacking process after the program returns from ISR;
- The power consumption is reduced since:
  - The execution of unnecessary program in the main thread is avoided;
  - The unnecessary stacking and unstacking operation is avoided.



Copyright © ARM Ltd 2013

Wakeup Interrupt Controller

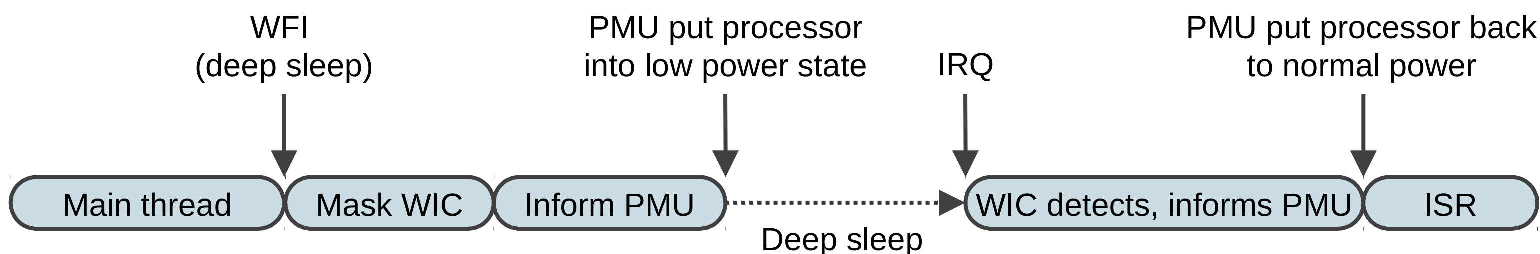
- The Wakeup Interrupt Controller (WIC) is an optional component used to make the wakeup decision in deep sleep mode;
  - WIC does not require extra programmable registers;
  - Usually requires a system-level Power Management Unit (PMU);
  - Used for deep sleep mode.



Copyright © ARM Ltd 2013

Enter and Exit Deep Sleep Mode

1. WFI instruction is executed for deep sleep mode;
2. Processor sends mask information to WIC;
3. Processor informs PMU to put into low power state;
4. Processor deep sleep;
5. IRQ occurs;
6. WIC firstly detects IRQ, and informs PMU to put power back to normal state;
7. WIC wakes up processor to start the ISR operation;



Copyright © ARM Ltd 2013

Developing Low-Power Applications

- Despite of the low-power features of the processor, there are various methods that can be taken to reduce power consumption of an application, for example:
  - Run the processor at a suitable clock frequency;
  - Disable a peripheral when not used;
  - For interrupt-driven application, the processor should stay in sleep mode as much as possible, e.g. use Sleep-on-Exit;
  - Optimize the application code for speed to reduce active cycles (this may be at the cost of a larger code size);
  - Turn off some of the clock signals or power supply when that piece of circuit is not used;
  - And so forth…

Useful Resources

- Reference1
  - Book: “The Definitive Guide to the ARM Cortex-M0” by Joseph Yiu, ISBN-10: 0123854776, ISBN-13: 978-0123854773, March 11, 2011
- Reference2
  - ARM v6-M Architecture Reference Manual:

    http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0419c/index.html
- Reference3
  - Cortex-M0 Technical Reference Manual:

    http://infocenter.arm.com/help/topic/com.arm.doc.ddi0432c/DDI0432C_cortex_m0_r0p0_trm.pdf

Copyright © ARM Ltd 2013